

Introduction to Python Programming



Gowrishankar S.
Veena A.



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Introduction to **Python Programming**



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Introduction to Python Programming

Gowrishankar S.
Veena A.



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2019 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper

International Standard Book Number-13: 978-0-8153-9437-2 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: S, Gowrishankar, author. | A, Veena author.
Title: Introduction to Python programming / Gowrishankar S, Veena A.
Description: Boca Raton : Taylor & Francis, a CRC title, part of the Taylor & Francis imprint, a member of the Taylor & Francis Group, the academic division of T&F Informa, plc, 2018. | Includes bibliographical references and index.
Identifiers: LCCN 2018046894 | ISBN 9780815394372 (hardback : alk. paper) | ISBN 9781351013239 (ebook)
Subjects: LCSH: Python (Computer program language)
Classification: LCC QA76.73.P98 S2325 2018 | DDC 005.13/3--dc23
LC record available at <https://lcn.loc.gov/2018046894>

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

Dedicated to my wife Roopa K. M. and to my sister Ashwini S. Nath.

—Dr. Gowrishankar S.

*I would love to dedicate this book to my parents, and my family
for their love, support and encouragement.*

—Veena A.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

Preface.....	xiii
Acknowledgment.....	xvii
Authors	xix
1. Introduction	1
1.1 What Is a Program?	1
1.2 Programming Languages.....	2
1.2.1 Machine Language.....	2
1.2.2 Assembly Language.....	3
1.2.3 High-Level Language	3
1.3 Software Development.....	5
1.4 History of Python Programming Language.....	7
1.5 Thrust Areas of Python.....	8
1.5.1 Academia.....	9
1.5.2 Scientific Tools	10
1.5.3 Machine Learning.....	10
1.5.4 Natural Language Processing	10
1.5.5 Data Analysis.....	10
1.5.6 Statistics.....	11
1.5.7 Hypertext Transfer Protocol (HTTP) Library.....	11
1.5.8 Database Connectors/ORM/NoSQL Connectors	11
1.5.9 Web Frameworks.....	11
1.5.10 Cloud Computing.....	11
1.5.11 Python Distributions	12
1.5.12 IDE Available	12
1.5.13 Community	12
1.5.14 Python Stack in Industry	12
1.6 Installing Anaconda Python Distribution	13
1.7 Installing PyCharm IDE to Set Up a Python Development Environment.....	16
1.8 Creating and Running Your First Python Project.....	19
1.9 Installing and Using Jupyter Notebook.....	23
1.9.1 Starting Jupyter Notebook	24
1.10 Open Source Software	27
1.10.1 Why Do People Prefer Using Open Source Software?	28
1.10.2 Doesn't "Open Source" Just Mean Something Is Free of Charge?	29
1.10.3 Open Source Licenses.....	29
1.11 Summary.....	32
Multiple Choice Questions	32
Review Questions	34
2. Parts of Python Programming Language	35
2.1 Identifiers	35
2.2 Keywords	36

2.3	Statements and Expressions	36
2.4	Variables	37
2.4.1	Legal Variable Names.....	37
2.4.2	Assigning Values to Variables	37
2.5	Operators.....	38
2.5.1	Arithmetic Operators.....	39
2.5.2	Assignment Operators.....	40
2.5.3	Comparison Operators	42
2.5.4	Logical Operators.....	43
2.5.5	Bitwise Operators.....	44
2.6	Precedence and Associativity	47
2.7	Data Types.....	48
2.7.1	Numbers	48
2.7.2	Boolean	48
2.7.3	Strings	48
2.7.4	None	49
2.8	Indentation.....	49
2.9	Comments	50
2.9.1	Single Line Comment	50
2.9.2	Multiline Comments.....	50
2.10	Reading Input.....	50
2.11	Print Output.....	51
2.11.1	<i>str.format()</i> Method	51
2.11.2	f-strings.....	53
2.12	Type Conversions.....	54
2.12.1	The <i>int()</i> Function	54
2.12.2	The <i>float()</i> Function.....	55
2.12.3	The <i>str()</i> Function	55
2.12.4	The <i>chr()</i> Function	56
2.12.5	The <i>complex()</i> Function	56
2.12.6	The <i>ord()</i> Function	57
2.12.7	The <i>hex()</i> Function.....	57
2.12.8	The <i>oct()</i> Function.....	57
2.13	The <i>type()</i> Function and Is Operator	58
2.14	Dynamic and Strongly Typed Language	58
2.15	Summary.....	59
	Multiple Choice Questions	60
	Review Questions	65
3.	Control Flow Statements.....	67
3.1	The <i>if</i> Decision Control Flow Statement	68
3.2	The <i>if...else</i> Decision Control Flow Statement.....	69
3.3	The <i>if...elif...else</i> Decision Control Statement	71
3.4	Nested <i>if</i> Statement	73
3.5	The <i>while</i> Loop.....	74
3.6	The <i>for</i> Loop	79
3.7	The <i>continue</i> and <i>break</i> Statements	81

3.8	Catching Exceptions Using <i>try</i> and <i>except</i> Statement	84
3.8.1	Syntax Errors.....	84
3.8.2	Exceptions	84
3.8.3	Exception Handling Using <i>try...except...finally</i>	85
3.9	Summary	89
	Multiple Choice Questions	90
	Review Questions	93
4.	Functions	95
4.1	Built-In Functions	95
4.2	Commonly Used Modules.....	97
4.3	Function Definition and Calling the Function	99
4.4	The <i>return</i> Statement and <i>void</i> Function	103
4.5	Scope and Lifetime of Variables	106
4.6	Default Parameters	108
4.7	Keyword Arguments.....	109
4.8	*args and **kwargs.....	110
4.9	Command Line Arguments	112
4.10	Summary	113
	Multiple Choice Questions	113
	Review Questions	117
5.	Strings.....	119
5.1	Creating and Storing Strings.....	119
5.1.1	The <i>str()</i> Function	120
5.2	Basic String Operations.....	120
5.2.1	String Comparison.....	122
5.2.2	Built-In Functions Used on Strings	122
5.3	Accessing Characters in String by Index Number.....	123
5.4	String Slicing and Joining.....	124
5.4.1	Specifying Steps in Slice Operation.....	126
5.4.2	Joining Strings Using <i>join()</i> Method	127
5.4.3	Split Strings Using <i>split()</i> Method	127
5.4.4	Strings Are Immutable	128
5.4.5	String Traversing	128
5.5	String Methods.....	131
5.6	Formatting Strings.....	138
5.6.1	Format Specifiers	140
5.6.2	Escape Sequences	141
5.6.3	Raw Strings	142
5.6.4	Unicodes	142
5.7	Summary	143
	Multiple Choice Questions	143
	Review Questions	146
6.	Lists.....	149
6.1	Creating Lists.....	149
6.2	Basic List Operations.....	151
6.2.1	The <i>list()</i> Function.....	151

6.3	Indexing and Slicing in Lists.....	152
6.3.1	Modifying Items in Lists.....	153
6.4	Built-In Functions Used on Lists	155
6.5	List Methods.....	156
6.5.1	Populating Lists with Items.....	158
6.5.2	Traversing of Lists.....	159
6.5.3	Nested Lists.....	167
6.6	The <i>del</i> Statement	169
6.7	Summary.....	170
	Multiple-Choice Questions.....	170
	Review Questions	173
7.	Dictionaries.....	175
7.1	Creating Dictionary.....	175
7.2	Accessing and Modifying <i>key:value</i> Pairs in Dictionaries.....	178
7.2.1	The <i>dict()</i> Function.....	179
7.3	Built-In Functions Used on Dictionaries	179
7.4	Dictionary Methods.....	181
7.4.1	Populating Dictionaries with <i>key:value</i> Pairs.....	183
7.4.2	Traversing of Dictionary	185
7.5	The <i>del</i> Statement	193
7.6	Summary.....	193
	Multiple Choice Questions.....	193
	Review Questions	198
8.	Tuples and Sets.....	201
8.1	Creating Tuples	201
8.2	Basic Tuple Operations.....	203
8.2.1	The <i>tuple()</i> Function	204
8.3	Indexing and Slicing in Tuples	205
8.4	Built-In Functions Used on Tuples	207
8.5	Relation between Tuples and Lists.....	208
8.6	Relation between Tuples and Dictionaries.....	209
8.7	Tuple Methods.....	210
8.7.1	Tuple Packing and Unpacking	211
8.7.2	Traversing of Tuples	211
8.7.3	Populating Tuples with Items.....	212
8.8	Using <i>zip()</i> Function.....	216
8.9	Sets	216
8.10	Set Methods	218
8.10.1	Traversing of Sets	219
8.11	Frozenset.....	221
8.12	Summary.....	222
	Multiple Choice Questions.....	222
	Review Questions	227

9. Files.....	229
9.1 Types of Files	230
9.1.1 File Paths.....	231
9.1.2 Fully Qualified Path and Relative Path.....	232
9.2 Creating and Reading Text Data.....	233
9.2.1 Creating and Opening Text Files	233
9.2.2 File <i>close()</i> Method	235
9.2.3 Use of <i>with</i> Statements to Open and Close Files.....	237
9.2.4 File Object Attributes.....	239
9.3 File Methods to Read and Write Data	239
9.4 Reading and Writing Binary Files	247
9.5 The Pickle Module	249
9.6 Reading and Writing CSV Files	251
9.7 Python <i>os</i> and <i>os.path</i> Modules	257
9.8 Summary	261
Multiple Choice Questions	262
Review Questions	265
10. Regular Expression Operations.....	267
10.1 Using Special Characters	267
10.1.1 Using <i>r</i> Prefix for Regular Expressions.....	272
10.1.2 Using Parentheses in Regular Expressions	272
10.2 Regular Expression Methods	273
10.2.1 Compiling Regular Expressions Using <i>compile()</i> Method of <i>re</i> Module.....	273
10.2.2 Match Objects	274
10.3 Named Groups in Python Regular Expressions	282
10.4 Regular Expression with <i>glob</i> Module	282
10.5 Summary	284
Multiple Choice Questions	284
Review Questions	287
11. Object-Oriented Programming	289
11.1 Classes and Objects	289
11.2 Creating Classes in Python	291
11.3 Creating Objects in Python	293
11.4 The Constructor Method	294
11.5 Classes with Multiple Objects.....	297
11.5.1 Using Objects as Arguments.....	301
11.5.2 Objects as Return Values.....	303
11.6 Class Attributes versus Data Attributes.....	306
11.7 Encapsulation	307
11.7.1 Using Private Instance Variables and Methods.....	309
11.8 Inheritance	311
11.8.1 Accessing the Inherited Variables and Methods	312
11.8.2 Using <i>super()</i> Function and Overriding Base Class Methods.....	314

11.8.3	Multiple Inheritances.....	317
11.8.4	Method Resolution Order (MRO)	320
11.9	The Polymorphism	328
11.9.1	Operator Overloading and Magic Methods.....	331
11.10	Summary.....	335
	Multiple Choice Questions.....	336
	Review Questions	338
12.	Introduction to Data Science.....	341
12.1	Functional Programming.....	341
12.1.1	Lambda.....	341
12.1.2	Iterators	342
12.1.3	Generators	343
12.1.4	List Comprehensions	344
12.2	JSON and XML in Python	346
12.2.1	Using JSON with Python	347
12.2.2	Using Requests Module.....	353
12.2.3	Using XML with Python	355
12.2.4	JSON versus XML	359
12.3	NumPy with Python	359
12.3.1	NumPy Arrays Creation Using <i>array()</i> Function.....	360
12.3.2	Array Attributes	361
12.3.3	NumPy Arrays Creation with Initial Placeholder Content.....	362
12.3.4	Integer Indexing, Array Indexing, Boolean Array Indexing, Slicing and Iterating in Arrays	364
12.3.5	Basic Arithmetic Operations on NumPy Arrays.....	367
12.3.6	Mathematical Functions in NumPy	368
12.3.7	Changing the Shape of an Array	369
12.3.8	Stacking and Splitting of Arrays.....	370
12.3.9	Broadcasting in Arrays.....	371
12.4	Pandas.....	374
12.4.1	Pandas Series	375
12.4.2	Pandas DataFrame	380
12.5	Altair.....	398
12.6	Summary.....	409
	Multiple Choice Questions.....	410
	Review Questions	413
	Appendix-A: Debugging Python Code.....	415
	Bibliography.....	425
	Solutions	427
	Index	437

Preface

This book presents an intuitive approach to the concepts of Python Programming for students. It is appropriate for courses generally known as “Introduction to Python Programming.” We have tried to write a book that assists students in discovering the power of Python programming. We have taken into account the reality that students taking “Introduction to Python Programming” course are likely to come from a variety of disciplines. In addition to Computer Science majors, there tend to be students from other majors like other engineering streams, physics, chemistry, biology, environmental science, geography, economics, psychology and business.

This book differs from traditional texts not only in its philosophy but also in its overall focus, level of activities, development of topics, and attention to programming details. The emphasis is on understanding Python programming concepts. Reading a programming book is different from reading a newspaper or a novel. Don’t be discouraged if you have to read a passage more than once in order to understand it. We recommend that you keep this book for reference purposes after you finish working through the course. Because you will likely forget some of the specific details of Python programming language, the book will serve as a useful reminder. Students will appreciate the many programming examples within the text. Programs are carefully selected to bring the theoretical concepts to fruition. Our aim is to get the reader to productivity as quickly as possible without sacrificing the overall flow quality.

In fact, if you are a novice programmer, with some dedication and hard work you should be able to learn Python Programming as your first programming language. As we introduce each new feature of the language, we usually provide a complete program as an example to illustrate the feature. Just as a picture is worth a thousand words, so is a properly chosen programming example. We present the material in a way to encourage student thinking and we show students how to generalize key concepts once they are introduced, which can be used to solve real-world problems. Programming questions that accompany the end of each chapter are based on worked examples which help students to gain solid knowledge of the basics and assess their own level of understanding before moving on. Students are highly encouraged to solve these programs to gain a solid hold on Python programming language.

This book takes you through step by step process of learning the Python programming language. Each line of the code is marked with numbers and is explained in detail. In this book all the names of variables, strings, lists, dictionaries, tuples, functions, methods and classes consist of several natural words and in the explanation part they are written in italics to indicate the readers that they are part of programming code and to distinguish them from normal words. This programming style of using readable natural names makes the reading of code lot easier and prevents programming errors. Learning outcome component is mentioned at the beginning of each chapter that calls the attention of the readers to important items in the chapter. A summary of important concepts appears at the end of each chapter.

We hope that students using the book will come away with an appreciation of the beauty, power, and tremendous utility of Python programming language and that they will have fun along the way.

Organization of Chapters

Here's a brief rundown of what you will find in each chapter:

- In [Chapter 1](#), the advantages of using Python programming language is discussed and the scope of Python's reach, and all the different areas of application development in which Python plays a part is identified. This chapter also covers the downloading and installation of Anaconda distribution and PyCharm IDE. You will be guided towards setting up your own Python development environment. You will understand the meaning of Open Source Software and its various licenses.
- In [Chapter 2](#), you will learn the basic building blocks of Python programming language like arithmetic operators, data types, operator precedence and associativity and construct complex expressions using operators. This chapter teaches you the syntax of Python language which you need to know before writing Python programs.
- In [Chapter 3](#), Python statements such as `if`, `if...else`, `if...elif...else` are taught which are used to transfer the control from one part of the program to another. Loops to run one or more statements repeatedly are examined. Controlling the flow of execution is carried out through the `break` and `continue` statements. To take advantage of the iterative capabilities in a computer, you need to know how to include loops and conditional logic within your program. How to deal with errors in the input is also part of this chapter.
- In [Chapter 4](#), you will determine how to create functions, pass values to and return values from the function. In this chapter, commonly used modules and built-in functions are identified. This chapter also includes discussion on different function features like keyword arguments and variable number of arguments. You'll learn how to use functions to make your code more efficient and flexible.
- In [Chapter 5](#), some of the more commonly occurring string tasks, such as concatenating strings, trimming of white spaces, splitting string as well as finding substrings within strings is covered. Various string methods to manipulate strings are described in detail. Indexing, slicing and joining of strings are illustrated through figures. Formatting of the string using f-strings is also outlined.
- In [Chapter 6](#), one of the main pillars of Python programming language and the built-in data type, Lists, is discussed which acts as a container to hold items of different types. Creating, slicing and indexing of lists is elaborated with high-grade examples. Use of list methods in manipulating Python programs is demonstrated.
- In [Chapter 7](#), another built-in data type in Python programming language, Dictionary, is covered in depth. Accessing and modifying the key:value pairs of Dictionaries is demonstrated.
- In [Chapter 8](#), role of tuples in returning multiple values from functions and storing heterogeneous elements of fixed sized is discussed. Mathematical operations like union and intersection are demonstrated using sets. Different methods supported by tuples and sets are identified.
- In [Chapter 9](#), you will deal with file-based input and output functions, including how to load files, perform basic operations, and save the results back to disk.

The difference between a text file and a binary file is covered. Functions needed for manipulation of CSV files programmatically are also addressed. This chapter shows the modules required for navigating and accessing file systems using the Python programming language.

In [Chapter 10](#), regular expression concepts are introduced. You explore the use of regular expressions in Python programs through `re` module.

In [Chapter 11](#), you will look at how Object-oriented paradigm is implemented in Python programming language. You will look at creating classes and objects. Various Object-oriented concepts like encapsulation, inheritance and polymorphism are explained. This chapter also introduces the concept of operator overloading which plays a pivotal role in program development. These concepts transcend a particular programming language and are important to the success of an Object-oriented programming solution.

In [Chapter 12](#), you begin with Functional programming features of Python programming language. You will see how to handle common data formats like JSON and XML. Modules like NumPy and pandas, required to develop Data Science solutions, are also touched upon in this chapter. Altair data visualization library for Python is introduced in this chapter.

In the [Appendix-A](#), steps involved in debugging a Python program are listed.

Who Should Read This Book?

This book is for beginners to Python programming who are interested in learning the basics of Python programming language.

Availability of Source Code

As you work through the programming examples given in the book, you can either type the programs manually or download the programs from https://github.com/gowrishankarnath/Introduction_To_Python_Programming. You are strongly encouraged to run each of these programs and compare the results obtained in your system to those shown in the text. By doing so, you will learn the language and its syntax.

Software Requirements

All the code in the book works with Python 3.6 version or above. Since f-strings are used throughout the book which was introduced in Python 3.6 version, you need to have Python interpreter 3.6 or above for the code to work as it is. Install the latest version of 64-bit Anaconda Distribution with support for Python 3.6 or above. If you want to make the code work with Python 3.1 to 3.5 versions, use `str.format()` statements within the `print()` function instead of f-strings. The rest of the statements within the code remains the same and the code should work fine. It is highly recommended that you use the latest version of PyCharm Community Edition to execute the programs. You will gain valuable insights in to Python programming language by using this IDE, no matter whether you are a beginner or an experienced professional.

Icons Used



It is a technical trivia which you might find interesting. Please take note of it.



There are a few concepts which we may have to glance at beforehand itself without dwelling deep into it. But if you want to find more about the same then we indicate it in here.

Support for the Book

We will try to address your questions within an appropriate timeframe. Understand that we have a day job just like you and we may not be able to respond immediately. Rest assured we will respond to genuine questions and we do not like to keep our readers in the dark whatsoever.

This Book Is Not for You

If you are an advanced Python Programmer or if you are a Professional who is well versed in another programming language, then this book is not for you. The programming examples and the explanations are kept simple. Please understand that with your advanced knowledge you might feel that we are apparently dealing with basic stuff but what might be a basic material to you is not the same to someone who is learning to write code for the first time.

Errata

We hope to improve this book continually. If you have any suggestions for improving this text or if an error should be found, the authors would be grateful if notification were sent by e-mail to gowrishankarnath@acm.org. To ensure your messages do not end up in our junk mail folder, please include subject as "Introduction to Python Programming."

Acknowledgment

Writing a book requires the contribution of many individuals. I have been blessed with the support of many people who have encouraged me all along the way to ensure this project reached the stage of completion. I have tried to acknowledge every one of those of whom I'm conscious of and hope others will forgive me for lapses in my memory.

I would like to express deep appreciation to my co-author Mrs. Veena A., Assistant Professor, Dr.AIT, who was absolutely wonderful to work with in crafting this book.

I would like to thank the Management and Principal of Dr. Ambedkar Institute of Technology (Dr.AIT), Bengaluru; Sri S. Mariswamy (The Honourable Chairman, PVPWT), Sri A. R. Krishnamurthy (The Honourable Secretary, PVPWT), Sri P. L. Nanjundaswamy (Trustee, PVPWT), Sri S. Shivamallu (Trustee, PVPWT), Dr. M. Mahadeva (Trustee, PVPWT) and Dr. C. Nanjundaswamy (Principal, Dr.AIT) for their positive influence, kind support and encouragement during this project.

I wish to thank the following members for providing direction on the development of this book; Dr. H. N. Jagannatha Reddy (Registrar, Visvesvaraya Technological University), Dr. Subir Kumar Sarkar (Professor, Jadavpur University), Dr. T. G. Basavaraju (Professor, Government SKSJT), Dr. C. N. Ravi Kumar (Principal, Mysore College of Engineering and Management), Dr. Basavaraj Anami (Principal, KLE Institute of Technology), Dr. G. N. Krishna Murthy (Principal, B.N.M. Institute of Technology), Dr. H. S. Arvinda (Professor, JSS Academy of Technical Education), Dr. D. H. Manjaiah (Professor, Mangalore University), Dr. H. S. Guruprasad (Professor, BMS College of Engineering), Dr. S. N. Chandrashekara (Professor, C Byregowda Institute of Technology), and Mr. Sharath Malve (Network Architect, Honeywell India).

I am lucky to work with wonderful colleagues at Dr.Ambedkar Institute of Technology, Bengaluru, India. In one way or another, they all have influenced me and my teaching which helped in shaping this book. I thank them all and would like to add particular mention of these; Dr. Siddaraju (Professor, Dr.AIT), Dr. Meenakshi M. Bhat (Professor, Dr.AIT), Mrs. Asha (Associate Professor, Dr.AIT), Mrs. Leena Giri G (Associate Professor, Dr.AIT), Mr. Shamshekhar S. Patil (Associate Professor, Dr.AIT), Mr. A. H. Srinivasa (Associate Professor, Dr.AIT), Mr. D. Suresha (Associate Professor, Dr.AIT), Mr. G. Harish (Associate Professor, Dr.AIT), Dr. S. Prakash (Associate Professor, Dr.AIT), Mr. M. V. Praveena (Assistant Professor, Dr.AIT) and Mrs. Lavanya Santhosh (Assistant Professor, Dr.AIT).

Here is the list of developers who have enthused me a lot towards programming: Wes McKinney (Creator, pandas library), Scott Hanselman (Principal Program Manager, Microsoft), Armin Ronacher (Creator, Flask Web Framework), Kenneth Reitz (Creator, Requests library), Jake Vanderplas (Co-creator, Altair library), Brian Granger (Co-founder, Project Jupyter), Adrian Holovaty (Co-creator, Django Web Framework), Hadley Wickham (Chief Scientist, RStudio), David Robinson (Chief Data Scientist, DataCamp), Jon Skeet (Developer, Google) and Mara Averick (Tidyverse Dev Advocate, RStudio).

We owe a deep debt of gratitude to Mrs. Aastha Sharma, Senior Commission Editor, CRC Press/Taylor & Francis, and Ms. Shikha Garg, Editorial Assistant, CRC Press/Taylor & Francis, for keeping faith in us even when deadlines slipped and for keeping our book on the track by responding to our questionnaires promptly. We are grateful to various

reviewers who reviewed this book and provided feedback and corrections. Your suggestions have significantly improved this book. I like to thank Mrs. Shannon Stanton, the production manager from Lumina Datamatics for her untiring effort in reproducing various drafts of the manuscript before sending the book to print. We also express our thanks to all the people at CRC Press/Taylor & Francis who contributed their efforts to the production of this book.

I cannot forget the unending love and support given by my parents, Pushpalatha S. Nath and Dr. S. Surendranath. It is from them I learned the virtue of patience and that never-say-die attitude. Without their proper guidance I would have become lost in due course of my life. What I am today is only because of them. My sisters, Ashwini S. Nath and Dr. Shalini S. Nath, are the ones with whom I shared all the finer moments of my growing years. You all are my inspiration and motivation for continuing to improve my knowledge and move my career forward.

I now know that writing a book is a colossal task that keeps a person away from family activities. No words can describe the amount of sacrifice my wife Roopa K. M. and son Yashmith G. Nath have done during the preparation of this book and those silent tears that have been shed for sacrificing the family time together. Their cheerful smile, devoted accompaniment and constant love kept me moving forward.

I would also like to thank all my friends and relatives who in some way have helped me to achieve this task.

Finally, I would like to thank the Almighty God for giving me that inner strength to complete this project.

Dr. Gowrishankar S.

“Introduction to Python Programming” was a delight and a brainstorming kind of a project. Making this book a reality takes many dedicated people and it is my great pleasure to acknowledge their contributions.

I am grateful to Dr. Gowrishankar S., who first conceived the idea of writing this book and kept faith in me and gave me an opportunity to be the co-author of this book.

I wish to acknowledge the direct and indirect contributions and assistance of various colleagues and friends with whom I have collaborated. I would like to thank our publisher Taylor & Francis/CRC Press, for accepting our book proposal. I would like to thank our team at Taylor & Francis/CRC Press especially Aastha Sharma, Senior Commissioning Editor and Shikha Garg, Editorial Assistant for the excellent collaboration, follow-up and timely replies to emails. I also thank the production team as without them this project would not have achieved what it is now.

I anticipate the readers will enjoy the book and welcome any suggestions and feedback for the improvement of this book.

Veena A.

Authors



Dr. Gowrishankar S. is currently working as Associate Professor in the Department of Computer Science and Engineering at Dr.Ambedkar Institute of Technology, Bengaluru, India.

He earned his PhD in Engineering from Jadavpur University, Kolkata, India in 2010 and MTech in Software Engineering and BE in Computer Science and Engineering from Visvesvaraya Technological University (VTU), Belagavi, India, in the years 2005 and 2003 respectively.

From 2011 to 2014 he worked as senior research scientist and tech lead at Honeywell Technology Solutions, Bengaluru, India. He has been awarded with the Technical and Innovation Award and Individual Excellence Award for his contribution towards

successful delivery of projects at Honeywell Technology Solutions.

He has published several papers in various reputed *International Journals and Conferences*. He is serving as editor and reviewer for various prestigious International Journals. He is also member of IEEE, ACM, CSI and ISTE.

He has delivered many keynote addresses and invited talks throughout India on a variety of subjects related to Computer Science and Engineering. He was instrumental in organizing several conferences, workshops and seminars. He has also served on the panel of number of Academic Bodies of Universities and Autonomous Colleges as a BOS and BOE member.

His current research interests are mainly focused on Data Science, including its technical aspects as well as its applications and implications. Specifically, he is interested in the applications of Machine Learning, Data Mining and Big Data analytics in Healthcare. He writes articles on his personal blog at <http://www.gowrishankarnath.com>. His Twitter handle is @g_s_nath.



Mrs. Veena A. is currently working as Assistant Professor in the Department of Computer Science and Engineering at Dr.Ambedkar Institute of Technology, Bengaluru, India, since January 2016.

She completed her MTech in Computer Science and Engineering from PESIT, Bengaluru, India and BE in Information Science and Engineering from Visvesvaraya Technological University (VTU), Belagavi, India, in the years 2011 and 2004 respectively.

From 2004 to 2006 she worked as Software Developer at Envision Network Technologies, Bengaluru, India, and worked as member technical staff from 2006 to 2007 at CDAC, Pune, India.

She has published several papers in various reputed International Journals and Conferences.

Her current research interests are Machine Learning, Data Mining and Big Data Analytics in Healthcare.

She is currently pursuing her PhD in Computer Science and Engineering from VTU, Belagavi, India.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

1

Introduction

AIM

Setup a Python development environment and understand the importance of Open source software and its different licenses.

LEARNING OUTCOMES

At the end of the chapter, you are expected to

- Identify various domains where Python plays a significant role.
- Install and run the Python interpreter.
- Create and execute Python programs using PyCharm IDE and Jupyter Notebook.
- Understand the meaning of Open Source Software and its different licenses.

Python is a free general-purpose programming language with beautiful syntax. It is available across many platforms including Windows, Linux and Mac OS. Due to its inherently easy to learn nature along with Object Oriented features, Python is used to develop and demonstrate applications quickly. Python has the “batteries included” philosophy wherein the standard programming language comes with a rich set of built-in libraries. It’s a known fact that developers spend most of the time reading the code than writing it and Python can speed up software development. Hosting solutions for Python applications are also very cheap. Python Software Foundation (PSF) nurtures the growth of Python programming language. A versatile language like Python can be used not only to write simple scripts for handling file operations but also to develop massively trafficked websites for corporate IT organizations.

1.1 What Is a Program?

The ability to code computer programs is an important part of literacy in today’s society. A program is a set of instructions instructing a computer to do specific tasks. “Software” is a generic term used to describe computer programs. Scripts, applications,

programs and a set of instructions are all terms often used to describe software. The software can be categorized into three categories,

System software includes device drivers, operating systems (OSs), compilers, disk formatters, text editors and utilities helping the computer to operate more efficiently. System software serves as a base for application software. It is also responsible for managing hardware components.

Programming software is a set of tools to aid developers in writing programs. The various tools available are compilers, linkers, debuggers, interpreters and text editors.

Application software is intended to perform certain tasks. Examples of application software include office suites, gaming applications, database systems and educational software. Application software can be a single program or a collection of small programs. This type of software is what consumers most typically think of as “Software.”

There are myriad of areas where programs are used like supermarkets, banks, insurance industries, process control, hospitals, offices, government institutions, education, research, telecommunication, transport industry, police, defense, multimedia applications, entertainment systems, library services and many more.

1.2 Programming Languages

Computers cannot write programs on their own as they don't understand human needs unless we communicate with the computer through programming languages. A programming language is a computer language engineered to communicate instructions to a machine. Programs are created through programming languages to control the behavior and output of a machine through accurate algorithms, similar to the human communication process.

1.2.1 Machine Language

Machine language, also called machine code, is a low-level computer language that is designed to be directly understandable by a computer and it is the language into which all programs must be converted before they can be run. It is entirely comprised of binary, 0's and 1's. In machine language, all instructions, memory locations, numbers and characters are represented in 0's and 1's. For example, a typical piece of machine language might look like, 00000100 10000000.

The main advantage of machine language is that it can run and execute very fast as the code will be directly executed by a computer and the programs efficiently utilize memory.

Some of the disadvantages of machine language are,

- Machine language is almost impossible for humans to use because it consists entirely of numbers.
- Machine language programs are hard to maintain and debug.
- Machine language has no mathematical functions available.
- Memory locations are manipulated directly, requiring the programmer to keep track of every memory location.

1.2.2 Assembly Language

Machine language is extremely difficult for humans to read because it consists merely of patterns of bits (i.e., 0's and 1's). Thus, programmers who want to work at the machine language level instead usually use assembly language, which is a human-readable notation for the machine language. Assembly language replaces the instructions represented by patterns of 0's and 1's with alphanumeric symbols also called as mnemonics in order to make it easier to remember and work with them including reducing the chances of making errors. For example, the code to perform addition and subtraction is,

```
ADD 3, 5, result
```

```
SUB 1, 2, result
```

Because of alphanumeric symbols, assembly language is also known as *Symbolic Programming Language*. The use of mnemonics is an advantage over machine language. Since the computer cannot understand assembly language, it uses another program called assembler. Assembler is a program that is used to convert the alphanumeric symbols written in assembly language to machine language and this machine language can be directly executed on the computer.

Some of the disadvantages of Assembly language are,

- There are no symbolic names for memory locations.
- It is difficult to read.
- Assembly language is machine-dependent making it difficult for portability.

1.2.3 High-Level Language

High-level language is more like human language and less like machine language. High-level languages are written in a form that is close to our human language, enabling programmers to just focus on the problem being solved. High-level languages are platform independent which means that the programs written in a high-level language can be executed on different types of machines. A program written in the high-level language is called source program or source code and is any collection of human-readable computer instructions. However, for a computer to understand and execute a source program written in high-level language, it must be translated into machine language. This translation is done using either compiler or interpreter.

Advantages

- Easier to modify, faster to write code and debug as it uses English like statements.
- Portable code, as it is designed to run on multiple machines.

A compiler is a system software program that transforms high-level source code written by a software developer in a high-level programming language into a low-level machine language. The process of converting high-level programming language into machine language is known as compilation. Compilers translate source code all at once and the computer then executes the machine language that the compiler produced. The generated machine language can be later executed many times against different data each time. Programming languages like C, C++, C# and Java use compilers. Compilers can be

classified into native-code compilers and cross compilers based on their input language, output language and the platform they run on. A compiler that is intended to produce machine language to run on the same platform that the compiler itself runs on is called a native-code compiler. A cross compiler produces machine language that is intended to run on a different platform than it runs on.

Not all source code is compiled. With some programming languages like Python, Ruby and Perl the source code is frequently executed directly using an interpreter rather than first compiling it and then executing the resulting machine language. An interpreter is a program that reads source code one statement at a time, translates the statement into machine language, executes the machine language statement, then continues with the next statement. It is generally faster to run compiled code than to run a program under an interpreter. This is largely because the interpreter must analyze each statement in the source code each time the program is executed and then perform the desired conversion, whereas this is not necessary with compiled code because the source code was fully analyzed during compilation. However, it can take less time to interpret source code than the total time needed to both compile and run it, and thus interpreting is frequently used when developing and testing source code for new programs.

A programming paradigm is a style, or “way” of programming. Major programming paradigms are,

- Imperative
- Logical
- Functional
- Object-Oriented

It can be shown that anything solvable using one of these paradigms can be solved using the others; however, certain types of problems lend themselves more naturally to specific paradigms and there will be some overlap between different paradigms.

Imperative

Imperative programming is a paradigm of computer programming in which the program describes a sequence of steps that change the state of the computer as each one is executed in turn. Imperative programming explicitly tells the computer “how” to accomplish a certain goal. Structured programming, on the other hand, is a subset of Imperative programming, which emerged to remove the reliance on the GOTO statement by introducing looping structures. Then you also have Procedural programming, which is another subset of Imperative programming, where you use procedures to describe the commands the computer should perform. Procedural programming refers to the ability to combine a sequence of instructions into a procedure so that these instructions can be invoked from many places without resorting to duplicating the same instructions. The difference between procedure and function is that functions return a value, and procedures do not. An assembly language is an imperative language which is NOT structured or procedural. Popular programming language like C is imperative and structured in nature.

Logical

The logical paradigm fits exceptionally well when applied to problem domains that deal with the extraction of knowledge from basic facts and rules. Rules are written as logical

clauses with a head and a body; for instance, "Y is true if X1, X2, and X3 are true." Facts are expressed similar to rules, but without a body; for instance, "Y is true." The idea in logical programming is that instead of telling the computer how to calculate things, you tell it what things are. Example: PROLOG.

Functional

In functional programming languages, functions are treated as first-class objects. In other words, you can pass a function as an argument to another function, or a function may return another function. Examples of functional programming languages are F#, LISP, Scheme, and Haskell.

Object-Oriented

The Object-oriented paradigm has gained enormous popularity in the recent decade. Object-oriented is the term used to describe a programming approach based on objects and classes. The object-oriented paradigm allows us to organize software as a collection of objects that consist of both data and behavior. This lets you have nice things like encapsulation, inheritance, and polymorphism. These properties are very important when programs become larger and larger. The object-oriented paradigm provides key benefits of reusable code and code extensibility. Examples of object-oriented languages are Python, C++, Java and C#.

It is important to note that many languages, such as Python and C++, support multiple paradigms. It is also true that even when a language is said to support a particular paradigm, it may not support all the paradigm's features. Not to mention that there is a lot of disagreement as to which features are required for a particular paradigm.

1.3 Software Development

Software development is a process by which stand-alone or individual software is created using a specific programming language. It involves writing a series of interrelated programming code, which provides the functionality of the developed software. Software development may also be called application development.

The process of software development goes through a series of stages in stepwise fashion known as the Software Development Life Cycle (SDLC). It is a systematic approach to develop software (FIGURE 1.1). It creates a structure for the developer to design, create and deliver high-quality software according to the requirements of the customer. It also provides a methodology for improving the quality of the desired product.



FIGURE 1.1

Different stages of Software Development Life Cycle.

The purpose of the SDLC process is to provide help in producing a product that is cost effective and of high quality. Different stages of the Software Development Life Cycle are,

Planning of Project. At this stage, the total number of resources required to implement this project is determined by estimating the cost and size of the software product.

Analysis and Requirement Gathering. At this stage, the maximum amount of information is collected from the client about the kind of software product he desires. Different questions are posed to the client like: Who is going to use the product? How will they use the product? What kind of data is given as input to the product? What kind of data is expected as output from the product? Questionnaires enable the development team to gather overall specification of the product in good detail. The software development team then analyzes all these requirements of the client, keeping in mind the design constraints, coding standards and its validity. The aim of analysis and requirements gathering stage is to understand the requirements of the client by all the members of the software development team and see how these requirements can be implemented.

Design. At this stage, the software development team analyzes whether the software can be implemented with all the features as specified by the client. Also, the development team has to convince the client about the financial feasibility and technological viability. The software development team has to select the programming language and the platform to implement the software that is best suited to satisfy the requirements of the client. Software design helps the development team to define and understand the overall architecture required for the software product and the approach is captured in detail in a design document.

Development. At this stage, the development team starts building the software according to the design document. The development team translates the design into a set of programs that adhere to coding standards of their organization. Coding is done by dividing the specification mentioned in the design document into different modules to provide a working and reliable product. This is the longest phase of SDLC.

Testing. At this stage, the software product is tested against the requirements specified by the client to ensure the product is working as expected. The testing team is mainly responsible for checking the system to weed out bugs and to verify that the software product is working as expected. Any bugs that are found in the process or any shortcomings in the features of the software product is conveyed to the software development to rectify. This is the last stage of overall software development before handing over the product to the client.

Deployment. At this stage, the product is released to the client to use after testing the product thoroughly to match the requirements of the client. The client needs to be trained in using the software and documents should be provided containing instructions on how to operate the software in a user-friendly language.

Maintenance. The process of taking care of the developed and deployed software product is known as Maintenance. When the customer starts using the deployed software product, unforeseen problems may come up and these need to be solved. Also, new requirements may come up at the client's workplace and the software needs to be updated and upgraded to accommodate these changes.

1.4 History of Python Programming Language

The history of the Python programming language dates back to the late 1980s. Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum ([FIGURE 1.2](#)) at CWI in the Netherlands as a successor to the ABC programming language capable of exception handling and interfacing with the Amoeba operating system. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community. He is the "Benevolent Dictator For Life" (BDFL), which means he continues to oversee Python development and retains the final say in disputes or arguments arising within the community.



FIGURE 1.2

Guido van Rossum—Creator and BDFL of Python Programming Language. (Image courtesy of Wikipedia.org)

Often people assume that the name Python was written after a snake. Even the logo of Python programming language ([FIGURE 1.3](#)) depicts the picture of two snakes, blue and yellow. But, the story behind the naming is somewhat different.



FIGURE 1.3

Python logo. (Image courtesy of Python.org)

Back in the 1970s, there was a popular BBC comedy TV show called "Monty Python's Flying Circus" and Van Rossum happened to be a big fan of that show. At the time when he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus." It occurred to him that he needed a name that was short, unique, and slightly mysterious, so he decided to call the language "Python."

Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code that would not be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale. Python is a multi-paradigm programming language having full support for Object-oriented programming and Structured programming and there are a number of language features which support Functional programming.

The first ever version of Python (i.e., Python 1.0) was introduced in 1991. Since its inception and introduction of Version 1, the evolution of Python has reached up to Version 3.x (till 2018). Python 2.0 was released on 16 October 2000 and had many major new features, including a cycle-detecting garbage collector and support for Unicode. With this release, the development process became more transparent and community-backed. Python 3.0 (initially called Python 3000 or py3k) was released on 3 December 2008 after a long testing period. It is a major revision of the language that is not completely backward-compatible with previous versions.

The language's core philosophy is summarized in the document *The Zen of Python*, which includes principles such as,

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

1.5 Thrust Areas of Python

Python is a top marketable professional skill known for its simplicity and developer friendliness. Python has a solid claim to being the fastest-growing major programming language. Since 2003, Python has been consistently ranked in the top ten most popular programming

languages as measured by the TIOBE Programming Community Index. As of April 2018, it is in the fourth position. Python is ranked at first position by IEEE Spectrum ranking of top programming languages for the year 2017 (FIGURE 1.4) and RedMonk Programming language rankings for the year 2018 has listed Python at the third position.




































Language Rank	Types	Spectrum Ranking
1. Python	 	100.0
2. C	  	99.7
3. Java	  	99.4
4. C++	  	97.2
5. C#	  	88.6
6. R		88.1
7. JavaScript	 	85.5
8. PHP		81.4
9. Go	 	76.1
10. Swift	 	75.3
11. Arduino		73.0
12. Ruby	 	72.4
13. Assembly		72.1
14. Scala	 	68.3
15. Matlab		68.0
16. HTML		67.0
17. Shell		66.3
18. Perl	 	57.6
19. Visual Basic		55.4
20. Cuda		53.9

FIGURE 1.4

Ranking of programming languages by IEEE. (Image courtesy of *IEEE Spectrum*, New York.)

1.5.1 Academia

Python is being offered as the introductory programming language in the majority of the computer science departments at various American universities. Python is being adapted by academia for research purposes at an accelerated rate and is competing with Matlab for the coveted title of most preferred language for research. There are a few advantages of Python over Matlab, like Matlab is not a real programming language but Python is. Python has lots of scientific tools which are almost as good as Matlab modules. Developers nowadays need to work in multiple languages and the majority of the languages, including Python, have their index starting from zero while Matlab index starts from 1, which may lead to more syntactical errors. Also, Matlab uses parentheses both for indexing and functions, while Python uses the square brackets for indexing and parentheses for functions which brings more clarity in the code. Matlab is closed and proprietary with a very expensive licensing agreement, while Python is free and open source. Raspberry Pi project was started by Raspberry Pi foundation which aims to bring computer knowledge to children, elderly people and

the lower strata of the society who are deprived of computer education. This foundation brings out Raspberry Pi devices, which are tiny, low-cost microcomputers that are powerful enough to do most of the work that can be done using a desktop. Python, due to its ease of learning, is recommended as the preferred programming language for Raspberry Pi.

1.5.2 Scientific Tools

Scientific tools are essential for simulating and analyzing complex systems. The Python ecosystem consists of these core scientific packages, namely SciPy library, NumPy, Jupyter, Sympy and Matplotlib. Most of these tools are available under Berkeley Software Distribution (BSD) license and can be used without any restrictions. SciPy library is mainly used for numerical integration and optimization. NumPy provides N-dimensional array objects which can be used to perform linear algebra, Fourier transform and other mathematical operations. Jupyter has revolutionized the way programming is done in Python. Jupyter provides an interactive web-based interface which can be invoked from a browser. Jupyter is used to write Python programs and create embeddable plots to visualize data. Sympy library is used to generate symbolic mathematics. Matplotlib is the oldest and most popular plotting library available for Python. With these tools, we have better chances of solving scientific problems and create working prototypes more quickly than any other competing tools.

1.5.3 Machine Learning

Machine Learning is an effective and adaptive tool to learn from experience and by a dataset. Many machine-learning algorithms and techniques have been developed that allow computers to learn. Machine Learning has its origin in Computer Science and Statistics. Scikit-Learn is a well-known Machine Learning tool built on top of other Python scientific tools like NumPy, SciPy and Matplotlib. This allows Scikit-Learn to be easily extended to implement new models. Scikit-Learn supports various models for Classification, Regression, Clustering, Model Selection, Dimensionality Reduction and Preprocessing. Some of the advantages of Scikit-Learn are integration of parallelization, consistent APIs, good documentation and it is available under BSD license as well as commercial license with full support.

1.5.4 Natural Language Processing

Natural language processing is used to read and understand the text. Natural Language Toolkit (NLTK) is the popular library used for natural language processing in Python. NLTK has numerous trained algorithms to understand the text. NLTK has huge corpora of datasets and lexical resources like journals, chat logs, movie reviews and many more. NLTK is available under Apache License V2.0.

1.5.5 Data Analysis

Pandas library changed the landscape of data analysis in Python altogether and is available under BSD license. Pandas is built on top of NumPy and has two important data structures, namely Series and DataFrame. It can hold any type of data like integers, floats, strings, objects and others. Each of the data stored in series is labeled after the index. DataFrame is a tabular data structure with labeled rows and columns similar to Excel spreadsheet. In the real world, data is never in order and pandas can be used to fill in missing data, reshaping of datasets, slicing, indexing, merging, and joining of datasets. Pandas can be used to read Comma-Separated Values (CSV) files, Microsoft Excel, Structured Query Language (SQL) database and Hierarchical Data Format (HDF5) format files.

1.5.6 Statistics

Statsmodels is a Python library used for statistical analysis. It supports various models and features like linear aggression models, generalized linear models, discrete choice models and functions for time series analysis. To ensure the accuracy of results Statsmodels is tested thoroughly by comparing it with other statistical packages. Statsmodels can also be used along with Pandas to fit statistical models. This package is available under modified BSD license. Statsmodels is used across various fields like economics, finance and engineering.

1.5.7 Hypertext Transfer Protocol (HTTP) Library

The Requests HTTP library is popularly referred to as the library written for humans. Python has a standard HTTP library called `urllib.request` to carry out most of the HTTP operations. But the Application Programming Interfaces (APIs) of `urllib.request` are not easy to use and are verbose. To overcome these problems Requests was created as a stand-alone library. Common HTTP verbs like POST, GET, PUT and DELETE which correspond to create, read, update and delete operations are fully supported. Also, Requests provides features like thread-safety, support for International Domains, Cookie Persistence and Connection Timeouts. Requests library is available under Apache license 2.0.

1.5.8 Database Connectors/ORM/NoSQL Connectors

Database connectors are drivers that allows us to query the database from the programming language itself. MySQL and PostgreSQL are the popular open source databases. MySQL-Python-Connector for Python from Oracle is the most popular Python connector available for MySQL and Psycopg2 is the Python connector widely used for PostgreSQL.

Object Role Modeling (ORM) is a powerful way of querying the database to achieve persistence so that data can live beyond the application process. There is a mismatch between the Object-oriented language models and the Relational databases leading to several problems like granularity, inheritance, identity, associations and navigations. ORM helps in mapping the data from Object-oriented languages to the Relational databases and follows the business layer logic. SQLAlchemy is a highly recommended ORM toolkit for Python applications to be deployed at the enterprise level. Python connectors are also available for popular NoSQL databases like MongoDB and Cassandra.

1.5.9 Web Frameworks

Django and Flask are the two most popular web frameworks. Both have different purposes. While Django is a full-fledged framework, Flask is a microframework that is used to build small applications with minimal requirements. Django has built-in support for various web-related services like caching, internationalization, serialization, ORM support and automatic admin interface, while Flask allows users to configure web services according to your needs by installing external libraries. Both of these frameworks are available under BSD derived licenses.

1.5.10 Cloud Computing

OpenStack is entirely written in Python and is used to create a scalable private and public cloud. OpenStack Foundation oversees the development of OpenStack software. OpenStack has decent load balancing, is highly reliable, vendor independent and has built-in security. OpenStack uses the dashboard as a central unit to manage network resources, processing power and storage in a data center. Linux distributions like Fedora and Ubuntu include

OpenStack as part of their package. Hosting of Python applications on a cloud platform is well supported by various cloud service providers like Google App Engine, Amazon Web Services (AWS), Heroku and Microsoft Azure.

1.5.11 Python Distributions

Python Software Foundation releases Python interpreter with standard libraries. But in order to use Python in a scientific or enterprise environment other packages need to be installed. Having these packages tested for compatibility with the latest release of Python is cumbersome and time-consuming. Anaconda and Enthought Canopy Express are two popular distributions that come with core Python interpreter and popular scientific tools to help us start working out of the box.

1.5.12 IDE Available

Integrated Development Environments (IDEs) help in the rapid development of the software and increase in productivity. PyCharm is the most popular IDE for Python programming. PyCharm comes in three flavors namely, Professional Edition, Community Edition and Educational Edition. PyCharm has advanced features like auto code completion, code highlighting, refactoring, remote development capabilities and support for various web frameworks. PyCharm is available for various platforms like Windows, Linux and OS X. Microsoft has released an extension for Visual Studio called Python Tools for Visual Studio (PTVS) which transforms Visual Studio IDE into a full-fledged Python IDE. Spyder is another IDE that comes as part of Anaconda distribution itself.

1.5.13 Community

Community is what really defines the success of Open Source projects. Development of projects is taken forward by adding new features and Community members play an important role in testing the software, recommending it to others and in documenting the software. Python community members are expected to follow the Python Code of Conduct and the Python community is generally considered very helpful. The Python community is very active and cordial in accommodating newbies. Python conferences are held regularly across the world wherein the core Python developers are invited to share their experience with other developers thus paving the way for more Python adaption across the horizon. Python language documentation is renowned for its depth and completeness.

1.5.14 Python Stack in Industry

Various companies use Python stack to power up their infrastructure. The popular online photo sharing service Instagram uses the Django framework for application development. At Mozilla, which develops the popular Firefox web browser, the majority of the web development is done using the Django framework. PayPal and eBay, where transactions worth billions of dollars take place every year, swear by the security features provided by Python libraries. Companies like Pinterest and Twilio have adapted Flask as their web development framework. Requests library is used in major projects of companies like Amazon, Google, Washington Post, Twitter and others. Python Scientific and data analysis tools are being used at LinkedIn, Boeing and NASA. Dropbox has hired Guido van Rossum, Father of Python programming language, to add new features to their existing

Python stack. Even though this is not a complete list of companies using Python, it surely indicates industry interest in using Python to solve challenging problems.

[Source: CSI Communications, Vol. 40, April 2016.]

1.6 Installing Anaconda Python Distribution

Anaconda is a free and open source distribution of the Python programming language for data science and machine-learning related applications such as large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. Package versions are managed by the package management system *conda*, which makes it quite simple to install, run, and update complex data science and machine learning software libraries like Scikit-learn, PyTorch, TensorFlow, and SciPy. Anaconda Distribution is used by over 6 million users, and it includes more than 250 popular data science packages suitable for Windows, Linux, and MacOS.

The steps described here work on the Windows 10 OS.

Step 1: Go to the link <https://www.continuum.io/downloads>. You have the option to download the 32-bit or the 64-bit version of either Python 2.7 or Python 3.6 supported Anaconda distribution. At the time of writing this book, Anaconda supported Python 3.6 version. As and when a new version of Python is released, Anaconda distribution will be updated to newer releases. In this book 64-bit Anaconda distribution supporting Python 3.6 is used to execute programs, so download the same version.

Step 2: Click on the executable file of Anaconda Python distribution which you have downloaded and the setup screen will start loading.

Step 3: You will get a welcome screen as shown in [FIGURE 1.5](#). Click on *Next* button.

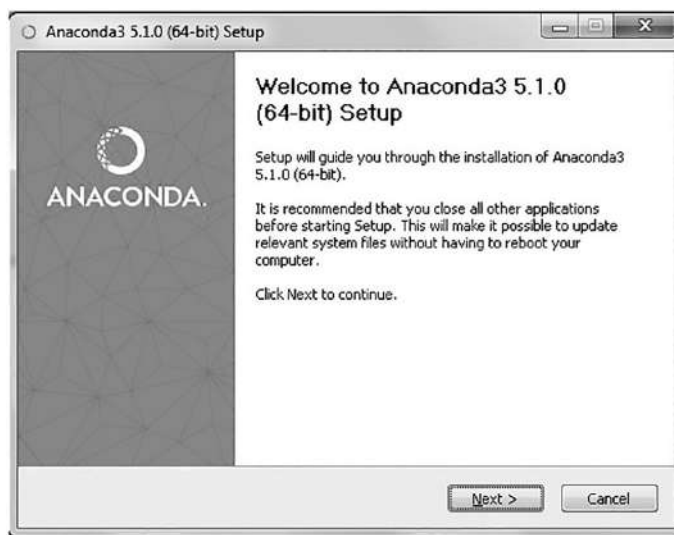


FIGURE 1.5

Welcome screen of Anaconda installation.

Step 4: You will get a License Agreement Screen, read the licensing terms and click on *I Agree* button.

Step 5: Assuming that you are the only user on your system, select *Just Me* radio button (FIGURE 1.6). Click on *Next* button.

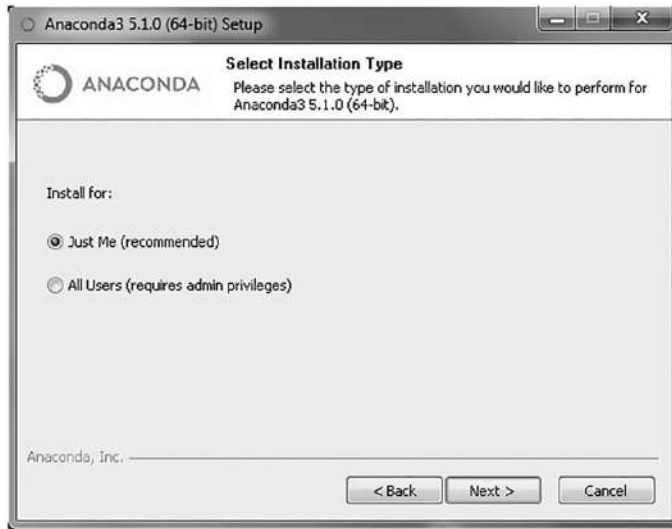


FIGURE 1.6
Selection of installation type.

Step 6: You need to choose a location to install Anaconda Python distribution. The default installation will be under *Users* folder. Change the destination folder to *C:\Anaconda3* to install Anaconda and click on *Next* button (FIGURE 1.7).

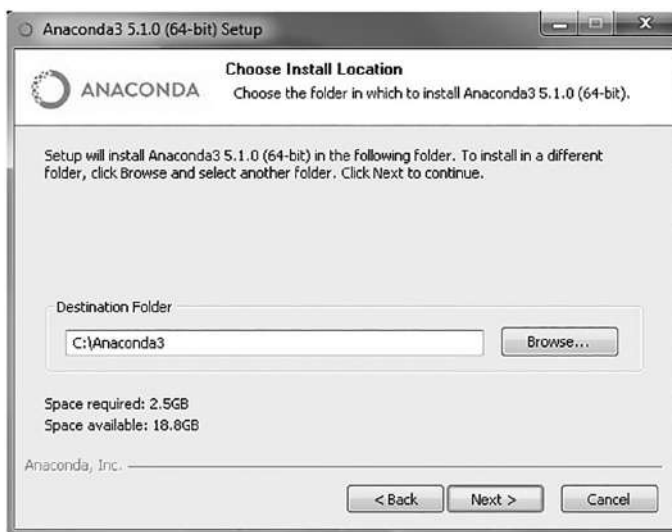


FIGURE 1.7
Choosing the destination folder.

Step 7: In the Advanced Installation Options screen, select all the check boxes. Ignore the warnings and click on *Install* button (FIGURE 1.8).

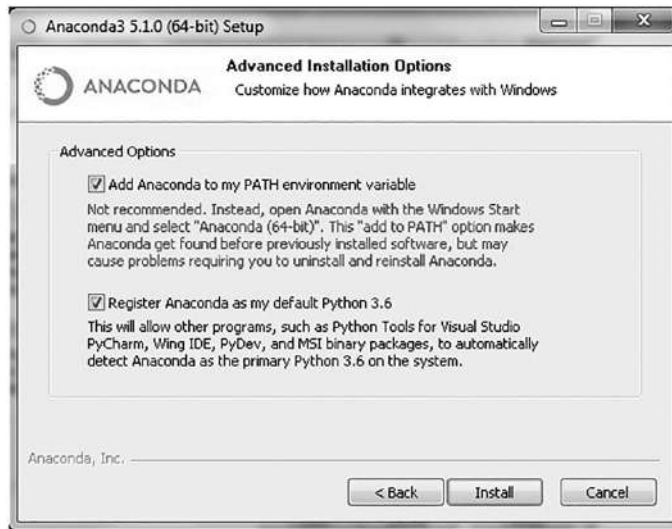


FIGURE 1.8

Selecting Advanced Installation Options.

Step 8: This starts the installation of Anaconda Python Distribution and once the installation is complete, click on *Next* button.

Step 9: Finish the setup by clicking on *Finish* button.

Step 10: To check whether the installation is working properly or not, go to the command prompt and type *python*. You should see a series of lines and a prompt as shown in FIGURE 1.9. This is Python Interactive mode. Here, the three greater-than signs “>>>” is the primary prompt of the interactive mode.

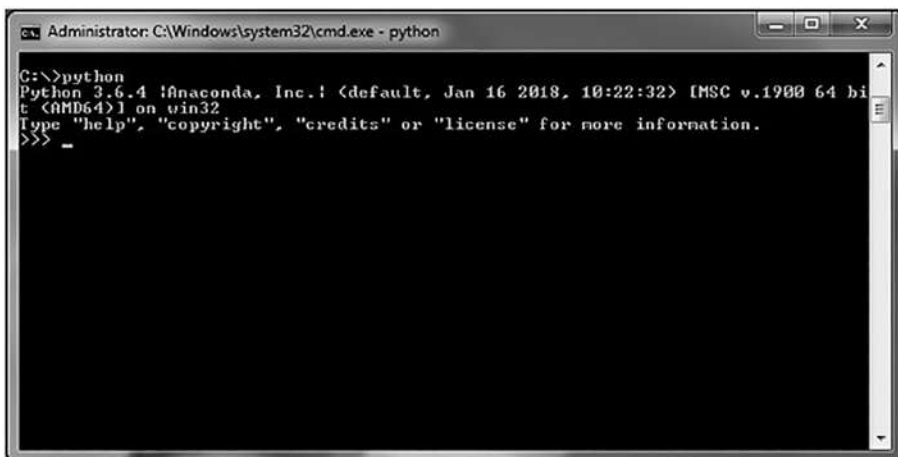


FIGURE 1.9

Python interactive shell.

1.7 Installing PyCharm IDE to Set Up a Python Development Environment

PyCharm is an Integrated Development Environment (IDE) used for Python programming language. It is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems and supports web development with Django web framework.

PyCharm is cross-platform, with the availability of Windows, MacOS and Linux versions. The Community Edition is released under the Apache License and there is also Professional Edition released under a proprietary license with added features like scientific tools, web development, Python Web Frameworks, Database and SQL support. PyCharm is designed by programmers, for programmers, to provide all the tools you need for productive Python development.

The steps described here work on the Windows 10 OS.

Step 1: Go to the link <https://www.jetbrains.com/PyCharm/download/>. You have the option to download either the PyCharm Professional Edition or the PyCharm Community Edition. If you have purchased a license for Professional Edition then go for it. Otherwise, you can download the Community edition which will suffice most of our requirements. All the programs in this book have been executed using PyCharm Community Edition IDE, so download the same edition.

Step 2: Click on the executable file which you have downloaded. You will be presented with the PyCharm Community Edition Setup screen. Click on *Next* button.

Step 3: Now you will be presented with Choose Install Location screen. Go with the default destination folder to install PyCharm Community Edition. Click on *Next* button.

Step 4: Select all the check boxes in the Installation Options screen except for the 32-bit launcher check box (since Windows 10 is a 64-bit OS, you don't need 32-bit launcher) and click on *Next* (FIGURE 1.10).

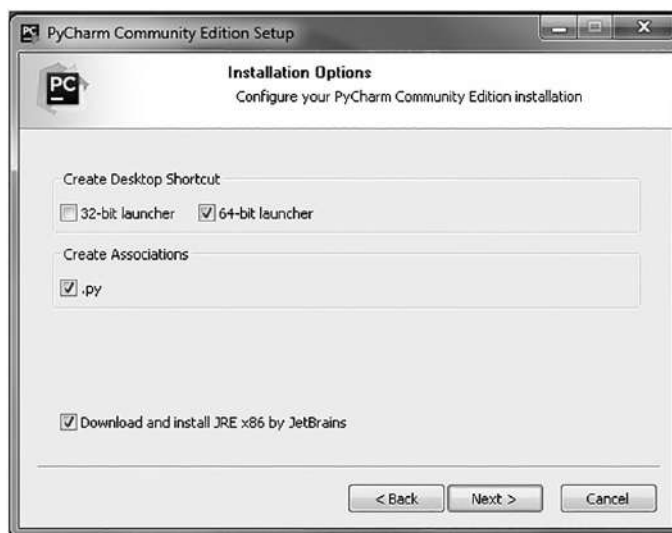


FIGURE 1.10
Installation options for PyCharm.

Step 5: Go with the default Start Menu Folder as shown on the screen and click on *Install* button. It will take some time for the installation to finish. Once the installation is done click on the *Finish* button.

Step 6: You will be asked whether you want to import previous PyCharm settings. Since we are starting on a clean slate, let's select the second radio button as shown in [FIGURE 1.11](#) and click on *OK* button.

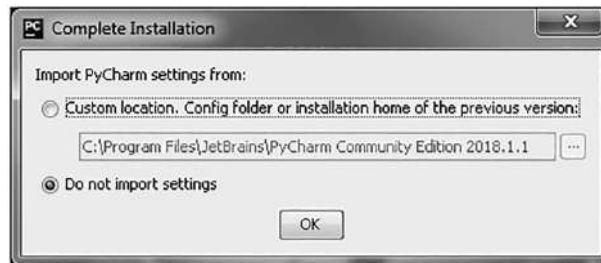


FIGURE 1.11
Importing PyCharm settings.

Step 7: You will be prompted with a Windows Security Alert. Do not worry about it. Click on *Allow Access* Button. Next screen will be Python community edition initial configuration. Let the default settings remain as it is and click on *OK* button. After you have completed initial PyCharm configuration, a customization screen will be displayed as shown in [FIGURE 1.12](#); click on *Skip Remaining and Set Defaults* button.

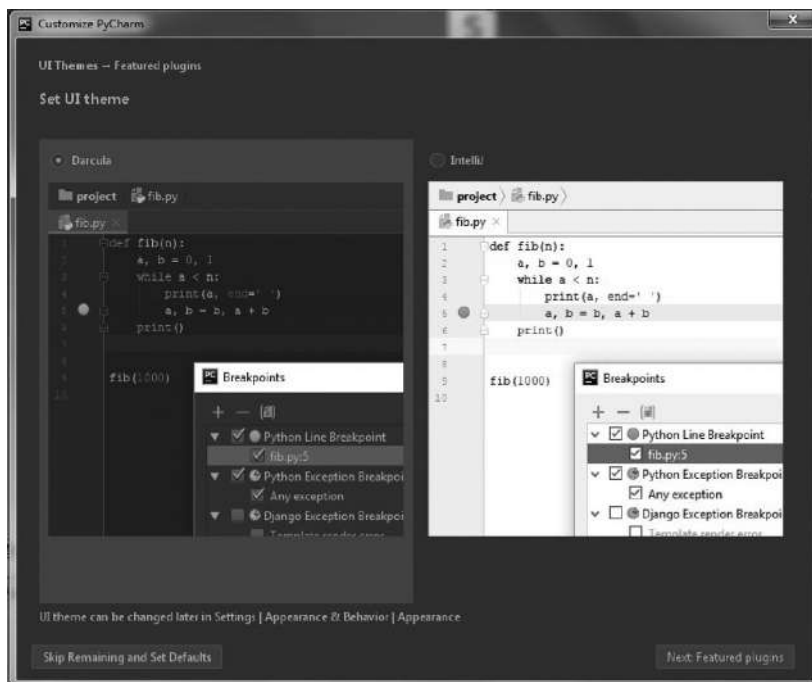


FIGURE 1.12
Customization of PyCharm.

Step 8: In the next screen, click on *Configure* pull down list and select *Settings* option as shown in [FIGURE 1.13](#).



FIGURE 1.13
Welcome screen for PyCharm.

Step 9: In the Default Settings screen, on the left pane, click on *Project Interpreter* as shown in [FIGURE 1.14](#).

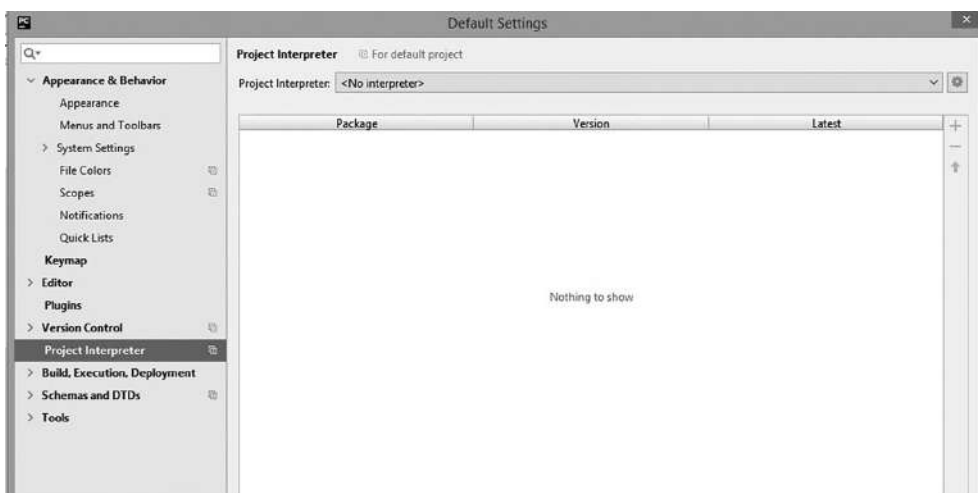


FIGURE 1.14
Default settings of PyCharm.

On the right pane, in the Project Interpreter option, click on the button having toothed wheel icon and select *Add*. In the Add Python Interpreter screen, on the left pane, click on *System Interpreter* and select the Python interpreter path from the *Interpreter* pull down list as shown in [FIGURE 1.15](#). Click on *OK* button.

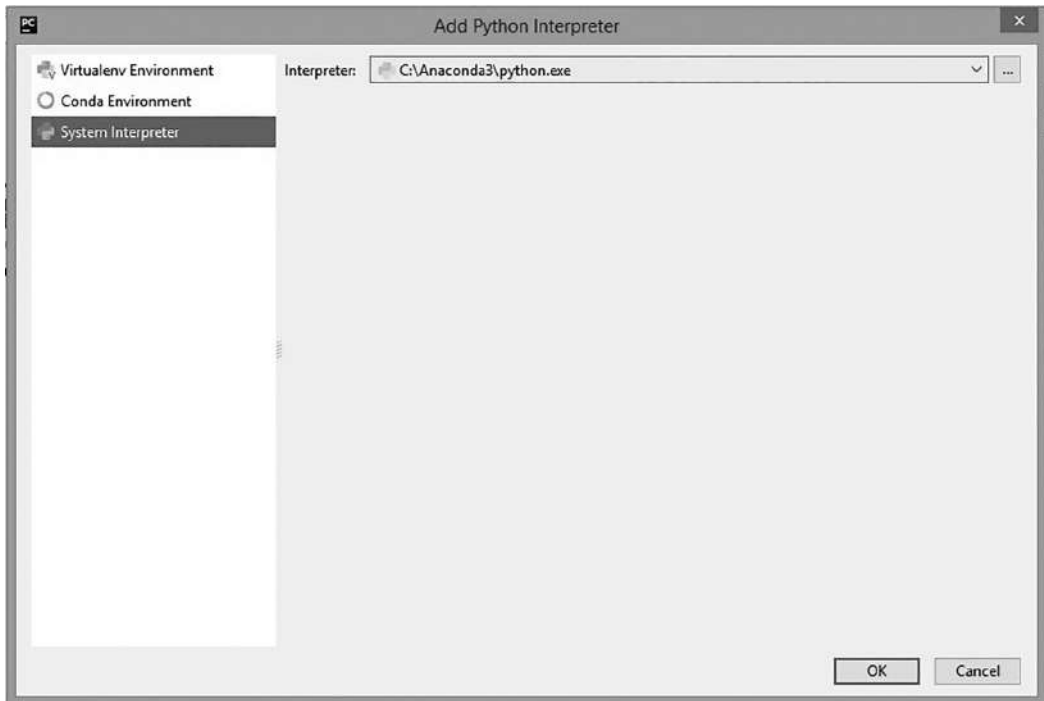


FIGURE 1.15
Adding Python Interpreter.

Step 10: It will take some time to list all the packages. Once done click on *OK* button.

Step 11: You will be again presented with the Welcome screen as shown in [FIGURE 1.13](#). Now, to work with PyCharm IDE, click on *Create New Project* option. In the next section, steps to create and execute Python program are discussed in detail.

1.8 Creating and Running Your First Python Project

Before you start make sure that the following prerequisites are met:

- You are working with PyCharm Community Edition or Professional.
- You have installed Python 3.6 supported Anaconda distribution.

Below steps describe the process of creating and running a Python project.

Step 1: Click on the *JetBrains Community Edition* shortcut icon and you will be presented with a welcome screen (FIGURE 1.13) and click on *Create New Project*. This screen is presented initially when you are creating a project for the first time. For the subsequent project creation, Go to *File* → *New Project* and go to Step 2. Create a folder called *PyWork* in *C:* drive.

Step 2: For the Location option, browse to the *PyWork* folder which you had created in *C:* drive in Step 1. Now give a name to your Python Project. For the purpose of the demo, the project name is given as *FirstProgram* (FIGURE 1.16). Expand the project Interpreter and click on *Existing interpreter* radio button. Select Python Interpreter path from *Interpreter* pull down list if it is not already selected. Then, click on the *Create* button.

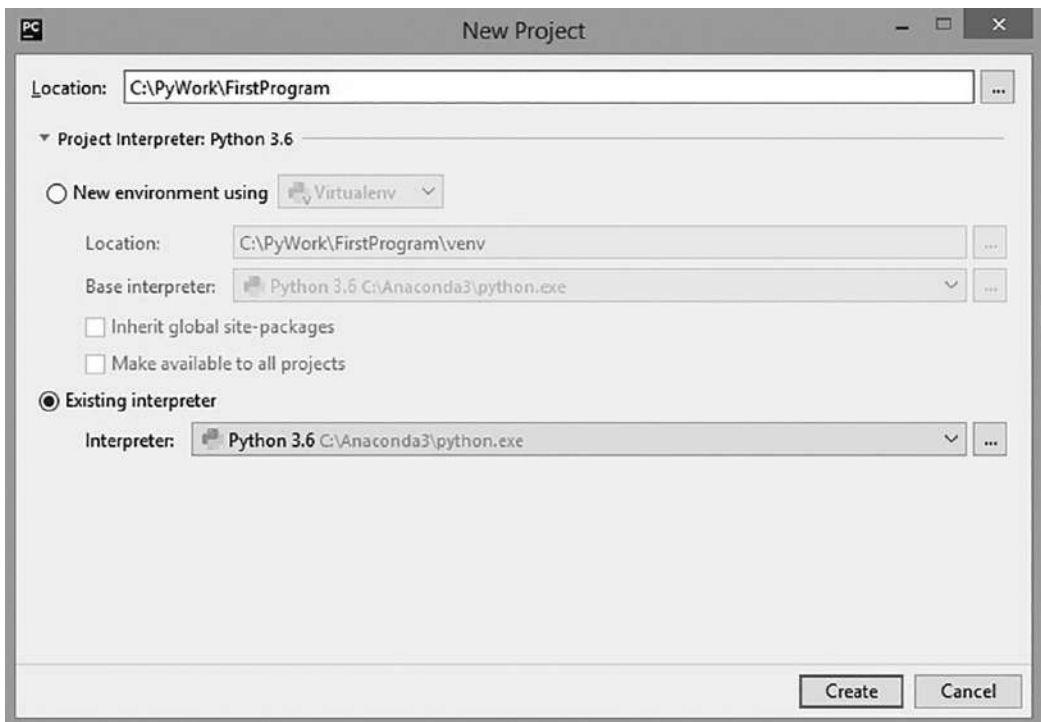


FIGURE 1.16
Adding a new Python project.

Step 3: On the left pane of the presented screen you can see the Project Name, which in our case is *FirstProgram*. Right-click on *FirstProgram* → Select *New* → Select *Python File* as shown in FIGURE 1.17.

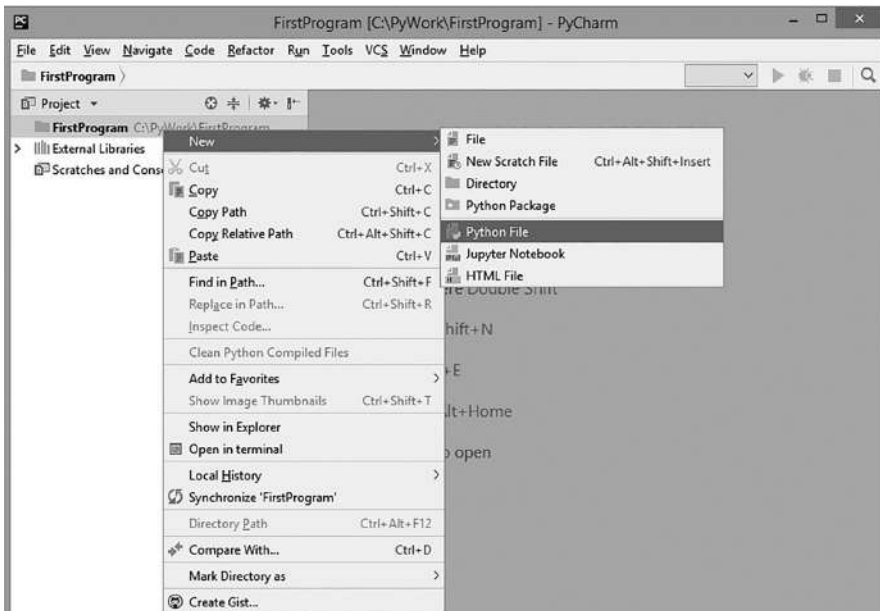


FIGURE 1.17
Creating a new Python file.

Step 4: Give a name to the Python File. For the purpose of this demo, let's name it as *HelloWorld* as shown in [FIGURE 1.18](#). No need to specify any extension. The PyCharm IDE itself will attach the *.py* extension to the file name which you have specified. Click on *OK* button.

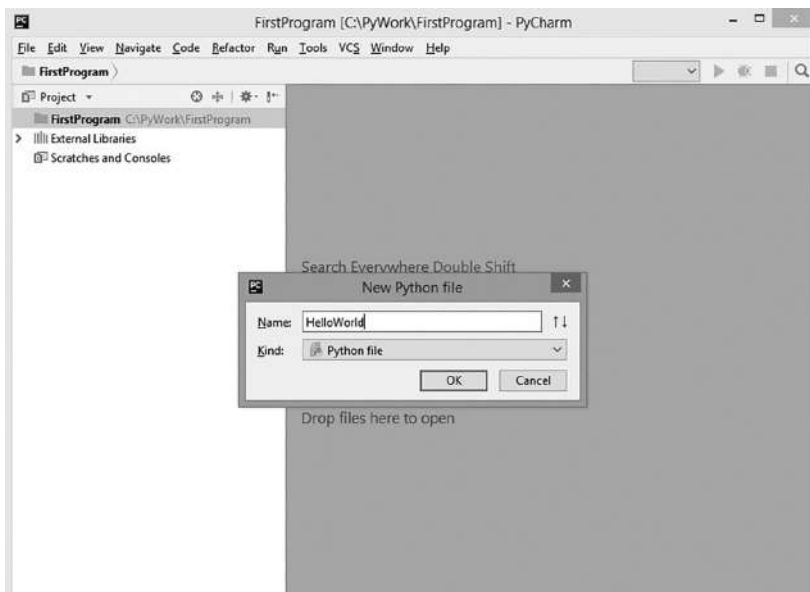


FIGURE 1.18
Naming of Python file.

Step 5: On the left pane double click on the File name which you have created (in our case it is *HelloWorld.py*). This should open an empty file on the right side of the editor in which you type the following statement ([FIGURE 1.19](#)).

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

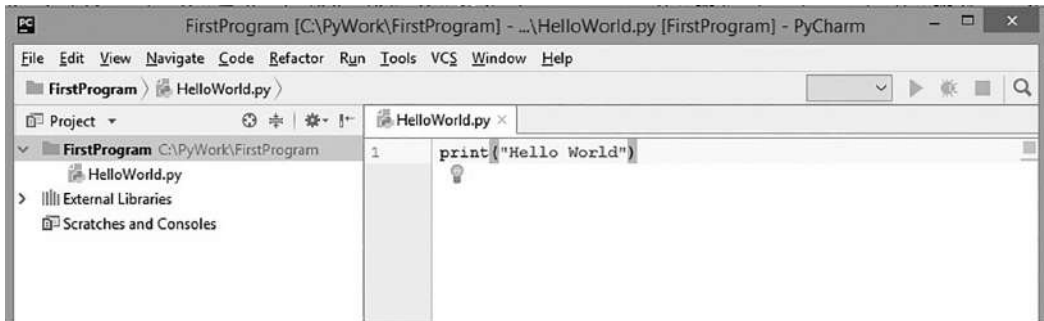


FIGURE 1.19
Code for HelloWorld.py program.

Step 6: To execute the above code, go to Run menu → and click on *Run* as shown in [FIGURE 1.20](#).

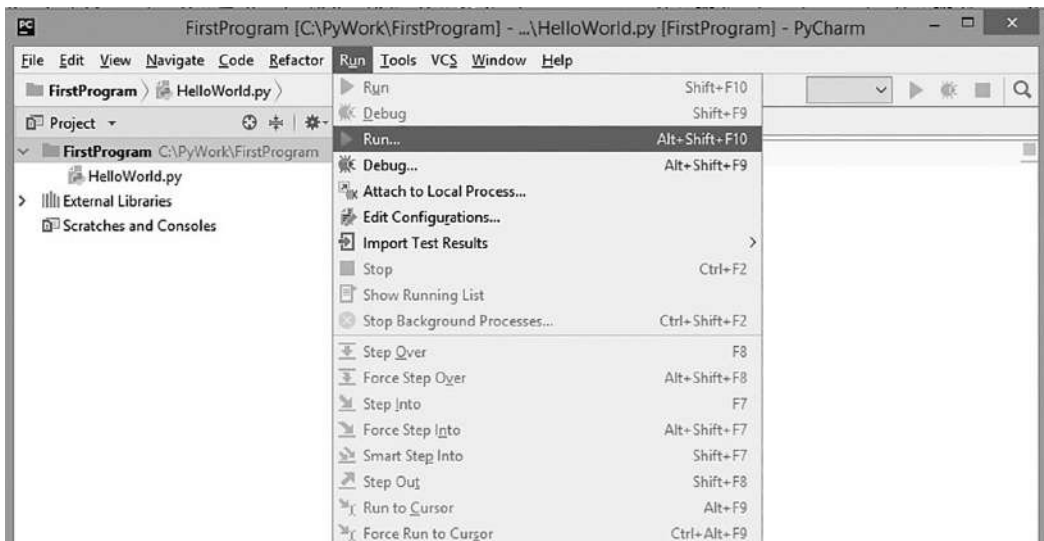


FIGURE 1.20
Execute HelloWorld.py program.

Another way of executing the above program is to right-click on the Python File Name (in our case it is *HelloWorld.py*) and select *Run "HelloWorld"*.

Step 7: In the below screen ([FIGURE 1.21](#)), you can see the output in the output window of the PyCharm IDE.

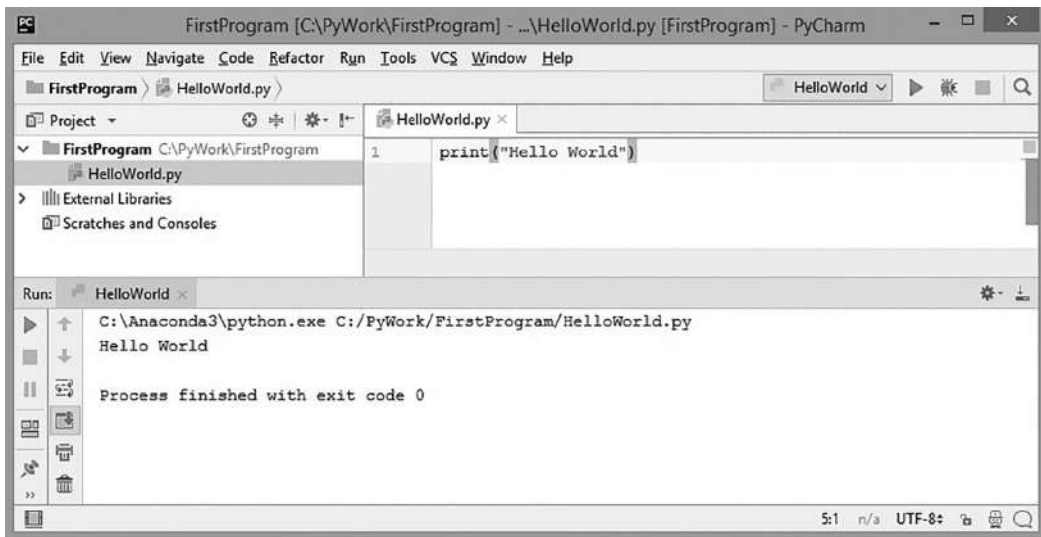


FIGURE 1.21
Output of HelloWorld.py program.

Step 8: You can add multiple Python files to the project and execute them individually. Create another Python File by following the steps 3, 4 and 5. Name the Python file as *SecondPythonFile* and in the editor type,

```
print("This is second file")
```

Right-click on the *SecondPythonFile.py* file and select Run "*SecondPythonFile*" to execute.

1.9 Installing and Using Jupyter Notebook

Jupyter Notebook is one of the many possible ways to interact with Python and the scientific libraries. Jupyter Notebook is an open source web application that uses a browser-based interface to Python with

- The choice to create and share documents.
- The ability to write and execute Python code.
- Formatted output in the browser, including tables, figures, equation, visualizations, etc.
- The option to mix in formatted text and mathematical expressions.

Because of these possibilities, Jupyter is fast turning into a major player in the scientific computing ecosystem.

1.9.1 Starting Jupyter Notebook

Step 1: Once you have installed Anaconda, you can start the Jupyter Notebook. Anaconda conveniently installs Python, the Jupyter Notebook, and other commonly used packages for scientific computing and data science. For demo purpose let's create a folder called *JupyterExample* in C:\ drive. Invoke command prompt and navigate to the folder by issuing the command `cd C:\JupyterExample`. Your Jupyter Notebook will be saved in this folder. You can start Jupyter Notebook by issuing the following command in the command prompt as shown in [FIGURE 1.22](#).

jupyter notebook



FIGURE 1.22
Command to start Jupyter Notebook.

You should see a series of lines in the command prompt as seen in [FIGURE 1.23](#). The output tells us the notebook is running at `http://localhost:8888/` where localhost is the name of the local machine and 8888 refers to port number 8888 on your computer. Thus, the Jupyter kernel is listening for Python commands on port 8888 of our local machine.

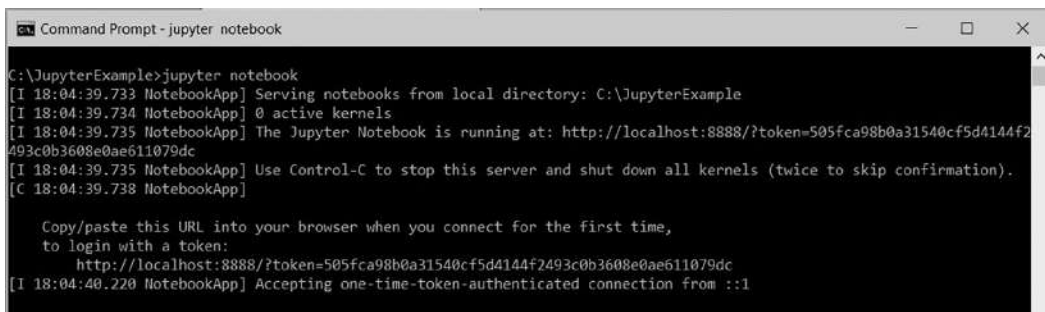


FIGURE 1.23
Output of *jupyter notebook* command.

Hopefully, your default browser has also opened up with a web page that looks something like [FIGURE 1.24](#). What you see here is called the Jupyter dashboard. If you look at the URL at the top, it should be localhost:8888 or similar, matching the message above.

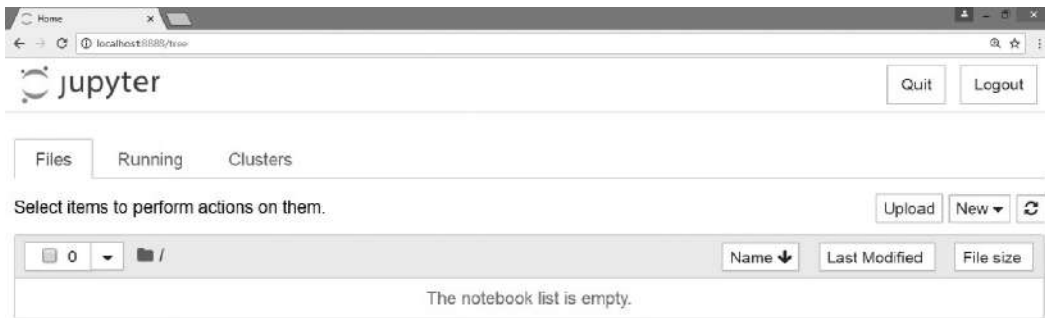


FIGURE 1.24
Jupyter dashboard.

Step 2: Assuming all this has worked OK, you can now click on *New* pull-down list at top right and select Python 3 option ([FIGURE 1.25](#)).

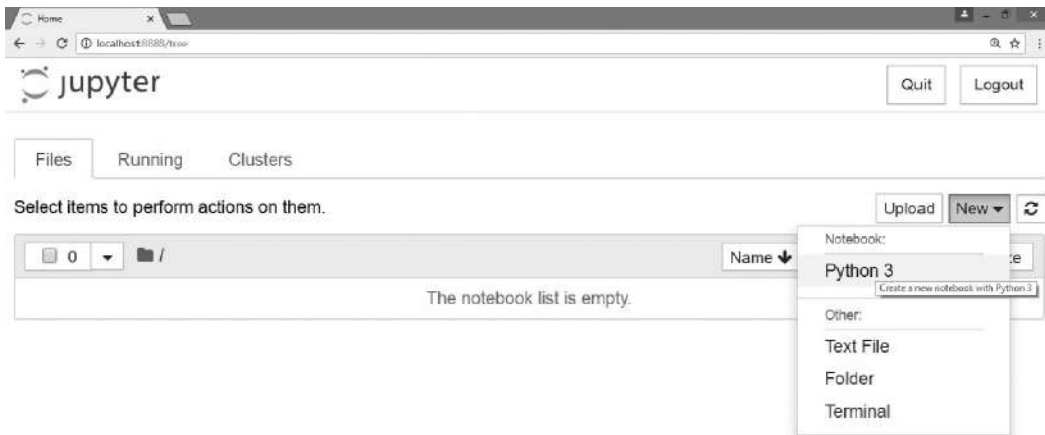


FIGURE 1.25
Creating new Jupyter Notebook.

Step 3: Here's what shows up on our machine. From the previous step, a new Jupyter Notebook for Python is created but is still *Untitled* ([FIGURE 1.26](#)). The blue border around the cell indicates that the particular cell is selected and you are in command mode. In command mode, you can cut, copy, paste and insert a new cell.

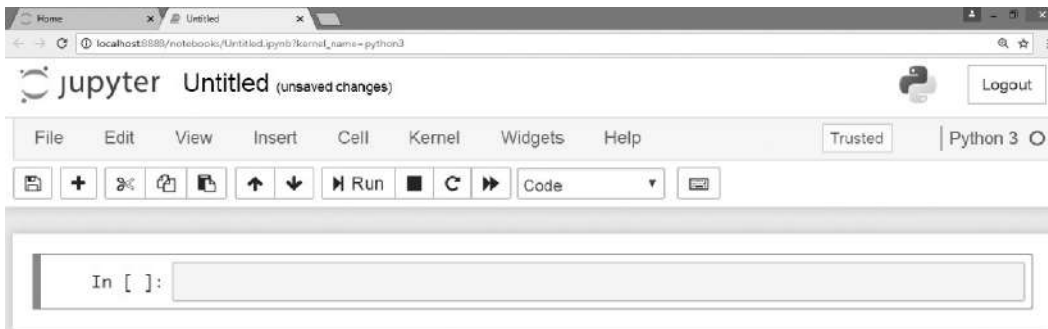


FIGURE 1.26
Untitled Jupyter Notebook.

To rename the notebook click on *Untitled* at the top. A Rename Notebook window pops up. Enter a name for the notebook. Let's name the notebook as *My_First_Notebook* (FIGURE 1.27). Click on *Rename* button. Notebook gets saved as *My_First_Notebook.ipynb* with *.ipynb* extension in *C:\JupyterExample* folder.



FIGURE 1.27
Rename Jupyter Notebook.

Step 4: The notebook itself consists of cells. Notice that in the FIGURE 1.28 the cell is surrounded by a green border. It means that the cell is in edit mode and you are permitted to type text into the selected cell. The edit mode is invoked when you click in the code area of that cell. You can switch to command mode from edit mode by pressing **Esc** key.

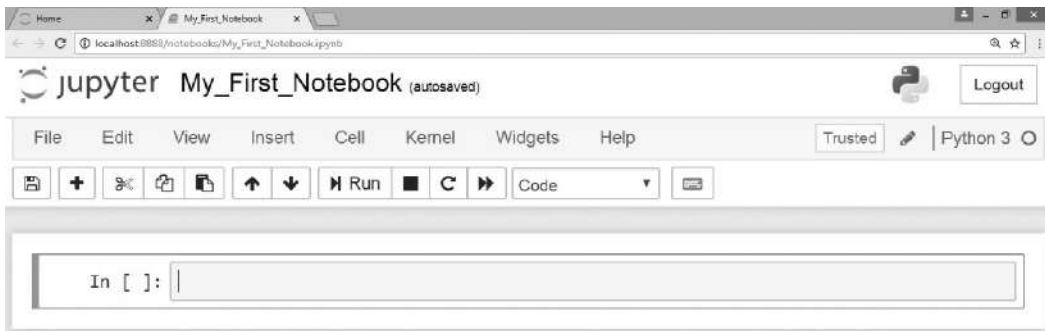


FIGURE 1.28
Jupyter Notebook in edit mode.

You can type in Python code and it will appear in the cell. Executing the code in this cell can be done by either clicking on the *Run* button or hitting Alt + Enter (FIGURE 1.29).

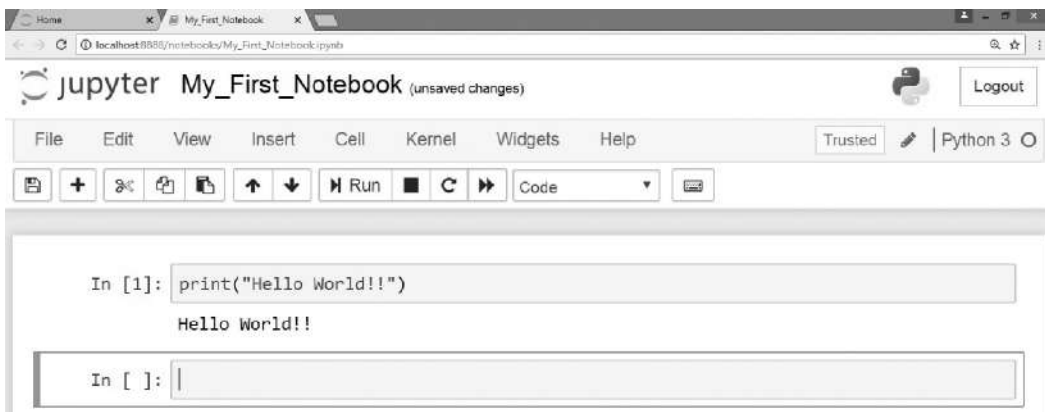


FIGURE 1.29
Executing Python code in Jupyter Notebook.

1.10 Open Source Software

The term “Open Source” refers to something people can modify and share because its design is publicly accessible. Open source software is software with source code that anyone can inspect, modify, and enhance. “Source code” is the part of the software that most computer users don’t ever see; it’s the code computer programmers can manipulate to change how a piece of software, a “program” or “application” works. Programmers who have access to a computer program’s source code can improve that program by adding

features to it or fixing parts that don't always work correctly. Some software has source code that only the person, team, or organization who created it and maintains exclusive control over it can modify. People call this kind of software "proprietary" or "closed source" software.

Only the original authors of proprietary software can legally copy, inspect, and alter that software. And in order to use proprietary software, computer users must agree (usually by signing a license displayed the first time they run the software) that they will not do anything with the software that the software's authors have not expressly permitted. Microsoft Office and Adobe Photoshop are examples of proprietary software. Open source software is different. Its authors make its source code available to others who would like to view that code, copy it, learn from it, alter it, or share it. LibreOffice and the GNU Image Manipulation Program are examples of open source software. As they do with proprietary software, users must accept the terms of a license when they use open source software but the legal terms of open source licenses differ dramatically from those of proprietary licenses.

By design, open source software licenses promote collaboration and sharing because they permit other people to make modifications to source code and incorporate those changes into their own projects. They encourage computer programmers to access, view, and modify open source software whenever they like, as long as they let others do the same when they share their work.

1.10.1 Why Do People Prefer Using Open Source Software?

People prefer open source software to proprietary software for a number of reasons, including:

Control. Many people prefer open source software because they have more control over that kind of software. They can examine the code to make sure it's not doing anything they don't want it to do, and they can change parts of it they don't like. Users who aren't programmers also benefit from open source software because they can use this software for any purpose they wish not merely the way someone else thinks they should.

Training. People like open source software because it helps them become better programmers. Because open source code is publicly accessible, students can easily study it as they learn to make better software. Students can also share their work with others, inviting comment and critique, as they develop their skills. When people discover mistakes in programs' source code, they can share these mistakes with others to help them avoid making the same mistakes themselves.

Security. Some people prefer open source software because they consider it more secure and stable than proprietary software. Because anyone can view and modify open source software, someone might spot and correct errors or omissions that a program's original authors might have missed. Because so many programmers can work on a piece of open source software without asking for permission from original authors, they can fix, update, and upgrade open source software more quickly than they can do for a proprietary software.

Stability. Many users prefer open source software to proprietary software for important, long-term projects. Because programmers publicly distribute the source code

for open source software, users relying on that software for critical tasks can be sure their tools won't disappear or fall into disrepair if their original creators stop working on them. Additionally, open source software tends to both incorporate and operate according to open standards.

1.10.2 Doesn't "Open Source" Just Mean Something Is Free of Charge?

No. This is a common misconception about what "open source" implies, and the concept's implications are not only economic.

Open source software programmers can charge money for the open source software they create or to which they contribute. But in some cases, because an open source license might require them to release their source code when they sell software to others, some programmers find that charging users money for software services and support (rather than for the software itself) is more lucrative. This way, their software remains free of charge, and they make money helping others install, use, and troubleshoot it.

While some open source software may be free of charge, skill in programming and troubleshooting open source software can be quite valuable. Many employers specifically seek to hire programmers with experience working on open source software.

[Source: Adapted with kind permission from <https://opensource.com/resources/what-open-source>.]

1.10.3 Open Source Licenses

Open source software plays a very important role in our daily life. Some of the most popular open source software are Android, Firefox, Linux, WordPress, 7-Zip, BitTorrent, Python and others. But did you know that not all open source licenses are the same? What are the differences and what do they mean for you? If you write open source software, which should you use?

Open source licenses make it easy for others to contribute to a project without having to seek special permission. It also protects you as the original creator, making sure you at least get some credit for your contributions. It also helps to prevent others from claiming your work as their own.

Apache License

Apache Software Foundation has authored the Apache License. Apache Software Foundation is well known for various software like Apache HTTP Web Server that powers a large percentage of the Internet's websites, Hadoop that supports the processing of large datasets and many more. There are more than 350 open source initiatives housed at the Apache Software Foundation. Version 2.0 was released in 2004 to make it easier for non-Apache projects to use the license and remains one of the most popular licenses to date. Some of the features of this license are,

- It allows you to freely download and use Apache software, in whole or in part, for personal or commercial purposes.
- It requires you to include a copy of the license in any redistribution you may make that includes Apache software.
- Software can be modified to create derivative works and can also be distributed with the same or different license.

- Owner of software cannot be held liable for damages and owner's software trademarks cannot be used in derivative works.
- Significant changes to original software must be noted.

The Apache License allows you to patent the derivative works.

BSD License

The *BSD license* is a class of extremely simple and very liberal licenses for computer software that was originally developed at the University of California at Berkeley (UCB) in 1990 and revised twice, being simplified further with each revision. Some of the features of this license are,

- Software can be used commercially.
- Software can be modified to create derivative works which can also be distributed and commercialized.
- Owner of software cannot be held liable for damages and the original copyright and license text must be included with the distributions.

BSD license allows you to commercialize even the derivative works of a software. While Apache license allows a patent grant for derivative works, the BSD license does not allow the same. You are not required to return any modified or improved code to the BSD-licensed software. Due to the extremely minimal restrictions of BSD license, software released under BSD can also be used in proprietary software.

GNU General Public License

Richard Stallman, an American free software movement activist and programmer launched the GNU Project in September 1983 to create a Unix-like computer operating system composed entirely of free software. With this, he also launched the Free Software Foundation movement and wrote the GNU General Public License (GNU GPL). The GNU GPL is the free software license par excellence. It sets forth distribution conditions that guarantee freedom for its users. A program protected by the GNU GPL is free, but the GNU GPL also stipulates that all programs derived from such a program preserve the same freedom. The license has been revised twice with each revision addressing significant issues that arose in previous versions. Version 3 was published in 2007. Some of the features of this license are,

- Software can be modified to create derivative works and the derivative works can be distributed provided that it is also licensed under GNU GPL 3 as well.
- Distributions must include original software, all source code, original copyright and license text. Owner of the software cannot be held liable for damages.

The GNU GPL guarantees “software freedom” to all users with the freedom to run, copy, distribute, study, modify, and improve the software. GNU GPL license ensures that the open source software remains open source always even when extended into derivative works. However, you should be cautious before incorporating GPL code into your own code as the

GNU GPL license requires that you license your entire project under GNU GPL as well. This ensures that the GNU GPL licensed software and its derivatives remain free always.

MIT License

The MIT license is probably the shortest and broadest of all the popular open source licenses. MIT licensing terms are very loose and more open than most other licenses. Some of the features of this license are,

- Software licensed under MIT license can be used, copied and modified however you want. MIT licensed software can be used with any other project, can be copied any number of times and can be changed however you want.
- MIT licenses software can be given away for free or you can even sell it. There is no restriction on how you intend to distribute the software.
- The only requirement is that the software should be accompanied by the MIT license agreement.

The MIT license is the least protective license out there. It basically says that anyone can do whatever they want with the MIT licensed software, as long as it is accompanied by MIT license.

Creative Commons Licenses

Creative Commons is a global nonprofit organization that enables sharing and reuse of creativity and knowledge. Creative Commons licenses provide an easy way to manage the copyright terms that attach automatically to all creative material. Creative Commons licenses allow the creative material to be shared and reused. Creative Commons licenses may be applied to any type of work, including educational resources, music, photographs, databases, government and public-sector information, and many other types of material. The only categories of works for which Creative Commons does not recommend its licenses are computer software and hardware. The latest version of the Creative Commons licenses is version 4.0. A Creative Commons license has four basic parts, which can be endorsed individually or in combination. You will find a brief description of each part below.

Attribution. All Creative Commons licenses require that others who use your work in any way must give you credit the way you request, but not in a way that suggests you endorse them or their use. If they want to use your work without giving you credit or for endorsement purposes, they must get your permission first. Beyond that, the work can be modified, distributed, copied and otherwise used.

Share Alike. You let others copy, distribute, display, perform, and modify your work, as long as they distribute any modified work on the same terms.

Non-Commercial. You let others copy, distribute, display, perform and modify and use your work for any purpose other than commercially unless they get your permission first.

No Derivative. You let others copy, distribute, display and perform only original copies of your work. If they want to modify your work, they must get your permission first.

As mentioned before, these parts of the Creative Commons licenses can be mixed together. The least protective Creative Commons license would be the “Attribution” license, which means that as long as people credit you, they can do whatever they like with the work. The most protective Creative Commons license would be the “Attribution, Non-Commercial, No Derivatives” license, which means that others can freely share your work, but should not change it or charge for it, and they must attribute it to you as the creator of the work. Creative Commons licenses give everyone from individual creators to large companies and institutions a simple, standardized way to grant copyright permissions to their creative work.

[Source: Adapted with kind permission from Joel Lee, makeuseof.com.]

1.11 Summary

- Overview of various Python libraries available under different categories is given.
- Python due to its flexibility and simplicity reduces the amount of time taken from conceptualization of an idea to building the application and marketing it, resulting in more demand for Python programmers in Enterprise setup.
- The younger generation needs to be motivated to consider Python as their first programming language to express their thoughts.
- Anaconda is the most popular Python distribution.
- PyCharm is one of the full-fledged IDEs to work with.
- Open source software has played a very important role in shaping today’s world.

Multiple Choice Questions

1. A program must be converted to _____ language to be executed by a computer.
 - a. Assembly
 - b. Machine
 - c. High level
 - d. Very high level
2. _____ is a Logical programming language.
 - a. PROLOG
 - b. Python
 - c. C#
 - d. Java

3. The program written only using 0's and 1's is
 - a. PHP
 - b. High level
 - c. Python
 - d. Machine
4. The founder of Python is
 - a. Charles Babbage
 - b. Guido van Rossum
 - c. Dennis Ritchie
 - d. Larry Wall
5. Python is a compiled language.
 - a. True
 - b. False
 - c. Can't say
 - d. None of these
6. This programming paradigm emerged to remove the reliance on the GOTO statements.
 - a. Structured
 - b. Object-oriented
 - c. Logical
 - d. Functional
7. Which Python library is popularly referred to as the HTTP library written for humans.
 - a. Receive
 - b. Requests
 - c. Sockets
 - d. Send
8. In which phase of SDLC does the software developer analyses whether software can be prepared to fulfill all the requirements of the end user?
 - a. Design
 - b. Development
 - c. Testing
 - d. Planning
9. This license allows a patent grant for derivative works.
 - a. BSD License
 - b. Apache License
 - c. MIT License
 - d. CC License

10. A group of people maintain exclusive control over the source code of a software. Such software is called
 - a. Freeware
 - b. Shareware
 - c. Proprietary
 - d. Adware
-

Review Questions

1. What is a programming language?
2. Briefly explain the steps to install Anaconda.
3. Describe the steps to install PyCharm.
4. Outline the advantages and disadvantages of machine language.
5. Why do we need programs? Comment on this.
6. Outline the advantages and disadvantages of high-level language.
7. Give a brief explanation of the history of Python.
8. Differentiate between Interpreter and Compiler.
9. Mention disadvantages of Assembly language.
10. Discuss various steps involved in the software development life cycle.
11. Give a brief description of open source software.
12. Explain the different types of licenses under which open source software can be released.

2

Parts of Python Programming Language

AIM

Understand various Operators, Expression, Data Types, User input and print statements upon which multifaceted operations can be built in Python Programming Language.

LEARNING OUTCOMES

After completing this chapter, you should be able to

- Define identifiers, keywords, operators and expressions.
- Use different operators, expressions and variables available in Python.
- Build complex expressions using operators.
- Determine the data types of values.
- Use indentation and comments in writing Python programs.

In this chapter, we discuss the fundamentals of Python Programming language which you need to know before writing simple Python programs. This chapter describes how Python programs should work at the most basic level and gives details about operators, types and keywords upon which complex solutions can be built. Once you gain familiarity with these foundational elements of Python programming language, then you will appreciate how a lot of functionality can be accomplished with less verbose code.

2.1 Identifiers

An identifier is a name given to a variable, function, class or module. Identifiers may be one or more characters in the following format:

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myCountry, other_1 and good_morning, all are valid examples. A Python identifier can begin with an alphabet (A – Z and a – z and _).
- An identifier cannot start with a digit but is allowed everywhere else. 1plus is invalid, but plus1 is perfectly fine.

- Keywords cannot be used as identifiers.
- One cannot use spaces and special symbols like !, @, #, \$, % etc. as identifiers.
- Identifier can be of any length.

2.2 Keywords

Keywords are a list of reserved words that have predefined meaning. Keywords are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name. Attempting to use a keyword as an identifier name will cause an error. The following [TABLE 2.1](#) shows the Python keywords.

TABLE 2.1
List of Keywords in Python

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

2.3 Statements and Expressions

A statement is an instruction that the Python interpreter can execute. Python program consists of a sequence of statements. Statements are everything that can make up a line (or several lines) of Python code. For example, `z = 1` is an assignment statement.

Expression is an arrangement of values and operators which are evaluated to make a new value. Expressions are statements as well. A value is the representation of some entity like a letter or a number that can be manipulated by a program. A single value `>>> 20` or a single variable `>>> z` or a combination of variable, operator and value `>>> z + 20` are all examples of expressions. An expression, when used in interactive mode is evaluated by the interpreter and result is displayed instantly. For example,

```
>>> 8 + 2
10
```

But the same expression when used in Python program does not show any output altogether. You need to explicitly print the result.

2.4 Variables

Variable is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution. In Python, there is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type. *To define a new variable in Python, we simply assign a value to a name.* If a need for variable arises you need to think of a variable name based on the rules mentioned in the following subsection and use it in the program.

2.4.1 Legal Variable Names

Follow the below-mentioned rules for creating legal variable names in Python.

- Variable names can consist of any number of letters, underscores and digits.
- Variable should not start with a number.
- Python Keywords are not allowed as variable names.
- Variable names are case-sensitive. For example, computer and Computer are different variables.

Also, follow these guidelines while naming a variable, as having a consistent naming convention helps in avoiding confusion and can reduce programming errors.

- Python variables use lowercase letters with words separated by underscores as necessary to improve readability, like this `whats_up`, `how_are_you`. Although this is not strictly enforced, it is considered a best practice to adhere to this convention.
- Avoid naming a variable where the first character is an underscore. While this is legal in Python, it can limit the interoperability of your code with applications built by using other programming languages.
- Ensure variable names are descriptive and clear enough. This allows other programmers to have an idea about what the variable is representing.

2.4.2 Assigning Values to Variables

The general format for assigning values to variables is as follows:

variable_name = expression

The equal sign (=) also known as simple assignment operator is used to assign values to variables. In the general format, the operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the expression which can be a value or any code snippet that results in a value. That value is stored in the variable on the execution of the assignment statement. Assignment operator should not be confused with the = used in algebra to denote equality. For example, enter the code shown below in interactive mode and observe the results.

1. `>>> number =100`
2. `>>> miles =1000.0`

```
3. >>> name ="Python"
4. >>> number
    100
5. >>> miles
    1000.0
6. >>> name
    'Python'
```

In ① integer type value is assigned to a variable *number*, in ② float type value has been assigned to variable *miles* and in ③ string type value is assigned to variable *name*. ④, ⑤ and ⑥ prints the value assigned to these variables.

In Python, not only the value of a variable may change during program execution but also the type of data that is assigned. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable. A new assignment overrides any previous assignments. For example,

```
1. >>> century = 100
2. >>> century
    100
3. >>> century = "hundred"
4. >>> century
    'hundred'
```

In ① an integer value is assigned to *century* variable and then in ③ you are assigning a string value to *century* variable. Different values are printed in each case as seen in ② and ④.

Python allows you to assign a single value to several variables simultaneously. For example,

```
1. >>> a = b = c =1
2. >>> a
    1
3. >>> b
    1
4. >>> c
    1
```

An integer value is assigned to variables *a*, *b* and *c* simultaneously ①. Values for each of these variables are displayed as shown in ②, ③ and ④.

2.5 Operators

Operators are symbols, such as $+$, $-$, $=$, $>$, and $<$, that perform certain mathematical or logical operation to manipulate data values and produce a result based on some rules. An operator manipulates the data values called operands.

Consider the expression,

```
>>> 4 + 6
```

where 4 and 6 are operands and + is the operator.

Python language supports a wide range of operators. They are

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators

2.5.1 Arithmetic Operators

Arithmetic operators are used to execute arithmetic operations such as addition, subtraction, division, multiplication etc. The following [TABLE 2.2](#) shows all the arithmetic operators.

TABLE 2.2

List of Arithmetic Operators

Operator	Operator Name	Description	Example
+	Addition operator	Adds two operands, producing their sum.	$p + q = 5$
-	Subtraction operator	Subtracts the two operands, producing their difference.	$p - q = -1$
*	Multiplication operator	Produces the product of the operands.	$p * q = 6$
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	$q / p = 1.5$
%	Modulus operator	Divides left hand operand by right hand operand and returns a remainder.	$q \% p = 1$
**	Exponent operator	Performs exponential (power) calculation on operators.	$p ** q = 8$
//	Floor division operator	Returns the integral part of the quotient.	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$

Note: The value of p is 2 and q is 3.

For example,

1.

```
>>> 10+35
45
```
2.

```
>>> -10+35
25
```
3.

```
>>> 4*2
8
```
4.

```
>>> 4**2
16
```

```
5. >>> 45/10
4.5
6. >>> 45//10.0
4.0
7. >>> 2025%10
5
8. >>> 2025//10
202
```

Above code illustrates various arithmetic operations ①–⑧.

2.5.2 Assignment Operators

Assignment operators are used for assigning the values generated after evaluating the right operand to the left operand. Assignment operation always works from right to left. Assignment operators are either simple assignment operator or compound assignment operators. Simple assignment is done with the equal sign (=) and simply assigns the value of its right operand to the variable on the left. For example,

```
1. >>> x = 5
2. >>> x = x + 1
3. >>> x
6
```

In ① you assign an integer value of 5 to variable *x*. In ② an integer value of 1 is added to the variable *x* on the right side and the value 6 after the evaluation is assigned to the variable *x*. The latest value stored in variable *x* is displayed in ③.

Compound assignment operators support shorthand notation for avoiding the repetition of the left-side variable on the right side. Compound assignment operators combine assignment operator with another operator with = being placed at the end of the original operator.

For example, the statement

```
>>> x = x + 1
```

can be written in a compactly form as shown below.

```
>>> x += 1
```

If you try to update a variable which doesn't contain any value, you get an error.

```
1. >>> z = z + 1
NameError: name 'z' is not defined
```

Trying to update variable *z* which doesn't contain any value results in an error because Python evaluates the right side before it assigns a value to *z* ①.

```
1. >>> z = 0
2. >>> x = z + 1
```

Before you can update a variable ②, you have to assign a value to it ①.

The following [TABLE 2.3](#) shows all the assignment operators.

TABLE 2.3

List of Assignment Operators

Operator	Operator Name	Description	Example
=	Assignment	Assigns values from right side operands to left side operand.	$z = p + q$ assigns value of $p + q$ to z
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand.	$z += p$ is equivalent to $z = z + p$
-=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand.	$z -= p$ is equivalent to $z = z - p$
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand.	$z *= p$ is equivalent to $z = z * p$
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand.	$z /= p$ is equivalent to $z = z / p$
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand.	$z **= p$ is equivalent to $z = z ** p$
//=	Floor Division Assignment	Produces the integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	$z //= p$ is equivalent to $z = z // p$
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	$z \% = p$ is equivalent to $z = z \% p$

For example,

```

1. >>> p = 10
2. >>> q = 12
3. >>> q += p
4. >>> q
    22
5. >>> q *= p
6. >>> q
    220
7. >>> q /= p
8. >>> q
    22.0
9. >>> q %= p
10. >>> q
    2.0
11. >>> q **= p
12. >>> q
    1024.0
13. >>> q //= p
14. >>> q
    102.0

```

Above code illustrates various assignment operations ①–⑭.



Learned readers coming from other languages should note that Python programming language doesn't support Autoincrement (++) and Autodecrement (--) operators.

2.5.3 Comparison Operators

When the values of two operands are to be compared then comparison operators are used. The output of these comparison operators is always a Boolean value, either True or False. The operands can be Numbers or Strings or Boolean values. Strings are compared letter by letter using their ASCII values, thus, "P" is less than "Q", and "Aston" is greater than "Asher". [TABLE 2.4](#) shows all the comparison operators.

TABLE 2.4

List of Comparison Operators

Operator	Operator Name	Description	Example
==	Equal to	If the values of two operands are equal, then the condition becomes True.	(p == q) is not True.
!=	Not Equal to	If values of two operands are not equal, then the condition becomes True.	(p != q) is True
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes True.	(p > q) is not True.
<	Lesser than	If the value of left operand is less than the value of right operand, then condition becomes True.	(p < q) is True.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes True.	(p >= q) is not True.
<=	Lesser than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes True.	(p <= q) is True.

Note: The value of p is 10 and q is 20.

For example,

1. `>>>10 == 12`
False
2. `>>>10 != 12`
True
3. `>>>10 < 12`
True
4. `>>>10 > 12`
False
5. `>>>10 <= 12`
True

```

6. >>>10 >= 12
    False
7. >>> "P" < "Q"
    True
8. >>> "Aston" > "Asher"
    True
9. >>> True == True
    True

```

Above code illustrates various comparison operations ①–⑨.



Don't confuse the equality operator `==` with the assignment operator `=`. The expression `x==y` compares `x` with `y` and has the value `True` if the values are the same. The expression `x=y` assigns the value of `y` to `x`.

2.5.4 Logical Operators

The logical operators are used for comparing or negating the logical values of their operands and to return the resulting logical value. The values of the operands on which the logical operators operate evaluate to either `True` or `False`. The result of the logical operator is always a Boolean value, `True` or `False`. [TABLE 2.5](#) shows all the logical operators.

TABLE 2.5

List of Logical Operators

Operator	Operator Name	Description	Example
and	Logical AND	Performs AND operation and the result is <code>True</code> when both operands are <code>True</code>	<code>p</code> and <code>q</code> results in <code>False</code>
or	Logical OR	Performs OR operation and the result is <code>True</code> when any one of both operand is <code>True</code>	<code>p</code> or <code>q</code> results in <code>True</code>
not	Logical NOT	Reverses the operand state	not <code>p</code> results in <code>False</code>

Note: The Boolean value of `p` is `True` and `q` is `False`.

The Boolean logic Truth table is shown in [TABLE 2.6](#).

TABLE 2.6

Boolean Logic Truth Table

P	Q	P and Q	P or Q	Not P
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

For example,

1. `>>> True and False`
`False`
2. `>>> True or False`
`True`
3. `>>> not(True) and False`
`False`
4. `>>> not(True and False)`
`True`
5. `>>> (10 < 0) and (10 > 2)`
`False`
6. `>>> (10 < 0) or (10 > 2)`
`True`
7. `>>> not(10 < 0) or (10 > 2)`
`True`
8. `>>> not(10 < 0 or 10 > 2)`
`False`

Above code illustrates various comparison operations ①–⑧.

As logical expressions are evaluated left to right, they are tested for possible “short-circuit” valuation using the following rules:

1. `False and (some_expression)` is short-circuit evaluated to `False`.
2. `True or (some_expression)` is short-circuit evaluated to `True`.

The rules of logic guarantee that these evaluations are always correct. The *some_expression* part of the above expressions is not evaluated, so any side effects of doing so do not take effect. For example,

1. `>>> 1 > 2 and 9 > 6`
`False`
2. `>>> 3 > 2 or 8 < 4`
`True`

In ① the expression `1 > 2` is evaluated to *False*. Since *and* operator is used in the statement the expression is evaluated to *False* and the remaining expression `9 > 6` is not evaluated. In ② the expression `3 > 2` is evaluated to *True*. As *or* operator is used in the statement the expression is evaluated to *True* while the remaining expression `8 < 4` is ignored.

2.5.5 Bitwise Operators

Bitwise operators treat their operands as a sequence of bits (zeroes and ones) and perform bit by bit operation. For example, the decimal number ten has a binary representation

of 1010. Bitwise operators perform their operations on such binary representations, but they return standard Python numerical values. [TABLE 2.7](#) shows all the bitwise operators.

TABLE 2.7

List of Bitwise Operators

Operator	Operator Name	Description	Example
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands are 1s.	$p \& q = 12$ (means 0000 1100)
	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.	$p q = 61$ (means 0011 1101)
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.	$(p \wedge q) = 49$ (means 0011 0001)
~	Binary Ones Complement	Inverts the bits of its operand.	$(\sim p) = -61$ (means 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	$p << 2 = 240$ (means 1111 0000)
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	$p >> 2 = 15$ (means 0000 1111)

Note: The value of p is 60 and q is 13.

The Bitwise Truth table is shown in [TABLE 2.8](#).

TABLE 2.8

Bitwise Truth Table

P	Q	P & Q	P Q	P ^ Q	~P
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	



A sequence consisting of ones and zeroes is known as *binary*. The smallest amount of information that you can store in a computer is known as a *bit*. A *bit* is represented as either 0 or 1.

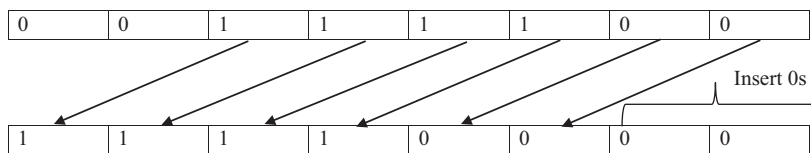
[FIGURE 2.1](#) shows examples for bitwise logical operations. The value of operand a is 60 and value of operand b is 13.

Bitwise and (&)	Bitwise or ()
a = 0011 1100 → (60)	a = 0011 1100 → (60)
b = 0000 1101 → (13)	b = 0000 1101 → (13)
a & b = 0000 1100 → (12)	a b = 0011 1101 → (61)
Bitwise exclusive or (^)	One's Complement (~)
a = 0011 1100 → (60)	a = 0011 1100 → (60)
b = 0000 1101 → (13)	~ a = 1100 0011 → (-61)
a ^ b = 0011 0001 → (49)	
Binary left shift (<<)	Binary right shift (>>)
a = 0011 1100 → (60)	a = 0011 1100 → (60)
a << 2 = 1111 0000 → (240)	a >> 2 = 0000 1111 → (15)
left shift of 2 bits	right shift of 2 bits

FIGURE 2.1

Examples of bitwise logical operators.

FIGURE 2.2 shows how the expression `60 << 2` would be evaluated in a byte.

**FIGURE 2.2**

Example of bitwise left shift of two bits.

Due to this operation,

- Each of the bits in the operand (60) is shifted two places to the left.
- The two bit positions emptied on the right end are filled with 0s.
- The resulting value is 240.

For example,

1. `>>> p = 60`
2. `>>> p << 2`
240
3. `>>> p = 60`
4. `>>> p >> 2`
15
5. `>>> q = 13`
6. `>>> p & q`
12
7. `>>> p | q`
61

```

8. >>> ~p
    -61
9. >>> p << 2
    240
10. >>> p >> 2
    15

```

Above code illustrates various Bitwise operations ①–⑩.

2.6 Precedence and Associativity

Operator precedence determines the way in which operators are parsed with respect to each other. Operators with higher precedence become the operands of operators with lower precedence. Associativity determines the way in which operators of the same precedence are parsed. Almost all the operators have left-to-right associativity. Operator precedence is listed in [TABLE 2.9](#) starting with the highest precedence to lowest precedence.

TABLE 2.9

Operator Precedence in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=,	Comparisons,
is, is not, in, not in	Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Consider the following code,

```

1. >>> 2 + 3 * 6
    20
2. >>> (2 + 3) * 6
    30
3. >>> 6 * 4 / 2
    12.0

```

Expressions with higher-precedence operators are evaluated first. In ① multiplication `*` is having precedence over addition. So, `3 * 6` gets evaluated first and the result is added to 2. This behavior can be overridden using parentheses as shown in ②. Parentheses have the highest precedence and the expression inside the parentheses gets evaluated first, which in our case is `2 + 3` and the result is multiplied with 6. In ③ both multiplication and division have the same precedence hence starting from left to right, the multiplication operator is evaluated first and the result is divided by 2.

2.7 Data Types

Data types specify the type of data like numbers and characters to be stored and manipulated within a program. Basic data types of Python are

- Numbers
- Boolean
- Strings
- None

2.7.1 Numbers

Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as `int`, `float` and `complex` class in Python. Integers can be of any length; it is only limited by the memory available. A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is an integer, 1.0 is floating point number. Complex numbers are written in the form, `x + yj`, where `x` is the real part and `y` is the imaginary part.

2.7.2 Boolean

Booleans may not seem very useful at first, but they are essential when you start using conditional statements. In fact, since a condition is really just a yes-or-no question, the answer to that question is a Boolean value, either `True` or `False`. The Boolean values, `True` and `False` are treated as reserved words.

2.7.3 Strings

A string consists of a sequence of one or more characters, which can include letters, numbers, and other types of characters. A string can also contain spaces. You can use single quotes or double quotes to represent strings and it is also called a string literal. Multiline strings can be denoted using triple quotes, `'''` or `"""`. These are fixed values, not variables that you literally provide in your script. For example,

1. `>>> s = 'This is single quote string'`
2. `>>> s = "This is double quote string"`

3.

```
>>> s = """This is  
Multiline  
string"""
```
4.

```
>>> s = "a"
```

In ① a string is defined using single quotes, in ② a string is defined using double quotes and a multiline string is defined in ③, a single character is also treated as string ④.



You use literals to represent values in Python. These are fixed values, not variables that you literally provide in your program. A string literal is zero or more characters enclosed in double (") or single (') quotation marks, an integer literal is 12 and float literal is 3.14.

2.7.4 None

None is another special data type in Python. *None* is frequently used to represent the absence of a value. For example,

1.

```
>>> money = None
```

None value is assigned to variable *money* ①.

2.8 Indentation

In Python, Programs get structured through indentation ([FIGURE 2.3](#)). Usually, we expect indentation from any program code, but in Python it is a requirement and not a matter of style. This principle makes the code look cleaner and easier to understand and read. Any statements written under another statement with the same indentation is interpreted to

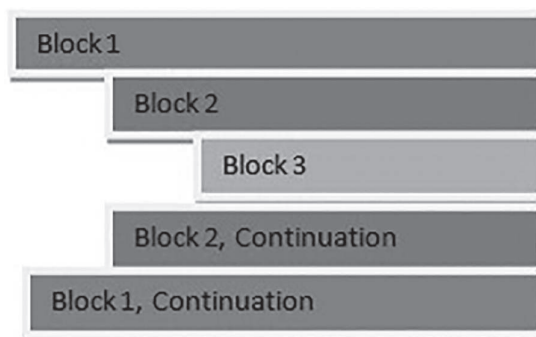


FIGURE 2.3
Code blocks and indentation in Python.

belong to the same code block. If there is a next statement with less indentation to the left, then it just means the end of the previous code block.

In other words, if a code block has to be deeply nested, then the nested statements need to be indented further to the right. In the above diagram, *Block 2* and *Block 3* are nested under *Block 1*. Usually, four whitespaces are used for indentation and are preferred over tabs. Incorrect indentation will result in *IndentationError*.

2.9 Comments

Comments are an important part of any program. A comment is a text that describes what the program or a particular part of the program is trying to do and is ignored by the Python interpreter. Comments are used to help you and other programmers understand, maintain, and debug the program. Python uses two types of comments: single-line comment and multiline comments.

2.9.1 Single Line Comment

In Python, use the hash (#) symbol to start writing a comment. Hash (#) symbol makes all text following it on the same line into a comment. For example,

```
#This is single line Python comment
```

2.9.2 Multiline Comments

If the comment extends multiple lines, then one way of commenting those lines is to use hash (#) symbol at the beginning of each line. For example,

```
#This is  
#multiline comments  
#in Python
```

Another way of doing this is to use triple quotes, either `'''` or `"""`. These triple quotes are generally used for multiline strings. However, they can be used as a multiline comment as well. For example,

```
"""This is  
multiline comment  
in Python using triple quotes"""
```

2.10 Reading Input

In Python, `input()` function is used to gather data from the user. The syntax for input function is,

```
variable_name = input([prompt])
```

prompt is a string written inside the parenthesis that is printed on the screen. The *prompt* statement gives an indication to the user of the value that needs to be entered through the keyboard. When the user presses Enter key, the program resumes and *input* returns what the user typed as a string. Even when the user inputs a number, it is treated as a string which should be casted or converted to number explicitly using appropriate type casting function.

1. `>>> person = input("What is your name?")`
2. What is your name? Carrey
3. `>>> person`
`'Carrey'`

① the *input()* function prints the prompt statement on the screen (in this case "What is your name?") indicating the user that keyboard input is expected at that point and then it waits for a line to be typed in. User types in his response in ②. The *input()* function reads the line from the user and converts the line into a string. As can be seen in ③, the line typed by the user is assigned to the *person* variable.



A function is a piece of code that is called by name. It can be passed data to operate on (i.e., the arguments) and can optionally return data (the return value). You shall learn more about functions in [Chapter 4](#).

2.11 Print Output

The *print()* function allows a program to display text onto the console. The print function will print everything as strings and anything that is not already a string is automatically converted to its string representation. For example,

1. `>>> print("Hello World!!")`
Hello World!!

① prints the string Hello World!! onto the console. Notice that the string Hello World is enclosed within double quotes inside the *print()* function.

Even though there are different ways to print values in Python, we discuss two major string formats which are used inside the *print()* function to display the contents onto the console as they are less error prone and results in cleaner code. They are

1. *str.format()*
2. f-strings

2.11.1 *str.format()* Method

Use *str.format()* method if you need to insert the value of a variable, expression or an object into another string and display it to the user as a single string. The *format()* method returns

a new string with inserted values. The `format()` method works for all releases of Python 3.x. The `format()` method uses its arguments to substitute an appropriate value for each format code in the template.

The syntax for `format()` method is,

```
str.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

where `p0, p1, ...` are called as positional arguments and, `k0, k1, ...` are keyword arguments with their assigned values of `v0, v1, ...` respectively.

Positional arguments are a list of arguments that can be accessed with an index of argument inside curly braces like `{index}`. Index value starts from zero.

Keyword arguments are a list of arguments of type *keyword = value*, that can be accessed with the name of the argument inside curly braces like `{keyword}`.

Here, *str* is a mixture of text and curly braces of indexed or keyword types. The indexed or keyword curly braces are replaced by their corresponding argument values and is displayed as a single string to the user.



The term “Method” is used almost exclusively in Object-oriented programming.

'Method' is the object-oriented word for 'function'. A Method is a piece of code that is called by a name that is associated with an object. You shall learn more about Classes and Objects in [Chapter 11](#).

Program 2.1: Program to Demonstrate `input()` and `print()` Functions

1. `country = input("Which country do you live in?")`
2. `print("I live in {}".format(country))`



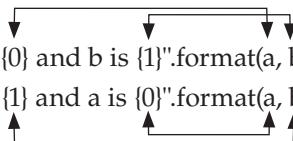
OUTPUT

```
Which country do you live in? India
I live in India
```

The 0 inside the curly braces `{0}` is the index of the first (0th) argument (here in our case, it is variable `country` ①) whose value will be inserted at that position ②.

Program 2.2: Program to Demonstrate the Positional Change of Indexes of Arguments

1. `a = 10`
2. `b = 20`
3. `print("The values of a is {0} and b is {1}".format(a, b))`
4. `print("The values of b is {1} and a is {0}".format(a, b))`



OUTPUT

```
The values of a is 10 and b is 20
The values of b is 20 and a is 10
```

You can have as many arguments as you want, as long as the indexes in curly braces have a matching argument in the argument list ③. {0} index gets replaced with the data value of variable a ① and {1} with the data value of variable b ②. This allows for re-arranging the order of display without changing the arguments ④.

```
1. >>> print("Give me {ball} ball".format(ball = "tennis"))
      Give me tennis ball
```

The keyword argument {ball} gets replaced with its assigned value ①.

2.11.2 f-strings

Formatted strings or f-strings were introduced in Python 3.6. A *f-string* is a string literal that is prefixed with "f". These strings may contain replacement fields, which are expressions enclosed within curly braces {}. The expressions are replaced with their values. In the real world, it means that you need to specify the name of the variable inside the curly braces to display its value. An f at the beginning of the string tells Python to allow any currently valid variable names within the string.



We use f-strings along within *print()* function to print the contents throughout this book, as f-strings are the most practical and straightforward way of formatting strings unless some special case arises.

Program 2.3: Code to Demonstrate the Use of f-strings with print() Function

```
1. country = input("Which country do you live in?")
2. print(f"I live in {country}")
```

OUTPUT

```
Which country do you live in? India
I live in India
```

Input string is assigned to variable *country* ①. Observe the character *f* prefixed before the quotes and the variable name is specified within the curly braces ②.

Program 2.4: Given the Radius, Write Python Program to Find the Area and Circumference of a Circle

```
1. radius = int(input("Enter the radius of a circle"))
2. area_of_a_circle = 3.1415 * radius * radius
3. circumference_of_a_circle = 2 * 3.1415 * radius
4. print(f"Area={area_of_a_circle}andCircumference={circumference_of_a_circle}")
```

OUTPUT

Enter the radius of a circle 5

Area = 78.53750000000001 and Circumference = 31.415000000000003

Get input for *radius* from the user ① and ② to calculate the *area* of a circle using the formula πr^2 and ③ *circumference* of a circle is calculated using the formula $2\pi r$. Finally, ④ print the results.

Program 2.5: Write Python Program to Convert the Given Number of Days to a Measure of Time Given in Years, Weeks and Days. For Example, 375 Days Is Equal to 1 Year, 1 Week and 3 Days (Ignore Leap Year).

```
1. number_of_days = int(input("Enter number of days"))
2. number_of_years = int(number_of_days/365)
3. number_of_weeks = int(number_of_days % 365 / 7)
4. remaining_number_of_days = int(number_of_days % 365 % 7)
5. print(f"Years = {number_of_years}, Weeks = {number_of_weeks}, Days = {remaining_number_of_days}")
```

OUTPUT

Enter number of days375

Years = 1, Weeks = 1, Days = 3

Total number of days is specified by the user ①. Number of years ② is calculated by dividing the total number of days by 365. To calculate the number of weeks ③, deduct the number of days using $\% 365$ and divide by 7. Now deduct the number of days using $\% 365$ and number of weeks by $\% 7$ to calculate the remaining number of days ④. Finally, display the results ⑤.

2.12 Type Conversions

You can explicitly cast, or convert, a variable from one type to another.

2.12.1 The *int()* Function

To explicitly convert a float number or a string to an integer, cast the number using *int()* function.

Program 2.6: Program to Demonstrate int() Casting Function

```
1. float_to_int = int(3.5)
2. string_to_int = int("1") #number treated as string
3. print(f"After Float to Integer Casting the result is {float_to_int}")
4. print(f"After String to Integer Casting the result is {string_to_int}")
```

OUTPUT

After Float to Integer Casting the result is 3

After String to Integer Casting the result is 1

Convert float and string values to integer ①–② and display the result ③–④.

1. >>>numerical_value = input("Enter a number")
Enter a number 9
2. >>> numerical_value
'9'
3. >>> numerical_value = int(input("Enter a number"))
Enter a number 9
4. >>> numerical_value
9

①–② User enters a value of 9 which gets assigned to variable *numerical_value* and the value is treated as string type. In order to assign an integer value to the variable, you have to enclose the *input()* function within the *int()* function which converts the input string type to integer type ③–④. A string to integer conversion is possible only when the string value is inherently a number (like “1”) and not a character.

2.12.2 The *float()* Function

The *float()* function returns a floating point number constructed from a number or string.

Program 2.7: Program to Demonstrate float() Casting Function

1. int_to_float = float(4)
2. string_to_float = float("1") #number treated as string
3. print(f"After Integer to Float Casting the result is {int_to_float}")
4. print(f"After String to Float Casting the result is {string_to_float}")

OUTPUT

After Integer to Float Casting the result is 4.0

After String to Float Casting the result is 1.0

Convert integer and string values to float ①–② and display the result ③–④.

2.12.3 The *str()* Function

The *str()* function returns a string which is fairly human readable.

Program 2.8: Program to Demonstrate str() Casting Function

1. int_to_string = str(8)
2. float_to_string = str(3.5)
3. print(f"After Integer to String Casting the result is {int_to_string}")
4. print(f"After Float to String Casting the result is {float_to_string}")

OUTPUT

After Integer to String Casting the result is 8

After Float to String Casting the result is 3.5

Here, integer and float values are converted ①–② to string using *str()* function and results are displayed ③–④.

2.12.4 The *chr()* Function

Convert an integer to a string of one character whose ASCII code is same as the integer using *chr()* function. The integer value should be in the range of 0–255.

Program 2.9: Program to Demonstrate *chr()* Casting Function

1. `ascii_to_char = chr(100)`
2. `print(f'Equivalent Character for ASCII value of 100 is {ascii_to_char}')`

OUTPUT

Equivalent Character for ASCII value of 100 is d

An integer value corresponding to an ASCII code is converted ① to the character and printed ②.

2.12.5 The *complex()* Function

Use *complex()* function to print a complex number with the value `real + imag*j` or convert a string or number to a complex number. If the first argument for the function is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the function serves as a numeric conversion function like `int()`, `long()` and `float()`. If both arguments are omitted, the *complex()* function returns `0j`.

Program 2.10: Program to Demonstrate *complex()* Casting Function

1. `complex_with_string = complex("1")`
2. `complex_with_number = complex(5, 8)`
3. `print(f'Result after using string in real part {complex_with_string}')`
4. `print(f'Result after using numbers in real and imaginary part {complex_with_number}')`

OUTPUT

Result after using string in real part (1+0j)

Result after using numbers in real and imaginary part (5+8j)

The first argument is a string ①. Hence you are not allowed to specify the second argument. In ② the first argument is an integer type, so you can specify the second argument which is also an integer. Results are printed out in ③ and ④.

2.12.6 The *ord()* Function

The *ord()* function returns an integer representing Unicode code point for the given Unicode character.

Program 2.11: Program to Demonstrate *ord()* Casting Function

```
1. unicode_for_integer = ord('4')
2. unicode_for_alphabet = ord("Z")
3. unicode_for_character = ord("#")
4. print(f'Unicode code point for integer value of 4 is {unicode_for_integer}')
5. print(f'Unicode code point for alphabet 'A' is {unicode_for_alphabet}')
6. print(f'Unicode code point for character '$' is {unicode_for_character}')
```

OUTPUT

```
Unicode code point for integer value of 4 is 52
Unicode code point for alphabet 'A' is 90
Unicode code point for character '$' is 35
```

The *ord()* function converts an integer ①, alphabet ② and a character ③ to its equivalent Unicode code point integer value and prints the result ④–⑥.

2.12.7 The *hex()* Function

Convert an integer number (of any size) to a lowercase hexadecimal string prefixed with “0x” using *hex()* function.

Program 2.12: Program to Demonstrate *hex()* Casting Function

```
1. int_to_hex = hex(255)
2. print(f'After Integer to Hex Casting the result is {int_to_hex}')
```

OUTPUT

```
After Integer to Hex Casting the result is 0xff
```

Integer value of 255 is converted to equivalent hex string of 0xff ① and result is printed as shown in ②.

2.12.8 The *oct()* Function

Convert an integer number (of any size) to an octal string prefixed with “0o” using *oct()* function.

Program 2.13: Program to Demonstrate *oct()* Casting Function

```
1. int_to_oct = oct(255)
2. print(f'After Integer to Hex Casting the result is {int_to_oct}')
```

OUTPUT

After Integer to Hex Casting the result is 0o377

Integer value of 255 is converted to equivalent oct string of 0o377 ① and result is printed as shown in ②.

2.13 The *type()* Function and Is Operator

The syntax for *type()* function is,

type(object)

The *type()* function returns the data type of the given object.

1. >>> type(1)
 <class 'int'>
2. >>> type(6.4)
 <class 'float'>
3. >>> type("A")
 <class 'str'>
4. >>> type(True)
 <class 'bool'>

The *type()* function comes in handy if you forget the type of variable or an object during the course of writing programs.

The operators *is* and *is not* are identity operators. Operator *is* evaluates to *True* if the values of operands on either side of the operator point to the same object and *False* otherwise. The operator *is not* evaluates to *False* if the values of operands on either side of the operator point to the same object and *True* otherwise.

1. >>> x = "Seattle"
2. >>> y = "Seattle"
3. >>> x is y
 True

Both *x* and *y* point to same object "Seattle" ①–② and results in *True* when evaluated using *is* operator ③.

2.14 Dynamic and Strongly Typed Language

Python is a dynamic language as the type of the variable is determined during run-time by the interpreter. Python is also a strongly typed language as the interpreter keeps track of all the variables types. In a strongly typed language, you are simply not allowed to do anything that's incompatible with the type of data you are working with. For example,

1. `>>> 5 + 10`

15

2. `>>> 1 + "a"`

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

In ① values which are of integer types are added and they are compatible. However, in ② when you try to add 1, which is an integer type with "a" which is string type, then it results in Traceback as they are not compatible.

In Python, Traceback is printed when an error occurs. The last line tells us the kind of error that occurred which in our case is the unsupported operand type(s).

2.15 Summary

- Identifier is a name given to the variable and must begin with a letter or underscore and can include any number of letter, digits or underscore.
- Keywords have predefined meaning.
- A Variable holds a value that may change.
- Python supports the following operators:
 - Arithmetic Operators
 - Comparison
 - Assignment
 - Logical
 - Bitwise
 - Identity
- Statement is a unit of code that the Python interpreter can execute.
- An expression is a combination of variables, operators, values and reserved keywords.
- The Hash (#) and Triple Quotes ("'' ''") are used for commenting.
- Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.
- The `print()` and `input()` functions handle simple text output (to the screen) and input (from the keyboard).
- The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, or floating-point number form of the value they are passed.
- A docstring is a string enclosed by triple quotation marks and provides program documentation.

- Comments are pieces of code that are not evaluated by the interpreter but can be read by programmers to obtain information about a program.
 - The type conversion functions can be used to convert a value of one type to a value of another type after input.
-

Multiple Choice Questions

1. Which of the following are invalid identifiers in Python?
 - a. Total-sum
 - b. Error
 - c. Error_count
 - d. None of these
2. A _____ is a sequence of one or more characters used to provide a name for a given program element.
 - a. Identifier
 - b. Variable
 - c. String
 - d. Character
3. Identify the invalid identifier below.
 - a. _2017discount
 - b. Profit
 - c. Total-discount
 - d. Totaldiscount
4. _____ are not allowed as part of an identifier.
 - a. Spaces
 - b. Numbers
 - c. Underscore
 - d. All of these
5. Identifiers may contain letters and digits, but cannot begin with a _____.
 - a. Character
 - b. Digit
 - c. Underscore
 - d. Special Symbols
6. Which is not a reserved keyword in Python?
 - a. insert
 - b. except
 - c. import
 - d. yield

7. Identify the invalid keyword below.
- and
 - as
 - while
 - until
8. _____ is an identifier that has predefined meaning.
- variable
 - identifier
 - keyword
 - None of these
9. Bitwise _____ operator gives 1 if one of the bit is zero and the other is 1.
- or
 - and
 - xor
 - not
10. Guess the output of the following code.
- $1 > 2$ and $9 > 6$
- True
 - False
 - Machine Dependent
 - Error
11. How many operands are there in the following arithmetic expression?
- $6 * 35 + 8 - 25$
- 4
 - 3
 - 5
 - 8
12. How many binary operators are there in the following arithmetic expression?
- $- 6 + 10 / (23 + 56)$
- 2
 - 3
 - 4
 - 5
13. Which operator returns the remainder of the operands?
- /
 - //
 - %
 - **

14. A _____ is a name that is associated with a value.
- identifier
 - keyword
 - variable
 - None of these
15. Guess the output of the following expression.
- `float(22//3+3/3)`
- 8
 - 8.0
 - 8.3
 - 8.333
16. What value does the following expression evaluate to?
- `2 + 9 * ((3 * 12) - 8) / 10`
- 27
 - 27.2
 - 30.8
 - None of these
17. _____ and _____ are two ways to comment in Python.
- Single and Multilevel comments
 - Single line and Double line comments
 - One and Many line comments
 - Single line and Multiline comments
18. Single-line comments start with the _____ symbol.
- `##`
 - `#`
 - `*`
 - `&`
19. Multiline comments can be done by adding _____ on each end of the comment.
- `"""`(triple quote)
 - `#` (Hash)
 - `$` (dollar)
 - `%` (modulus)
20. Python programs get structured through _____.
- Alignment
 - Indentation
 - Justification
 - None

21. In Python, Indentation is a _____ and not a matter of style.
 - a. Requirement
 - b. Refinement
 - c. Not required
 - d. Not Refined
22. Which of the following is correct about Python?
 - a. Python is a high-level, interpreted, interactive and object-oriented language.
 - b. Python is designed to be highly readable.
 - c. It uses English keywords frequently and has fewer syntactical constructions.
 - d. All of the above.
23. Which of the following function is used to read data from the keyboard?
 - a. function()
 - b. str()
 - c. input()
 - d. print()
24. The one's complement of 60 is given by _____.
 - a. -61
 - b. -60
 - c. -59
 - d. +59
25. The operators *is* and *is not* are _____.
 - a. Identity Operators
 - b. Comparison Operators
 - c. Membership Operators
 - d. Unary Operators
26. In Python an identifier is _____.
 - a. Machine Dependent
 - b. Keyword
 - c. Case Sensitive
 - d. Constant
27. Which of the following operator is truncation division operator?
 - a. /
 - b. %
 - c. |
 - d. //

28. The expression that requires type conversion when evaluated is _____.
a. $4.7 * 6.3$
b. $1.7 \% 2$
c. $3.4 + 4.6$
d. $79 * 6.3$
29. The operator that has the highest precedence is _____.
a. $<<$ and $>>$
b. $**$
c. $+$
d. $\%$
30. The expression that results in an error is _____.
a. `int('10.8')`
b. `float(10)`
c. `int(10)`
d. `float(10.8)`
31. Which of the following expression is an example of type conversion?
a. $4.0 + \text{float}(3)$
b. $5.3 + 6.3$
c. $5.0 + 3$
d. $3 + 7$
32. What is the output when the following statement is executed?
`>>>print('new' 'line')`
a. Error
b. Output equivalent to `print 'new\nline'`
c. new line
d. newline
33. What is the output when the following statement is executed?
`print(0xD + 0xE + 0xF)`
a. Error
b. `0XD0XE0XF`
c. `0X22`
d. 42
34. What is the output of `print (0.1 + 0.2 == 0.3)`?
a. True
b. False
c. Error
d. Machine dependent

35. Which of the following is not a complex number?
- a. $l = 4 + 5j$
 - b. $l = \text{complex}(4,5)$
 - c. $l = 4 + 5i$
 - d. $l = 4 + 5j$
36. Guess the output of the expression.
- ```
x = 15
y = 12
x & y
```
- a. 1101
  - b. b1101
  - c. 0b1101
  - d. 12
37. Incorrect Indentation results in \_\_\_\_\_.
- a. IndentationError
  - b. NameError
  - c. TypeError
  - d. SyntaxError
38. The function that converts an integer to a string of one character whose ASCII code is same as the integer is \_\_\_\_\_.
- a. `chr(x)`
  - b. `ord(x)`
  - c. `eval(x)`
  - d. `input(x)`

---

## Review Questions

1. Explain different Operators in Python with examples.
2. Define a variable. How to assign values to them?
3. Briefly explain binary left shift and binary right shift operators with examples.
4. Explain precedence and associativity of operators with examples.
5. Outline different assignment operators with examples.
6. Briefly explain how to read data from the keyboard.
7. Explain Type conversion in Python with examples.
8. Write a short note on data types in Python.

9. Write a program to read two integers and perform arithmetic operations on them (addition, subtraction, multiplication and division).
10. Write a program to read the marks of three subjects and find the average of them.
11. Write a program to convert kilogram into pound.
12. Surface area of a prism can be calculated if the lengths of the three sides are known. Write a program that takes the sides as input (read it as integer) and prints the surface area of the prism ( $\text{Surface Area} = 2ab + 2bc + 2ca$ ).
13. A plane travels 395,000 meters in 9000 seconds. Write a program to find the speed of the plane ( $\text{Speed} = \text{Distance} / \text{Time}$ ).
14. You need to empty out the rectangular swimming pool which is 12 meters long, 7 meters wide and 2 meter depth. You have a pump which can move 17 cubic meters of water in an hour. Write a program to find how long it will take to empty your pool? ( $\text{Volume} = l * w * h$ , and  $\text{flow} = \text{volume}/\text{time}$ ).
15. Write a program to convert temperature from centigrade (read it as float value) to Fahrenheit.
16. Write a program that calculates the number of seconds in a day.
17. A car starts from a stoplight and is traveling with a velocity of 10 m/sec east in 20 seconds. Write a program to find the acceleration of the car. ( $\text{acc} = (v_{\text{final}} - v_{\text{initial}}) / \text{time}$ ).

# 3

## Control Flow Statements

### AIM

Understand if, if...else and if...elif...else statements and use of control flow statements that provide a means for looping over a section of code multiple times within the program.

### LEARNING OUTCOMES

At the end of this chapter, you are expected to

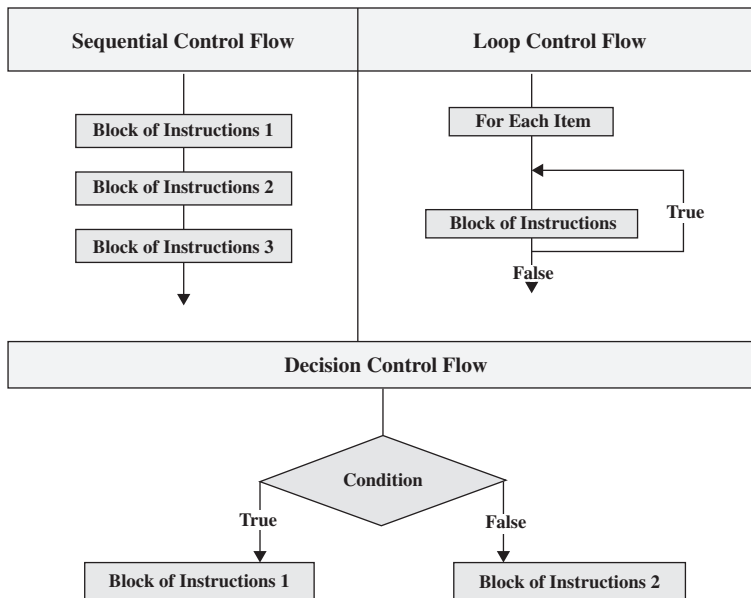
- Use if, if...else and if...elif...else statements to transfer the control from one part of the program to another.
- Write while and for loops to run one or more statements repeatedly.
- Control the flow of execution using *break* and *continue* statements.
- Improve the reliability of code by incorporating exception handling mechanisms through try-except blocks.

Python supports a set of control flow statements that you can integrate into your program. The statements inside your Python program are generally executed *sequentially* from top to bottom, in the order that they appear. Apart from sequential control flow statements you can employ decision making and looping control flow statements to break up the flow of execution thus enabling your program to conditionally execute particular blocks of code. The term *control flow* details the direction the program takes.

The control flow statements ([FIGURE 3.1](#)) in Python Programming Language are

1. **Sequential Control Flow Statements:** This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.
2. **Decision Control Flow Statements:** Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if...else and if...elif...else).
3. **Loop Control Flow Statements:** This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (*for* loop and *while* loop). Loop Control Flow Statements are also called Repetition statements or Iteration statements.

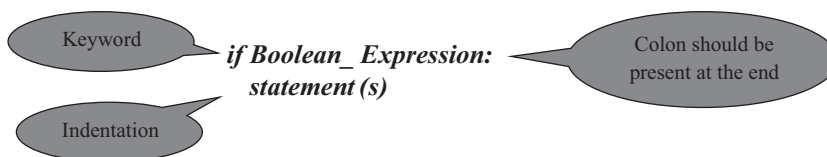




**FIGURE 3.1**  
Forms of control flow statements.

### 3.1 The *if* Decision Control Flow Statement

The syntax for *if* statement is,



The *if* decision control flow statement starts with *if* keyword and ends with a colon. The expression in an *if* statement should be a Boolean expression. The *if* statement decides whether to run some particular statement or not depending upon the value of the Boolean expression. If the Boolean expression evaluates to *True* then statements in the *if* block will be executed; otherwise the result is *False* then none of the statements are executed. In Python, the *if* block statements are determined through indentation and the first unindented statement marks the end. You don't need to use the `==` operator explicitly to check if a variable's value evaluates to *True* as the variable name can itself be used as a condition.

For example,

1. `>>> if 20 > 10:`
2. `... print(f"20 is greater than 10")`

**Output**

20 is greater than 10

In ①, the Boolean expression  $20 > 10$  is evaluated to Boolean *True* and the print statement ② is executed.

**Program 3.1: Program Reads a Number and Prints a Message If It Is Positive**

1. `number = int(input("Enter a number"))`
2. `if number >= 0:`
3.     `print(f"The number entered by the user is a positive number")`

**OUTPUT**

Enter a number 67

The number entered by the user is a positive number

The value entered by the user is read and stored in the *number* variable ①, the value in the *number* variable ② is checked to determine if it is greater than or equal to 0, if *True* then print the message ③.

**Program 3.2: Program to Read Luggage Weight and Charge the Tax Accordingly**

1. `weight = int(input("How many pounds does your suitcase weigh?"))`
2. `if weight > 50:`
3.     `print(f"There is a $25 surcharge for heavy luggage")`
4.     `print(f"Thank you")`

**OUTPUT**

How many pounds does your suitcase weigh? 75

There is a \$25 surcharge for heavy luggage

Thank you

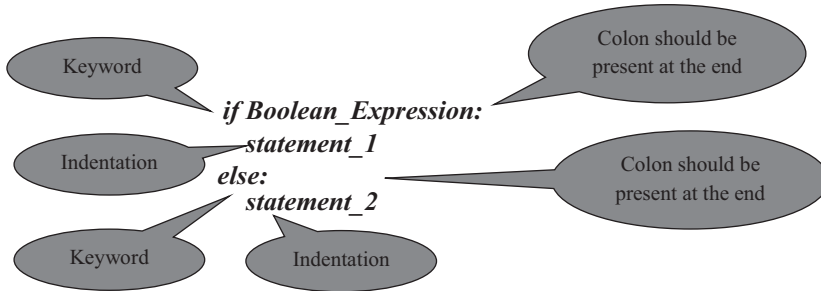
The *weight* of the luggage is read ① and if it is greater than 50 ② then extra charges are collected. Lines ③ and ④ are present in the indentation block of *if* statement. If the weight of the luggage is less than 50 then nothing is printed.

---

## 3.2 The *if...else* Decision Control Flow Statement

An *if* statement can also be followed by an *else* statement which is optional. An *else* statement does not have any condition. Statements in the *if* block are executed if the Boolean\_Expression is *True*. Use the optional *else* block to execute statements if the Boolean\_Expression is *False*. The *if...else* statement allows for a two-way decision.

The syntax for *if...else* statement is,



If the `Boolean_Expression` evaluates to `True`, then `statement_1` is executed, otherwise it is evaluated to `False` then `statement_2` is executed. Indentation is used to separate the blocks. After the execution of either `statement_1` or `statement_2`, the control is transferred to the next statement after the *if* statement. Also, *if* and *else* keywords should be aligned at the same column position.



Here, `statement`, `statement_1`, `statement_2` and so on can be either a single statement or multiple statements. `Boolean_Expression`, `Boolean_Expression_1`, `Boolean_Expression_2` and so on are expressions of the Boolean type which gets evaluated to either `True` or `False`.

### Program 3.3: Program to Find If a Given Number Is Odd or Even

1. `number = int(input("Enter a number"))`
2. `if number % 2 == 0:`
3.     `print(f"{number} is Even number")`
4. `else:`
5.     `print(f"{number} is Odd number")`

#### OUTPUT

```

Enter a number: 45
45 is Odd number

```

A number is read and stored in the variable named *number* ①. The *number* is checked using modulus operator ② to determine whether the *number* is perfectly divisible by 2 or not. If the *number* is perfectly divisible by 2, then the expression is evaluated to `True` and *number* is even ③. However, ④ if the expression evaluates to `False`, the *number* is odd ⑤.

### Program 3.4: Program to Find the Greater of Two Numbers

1. `number_1 = int(input("Enter the first number"))`
2. `number_2 = int(input("Enter the second number"))`

3. `if number_1 > number_2:`
4.   `print(f'{number_1} is greater than {number_2}')`
5. `else:`
6.   `print(f'{number_2} is greater than {number_1}')`

**OUTPUT**

Enter the first number 8  
 Enter the second number 10  
 10 is greater than 8

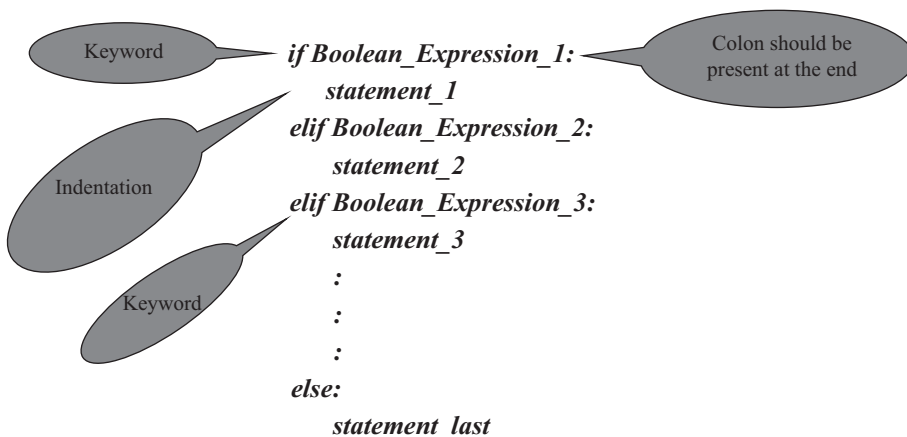
Two numbers are read using the input function and the values are stored in the variables `number_1` ① and `number_2` ②. The Boolean expression is evaluated ③ and if it is *True*, then line ④ is executed else ⑤ if the Boolean expression is evaluated to *False*, then line ⑥ is executed.

---

### 3.3 The *if...elif...else* Decision Control Statement

The *if...elif...else* is also called as multi-way decision control statement. When you need to choose from several possible alternatives, then an *elif* statement is used along with an *if* statement. The keyword '*elif*' is short for 'else if' and is useful to avoid excessive indentation. The *else* statement must always come last, and will again act as the default action.

The syntax for *if...elif...else* statement is,



This *if...elif...else* decision control statement is executed as follows:

- In the case of multiple Boolean expressions, only the first logical Boolean expression which evaluates to *True* will be executed.
- If `Boolean_Expression_1` is *True*, then `statement_1` is executed.
- If `Boolean_Expression_1` is *False* and `Boolean_Expression_2` is *True*, then `statement_2` is executed.

- If `Boolean_Expression_1` and `Boolean_Expression_2` are *False* and `Boolean_Expression_3` is *True*, then `statement_3` is executed and so on.
- If none of the `Boolean_Expression` is *True*, then `statement_last` is executed.



There can be zero or more *elif* parts each followed by an indented block, and the *else* part is optional. There can be only one *else* block. An *if...elif...else* statement is a substitute for the switch or case statements found in other programming languages.

**Program 3.5: Write a Program to Prompt for a Score between 0.0 and 1.0. If the Score Is Out of Range, Print an Error. If the Score Is between 0.0 and 1.0, Print a Grade Using the Following Table**

| Score      | Grade |
|------------|-------|
| $\geq 0.9$ | A     |
| $\geq 0.8$ | B     |
| $\geq 0.7$ | C     |
| $\geq 0.6$ | D     |
| $< 0.6$    | F     |

```

1. score = float(input("Enter your score"))
2. if score < 0 or score > 1:
3. print('Wrong Input')
4. elif score >= 0.9:
5. print('Your Grade is "A" ')
6. elif score >= 0.8:
7. print('Your Grade is "B" ')
8. elif score >= 0.7:
9. print('Your Grade is "C" ')
10. elif score >= 0.6:
11. print('Your Grade is "D" ')
12. else:
13. print('Your Grade is "F" ')

```

#### OUTPUT

```

Enter your score0.92
Your Grade is "A"

```

A number is read and is assigned to the variable *score* ①. If the *score* value is greater than 1 or less than 0 ② then we display an error message indicating to the user that it is a wrong input ③. If not, the *score* value is checked for different conditions based on the score table and the grade statements are printed accordingly ④–⑬.

**Program 3.6: Program to Display the Cost of Each Type of Fruit**

```

1. fruit_type = input("Enter the Fruit Type:")
2. if fruit_type == "Oranges":
3. print('Oranges are $0.59 a pound')
4. elif fruit_type == "Apples":
5. print('Apples are $0.32 a pound')
6. elif fruit_type == "Bananas":
7. print('Bananas are $0.48 a pound')
8. elif fruit_type == "Cherries":
9. print('Cherries are $3.00 a pound')
10. else:
11. print(f'Sorry, we are out of {fruit_type}')

```

**OUTPUT**

```

Enter the Fruit Type: Cherries
Cherries are $3.00 a pound

```


A string value is read and assigned to the variable *fruit\_type* ①. The value of the *fruit\_type* variable is checked against different strings ②–⑨. If the *fruit\_type* value matches with the existing string, then a message is printed else inform the user of the unavailability of the fruit ⑩–⑪.

---

**3.4 Nested if Statement**

In some situations, you have to place an *if* statement inside another statement. An *if* statement that contains another *if* statement either in its *if* block or *else* block is called a Nested *if* statement.

The syntax of the nested *if* statement is,



```

if Boolean_Expression_1:
 if Boolean_Expression_2:
 statement_1
 else:
 statement_2
else:
 statement_3

```

If the *Boolean\_Expression\_1* is evaluated to *True*, then the control shifts to *Boolean\_Expression\_2* and if the expression is evaluated to *True*, then *statement\_1* is executed, if the *Boolean\_Expression\_2* is evaluated to *False* then the *statement\_2* is executed. If the *Boolean\_Expression\_1* is evaluated to *False*, then *statement\_3* is executed.

**Program 3.7: Program to Check If a Given Year Is a Leap Year**

```

1. year = int(input('Enter a year'))
2. if year % 4 == 0:
3. if year % 100 == 0:
4. if year % 400 == 0:
5. print(f'{year} is a Leap Year')
6. else:
7. print(f'{year} is not a Leap Year')
8. else:
9. print(f'{year} is a Leap Year')
10. else:
11. print(f'{year} is not a Leap Year')

```

**OUTPUT**

```

Enter a year 2014
2014 is not a Leap Year

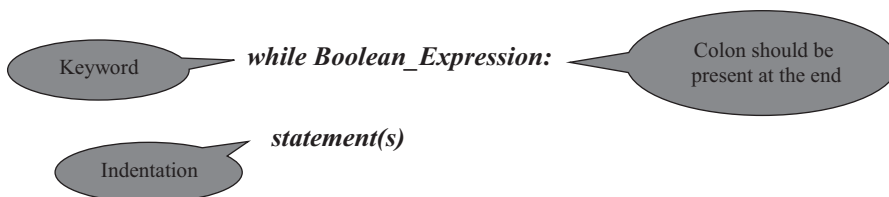
```

All years which are perfectly divisible by 4 are leap years except for century years (years ending with 00) which is a leap year only if it is perfectly divisible by 400. For example, years like 2012, 2004, 1968 are leap years but 1971, 2006 are not leap years. Similarly, 1200, 1600, 2000, 2400 are leap years but 1700, 1800, 1900 are not.

Read the value for *year* as input ①. Check whether the given *year* is divisible by 4 ② and also by 400 ④. If the condition is *True*, then the *year* is a leap year ⑤. If the *year* is divisible by 4 and not divisible by 100 ③ then the *year* is a leap year ⑨. If the condition at ② or ④ becomes *False*, then the *year* is not a leap year ⑦ and ⑩.

**3.5 The *while* Loop**

The syntax for *while* loop is,



The *while* loop starts with the *while* keyword and ends with a colon. With a *while* statement, the first thing that happens is that the Boolean expression is evaluated before the statements in the *while* loop block is executed. If the Boolean expression evaluates to *False*, then the statements in the *while* loop block are never executed. If the Boolean expression evaluates to *True*, then the *while* loop block is executed. After each iteration of the loop block, the Boolean expression is again checked, and if it is *True*, the loop is iterated again.

Each repetition of the loop block is called an iteration of the loop. This process continues until the Boolean expression evaluates to *False* and at this point the *while* statement exits. Execution then continues with the first statement after the *while* loop.

**Program 3.8: Write Python Program to Display First 10 Numbers Using *while* Loop Starting from 0**

```
1. i = 0
2. while i < 10:
3. print(f"Current value of i is {i}")
4. i = i + 1
```

**OUTPUT**

```
Current value of i is 0
Current value of i is 1
Current value of i is 2
Current value of i is 3
Current value of i is 4
Current value of i is 5
Current value of i is 6
Current value of i is 7
Current value of i is 8
Current value of i is 9
```

Variable *i* is assigned with 0 outside the loop ①. The expression *i < 10* is evaluated ②. If the value of *i* is less than 10 (i.e., *True*) then the body of the loop is executed. Value of *i* is printed ③ and *i* is incremented by 1 ④. This continues until the expression in *while* loop becomes *False*.

**Program 3.9: Write a Program to Find the Average of *n* Natural Numbers Where *n* Is the Input from the User**

```
1. number = int(input("Enter a number up to which you want to find the average"))
2. i = 0
3. sum = 0
4. count = 0
5. while i < number:
6. i = i + 1
7. sum = sum + i
8. count = count + 1
9. average = sum/count
10. print(f"The average of {number} natural numbers is {average}")
```

**OUTPUT**

```
Enter a number up to which you want to find the average 5
The average of 5 natural numbers is 3.0
```



The variables *i*, *sum* and *count* are assigned with zero ②, ③, ④. The expression *i* < *number* is evaluated ⑤. Since it is *True* for the first iteration, the body of the *while* loop gets executed. Variable *i* gets incremented ⑥ by value of 1 and it generates the required natural numbers. The *sum* variable adds the value of *sum* variable with the value of *i* variable ⑦, while the *count* variable keeps track of number of times the body of the loop gets executed ⑧. The loop gets repeated until the test expression becomes *False*. The *average* is calculated as *sum*/*count* ⑨ and displayed ⑩.

### Program 3.10: Program to Find the GCD of Two Positive Numbers

```

1. m = int(input("Enter first positive number"))
2. n = int(input("Enter second positive number"))
3. if m == 0 and n == 0:
4. print("Invalid Input")
5. if m == 0:
6. print(f"GCD is {n}")
7. if n == 0:
8. print(f"GCD is {m}")
9. while m != n:
10. if m > n:
11. m = m - n
12. if n > m:
13. n = n - m
14. print(f"GCD of two numbers is {m}")

```

#### OUTPUT

```

Enter first positive number8
Enter second positive number12
GCD of two numbers is 4

```

Read the value for *m* and *n* ①–②. If both *m* and *n* are zero then it is invalid input because zero cannot be divided by zero which is indeterminate ③–④. If either *m* or *n* is zero then the other one is gcd ⑤–⑧. If the value of *m* > *n* then *m* = *m* - *n* or if *n* > *m* then *n* = *n* - *m*. The logic in line ⑩–⑬ is repeated until the value of *m* is equal to the value of *n* ⑨. Then gcd will be either *m* or *n* ⑭.

### Program 3.11: Write Python Program to Find the Sum of Digits in a Number

```

1. number = int(input("Enter a number"))
2. result = 0
3. remainder = 0
4. while number != 0:
5. remainder = number % 10

```

```
6. result = result + remainder
7. number = int(number / 10)
8. print(f"The sum of all digits is {result}")
```

**OUTPUT**

```
Enter a number1234
The sum of all digits is 10
```

Read a number from user ① and store it in a variable called *number*. Assign zero to the variables *result* and *remainder* ②–③. Find the last digit of the number. To get the last digit of the *number* use modulus division by 10 and assign it the variable *remainder* ⑤. Add the last digit that you obtained with the *result* variable ⑥. Then remove the last digit from the *number* by dividing the *number* by 10 and cast it as int ⑦. Repeat the logic in line ⑤–⑦ till the variable *number* becomes 0 ④. Finally, you will be left with the sum of digits in the *result* variable ⑧.

**Program 3.12: Write a Program to Display the Fibonacci Sequences up to *n*th Term Where *n* is Provided by the User**

```
1. nterms = int(input('How many terms?'))
2. current = 0
3. previous = 1
4. count = 0
5. next_term = 0
6. if nterms <= 0:
7. print('Please enter a positive number')
8. elif nterms == 1:
9. print('Fibonacci sequence')
10. print('0')
11. else:
12. print("Fibonacci sequence")
13. while count < nterms:
14. print(next_term)
15. current = next_term
16. next_term = previous + current
17. previous = current
18. count += 1
```

**OUTPUT**

```
How many terms? 5
Fibonacci sequence
0
1
1
2
3
```

In a Fibonacci sequence, the next number is obtained by adding the previous two numbers. The first two numbers of the Fibonacci sequence are 0 and 1. The next number is obtained by adding 0 and 1 which is 1. Again, the next number is obtained by adding 1 and 1 which is 2 and so on. Get a number from user ① up to which you want to generate the Fibonacci sequence. Assign values to variables *current*, *previous*, *next\_term* and *count* ②–⑤. The variable *count* keeps track of number of times the *while* block is executed. User is required to enter a positive number to generate the Fibonacci sequence ⑥–⑦. If the user asks to generate a single number in the sequence, then print zero ⑧–⑩. The *next\_term* is obtained ④ by adding the *previous* and *current* variables and the logic from ⑤–⑧ is repeated until *while* block conditional expression becomes *False* ⑩.

**Program 3.13: Program to Repeatedly Check for the Largest Number Until the User Enters “done”**

```

1. largest_number = int(input("Enter the largest number initially"))
2. check_number = input("Enter a number to check whether it is largest or not")
3. while check_number != "done":
4. if largest_number > int(check_number):
5. print(f"Largest Number is {largest_number}")
6. else:
7. largest_number = int(check_number)
8. print(f"Largest Number is {largest_number}")
9. check_number = input("Enter a number to check whether it is largest or not")

```

**OUTPUT**

```

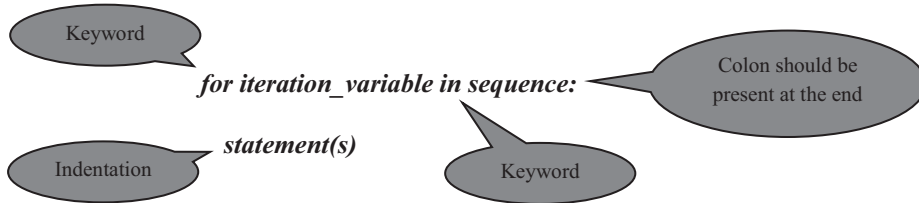
Enter the largest number initially 5
Enter a number to check whether it is largest or not 1
Largest Number is 5
Enter a number to check whether it is largest or not 10
Largest Number is 10
Enter a number to check whether it is largest or not 8
Largest Number is 10
Enter a number to check whether it is largest or not done

```

A number is read initially which is assumed to be the *largest\_number* ①. Then, the user is prompted to enter another number which is assigned to variable *check\_number* ②. Within the *while* loop ③ the value of *check\_variable* is compared with that of *largest\_variable* ④–⑤. If the *check\_variable* has a larger value then that value is assigned to *largest\_variable* ⑥–⑧. The user is again prompted to enter another value which is compared against the *largest\_number* value ⑨. This continues until the user enters the string "done" instead of a numerical value.

### 3.6 The *for* Loop

The syntax for the *for* loop is,



The *for* loop starts with *for* keyword and ends with a colon. The first item in the sequence gets assigned to the iteration variable *iteration\_variable*. Here, *iteration\_variable* can be any valid variable name. Then the statement block is executed. This process of assigning items from the sequence to the *iteration\_variable* and then executing the statement continues until all the items in the sequence are completed.

We take the liberty of introducing you to *range()* function which is a built-in function at this stage as it is useful in demonstrating *for* loop. The *range()* function generates a sequence of numbers which can be iterated through using *for* loop. The syntax for *range()* function is,

*range([start ,] stop [, step])*

Both start and step arguments are optional and the range argument value should always be an integer.

**start** → value indicates the beginning of the sequence. If the start argument is not specified, then the sequence of numbers start from zero by default.

**stop** → Generates numbers up to this value but not including the number itself.

**step** → indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

**NOTE:** The square brackets in the syntax indicate that these arguments are optional. You can leave them out.

#### Program 3.14: Demonstrate *for* Loop Using *range()* Function

1. `print("Only "stop" argument value specified in range function")`
2. `for i in range(3):`
3.     `print(f"{i}")`
4. `print("Both "start" and "stop" argument values specified in range function")`
5. `for i in range(2, 5):`
6.     `print(f"{i}")`
7. `print("All three arguments "start", "stop" and "step" specified in range function")`
8. `for i in range(1, 6, 3):`
9.     `print(f"{i}")`

**OUTPUT**

Only "stop" argument value specified in range function

0

1

2

Both "start" and "stop" argument values specified in range function

2

3

4

All three arguments "start", "stop" and "step" specified in range function

1

4

The function `range(3)` generates numbers starting from 0 to 2 ②. During the first iteration, the 0th value gets assigned to the iteration variable *i* and the same gets printed out ③. This continues to execute until all the numbers generated using `range()` function are assigned to the variable *i*. The function `range(2, 5)` ⑤ generates a sequence of numbers starting from 2 to 4 and the function `range(1, 6, 3)` generates ⑧ all the numbers starting from 1 up to 5 but the difference between each number is 2.

### **Program 3.15: Program to Iterate through Each Character in the String Using *for* Loop**

1. `for each_character in "Blue":`
2. `print(f"Iterate through character {each_character} in the string 'Blue'")`

**OUTPUT**

Iterate through character B in the string 'Blue'

Iterate through character l in the string 'Blue'

Iterate through character u in the string 'Blue'

Iterate through character e in the string 'Blue'

The iteration variable `each_character` is used to iterate through each character of the string "Blue" ① and each character is printed out in separate line ②.

### **Program 3.16: Write a Program to Find the Sum of All Odd and Even Numbers up to a Number Specified by the User.**

1. `number = int(input("Enter a number"))`
2. `even = 0`
3. `odd = 0`
4. `for i in range(number):`
5. `if i % 2 == 0:`
6. `even = even + i`
7. `else:`
8. `odd = odd + i`
9. `print(f"Sum of Even numbers are {even} and Odd numbers are {odd}"))`

**OUTPUT**

Enter a number 10

Sum of Even numbers are 20 and Odd numbers are 25

A range of numbers are generated using *range()* function ④. The numbers are segregated as odd or even by using the modulus operator ⑤. All the even numbers are added up and assigned to *even* variable and odd numbers are added up and assigned to *odd* variable ⑥–⑧ and print the result ⑨.

**Program 3.17: Write a Program to Find the Factorial of a Number**

```

1. number = int(input('Enter a number'))
2. factorial = 1
3. if number < 0:
4. print("Factorial doesn't exist for negative numbers")
5. elif number == 0:
6. print("The factorial of 0 is 1")
7. else:
8. for i in range(1, number + 1):
9. factorial = factorial * i
10. print(f"The factorial of number {number} is {factorial}")

```

**OUTPUT**

Enter a number 5

The factorial of number 5 is 120

The factorial of a non-negative integer  $n$  is denoted by  $n!$  which is the product of all positive integers less than or equal to  $n$  i.e.,  $n! = n * (n - 1) * (n - 2) * (n - 3) \dots 3 * 2 * 1$ . For example,

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

The value of  $0!$  is 1

Read a *number* from user ①. A value of 1 is assigned to variable *factorial* ②. To find the *factorial* of a number it has to be checked for a non-negative integer ③–④. If the user entered number is zero then the factorial is 1 ⑤–⑥. To generate numbers from 1 to the user entered number *range()* function is used. Every number is multiplied with the *factorial* variable and is assigned to the *factorial* variable itself inside the *for* loop ⑦–⑨. The *for* loop block is repeated for all the numbers starting from 1 up to the user entered number. Finally, the factorial value is printed ⑩.

---

**3.7 The *continue* and *break* Statements**

The *break* and *continue* statements provide greater control over the execution of code in a loop. Whenever the *break* statement is encountered, the execution control immediately jumps to the first instruction following the loop. To pass control to the next

iteration without exiting the loop, use the *continue* statement. Both *continue* and *break* statements can be used in *while* and *for* loops.

**Program 3.18: Program to Demonstrate Infinite *while* Loop and *break***

```
1. n = 0
2. while True:
3. print(f"The latest value of n is {n}")
4. n = n + 1
```

Here the *while* loop evaluates to *True* logical value always ② and it prints the latest value ③–④. This is an infinite loop with no end in sight. You need to press Ctrl + C to terminate this program. One way of ending this infinite loop is to use *break* statement along with *if* condition as shown in the following code.

```
1. n = 0
2. while True:
3. print(f"The latest value of n is {n}")
4. n = n + 1
5. if n > 5:
6. print(f"The value of n is greater than 5")
7. break
```

**OUTPUT**

```
The latest value of n is 0
The latest value of n is 1
The latest value of n is 2
The latest value of n is 3
The latest value of n is 4
The latest value of n is 5
The value of n is greater than 5
```

While this is an infinite loop ①–④, you can use this pattern to build useful loops as long as you explicitly add code to the body of the loop to ensure to exit from the loop using *break* statement upon satisfying a condition ⑤–⑦.

**Program 3.19: Write a Program to Check Whether a Number Is Prime or Not**

```
1. number = int(input('Enter a number > 1: '))
2. prime = True
3. for i in range(2, number):
4. if number % i == 0:
5. prime = False
6. break
```

7. if prime:
8.     print(f"{number} is a prime number")
9. else:
10.    print(f"{number} is not a prime number")

**OUTPUT**

Enter a number > 1: 7  
7 is a prime number

A prime number is a number which is divided by one and itself. For example, the number 7 is prime as it can be divided by 1 and itself, while number 10 can be divided by 2 and 5 other than 1 and itself, thus 10 can't be a prime number.

The user shall enter a value greater than one ①. Initially, the variable *prime* is assigned with *True* Boolean value ②. Use *range()* function to generate numbers starting from 2 up to number – 1, excluding the user entered *number* ③. The user entered *number* is checked using modulus operator ④ to determine whether the *number* is perfectly divisible by any number other than 1 and by itself. If it is divisible, then the variable *prime* is assigned *False* Boolean value ⑤ and we break out of the loop ⑥. But after completing the loop if *prime* remains *True* ⑦ then the user entered *number* is prime number ⑧ else ⑨ it is not a prime number ⑩.

**Program 3.20: Program to Demonstrate *continue* Statement**

1. n = 10
2. while n > 0:
3.     print(f"The current value of number is {n}")
4.     if n == 5:
5.         print(f"Breaking at {n}")
6.         n = 10
7.         continue
8.     n = n – 1

**OUTPUT**

The current value of number is 10  
The current value of number is 9  
The current value of number is 8  
The current value of number is 7  
The current value of number is 6  
The current value of number is 5  
Breaking at 5  
The current value of number is 10  
The current value of number is 9  
The current value of number is 8  
The current value of number is 7  
The current value of number is 6  
The current value of number is 5



In the above program, the while block is executed when the value of  $n$  is greater than zero ①–②. The value in the variable  $n$  is decremented ③ and printed in descending order ③ and when  $n$  becomes five ④–⑦ the control goes back to the beginning of the loop.

---

## 3.8 Catching Exceptions Using *try* and *except* Statement

There are at least two distinguishable kinds of errors:

1. Syntax Errors
2. Exceptions

### 3.8.1 Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python. For example,

1. while True
2.   print("Hello World)

#### OUTPUT

```
File "<ipython-input-3-c231969faf4f>", line 1
while True
 ^
SyntaxError: invalid syntax
```

In the output, the offending line is repeated and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected ①. The error is caused by a missing colon (‘:’). File name and line number are also printed so you know where to look in case the input came from a Python program file.

### 3.8.2 Exceptions

Exception handling is one of the most important feature of Python programming language that allows us to handle the errors caused by exceptions. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions.

An exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs in the program, execution gets terminated. In such cases, we get a system-generated error message. However, these exceptions can be handled in Python. By handling the exceptions, we can provide a meaningful message to the user about the issue rather than a system-generated message, which may not be understandable to the user.

Exceptions can be either built-in exceptions or user-defined exceptions. The interpreter or built-in functions can generate the built-in exceptions while user-defined exceptions are custom exceptions created by the user.

When the exceptions are not handled by programs it results in error messages as shown below.

1. `>>> 10 * (1/0)`  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
2. `>>> 4 + spam*3`  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
NameError: name 'spam' is not defined
3. `>>> '2' + 2`  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: Can't convert 'int' object to str implicitly

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are *ZeroDivisionError* ①, *NameError* ② and *TypeError* ③. The string printed as the exception type is the name of the *built-in exception* that occurred. The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback.

### 3.8.3 Exception Handling Using *try...except...finally*

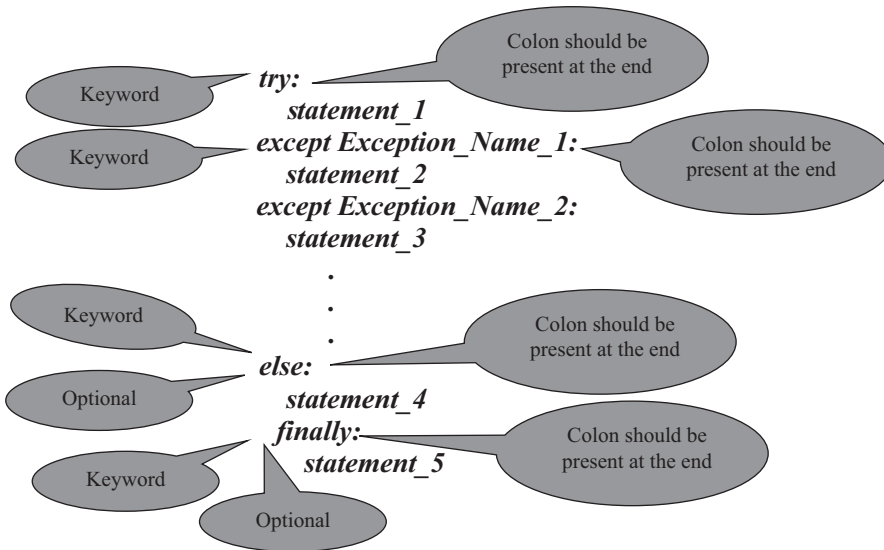
Handling of exception ensures that the flow of the program does not get interrupted when an exception occurs which is done by trapping run-time errors. Handling of exceptions results in the execution of all the statements in the program.



Run-time errors are those errors that occur during the execution of the program. These errors are not detected by the Python interpreter, because the code is syntactically correct.

It is possible to write programs to handle exceptions by using *try...except...finally* statements.

The syntax for *try...except...finally* is,



A *try* block consisting of one or more statements is used by Python programmers to partition code that might be affected by an exception. The associated *except* blocks are used to handle any resulting exceptions thrown in the *try* block. That is, you want the *try* block to succeed, and if it does not succeed, you want the control to pass to the *catch* block. If any statement within the *try* block throws an exception, control immediately shifts to the *catch* block. If no exception is thrown in the *try* block, the *catch* block is skipped.

There can be one or more *except* blocks. Multiple *except* blocks with different exception names can be chained together. The *except* blocks are evaluated from top to bottom in your code, but only one *except* block is executed for each exception that is thrown. The first *except* block that specifies the exact exception name of the thrown exception is executed. If no *except* block specifies a matching exception name then an *except* block that does not have an exception name is selected, if one is present in the code. Instead of having multiple *except* blocks with multiple exception names for different exceptions, you can combine multiple exception names together separated by a comma (also called parenthesized tuples) in a single *except* block. The syntax for combining multiple exception names in an *except* block is,

```
except (Exception_Name_1, Exception_Name_2, Exception_Name_3):
 statement(s)
```

where *Exception\_Name\_1*, *Exception\_Name\_2* and *Exception\_Name\_3* are different exception names.



You shall learn more about tuples in [Chapter 8](#).

The *try...except* statement has an optional *else* block, which, when present, must follow all *except* blocks. It is useful for code that must be executed if the *try* block does not raise an exception. The use of the *else* block is better than adding additional code to the *try* block because it avoids accidentally catching an exception that wasn't raised by the code being protected by the *try...except* statement.

The *try* statement has another optional block which is intended to define clean-up actions that must be executed under all circumstances. A *finally* block is always executed before leaving the *try* statement, whether an exception has occurred or not. When an exception has occurred in the *try* block and has not been handled by an *except* block, it is re-raised after the *finally* block has been executed. The *finally* clause is also executed "on the way out" when any other clause of the *try* statement is left via a *break*, *continue* or *return* statement.

You can also leave out the name of the exception after the *except* keyword. This is generally not recommended as the code will now be catching different types of exceptions and handling them in the same way. This is not optimal as you will be handling a *TypeError* exception the same way as you would have handled a *ZeroDivisionError* exception. When handling exceptions, it is better to be as specific as possible and only catch what you can handle.

### Program 3.21: Program to Check for *ValueError* Exception

```
1. while True:
2. try:
3. number = int(input("Please enter a number: "))
4. print(f"The number you have entered is {number}")
5. break
6. except ValueError:
7. print("Oops! That was no valid number. Try again...")
```

#### OUTPUT

```
Please enter a number: g
Oops! That was no valid number. Try again...
Please enter a number: 4
The number you have entered is 4
```

First, the *try* block (the statement(s) between the *try* and *except* keywords) is executed ②–⑤ inside the *while* loop ①. If no exception occurs, the *except* block is skipped and execution of the *try* statement is finished. If an exception occurs during execution of the *try* block statements, the rest of the statements in the *try* block is skipped. Then if its type matches the exception named after the *except* keyword, the *except* block is executed ⑥–⑦, and then execution continues after the *try* statement. When a variable receives an inappropriate value then it leads to *ValueError* exception.

### Program 3.22: Program to Check for *ZeroDivisionError* Exception

```
1. x = int(input("Enter value for x: "))
2. y = int(input("Enter value for y: "))
```

```

3. try:
4. result = x / y
5. except ZeroDivisionError:
6. print("Division by zero!")
7. else:
8. print(f"Result is {result}")
9. finally:
10. print("Executing finally clause")

```

**OUTPUT****Case 1**

```

Enter value for x: 8
Enter value for y: 0
Division by zero!
Executing finally clause

```

**Case 2**

```

Enter value for x: p
Enter value for y: q
Executing finally clause
ValueError Traceback (most recent call last)
<ipython-input-16-271d1f4e02e8> in <module>()
ValueError: invalid literal for int() with base 10: 'p'

```

In the above example divide by zero exception is handled. In line ① and ②, the user entered values are assigned to  $x$  and  $y$  variables. Line ④ is enclosed within line ③ which is a *try* clause. If the statements enclosed within the *try* clause raise an exception then the control is transferred to line ⑤ and divide by zero error is printed ⑥. As can be seen in Case 1, *ZeroDivisionError* occurs when the second argument of a division or modulo operation is zero. If no exception occurs, the *except* block is skipped and result is printed out ⑦–⑧. In Case 2, as you can see, the *finally* clause is executed in any event ⑨–⑩. The *ValueError* raised by dividing two strings is not handled by the *except* clause and therefore re-raised after the *finally* clause has been executed.

**Program 3.23: Write a Program Which Repeatedly Reads Numbers Until the User Enters 'done'. Once 'done' Is Entered, Print Out the Total, Count, and Average of the Numbers. If the User Enters Anything Other Than a Number, Detect Their Mistake Using *try* and *except* and Print an Error Message and Skip to the Next Number**

```

1. total = 0
2. count = 0
3. while True:
4. num = input("Enter a number: ")
5. if num == 'done':
6. print(f"Sum of all the entered numbers is {total}")
7. print(f"Count of total numbers entered {count}")
8. print(f"Average is {total / count}")

```

```
9. break
10. else:
11. try:
12. total += float(num)
13. except:
14. print("Invalid input")
15. continue
16. count += 1
```

**OUTPUT**

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: 5
Enter a number: done
Sum of all the entered numbers is 15.0
Count of total numbers entered 5
Average is 3.0
```

The program prompts the user ④ to enter a series of numbers until the user enters the word "done" ③. Assign zero to the variables *total* and *count* ①–②. Check whether the user has entered the word "done" or a numerical value ⑤. If it is other than the word "done" then the value entered by the user is added to the *total* variable ⑩–⑫. If the value entered is a value other than numerical value and other than "done" string value then an exception is raised ⑬–⑭ and the program continues ⑮ with the next iteration prompting the user to enter the next value. The *count* variable ⑯ keeps track of number of times the user has entered a value. If the user enters "done" string value ⑤ then calculate and display the average ⑥–⑧. At this stage break from the loop and stop the execution of the program ⑨.

---

### 3.9 Summary

- An if statement always starts with *if* clause. It can also have one or more *elif* clauses and a concluding *else* clause, but those clauses are optional.
- When an error occurs at run time, an exception is thrown and exceptions must be handled to end the programs gracefully.
- Python allows try-except and can have multiple except blocks for a single try block. Developers can create and raise their own exceptions.
- The while statement loops through a block of statements until the condition becomes false.
- The for statement can loop through a block of statements once for each item sequence.
- A break statement breaks out of a loop by jumping to the end of it.
- A continue statement continues a loop by jumping to the start of it.

---

## Multiple Choice Questions

1. \_\_\_\_\_ control statement repeatedly executes a set of statements.
  - a. Iterative
  - b. Conditional
  - c. Multi-way
  - d. All of these
2. Deduce the output of the following code.

```
if False and False:
 print("And Operation")
elif True or False:
 print("Or operation")
else:
 print("Default case")
```

  - a. And Operation
  - b. Or Operation
  - c. Default Case
  - d. B and C option
3. Predict the output of the following code.

```
i = 1
while True:
 if i%2 == 0:
 break
 print(i)
 i += 1
```

  - a. 1
  - b. 12
  - c. 123
  - d. None of these
4. Which keyword is used to take the control to the beginning of the loop?
  - a. exit
  - b. break
  - c. continue
  - d. None of these
5. The step argument in range() function \_\_\_\_\_.
  - a. indicates the beginning of the sequence
  - b. indicates the end of the sequence
  - c. indicates the difference between every two consecutive numbers in the sequence
  - d. generates numbers up to a specified value

6. The symbol that is placed at the end of if condition is
- a. ;
  - b. :
  - c. &
  - d. ~
7. What is the keyword that is used to come out of a loop only for that iteration?
- a. break
  - b. return
  - c. continue
  - d. if
8. Judge the output of the following code snippet.
- ```
for i in range(10):
    if i == 5:
        break
    else:
        print(i)
```
- a. 0 1 2 3 4
 - b. 0 1 2 3 4 5
 - c. 0 1 2 3
 - d. 1 2 3 4 5
9. Predict the output of the following code snippet.
- ```
while True:
 print(True)
 break
```
- a. True
  - b. False
  - c. None
  - d. Syntax error
10. The output of the below expression is
- ```
>>>10 * (1/0).
```
- a. OverflowError
 - b. ZeroDivisionError
 - c. NameError
 - d. TypeError
11. How many except statements can a try-except block have?
- a. Zero
 - b. One
 - c. More than one
 - d. More than zero

12. When will the else part of the try-except-else be executed?
 - a. Always
 - b. When an exception occurs
 - c. When no exception occurs
 - d. When an exception occurs in a try block
13. When is the finally block executed?
 - a. When an exception occurs
 - b. When there is no exception
 - c. Only if some condition that has been specified is satisfied
 - d. always
14. The keyword that is not used as an exception handling in Python?
 - a. try
 - b. except
 - c. accept
 - d. finally
15. An exception is
 - a. A object
 - b. A special function
 - c. A special module
 - d. A module
16. The set of statements that will be executed whether an exception is thrown or not?
 - a. except
 - b. else
 - c. finally
 - d. assert
17. Predict the output of the following code snippet.

```
while True
    print("Hello World")
```

 - a. Syntax Error
 - b. Logical Error
 - c. Run-time error
 - d. None of these
18. Gauge the output of the following statement?

```
int("65.43")
```

 - a. Import error
 - b. Value error
 - c. Type error
 - d. Name error

19. The error that is not a standard exception in Python.
 - a. Name Error
 - b. Assignment Error
 - c. IO Error
 - d. Value Error
 20. The function that generates a sequence of numbers which can be iterated through using *for* loop.
 - a. `input()`
 - b. `range()`
 - c. `list()`
 - d. `raw_input()`
 21. What is the output of the following code snippet?

```
x = 'abcd'
for i in x:
    print(i)
```

 - a. abcd
 - b. 0 1 2 3
 - c. iiii
 - d. Traceback
 22. The function of while loop is
 - a. Repeat a chunk of code a given number of times.
 - b. Repeat a chunk of code until a condition is true.
 - c. Repeat a chunk of code until a condition is false.
 - d. Repeat a chunk of code indefinitely.
-

Review Questions

1. Briefly explain the conditional statements available in Python.
2. Explain the syntax of for loop with an example.
3. What is the purpose of using break and continue?
4. Differentiate the syntax of *if...else* and *if...elif...else* with an example.
5. Explain the use of `range()` function with an example.
6. Why would you use a try-except statement in a program?
7. Explain the syntax of *while* loop with an example.
8. Differentiate between syntax error and an exception.
9. Explain the syntax of the try-except-finally block.

10. Write a program to read the Richter magnitude value from the user and display the result for the following conditions using *if...elif...else* statement.

Richter Magnitude	Information
> 1.0 and < 2.0	Microearthquakes not felt or rarely felt
> 2.0 and < 4.0	Very rarely causes damage
> 4.0 and < 5.0	Damage done to weak buildings
> 5.0 and < 6.0	Cause damage to the poorly constructed building
> 6.0 and < 7.0	Causes damage to well-built structures
> 7.0 and < 8.0	Causes damage to most buildings
> 8.0 and < 9.0	Causes major destruction
> 9.0	Causes unbelievable damage

11. Write a program to display Pascal's triangle.
 12. Write a program to display the following pattern using nested loops.

1	1
22	21
333	321
4444	4321
55555	54321

13. Write a program that uses a while loop to add up all the even numbers between 100 and 200.
 14. Write a program to print the sum of the following series
 a. $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$
 b. $\frac{1}{1} + \frac{2^2}{2} + \frac{3^3}{3} + \dots + \frac{n^n}{n}$
 15. Write a program to find the depreciation value of an asset(property) by reading the purchase value of the asset (amt), year of the service (year) and the value of depreciation.

4

Functions

AIM

Learn to define and invoke functions and comprehend the use of arguments and parameters to pass information to a function as well as return information from a function.

LEARNING OUTCOMES

At the end of this chapter, you are expected to

- Understand the purpose of functions in Python.
- Define and invoke functions.
- Receive the returned data from functions.
- Understand the use of modules and built-in functions in Python.
- Determine the scope of variables.
- Recognize different forms of function arguments.

Functions are one of the fundamental building blocks in Python programming language. Functions are used when you have a block of statements that needs to be executed multiple times within the program. Rather than writing the block of statements repeatedly to perform the action, you can use a function to perform that action. This block of statements are grouped together and is given a name which can be used to invoke it from other parts of the program. You write a function once but can execute it any number of times you like. Functions also reduce the size of the program by eliminating rudimentary code. Functions can be either Built-in Functions or User-defined functions.

4.1 Built-In Functions

The Python interpreter has a number of functions that are built into it and are always available. You have already looked at some of the built-in functions like *input()*, *print()*, *range()* and others in previous chapters. Let's discuss few more built-in functions ([TABLE 4.1](#)).

TABLE 4.1

A Few Built-in Functions in Python

Function Name	Syntax	Explanation
abs()	<i>abs(x)</i> where x is an integer or floating-point number.	The abs() function returns the absolute value of a number.
min()	<i>min(arg_1, arg_2, arg_3, ..., arg_n)</i> where arg_1, arg_2, arg_3 are the arguments.	The min() function returns the smallest of two or more arguments.
max()	<i>max(arg_1, arg_2, arg_3, ..., arg_n)</i> where arg_1, arg_2, arg_3 are the arguments.	The max() function returns the largest of two or more arguments.
divmod()	<i>divmod(a, b)</i> where a and b are numbers representing numerator and denominator.	The divmod() function takes two numbers as arguments and return a pair of numbers consisting of their quotient and remainder. For example, if a and b are integer values, then the result is the same as (a // b, a % b). If either a or b is a floating-point number, then the result is (q, a % b), where q is the whole number of the quotient.
pow()	<i>pow(x, y)</i> where x and y are numbers.	The pow(x, y) function returns x to the power y which is equivalent to using the power operator: x**y.
len()	<i>len(s)</i> where s may be a string, byte, list, tuple, range, dictionary or a set.	The len() function returns the length or the number of items in an object.

For example,

```

1. >>> abs(-3)
   3
2. >>> min(1, 2, 3, 4, 5)
   1
3. >>> max(4, 5, 6, 7, 8)
   8
4. >>> divmod(5, 2)
   (2, 1)
5. >>> divmod(8.5, 3)
   (2.0, 2.5)
6. >>> pow(3, 2)
   9
7. >>> len("Japan")
   5

```

Demonstration of various built-in functions ①–⑦.

4.2 Commonly Used Modules

Modules in Python are reusable libraries of code having *.py* extension, which implements a group of methods and statements. Python comes with many built-in modules as part of the standard library.

To use a module in your program, import the module using *import* statement. All the *import* statements are placed at the beginning of the program.

The syntax for import statement is,

 *import module_name*

For example, you can import the *math* module as

1. `>>>import math`

The *math* module is part of the Python standard library which provides access to various mathematical functions and is always available to the programmer ①.

The syntax for using a function defined in a module is,

module_name.function_name()

The module name and function name are separated by a dot.

Here we list some of the functions supported by *math* module.

1. `>>> import math`
2. `>>> print(math.ceil(5.4))`
6
3. `>>> print(math.sqrt(4))`
2.0
4. `>>> print(math.pi)`
3.141592653589793
5. `>>> print(math.cos(1))`
0.5403023058681398
6. `>>> math.factorial(6)`
720
7. `>>> math.pow(2, 3)`
8.0

Import the *math* module at the beginning ①. The *math.ceil(x)* function returns the ceiling of *x*, the smallest integer greater than or equal to the number ②, *math.sqrt(x)*, returns the square root of *x* ③, *math.pi* is the mathematical constant $\pi = 3.141592\dots$, to available precision ④, *math.cos(x)* returns the cosine of *x* radians ⑤, *math.factorial(x)* returns *x* factorial ⑥, *math.pow(x, y)* returns *x* raised to the power *y* ⑦.

The built-in function *dir()* returns a sorted list of comma separated strings containing the names of functions, classes and variables as defined in the module. For example, you can find all the functions supported by the *math* module by passing the module name as an argument to the *dir()* function.

```
1. >>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Various functions associated with *math* module is displayed ①

Another built-in function you will find useful is the *help()* function which invokes the built-in help system. The argument to the *help()* function is a string, which is looked up as the name of a module, function, class, method, keyword, or documentation topic, and then a related help page is printed in the console. For example, if you want to find information about *gcd()* function in *math* module then pass the function name as an argument without parenthesis.

```
1. >>> help(math.gcd)
Help on built-in function gcd in module math:
gcd(...)
gcd(x, y) -> int
greatest common divisor of x and y
```

Help page related to *math.gcd* is printed in the console ①.

Another useful module in the Python standard library is the *random* module which generates random numbers.

```
1. >>> import random
2. >>> print(random.random())
0.2551941897892144
3. >>> print(random.randint(5,10))
9
```

First, you need to import the *random* module to use any of its functions ①. The *random()* function generates a random floating-point number between 0 and 1 and it produces a different value each time it is called ②. The syntax for *random.randint()* function is *random.randint(start, stop)* which generates a integer number between start and stop argument numbers (including both) ③.

Third-party modules or libraries can be installed and managed using Python's package manager *pip*.

The syntax for *pip* is,

pip install module_name

Arrow is a popular Python library that offers a sensible, human-friendly approach to creating, manipulating, formatting and converting dates, times, and timestamps.

To install the *arrow* module, open a command prompt window and type the below command from any location.

1. C:\> pip install arrow

Install *arrow* module using *pip* command ①.

Below code shows a simple usage of arrow module.

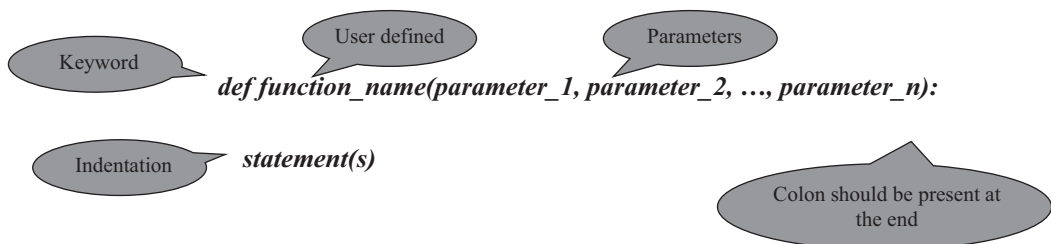
1. >>> import arrow
2. >>> a = arrow.utcnow()
3. >>> a.now()
<Arrow [2017-12-23T20:45:14.490380+05:30]>

UTC is the time standard commonly used across the world. The world's timing centers have agreed to keep their time scales closely synchronized—or coordinated—therefore the name Coordinated Universal Time ②. Current date including time is displayed using *now()* function ③.

4.3 Function Definition and Calling the Function

You can create your own functions and use them as and where it is needed. User-defined functions are reusable code blocks created by users to perform some specific task in the program.

The syntax for function definition is,



In Python, a function definition consists of the *def* keyword, followed by

1. The name of the function. The function's name has to adhere to the same naming rules as variables: use letters, numbers, or an underscore, but the name cannot start with a number. Also, you cannot use a keyword as a function name.
2. A list of parameters to the function are enclosed in parentheses and separated by commas. Some functions do not have any parameters at all while others may have one or more parameters.

3. A colon is required at the end of the function header. The first line of the function definition which includes the name of the function is called the function header.
4. Block of statements that define the body of the function start at the next line of the function header and they must have the same indentation level.

The *def* keyword introduces a function definition. The term parameter or formal parameter is often used to refer to the variables as found in the function definition.



The first statement among the block of statements within the function definition can optionally be a documentation string or docstring. There are tools which use docstrings to produce online documents or printed documentation automatically. Triple quotes are used to represent docstrings. For example,

```
""" This is single line docstring """
OR
""" This is
    multiline
    docstring """
```

Defining a function does not execute it. Defining a function simply names the function and specifies what to do when the function is called. Calling the function actually performs the specified actions with the indicated parameters.

The syntax for function call or calling function is,

function_name(argument_1, argument_2,...,argument_n)

Arguments are the actual value that is passed into the calling function. There must be a one to one correspondence between the formal parameters in the function definition and the actual arguments of the calling function. When a function is called, the formal parameters are temporarily “bound” to the arguments and their initial values are assigned through the calling function.

A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function. Normally, statements in the Python program are executed one after the other, in the order in which they are written. Function definitions do not alter the flow of execution of the program. When you call a function, the control flows from the calling function to the function definition. Once the block of statements in the function definition is executed, then the control flows back to the calling function and proceeds with the next statement. Python interpreter keeps track of the flow of control between different statements in the program.



When the control returns to the calling function from the function definition then the formal parameters and other variables in the function definition no longer contain any values.

Before executing the code in the source program, the Python interpreter automatically defines few special variables. If the Python interpreter is running the source program as a stand-alone main program, it sets the special built-in `__name__` variable to have a string value `"__main__"`. After setting up these special variables, the Python interpreter reads the program to execute the code found in it. All of the code that is at indentation level 0 gets executed. Block of statements in the function definition is not executed unless the function is called.

```
if __name__ == "__main__":  
    statement(s)
```

The special variable, `__name__` with `"__main__"`, is the entry point to your program. When Python interpreter reads the *if* statement and sees that `__name__` does equal to `"__main__"`, it will execute the block of statements present there.

Program 4.1: Program to Demonstrate a Function with and without Arguments

```
1. def function_definition_with_no_argument():  
2.     print("This is a function definition with NO Argument")  
  
3. def function_definition_with_one_argument(message):  
4.     print(f"This is a function definition with {message}")  
  
5. def main():  
6.     function_definition_with_no_argument()  
7.     function_definition_with_one_argument("One Argument")  
  
8. if __name__ == "__main__":  
9.     main()
```

OUTPUT

```
This is a function definition with NO Argument  
This is a function definition with One Argument
```

When you code Python programs, it is a best practice to put all the relevant necessary calling functions inside the `main()` function definition. Since the above program is a stand-alone main source program, Python interpreter assigns the string value `"__main__"` to the built-in special variable `__name__` which is checked for equality using *if* condition ⑧–⑨ and is also the entry point of your program. If the condition is *True* then the Python interpreter calls the `main()` function. In the `main()` ⑤ function definition there are two calling functions. You can have any number of function definitions and their *calling functions* in your program. When function at ⑥ is called without any arguments the control flows to function definition at ① and displays the statement ②. After executing the function definition at ① the control comes back to the place in the main function from where it had left and starts executing the next statement. At this stage, calling the function at ⑦ with one argument which is a string value is executed. The control moves over to the function definition at ③ which has one parameter and assigns the string value to the `message` parameter. The passed string value is displayed at ④.

Program 4.2: Program to Find the Area of Trapezium Using the Formula
Area = (1/2) * (a + b) * h Where a and b Are the 2 Bases
of Trapezium and h Is the Height

```
1. def area_trapezium(a, b, h):
2.     area = 0.5 * (a + b) * h
3.     print(f"Area of a Trapezium is {area}")

4. def main():
5.     area_trapezium(10, 15, 20)

6. if __name__ == "__main__":
7.     main()
```

OUTPUT

Area of a Trapezium is 250.0

Here the function definition *area_trapezium(a, b, h)* uses three formal parameters *a*, *b*, *h* ① to stand for the actual values passed by the user in the calling function *area_trapezium(10, 15, 20)* ⑤. The arguments in the calling function are numbers. The variables *a*, *b* and *h* are assigned with values of 10, 15 and 20 respectively. The area of a trapezium is calculated using the formula $0.5 * (a + b) * h$ and the result is assigned to the variable *area* ② and the same is displayed ③.

Program 4.3: Program to Demonstrate Using the Same Variable Name
in Calling Function and Function Definition

```
1. god_name = input("Who is the God of Seas according to Greek Mythology?")
2. def greek_mythology(god_name):
3.     print(f"The God of seas according to Greek Mythology is {god_name}")

4. def main():
5.     greek_mythology(god_name)

6. if __name__ == "__main__":
7.     main()
```

OUTPUT

Who is the God of Seas according to Greek Mythology? Poseidon
The God of seas according to Greek Mythology is Poseidon

The arguments passed by the calling program and the parameters used to receive the values in the function definition may have the same variable names. However, it is imperative to recognize that they are entirely independent variables as they exist in different scope. In the above code, the variable *god_name* appearing in the calling function ⑤ and in the function definition ② are different variables. The function *greek_mythology()* is a void function.



Scope refers to the visibility of variables. In other words, which parts of your program can see or use it.

4.4 The *return* Statement and *void* Function

Most of the times you may want the function to perform its specified task to calculate a value and return the value to the calling function so that it can be stored in a variable and used later. This can be achieved using the optional *return* statement in the function definition.

The syntax for *return* statement is,

Keyword

return [*expression_list*]

If an expression list is present, it is evaluated, else *None* is substituted. The *return* statement leaves the current function definition with the *expression_list* (or *None*) as a return value. The *return* statement terminates the execution of the function definition in which it appears and returns control to the calling function. It can also return an optional value to its calling function.

In Python, it is possible to define functions without a *return* statement. Functions like this are called **void functions**, and they return *None*.



Functions without a *return* statement do return a value, albeit a rather boring one. This value is called *None* (it is a built-in name) which stands for “nothing”. Writing the value *None* is normally suppressed by the interpreter if it would be the only value written.

If you want to return a value using *return* statement from the function definition, then you have to assign the result of the function to a variable. A function can return only a single value, but that value can be a list or tuple.



You shall learn more about lists and tuples in [Chapters 6](#) and [8](#) respectively.

In some cases, it makes sense to return multiple values if they are related to each other. If so, return the multiple values separated by a comma which by default is constructed as a tuple by Python. When calling function receives a tuple from the function definition, it is common to assign the result to multiple variables by specifying the same number of variables on the left-hand side of the assignment as there were returned from the function definition. This is called tuple unpacking.

Program 4.4: Program to Demonstrate the Return of Multiple Values from a Function Definition

```

1. def world_war():
2.     alliance_world_war = input("Which alliance won World War 2?")
3.     world_war_end_year = input("When did World War 2 end?")
4.     return alliance_world_war, world_war_end_year

5. def main():
6.     alliance, war_end_year = world_war()
7.     print(f"The war was won by {alliance} and the war ended in {war_end_year}")

8. if __name__ == "__main__":
9.     main()

```

OUTPUT

```

Which alliance won World War 2? Allies
When did World War 2 end? 1945
The war was won by Allies and the war ended in 1945

```

Line ⑧ is the entry point of the program and you call the function *main()* ⑨. In the function definition *main()* you call another function *world_war()* ⑥. The block of statements in function definition *world_war()* ① includes statements to get input from the user and a *return* statement ②–④. The *return* statement returns multiple values separated by a comma. Once the execution of function definition is completed, the values from function definition *world_war()* are returned to the calling function. At the calling function on the left-hand side of the assignment ⑥ there should be matching number of variables to store the values returned from the *return* statement. At ⑦ returned values are displayed.

Program 4.5: Calculate the Value of sin(x) up to n Terms Using the Series

$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$ Where x Is in Degrees

```

1. import math
2. degrees = int(input("Enter the degrees"))
3. nterms = int(input("Enter the number of terms"))
4. radians = degrees * math.pi / 180

5. def calculate_sin():

```

```

6.  result = 0
7.  numerator = radians
8.  denominator = 1
9.  for i in range(1, nterms+1):
10.     single_term = numerator / denominator
11.     result = result + single_term
12.     numerator = -numerator * radians * radians
13.     denominator = denominator * (2 * i) * (2 * i + 1)
14.  return result

15. def main():
16.     result = calculate_sin()
17.     print(f"value is sin(x) calculated using the series is {result} ")

18. if __name__ == "__main__":
19.     main()

```

OUTPUT

Enter the degrees45

Enter the number of terms5

value is sin(x) calculated using the series is 0.7071067829368671

All the trigonometric functions require their operands to be in radians. So, convert the degrees to radians by multiplying it by $\pi / 180$ ④. Next, a *for* loop ⑨ is required since we have to add and subtract each of these *single_term* ⑩. Let us take an index *i* for running the loop from 1 to *nterms* + 1. Let us get the first *single_term* $x / 1!$ where the variable *numerator* represents *x* and the variable *denominator* represents $1!$. Add the first *single_term* to the result which is initialized to 0 initially ⑪. Now we have to get the second *single_term*. The numerator for the second *single_term* can be obtained by multiplying $-x^2$ to previous value of *numerator* variable which was *x* ⑫ and denominator which is $3!$ can be got by multiplying the previous *denominator* variable by $(2 * i) * (2 * i + 1)$ where the value of *i* is 1 ⑬. So, the result will be $1 \times (2 \times 1) \times (2 \times 1 + 1) = 1 \times 2 \times 3 = 6$. Now, you have got the second *single_term* which is added to the result. Run the loop up to *nterms* + 1 to get the value of sin(*x*) which is stored in *result* variable and is returned to the *calculate_sin()* calling function to print the result.

Program 4.6: Program to Check If a 3 Digit Number Is Armstrong Number or Not

```

1. user_number = int(input("Enter a 3 digit positive number to check for Armstrong
   number"))

2. def check_armstrong_number(number):
3.     result = 0
4.     temp = number
5.     while temp != 0:
6.         last_digit = temp % 10

```

```

7.     result += pow(last_digit, 3)
8.     temp = int(temp / 10)
9.     if number == result:
10.        print(f"Entered number {number} is a Armstrong number")
11.    else:
12.        print(f"Entered number {number} is not a Armstrong number")

13. def main():
14.     check_armstrong_number(user_number)

15. if __name__ == "__main__":
16.     main()

```

OUTPUT

```

Enter a 3 digit positive number to check for Armstrong number407
Entered number 407 is a Armstrong number

```

A 3 digit number is called a Armstrong number if the sum of cube of its digits is equal to the number itself. For example, 407 is an Armstrong number since $4^3 + 0^3 + 7^3 = 407$. For a number with 'n' digits, the power value should be n, i.e., for a 5 digit number 12345, we should check for $1^5 + 2^5 + 3^5 + 4^5 + 5^5$. Read a 3 digit positive number from the user and store it in a variable called *number* ①. Initialize the variables *result* and *temp* to zero and number respectively ③–④. The original number value is kept intact and the *temp* variable which stores the original number is used to perform manipulations. Find the last digit of the number. To get the last digit of the number in *temp* variable use modulus division by 10 and assign it to *last_digit* variable ⑥. Find the cube of the last digit and add it to the *result* variable ⑦. Then remove the last digit from the number in *temp* variable by dividing *temp* by 10 and cast it as *int* ⑧. Repeat the logic in line ⑤–⑧ till the variable *temp* becomes 0 ④. Finally, you will be left with a number that is equal to the sum of the cubes of its own digits. Check the *result* number against the original *number* to determine whether the number is Armstrong number or not.

4.5 Scope and Lifetime of Variables

Python programs have two scopes: global and local. A variable is a global variable if its value is accessible and modifiable throughout your program. Global variables have a global scope. A variable that is defined inside a function definition is a local variable. The lifetime of a variable refers to the duration of its existence. The local variable is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function definition. Local variables inside a function definition have local scope and exist as long as the function is executing.

It is possible to access global variables from inside a function, as long as you have not defined a local variable with the same name. A local variable can have the same name as a global variable, but they are totally different so changing the value of the local

variable has no effect on the global variable. Only the local variable has meaning inside the function in which it is defined.

Program 4.7: Program to Demonstrate the Scope of Variables

```
1. test_variable = 5
2. def outer_function():
3.     test_variable = 60
4.     def inner_function():
5.         test_variable = 100
6.         print(f"Local variable value of {test_variable} having local scope to inner function is displayed")
7.     inner_function()
8.     print(f"Local variable value of {test_variable} having local scope to outer function is displayed ")
9. outer_function()
10. print(f"Global variable value of {test_variable} is displayed ")
```

OUTPUT

```
Local variable value of 100 having local scope to inner function is displayed
Local variable value of 60 having local scope to outer function is displayed
Global variable value of 5 is displayed
```

The variable name *test_variable* appears at different stages of the program. None of these variables are related to each other and are totally independent. In line ① the *test_variable* has global scope and is global variable as it is available throughout the program. A *test_variable* is created at line ③ is a local variable and having scope within the *outer_function()* ②. In line ⑤ another *test_variable* is created and is a local variable of *inner_function()* ④ having local scope and its life time is as far as *inner_function()* is executing. The change of values to *test_variable* in function definitions does not have any impact on the global *test_variable*. The value stored in the *test_number* variable is printed out as per its scope during the execution of the program ⑥, ⑧, ⑩.



It is not recommended to access global variables from inside the definition of the function. If there is a need by function to access an external value then it should be passed as a parameter to that function.

You can nest a function definition within another function definition. A nested function (inner function definition) can “inherit” the arguments and variables of its outer function definition. In other words, the inner function contains the scope of the outer

function. The inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function. The inner function definition can be invoked by calling it from within the outer function definition.

**Program 4.8: Calculate and Add the Surface Area of Two Cubes.
Use Nested Functions**

```
1. def add_cubes(a, b):
2.     def cube_surface_area(x):
3.         return 6 * pow(x, 2)
4.     return cube_surface_area(a) + cube_surface_area(b)

5. def main():
6.     result = add_cubes(2, 3)
7.     print(f"The surface area after adding two Cubes is {result}")

8. if __name__ == "__main__":
9.     main()
```

OUTPUT

The surface area after adding two Cubes is 78

The statement blocks in function definition *add_cubes(a, b)* ① has a nested function definition *cube_surface_area(x)* ②. The outer *add_cubes(a, b)* function definition has two parameters and inner function definition *cube_surface_area(x)* has one parameter which calculates the surface area of a cube ③. The parameters of *add_cubes(a, b)* function are passed as arguments to the calling function *cube_surface_area(x)*. The inner function is called within the outer function. The sum ④ of the surface area of two cubes is returned ⑥ and the result is printed out ⑦.

4.6 Default Parameters

In some situations, it might be useful to set a default value to the parameters of the function definition. This is where default parameters can help. Each default parameter has a default value as part of its function definition. Any calling function must provide arguments for all required parameters in the function definition but can omit arguments for default parameters. If no argument is sent for that parameter, the default value is used. Usually, the default parameters are defined at the end of the parameter list, after any required parameters and non-default parameters cannot follow default parameters. The default value is evaluated only once.

Program 4.9: Program to Demonstrate the Use of Default Parameters

```
1. def work_area(prompt, domain="Data Analytics"):
2.     print(f"{prompt} {domain}")

3. def main():
4.     work_area("Sam works in")
5.     work_area("Alice has interest in", "Internet of Things")

6. if __name__ == "__main__":
7.     main()
```

OUTPUT

```
Sam works in Data Analytics
Alice has interest in Internet of Things
```

There are two parameters in the function header for *work_area()* function definition. For the first parameter *prompt*, you have to specify the corresponding argument in the calling function. For the second parameter *domain*, a default value has been set ①. In the calling function, you may skip the second argument ④ as it is optional. In that case, the default value set for the parameter *domain* will be used in statements blocks of the function definition ②. If you specify the second argument in the calling function ⑤ then the default value assigned to the *domain* parameter will be overwritten with the latest value as specified in the second argument.

4.7 Keyword Arguments

Until now you have seen that whenever you call a function with some values as its arguments, these values get assigned to the parameters in the function definition according to their position. In the calling function, you can explicitly specify the argument name along with their value in the form *kwarg = value*. In the calling function, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the parameters in the function definition and their order is not important. No parameter in the function definition may receive a value more than once.

Program 4.10: Program to Demonstrate the Use of Keyword Arguments

```
1. def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
2.     print(f"This parrot wouldn't {action}, if you put {voltage}, volts through it.")
3.     print(f"Lovely plumage, the {type}")
4.     print(f"It's {state} !!!")
```

5. `parrot(1000)`
6. `parrot(voltage=1000)`
7. `parrot(voltage=1000000, action='VOOOOOM')`
8. `parrot('a thousand', state='pushing up the daisies')`

OUTPUT

This parrot wouldn't voom, if you put 1000, volts through it.
 Lovely plumage, the Norwegian Blue
 It's a stiff !!!

This parrot wouldn't voom, if you put 1000, volts through it.
 Lovely plumage, the Norwegian Blue
 It's a stiff !!!

This parrot wouldn't VOOOOOM, if you put 1000000, volts through it.
 Lovely plumage, the Norwegian Blue
 It's a stiff !!!

This parrot wouldn't voom, if you put a thousand, volts through it.
 Lovely plumage, the Norwegian Blue
 It's pushing up the daisies !!!

The `parrot()` function definition ① accepts one required parameter (voltage) and three optional parameters (state, action, and type). In line ⑤ one positional argument is specified. In line ⑥, keyword argument is used to pass a value to non-optional parameter. Two keyword arguments are specified in ⑦ and one positional and one keyword arguments are stated in ⑧.

Also, the following functional calls are invalid.

```
parrot()           # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220)   # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

4.8 *args and **kwargs

`*args` and `**kwargs` are mostly used as parameters in function definitions. `*args` and `**kwargs` allows you to pass a variable number of arguments to the calling function. Here variable number of arguments means that the user does not know in advance about how many arguments will be passed to the calling function. `*args` as parameter in function definition allows you to pass a non-keyworded, variable length tuple argument list to the calling function. `**kwargs` as parameter in function definition allows you to pass keyworded, variable length dictionary argument list to the calling function. `*args` must come after all the positional parameters and `**kwargs` must come right at the end.



It should be noted that the single asterisk (*) and double asterisk (**) are the important elements here and the words args and kwargs are used only by convention. The Python programming language does not enforce those words and the user is free to choose any words of his choice. Learned readers coming from C programming language should not mistake this asterisk for a pointer.

Program 4.11: Program to Demonstrate the Use of *args and **kwargs

```

1. def cheese_shop(kind, *args, **kwargs):
2.     print(f"Do you have any {kind} ?")
3.     print(f"I'm sorry, we're all out of {kind}")
4.     for arg in args:
5.         print(arg)
6.     print("-" * 40)
7.     for kw in kwargs:
8.         print(kw, ":", kwargs[kw])
9.
10. def main():
11.     cheese_shop("Limburger", "It's very runny, sir.",
12.                 "It's really very, VERY runny, sir.",
13.                 shop_keeper="Michael Palin",
14.                 client="John Cleese",
15.                 sketch="Cheese Shop Sketch")
16.
17. if __name__ == "__main__":
18.     main()

```

OUTPUT

```

Do you have any Limburger ?
I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shop_keeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch

```

The function definition `cheese_shop()` has three parameters where the first parameter is a positional parameter and the last two parameters are `*args` and `**kwargs` ①. In the

statement blocks of the function definition `*` and `**` are not used with args and kwargs. In the function call ⑩ the first argument is assigned to the first parameter in the function definition. After the first argument, the remaining series of arguments separated by commas are assigned to `*args` until the keyword arguments are encountered. All the keyword arguments in the calling function are assigned to `**kwargs`. You can work with `**kwargs` just like you work with dictionaries. The order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the calling function. Loops are used to display the items in args and kwargs ④–⑧.

4.9 Command Line Arguments

A Python program can accept any number of arguments from the command line. Command line arguments is a methodology in which user will give inputs to the program through the console using commands. You need to import `sys` module to access command line arguments. All the command line arguments in Python can be printed as a list of string by executing `sys.argv`.

Program 4.12: Program to Demonstrate Command Line Arguments in Python

```
1. import sys
2. def main():
3.     print(f"sys.argv prints all the arguments at the command line including file
      name {sys.argv}")
4.     print(f"len(sys.argv) prints the total number of command line arguments includ-
      ing file name {len(sys.argv)}")
5.     print("You can use for loop to traverse through sys.argv")
6.     for arg in sys.argv:
7.         print(arg)
8. if __name__ == "__main__":
9.     main()
```

OUTPUT

```
C:\Introduction_To_Python_Programming\Chapter_4>python Program_4.12.py arg_1
arg_2 arg_3
sys.argv prints all the arguments at the command line including file name ['Program_4.12.
py', 'arg_1', 'arg_2', 'arg_3']
len(sys.argv) prints the total number of command line arguments including file name 4
You can use for loop to traverse through sys.argv
Program_4.12.py
arg_1
arg_2
arg_3
```

To execute a command line argument program, you need to navigate to the directory where your program is saved. Then issue a command in the format *python file_name argument_1 argument_2 argument_3 argument_n*. Here *argument_1 argument_2* and so on can be any argument and should be separated by a space. Import *sys* module ① to access command line arguments. Print all the arguments including file name using *sys.argv* ③ but excluding the *python* command. The number of arguments passed at the command line can be obtained by *len(sys.argv)* ④. A *for* loop can be used to traverse through each of the arguments in *sys.argv* ⑤–⑦.

4.10 Summary

- Making programs modular and reusable is one of the fundamental goals of any programming language and functions help to achieve this goal.
 - A function is a code snippet that performs some task and can be called from another part of the code.
 - There are many built-in functions provided by Python such as *min()*, *pow()* and others and users can also create their own functions which are called as user-defined functions.
 - A function header begins with the *def* keyword followed by function's name and parameters and ends with a colon.
 - A function is called a void function if it does not return any value.
 - A global variable is a variable that is defined outside of any function definition and a local variable is a variable that is only accessible from within the function it resides.
 - Docstrings serve the same purpose as that of comments.
 - The syntax **args* allows to pass a variable number of arguments to the calling function.
 - The syntax ***kwargs* allows you to pass keyworded, variable length dictionary arguments to the calling function.
 - Command-line arguments in Python show up in *sys.argv* as a list of strings.
-

Multiple Choice Questions

1. A local variable in Python is a variable that is,
 - a. Defined inside every function
 - b. Local to the given program
 - c. Accessible from within the function
 - d. All of these
2. Which of the following statements are the advantages of using functions?
 - a. Reduce duplication of code
 - b. Clarity of code
 - c. Reuse of code
 - d. All of these

3. The keyword that is used to define the block of statements in function?
 - a. function
 - b. func
 - c. def
 - d. pi
4. The characteristics of docstrings are
 - a. suitable way of using documentation
 - b. Function should have a docstring
 - c. Can be accessed by `__doc__`
 - d. All of these
5. The two types of functions used in Python are
 - a. Built-in and user-defined
 - b. Custom function and user function
 - c. User function and system call
 - d. System function
6. _____ refers to built-in mathematical function.
 - a. sqrt
 - b. rhombus
 - c. add
 - d. sub
7. The variable defined outside the function is referred as
 - a. static
 - b. global
 - c. automatic
 - d. register
8. Functions without a return statement do return a value and it is
 - a. int
 - b. null
 - c. None
 - d. error
9. The data type of the elements in `sys.argv`?
 - a. set
 - b. list
 - c. tuple
 - d. string
10. The length of `sys.argv` is?
 - a. Total number of arguments excluding the filename
 - b. Total number of arguments including the filename
 - c. Only filename
 - d. Total number of arguments including Python Command

11. The syntax of keyword arguments specified in the function header?
 - a. * followed by an identifier
 - b. _ followed by an identifier
 - c. ** followed by an identifier
 - d. __ followed by an identifier
12. The number of arguments that can be passed to a function is
 - a. 0
 - b. 1
 - c. 0 or more
 - d. 1 or more
13. The library that is used to create, manipulate, format and convert dates, times and timestamps in Python is
 - a. Arrow
 - b. Pandas
 - c. Scipy
 - d. NumPy
14. The command line arguments is stored in
 - a. os.argv
 - b. sys.argv
 - c. argv
 - d. None
15. The command that is used to install a third-party module in Python is
 - a. pip
 - b. pipe
 - c. install_module
 - d. pypy
16. Judge the output of the following code.

```
import math
math.sqrt(36)
```

 - a. Error
 - b. -6
 - c. 6
 - d. 6.0
17. The function divmod(10,20) is evaluated as
 - a. (10%20,10//20)
 - b. (10//20,10%20)
 - c. (10//20,10*20)
 - d. (10/20,10%20)

18. Predict the output of the following code?

```
def tweet():  
    print("Python Programming!")  
tweet()
```

- a. Python Programming!
- b. Indentation Error
- c. Syntax Error
- d. Name Error

19. The output of the following code is

```
def displaymessage(message, times = 1):  
    print(message * times)  
    displaymessage("Data")  
    displaymessage("Science", 5)
```

- a. Data Science Science Science Science Science
- b. Data Science 5
- c. DataDataDataDataDataScience
- d. DataDataDataDataDataData

20. Guess the output of the following code

```
def quad(x):  
    return x * x * x * x  
  
x = quad(3)  
print(x)
```

- a. 27
- b. 9
- c. 3
- d. 81

21. The output of the following code is

```
def add(*args):  
    x = 0  
    for i in args:  
        x += i  
    return x  
  
print(add(1, 2, 3))  
print(add(1, 2, 3, 4, 5))
```

- a. 16 15
- b. 6 15
- c. 1 2 3
- d. 1 2 3 45

22. Gauge the output of the following code.
- ```
def foo():
 return total + 1
total = 0
print(foo())
```
- a. 1
  - b. 0
  - c. 11
  - d. 00
23. The default arguments specified in the function header is an
- a. Identifier followed by an = and the default value
  - b. Identifier followed by the default value within back-ticks
  - c. Identifier followed by the default value within []
  - d. Identifier followed by an #.
- 

## Review Questions

1. Define function. What are the advantages of using a function?
2. Differentiate between user-defined function and built-in functions.
3. Explain with syntax how to create a user-defined functions and how to call the user-defined function from the main function.
4. Explain the built-in functions with examples in Python.
5. Differentiate between local and global variables with suitable examples.
6. Explain the advantages of \*args and \*\*kwargs with examples.
7. Demonstrate how functions return multiple values with an example.
8. Explain the utility of docstrings?
9. Write a program using functions to perform the arithmetic operations.
10. Write a program to find the largest of three numbers using functions.
11. Write a Python program using functions to find the value of  ${}^nP_r$  and  ${}^nC_r$ .
12. Write a Python function named area that finds the area of a pentagon.
13. Write a program using functions to display Pascal's triangle.
14. Write a program using functions to print harmonic progression series and its sum till N terms.
15. Write a program using functions to do the following tasks:
  - a. Convert milliseconds to hours, minutes and seconds.
  - b. Compute a sales commission, given the sales amount and the commission rate.
  - c. Convert Celsius to Fahrenheit.
  - d. Compute the monthly payment, given the loan amount, number of years and the annual interest rate.



**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 5

---

## Strings

---

### AIM

Use indexing and built-in string methods to manipulate and process the string values.

### LEARNING OUTCOMES

At the end of the chapter, you are expected to

- Access individual characters in a string and apply basic string operations.
- Search and retrieve a substring from a string.
- Use string methods to manipulate strings.

String usage abounds in just about all types of applications. A string consists of a sequence of characters, which includes letters, numbers, punctuation marks and spaces. To represent strings, you can use a single quote, double quotes or triple quotes.

---

### 5.1 Creating and Storing Strings

Strings are another basic data type available in Python. They consist of one or more characters surrounded by matching quotation marks. For example,

```
1. >>> single_quote = 'This is a single message'
2. >>> double_quote = "Hey it is my book"
3. >>> single_char_string = "A"
4. >>> empty_string = ""
5. >>> empty_string = "
6. >>> single_within_double_quote = "Opportunities don't happen. You create
 them."
7. >>> double_within_single_quote = "Why did she call the man 'smart'?"
8. >>> same_quotes = 'I\'ve an idea'
9. >>> triple_quote_string = """This
 ... is
 ... triple
 ... quote"""
```

```
10. >>> triple_quote_string
 'This\nis\ntriple\nquote'
11. >>> type(single_quote)
 <class 'str'>
```

Strings are surrounded by either single ① or double quotation marks ②. To store a string inside a variable, you just need to assign a string to a variable. In the above code, all the variables on the left side of the assignment operator are string variables. A single character is also treated as string ③. A string does not need to have any characters in it. Both "" ④ and "" ⑤ are valid strings, called empty strings. A string enclosed in double quotation marks can contain single quotation marks ⑥. Likewise, a string enclosed in single quotation marks can contain double quotation marks ⑦. So basically, if one type of quotation mark surrounds the string, you have to use the other type within it. If you want to include the same quotation marks within a string as you have used to enclose the string, then you need to preface the inner quote with a backslash ⑧. If you have a string spanning multiple lines, then it can be included within triple quotes ⑨. All white spaces and newlines used inside the triple quotes are literally reflected in the string ⑩. You can find the type of a variable by passing it as an argument to *type()* function. Python strings are of *str* data type ⑪.

### 5.1.1 The *str()* Function

The *str()* function returns a string which is considered an informal or nicely printable representation of the given object. The syntax for *str()* function is,

***str(object)***

It returns a string version of the object. If the object is not provided, then it returns an empty string.

```
1. >>> str(10)
 '10'
2. >>> create_string = str()
3. >>> type(create_string)
 <class 'str'>
```

Here integer type is converted to string type. Notice the single quotes to represent the string ①. The *create\_string* is an empty string ② of type *str* ③.

---

## 5.2 Basic String Operations

In Python, strings can also be concatenated using + sign and \* operator is used to create a repeated sequence of strings.

```
1. >>> string_1 = "face"
2. >>> string_2 = "book"
3. >>> concatenated_string = string_1 + string_2
4. >>> concatenated_string
 'facebook'
```

```

5. >>> concatenated_string_with_space = "Hi " + "There"
6. >>> concatenated_string_with_space
 'Hi There'
7. >>> singer = 50 + "cent"
 Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 TypeError: unsupported operand type(s) for +: 'int' and 'str'
8. >>> singer = str(50) + "cent"
9. >>> singer
 '50cent'
10. >>> repetition_of_string = "wow" * 5
11. >>> repetition_of_string
 'wowwowwowwowwow'

```

Two string variables are assigned with "face"<sup>①</sup> and "book"<sup>②</sup> string values. The *string\_1* and *string\_2* are concatenated using + operator to form a new string. The new string *concatenated\_string*<sup>④</sup> has the values of both the strings<sup>③</sup>. As you can see in the output, there is no space between the two concatenated string values. If you need whitespace between concatenated strings<sup>⑥</sup>, all you need to do is include whitespace within a string like in<sup>⑤</sup>. You cannot use the + operator to concatenate values of two different types. For example, you cannot concatenate string data type with integer data type<sup>⑦</sup>. You need to convert integer type to string type and then concatenate the values<sup>⑧–⑨</sup>. You can use the multiplication operator \* on a string<sup>⑩</sup>. It repeats the string the number of times you specify and the string value "wow" is repeated five times<sup>⑪</sup>.



Python cannot concatenate string value with integer value since they are of different data types. You need to convert integer type to string type before concatenating integer and string values.

You can check for the presence of a string in another string using *in* and *not in* membership operators. It returns either a Boolean *True* or *False*. The *in* operator evaluates to *True* if the string value in the left operand appears in the sequence of characters of string value in right operand. The *not in* operator evaluates to *True* if the string value in the left operand does not appear in the sequence of characters of string value in right operand.

```

1. >>> fruit_string = "apple is a fruit"
2. >>> fruit_sub_string = "apple"
3. >>> fruit_sub_string in fruit_string
 True
4. >>> another_fruit_string = "orange"
5. >>> another_fruit_string not in fruit_string
 True

```

Statement ③ returns *True* because the string "apple" is present in the string "apple is a fruit". The *not in* operator evaluates to *True* as the string "orange" is not present in "apple is a fruit" string ⑤.

### 5.2.1 String Comparison

You can use (>, <, <=, >=, ==, !=) to compare two strings resulting in either Boolean *True* or *False* value. Python compares strings using ASCII value of the characters. For example,

```
1. >>> "january" == "jane"
 False
2. >>> "january" != "jane"
 True
3. >>> "january" < "jane"
 False
4. >>> "january" > "jane"
 True
5. >>> "january" <= "jane"
 False
6. >>> "january" >= "jane"
 True
7. >>> "filled" > ""
 True
```

Strings can be compared using various comparison operators ①–⑦. String equality is compared using == (double equal sign). String inequality is compared using != sign. Suppose you have string\_1 as "january" and string\_2 as "jane". The first two characters from string\_1 and string\_2 (j and j) are compared. As they are equal, the second two characters are compared (a and a). Because they are also equal, the third two characters (n and n) are compared. Since the third characters are also equal, the fourth character from both the strings are compared and because 'u' has greater ASCII value than 'e', string\_1 is greater than string\_2. You can even compare a string against an empty string.

### 5.2.2 Built-In Functions Used on Strings

There are many built-in functions for which a string can be passed as an argument (TABLE 5.1).

**TABLE 5.1**

Built-In Functions Used on Strings

| Built-In Functions | Description                                                                                                             |
|--------------------|-------------------------------------------------------------------------------------------------------------------------|
| len()              | The <i>len()</i> function calculates the number of characters in a string. The white space characters are also counted. |
| max()              | The <i>max()</i> function returns a character having highest ASCII value.                                               |
| min()              | The <i>min()</i> function returns character having lowest ASCII value.                                                  |

For example,

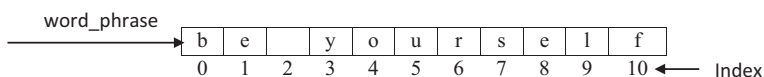
1. >>> count\_characters = len("eskimos")
2. >>> count\_characters  
7
3. >>> max("axel")  
'x'
4. >>> min("brad")  
'a'

The number of characters in the string "eskimos" ① is calculated using *len()* function ②. Characters with highest and lowest ASCII value are calculated using *max()* ③ and *min()* ④ functions.

### 5.3 Accessing Characters in String by Index Number

Each character in the string occupies a position in the string. Each of the string's character corresponds to an index number. The first character is at index 0; the next character is at index 1, and so on. The length of a string is the number of characters in it. You can access each character in a string using a subscript operator i.e., a square bracket. Square brackets are used to perform indexing in a string to get the value at a specific index or position. This is also called *subscript* operator.

The index breakdown for the string "be yourself" assigned to *word\_phrase* string variable is shown below.



The syntax for accessing an individual character in a string is as shown below.

*string\_name[index]*

where *index* is usually in the range of 0 to one less than the length of the string. The value of index should always be an integer and indicates the character to be accessed.

For example,

1. >>> word\_phrase = "be yourself"
2. >>> word\_phrase[0]  
'b'
3. >>> word\_phrase[1]  
'e'
4. >>> word\_phrase[2]  
,
5. >>> word\_phrase[3]  
'y'



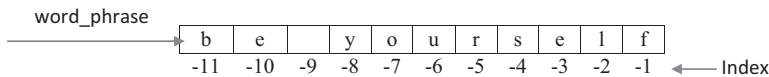
```

6. >>> word_phrase[10]
 'f'
7. >>> word_phrase[11]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
IndexError: string index out of range

```

By referring to the index numbers in square bracket, you can access individual characters in a string. The index number starts with zero corresponding to the first character in the string ②. The index number increases by one as we move to access the next letter to the right of the current letter ③–⑥. The whitespace character between *be* and *yourself* has its own index number, i.e., 2. The last character in the string is referenced by an index value which is the (size of the string – 1) or (len(string) – 1). If you try to specify an index number more than the number of characters in the string, then it results in *IndexError: string index out of range* error ⑦.

You can also access individual characters in a string using negative indexing. If you have a long string and want to access end characters in the string, then you can count backward from the end of the string starting from an index number of –1. The negative index breakdown for the string “*be yourself*” assigned to *word\_phrase* string variable is shown below.



```

1. >>> word_phrase[-1]
 'f'
2. >>> word_phrase[-2]
 'l'

```

By using negative index number of –1, you can print the character ‘f’ ①, the negative index number of –2 prints the character ‘l’ ②. You can benefit from using negative indexing when you want to access characters at the end of a long string.

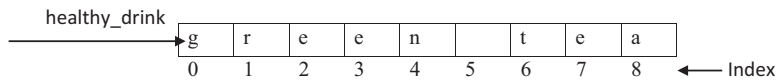
## 5.4 String Slicing and Joining

The "slice" syntax is a handy way to refer to sub-parts of sequence of characters within an original string. The syntax for string slicing is,

*string\_name[start:end[:step]]*

Colon is used to specify range values

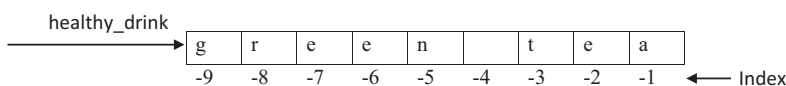
With string slicing, you can access a sequence of characters by specifying a range of index numbers separated by a colon. String slicing returns a sequence of characters beginning at *start* and extending up to but not including *end*. The *start* and *end* indexing values have to be integers. String slicing can be done using either positive or negative indexing.



1. `>>> healthy_drink = "green tea"`
2. `>>> healthy_drink[0:3]`  
'gre'
3. `>>> healthy_drink[:5]`  
'green'
4. `>>> healthy_drink[6:]`  
'tea'
5. `>>> healthy_drink[:]`  
'green tea'
6. `>>> healthy_drink[4:4]`  
''
7. `>>> healthy_drink[6:20]`  
'tea'

A substring is created when slicing the strings, which is basically a string that already exists within another string. A substring is any sequence of characters that is contained in a string. The string "green tea" is assigned to *healthy\_drink* variable ① and a sequence of characters or a substring is extracted from the beginning (0th Index) up to the third character (2nd Index) ②. In string slicing, *start* index value (including) is where slicing starts and *end* index value (excluding) is where the slicing ends. If the *start* index is omitted, the slicing starts from the first index number (0th Index) up to the *end* index (excluding) in the string. In ③ substring starting from 0th index to 4th index is printed. If the *end* index is omitted, slicing starts from the *start* index and goes up to the end of the string index. Substring starting from 6th index to the end of the string is displayed in ④. If both the *start* and *end* index values are omitted then the entire string is displayed ⑤. If the *start* index is equal to or higher than the *end* index, then it results in an empty string ⑥. If the *end* index number is beyond the end of the string, it stops at the end of the string ⑦.

Slicing can also be done using the negative integer numbers.



The negative index can be used to access individual characters in a string. Negative indexing starts with -1 index corresponding to the last character in the string and then the index decreases by one as we move to the left.

1. >>> healthy\_drink[-3:-1]  
'te'
2. >>> healthy\_drink[6:-1]  
'te'

You need to specify the lowest negative integer number in the start index position when using negative index numbers as it occurs earlier in the string ①. You can also combine positive and negative indexing numbers ②.

#### 5.4.1 Specifying Steps in Slice Operation

In the slice operation, a third argument called *step* which is an optional can be specified along with the *start* and *end* index numbers. This *step* refers to the number of characters that can be skipped after the *start* indexing character in the string. The default value of *step* is one. In the previous slicing examples, *step* is not specified and in its absence, a default value of one is used. For example,

1. >>> newspaper = "new york times"
2. >>> newspaper[0:12:4]  
'ny'
3. >>> newspaper[::4]  
'ny e'

In ② the slice [0:12:4] takes the character at 0th index which is *n* and will print every 4th character in the string extending till the 12th character (excluding). You can omit both *start* and *end* index values to consider the entire range of characters in the string by specifying two colons while considering the step argument which will specify the number of characters to skip ③.

#### Program 5.1: Write Python Code to Determine Whether the Given String Is a Palindrome or Not Using Slicing

1. def main():
2. user\_string = input("Enter string: ")
3. if user\_string == user\_string[::-1]:
4. print(f"User entered string is palindrome")
5. else:
6. print(f"User entered string is not a palindrome")
7. if \_\_name\_\_ == "\_\_main\_\_":
8. main()

#### OUTPUT

Case 1:

Enter string: madam

User entered string is palindrome

Case 2:

Enter string: cat

User entered string is not a palindrome

User entered string value is stored in *user\_string* variable ②. The *user\_string* is reversed using string slicing with  $-1$  ③ and is compared with non-reversed *user\_string*. If both are equal, then the string is palindrome ④ else it is not a palindrome ⑥.

#### 5.4.2 Joining Strings Using *join()* Method

Strings can be joined with the *join()* string. The *join()* method provides a flexible way to concatenate strings. The syntax of *join()* method is,

*string\_name.join(sequence)*

Here sequence can be string or list. If the sequence is a string, then *join()* function inserts *string\_name* between each character of the string sequence and returns the concatenated string. If the sequence is a list, then *join()* function inserts *string\_name* between each item of list sequence and returns the concatenated string. It should be noted that all the items in the list should be of string type.

1. >>> date\_of\_birth = ["17", "09", "1950"]
2. >>> ":".join(date\_of\_birth)  
'17:09:1950'
3. >>> social\_app = ["instagram", "is", "an", "photo", "sharing", "application"]
4. >>> " ".join(social\_app)  
'instagram is an photo sharing application'
5. >>> numbers = "123"
6. >>> characters = "amy"
7. >>> password = numbers.join(characters)
8. >>> password  
'a123m123y'

All the items in the list *date\_of\_birth* list variable is of string type ①. In ② the string ":" is inserted between each list item and the concatenated string is displayed. In *social\_app* list variable, all the list items are of string type ③. In ④, the *join()* method ensures a blank space is inserted between each of the list items in *social\_app* and the concatenated string is displayed. The variables *numbers* ⑤ and *characters* ⑥ are of string type. The string value of "123" is placed between each character of "amy" string resulting in 'a123m123y'. The string value of "123" is inserted between *a* and *m* and again between *m* and *y* and is assigned to *password* string variable.

#### 5.4.3 Split Strings Using *split()* Method

The *split()* method returns a list of string items by breaking up the string using the delimiter string. The syntax of *split()* method is,

`string_name.split([separator [, maxsplit]])`

Here *separator* is the delimiter string and is optional. A given string is split into list of strings based on the specified *separator*. If the *separator* is not specified then whitespace is considered as the delimiter string to separate the strings. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit* + 1 items). If *maxsplit* is not specified or -1, then there is no limit on the number of splits.

```
1. >>> inventors = "edison, tesla, marconi, newton"
2. >>> inventors.split(",")
 ['edison', ' tesla', ' marconi', ' newton']
3. >>> watches = "rolex hublot cartier omega"
4. >>> watches.split()
 ['rolex', 'hublot', 'cartier', 'omega']
```

The value in *inventors* string variable ① is separated based on "," (comma) ② separator. In ④ no separator is specified in *split()* method. Hence, the string variable *watches* ③ is separated based on whitespace.

#### 5.4.4 Strings Are Immutable

As strings are immutable, it cannot be modified. The characters in a string cannot be changed once a string value is assigned to string variable. However, you can assign different string values to the same string variable.

```
1. >>> immutable = "dollar"
2. >>> immutable[0] = "c"
 Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 TypeError: 'str' object does not support item assignment
3. >>> string_immutable = "c" + immutable[1:]
4. >>> string_immutable
 'collar'
5. >>> immutable = "rollar"
6. >>> immutable
 'rollar'
```

The string value "dollar" is assigned to string variable *immutable* ①. If you try to change the string by assigning a character in place of existing character through indexing, then it results in an error as the string is immutable ②. You cannot change the string value once it has been assigned to a string variable. What you can do is create a new string that is a variation of the original string ③–④. Also, assigning a whole new string value to existing string variable is allowed ⑤–⑥.

#### 5.4.5 String Traversing

Since the string is a sequence of characters, each of these characters can be traversed using the *for* loop.

**Program 5.2: Program to Demonstrate String Traversing Using the *for* Loop**

```
1. def main():
2. alphabet = "google"
3. index = 0
4. print(f"In the string '{alphabet}'")
5. for each_character in alphabet:
6. print(f"Character '{each_character}' has an index value of {index}")
7. index += 1
8. if __name__ == "__main__":
9. main()
```

**OUTPUT**

```
In the string 'google'
Character 'g' has an index value of 0
Character 'o' has an index value of 1
Character 'o' has an index value of 2
Character 'g' has an index value of 3
Character 'l' has an index value of 4
Character 'e' has an index value of 5
```

String value "google" is assigned to *alphabet* ② string variable and *index* variable is initialized to zero ③. The *for* statement makes it easy to loop over each character in a string. For each character traversed in the string ⑤ the *index* value is incremented by a value of one ⑦. Each character in the string and its corresponding *index* value is printed in ⑥.

**Program 5.3: Program to Print the Characters Which Are Common in Two Strings**

```
1. def common_characters(string_1, string_2):
2. for letter in string_1:
3. if letter in string_2:
4. print(f"Character '{letter}' is found in both the strings")
5. def main():
6. common_characters('rose', 'goose')
7. if __name__ == "__main__":
8. main()
```

**OUTPUT**

```
Character 'o' is found in both the strings
Character 's' is found in both the strings
Character 'e' is found in both the strings
```

Two string values, 'rose' and 'goose', are passed as arguments to *common\_characters()* function ⑥. The string values 'rose' and 'goose' are assigned to *string\_1* and *string\_2* parameter

variables ①. A *for* loop is used to traverse through each letter in the string value assigned to *string\_1* parameter variable ②. If any of the *letter* is found in *string\_2* parameter variable ③ then that character is printed out ④.

**Program 5.4: Write Python Program to Count the Total Number of Vowels, Consonants and Blanks in a String**

```

1. def main():
2. user_string = input("Enter a string: ")
3. vowels = 0
4. consonants = 0
5. blanks = 0
6. for each_character in user_string:
7. if(each_character == 'a' or each_character == 'e' or each_character == 'i' or
 each_character == 'o' or each_character == 'u'):
8. vowels += 1
9. elif "a" < each_character < "z":
10. consonants += 1
11. elif each_character == " ":
12. blanks += 1
13. print(f"Total number of Vowels in user entered string is {vowels}")
14. print(f"Total number of Consonants in user entered string is {consonants}")
15. print(f"Total number of Blanks in user entered string is {blanks}")
16. if __name__ == "__main__":
17. main()

```

**OUTPUT**

```

Enter a string: may god bless you
Total number of Vowels in user entered string is 5
Total number of Consonants in user entered string is 9
Total number of Blanks in user entered string is 3

```

User entered string is assigned to a *user\_string* ② variable. Initially, zero is assigned to vowels, consonants and blanks variables ③–⑤. Each of the characters in *user\_string* is traversed using *for* loop ⑥. If the character traversed in user entered string matches with any of the vowels then the *vowels* variable is incremented by one ⑦–⑧. If the character traversed is not a vowel then it is treated as consonant and the *consonants* variable is incremented by one ⑨–⑩. Also, characters are checked for " " (blank space) ⑪. If yes, then the *blanks* variable is incremented by one ⑫. The syntax "a" < each\_character < "z" is a shorthand syntax equivalent to each\_character > "a" and each\_character < "z". This shorthand syntax is called "chain comparison operation." This "chain comparison operation" is possible as all comparison operations in Python have the same priority, which is lower than that of

any arithmetic, shifting or bitwise operation and the comparison operation yields either Boolean *True* or *False*.

### Program 5.5: Write Python Program to Calculate the Length of a String Without Using Built-In *len()* Function

```

1. def main():
2. user_string = input("Enter a string: ")
3. count_character = 0
4. for each_character in user_string:
5. count_character += 1
6. print(f"The length of user entered string is {count_character} ")
7. if __name__ == "__main__":
8. main()

```

#### OUTPUT

```

Enter a string: To answer before listening that is folly and shame
The length of user entered string is 50

```

User entered string is assigned to *user\_string* variable ②. The *count\_character* variable is assigned to zero value ③. The *count\_character* acts like a counter which keeps track of number of characters and gets incremented by a value of one ⑤ when the *user\_string* is traversed using *for* loop ④.

## 5.5 String Methods

You can get a list of all the methods associated with string ([TABLE 5.2](#)) by passing the *str* function to *dir()*.

```

1. >>> dir(str)
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'cap-
italize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

```

Various methods associated with *str* are displayed ①.



**TABLE 5.2**

Various String Methods

| String Methods | Syntax                                         | Description                                                                                                                                                                                                                                                                                                          |
|----------------|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| capitalize()   | string_name.capitalize()                       | The <i>capitalize()</i> method returns a copy of the string with its first character capitalized and the rest lowercased.                                                                                                                                                                                            |
| casefold()     | string_name.casefold()                         | The <i>casefold()</i> method returns a casefolded copy of the string. Casefolded strings may be used for caseless matching.                                                                                                                                                                                          |
| center()       | string_name.center(width[, fillchar])          | The method <i>center()</i> makes string_name centered by taking width parameter into account. Padding is specified by parameter fillchar. Default filler is a space.                                                                                                                                                 |
| count()        | string_name.count(substring [, start [, end]]) | The method <i>count()</i> , returns the number of non-overlapping occurrences of substring in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.                                                                                                                         |
| endswith()     | string_name.endswith(suffix[, start[, end]])   | This method <i>endswith()</i> , returns <i>True</i> if the string_name ends with the specified suffix substring, otherwise returns <i>False</i> . With optional start, test beginning at that position. With optional end, stop comparing at that position.                                                          |
| find()         | string_name.find(substring[, start[, end]])    | Checks if substring appears in string_name or if substring appears in string_name specified by starting index <i>start</i> and ending index <i>end</i> . Return position of the first character of the first instance of string substring in string_name, otherwise return -1 if substring not found in string_name. |
| isalnum()      | string_name.isalnum()                          | The method <i>isalnum()</i> returns Boolean <i>True</i> if all characters in the string are alphanumeric and there is at least one character, else it returns Boolean <i>False</i> .                                                                                                                                 |
| isalpha()      | string_name.isalpha()                          | The method <i>isalpha()</i> , returns Boolean <i>True</i> if all characters in the string are alphabetic and there is at least one character, else it returns Boolean <i>False</i> .                                                                                                                                 |
| isdecimal()    | string_name.isdecimal()                        | The method <i>isdecimal()</i> , returns Boolean <i>True</i> if all characters in the string are decimal characters and there is at least one character, else it returns Boolean <i>False</i> .                                                                                                                       |
| isdigit()      | string_name.isdigit()                          | The method <i>isdigit()</i> returns Boolean <i>True</i> if all characters in the string are digits and there is at least one character, else it returns Boolean <i>False</i> .                                                                                                                                       |
| isidentifier() | string_name.isidentifier()                     | The method <i>isidentifier()</i> returns Boolean <i>True</i> if the string is a valid identifier, else it returns Boolean <i>False</i> .                                                                                                                                                                             |
| islower()      | string_name.islower()                          | The method <i>islower()</i> returns Boolean <i>True</i> if all characters in the string are lowercase, else it returns Boolean <i>False</i> .                                                                                                                                                                        |
| isspace()      | string_name.isspace()                          | The method <i>isspace()</i> returns Boolean <i>True</i> if there are only whitespace characters in the string and there is at least one character, else it returns Boolean <i>False</i> .                                                                                                                            |
| isnumeric()    | string_name.isnumeric()                        | The method <i>isnumeric()</i> , returns Boolean <i>True</i> if all characters in the string_name are numeric characters, and there is at least one character, else it returns Boolean <i>False</i> . Numeric characters include digit characters and all characters that have the Unicode numeric value property.    |

(Continued)

**TABLE 5.2 (Continued)**

Various String Methods

| String Methods | Syntax                                         | Description                                                                                                                                                                                                                                                                                      |
|----------------|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| istitle()      | string_name.istitle()                          | The method <i>istitle()</i> returns Boolean <i>True</i> if the string is a title cased string and there is at least one character, else it returns Boolean <i>False</i> .                                                                                                                        |
| isupper()      | string_name.isupper()                          | The method <i>isupper()</i> returns Boolean <i>True</i> if all cased characters in the string are uppercase and there is at least one cased character, else it returns Boolean <i>False</i> .                                                                                                    |
| upper()        | string_name.upper()                            | The method <i>upper()</i> converts lowercase letters in string to uppercase.                                                                                                                                                                                                                     |
| lower()        | string_name.lower()                            | The method <i>lower()</i> converts uppercase letters in string to lowercase.                                                                                                                                                                                                                     |
| ljust()        | string_name.ljust(width[, fillchar])           | In the method <i>ljust()</i> , when you provide the string to the method <i>ljust()</i> , it returns the string left justified. Total length of string is defined in first parameter of method width. Padding is done as defined in second parameter fillchar. (default is space).               |
| rjust()        | string_name.rjust(width[, fillchar])           | In the method <i>rjust()</i> , when you provide the string to the method <i>rjust()</i> , it returns the string right justified. The total length of string is defined in the first parameter of the method, width. Padding is done as defined in second parameter fillchar. (default is space). |
| title()        | string_name.title()                            | The method <i>title()</i> returns “titlecased” versions of string, that is, all words begin with uppercase characters and the rest are lowercase.                                                                                                                                                |
| swapcase()     | string_name.swapcase()                         | The method <i>swapcase()</i> returns a copy of the string with uppercase characters converted to lowercase and vice versa.                                                                                                                                                                       |
| splitlines()   | string_name.<br>splitlines([keepends])         | The method <i>splitlines()</i> returns a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.                                                                                                     |
| startswith()   | string_name.startswith(prefix[, start[, end]]) | The method <i>startswith()</i> returns Boolean <i>True</i> if the string starts with the prefix, otherwise return <i>False</i> . With optional start, test string_name beginning at that position. With optional end, stop comparing string_name at that position.                               |
| strip()        | string_name.strip([chars])                     | The method <i>strip()</i> returns a copy of the string_name in which specified <i>chars</i> have been stripped from both side of the string. If char is not specified then space is taken as default.                                                                                            |
| rstrip()       | string_name.rstrip([chars])                    | The method <i>rstrip()</i> removes all trailing whitespace of string_name.                                                                                                                                                                                                                       |
| lstrip()       | string_name.lstrip([chars])                    | The method <i>lstrip()</i> removes all leading whitespace in string_name.                                                                                                                                                                                                                        |
| replace()      | string_name.replace(old, new[, max])           | The method <i>replace()</i> replaces all occurrences of old in string_name with new. If the optional argument max is given, then only the first max occurrences are replaced.                                                                                                                    |
| zfill()        | string_name.zfill(width)                       | The method <i>zfill()</i> pads the string_name on the left with zeros to fill width.                                                                                                                                                                                                             |

Note: Replace the word *string\_name* mentioned in the syntax with the actual string name in your code.

For example,

```
1. >>> fact = "Abraham Lincoln was also a champion wrestler"
2. >>> fact.isalnum()
 False
3. >>> "sailors".isalpha()
 True
4. >>> "2018".isdigit()
 True
5. >>> fact.islower()
 False
6. >>> "TSAR BOMBA".isupper()
 True
7. >>> "columbus".islower()
 True
8. >>> warriors = "ancient gladiators were vegetarians"
9. >>> warriors.endswith("vegetarians")
 True
10. >>> warriors.startswith("ancient")
 True
11. >>> warriors.startswith("A")
 False
12. >>> warriors.startswith("a")
 True
13. >>> "cucumber".find("cu")
 0
14. >>> "cucumber".find("um")
 3
15. >>> "cucumber".find("xyz")
 -1
16. >>> warriors.count("a")
 5
17. >>> species = "charles darwin discovered galapagos tortoises"
18. >>> species.capitalize()
 'Charles darwin discovered galapagos tortoises'
19. >>> species.title()
 'Charles Darwin Discovered Galapagos Tortoises'
20. >>> "Tortoises".lower()
 'tortoises'
```

```

21. >>> "galapagos".upper()
 'GALAPAGOS'
22. >>> "Centennial Light".swapcase()
 'cENTENNIAL lIGHT'
23. >>> "history does repeat".replace("does", "will")
 'history will repeat'
24. >>> quote = " Never Stop Dreaming "
25. >>> quote.rstrip()
 ' Never Stop Dreaming'
26. >>> quote.lstrip()
 'Never Stop Dreaming '
27. >>> quote.strip()
 'Never Stop Dreaming'
28. >>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
 ['ab c', '\n', 'de fg', '\n']
29. >>> "scandinavian countries are rich".center(40)
 'scandinavian countries are rich'

```

Various operations on strings are carried out using string ①–② methods.

String methods like *capitalize()*, *lower()*, *upper()*, *swapcase()*, *title()* and *count()* are used for conversion purpose. String methods like *islower()*, *isupper()*, *isdecimal()*, *isdigit()*, *isnumeric()*, *isalpha()* and *isalnum()* are used for comparing strings. Some of the string methods used for padding are *rjust()*, *ljust()*, *zfill()* and *center()*. The string method *find()* is used to find substring in an existing string. You can use string methods like *replace()*, *join()*, *split()* and *splitlines()* to replace a string in Python.

**Program 5.6: Write Python Program That Accepts a Sentence and Calculate the Number of Words, Digits, Uppercase Letters and Lowercase Letters**

```

1. def string_processing(user_string):
2. word_count = 0
3. digit_count = 0
4. upper_case_count = 0
5. lower_case_count = 0
6. for each_char in user_string:
7. if each_char.isdigit():
8. digit_count += 1
9. elif each_char.isspace():
10. word_count += 1
11. elif each_char.isupper():
12. upper_case_count += 1

```

```

13. elif each_char.islower():
14. lower_case_count += 1
15. else:
16. pass
17. print(f"Number of digits in sentence is {digit_count}")
18. print(f"Number of words in sentence is {word_count + 1}")
19. print(f"Number of upper case letters in sentence is {upper_case_count}")
20. print(f"Number of lower case letters in sentence is {lower_case_count}")
21. def main():
22. user_input = input("Enter a sentence ")
23. string_processing(user_input)
24. if __name__ == "__main__":
25. main()

```

#### OUTPUT

```

Enter a sentence The Eiffel Tower in Paris is 324m tall
Number of digits in sentence is 3
Number of words in sentence is 8
Number of upper case letters in sentence is 4
Number of lower case letters in sentence is 24

```

Initially the variables *word\_count*, *digit\_count*, *upper\_case\_count* and *lower\_case\_count* are assigned to zero ②–⑤. For each character traversed through the string using *for* loop ⑥, the character is checked to see whether it is a digit or uppercase or lowercase. If any of these is *True*, then the corresponding variable *digit\_count* or *upper\_case* or *lower\_case* is incremented by one. If a space is encountered, then it indicates the end of a word and the variable *word\_count* is incremented by one. The number of whitespaces + 1 indicates the total number of words in a sentence ⑦–⑩. The *pass* statement allows you to handle the condition without the loop being impacted in any way. The *pass* is a null statement and nothing happens when *pass* is executed. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

#### Program 5.7: Write Python Program to Convert Uppercase Letters to Lowercase and Vice Versa

```

1. def case_conversion(user_string):
2. convert_case = str()
3. for each_char in user_string:
4. if each_char.isupper():
5. convert_case += each_char.lower()
6. else:
7. convert_case += each_char.upper()

```

```

8. print(f"The modified string is {convert_case}")
9. def main():
10. input_string = input("Enter a string ")
11. case_conversion(input_string)
12. if __name__ == "__main__":
13. main()

```

**OUTPUT**

```

Enter a string ExquiSITE
The modified string is eXQUIsite

```

Each character in the string *user\_string* is traversed using *for* loop ③ and check whether it is in uppercase ④. If Boolean *True* then convert the character to lowercase and concatenate it to *convert\_case* string ⑤ else convert the character to uppercase and concatenate with *convert\_case* string ⑦. Finally, print the *convert\_case* flipped string ⑧.

**Program 5.8: Write Python Program to Replace Comma-Separated Words with Hyphens and Print Hyphen-Separated Words in Ascending Order**

```

1. def replace_comma_with_hyphen(comma_separated_words):
2. split_words = comma_separated_words.split(",")
3. split_words.sort()
4. hyphen_separated_words = "-".join(split_words)
5. print(f"Hyphen separated words in ascending order are
 '{hyphen_separated_words}'")
6. def main():
7. comma_separated_words = input("Enter comma separated words ")
8. replace_comma_with_hyphen(comma_separated_words)
9. if __name__ == "__main__":
10. main()

```

**OUTPUT**

```

Enter comma separated words global,education
Hyphen separated words in ascending order are 'education-global'

```

The user enters a sequence of words separated by comma ⑦ which is passed as an argument to *replace\_comma\_with\_hyphen()* function ⑧. The comma-separated words are split based on "," (comma) and assigned to *split\_words* as a list of string items ②. Sort the words in the list ③. Join the words in the list with a hyphen between them using *join()* function ④. Print the sorted and hyphen-separated sequence of words ⑤.

### Program 5.9: Write Python Program to Count the Occurrence of User-Entered Words in a Sentence

```

1. def count_word(word_occurrence, user_string):
2. word_count = 0
3. for each_word in user_string.split():
4. if each_word == word_occurrence:
5. word_count += 1
6. print(f"The word '{word_occurrence}' has occurred {word_count} times")
7. def main():
8. input_string = input("Enter a string ")
9. user_word = input("Enter a word to count its occurrence ")
10. count_word(user_word, input_string)
11. if __name__ == "__main__":
12. main()

```

#### OUTPUT

```

Enter a string You cannot end a sentence with because because because is a conjunction
Enter a word to count its occurrence because
The word 'because' has occurred 3 times

```

The user enters a string to count its occurrence in a sentence and is stored in *user\_word* ⑨ variable and the user entered sentence is stored in *input\_string* ⑧ variable. These are passed as parameters to *count\_word()* function definition ①. The sentence is split using space as the reference to a list of words ③ and use *for* loop to iterate through each word. Check whether each word is equal to the user entered string value stored in *word\_occurrence* string variable ④. If *True*, the *word\_count* variable is incremented by one ⑤. Finally, the total number of occurrences of the user entered word in a sentence is printed out ⑥.

---

## 5.6 Formatting Strings

Python supports multiple ways to format text strings. These include *%-formatting* and *str.format()*. Each of these methods have their advantages, but in addition, have disadvantages that make them cumbersome to use in practice. A new string formatting mechanism referred to as "**f-strings**" is becoming popular among Python community, taken from the leading character used to denote such strings, and stands for "formatted strings". The f-strings provide a way to embed expressions inside strings literals, using a minimal syntax. A string literal is what we see in the source code of a Python program, including the quotation marks. It should be noted that an f-string is really an expression evaluated at run time and not a constant value. In Python source code, an f-string is a literal string, prefixed with 'f', which contains expressions within curly braces '{' and '}'.

The f-strings formatting is driven by the desire to have a simpler way to format strings in Python. The existing ways of formatting are either error-prone, inflexible, or cumbersome. The *%-formatting* is limited as to the types it supports. Only int, str and doubles can be formatted. All other types are either not supported or converted to one of these types before formatting. In addition, there's a well-known trap when a single value is passed. For example,

```
1. >>> almanac = 'nostradamus'
2. >>> 'seer: %s' % almanac
'seer: nostradamus'
```

①–② works well when single value is passed. But if the variable *almanac* were ever to be a tuple, the same code would fail. For example,

```
1. >>> almanac = ('nostradamus', 1567)
2. >>> 'seer: %s' % almanac
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
```

①–② Passing Multiple values not supported in *%-formatting*.

The *str.format()* formatting was added to address some of these problems with *%-formatting*. In particular, it uses standard function call syntax and therefore supports multiple parameters. However, *str.format()* is not without its issues. Chief among them is its verbosity. For example, in the following code the text value is repeated.

```
1. >>> value = 4 * 20
2. >>> 'The value is {value}.'.format(value=value)
'The value is 80.'
```

①–② Too much verbosity

Even in its simplest form, there is a bit of boilerplate, and the value that's inserted into the placeholder is sometimes far removed from where the placeholder is situated.

```
1. >>> 'The value is {}'.format(value)
'The value is 80.'
```

① Statement is not informative.

With an f-string, this becomes,

```
1. >>> f'The value is {value}.'
'The value is 80.'
```

① The f-strings provide a concise, readable way to include the value of Python expressions inside strings.



Backslashes may not appear inside the expression portions of f-strings, so you cannot use them. Backslash escapes may appear inside the string portions of an f-string. For example, to escape quotes inside f-strings:

```
1. >>> f{'\quoted string\'}
```

File "<stdin>", line 1

SyntaxError: f-string expression part cannot include a backslash

- ① Backslashes are not supported within the curly braces when f-strings are used. You can use a different type of quote inside the expression:

```
1. >>> f{'"quoted string"'}
```

'quoted string'

- ① Use different types of quotes within and outside the curly braces.

### 5.6.1 Format Specifiers

Format specifiers may also contain evaluated expressions. The syntax for f-string formatting operation is,

*f'string\_statements {variable\_name [: {width}.{precision}]}'*

The *f* character should be prefixed during f-string formatting. The *string\_statement* is a string consisting of a sequence of characters. Within curly braces, you specify the *variable\_name* whose value will be displayed. Specifying *width* and *precision* values are optional. If they are specified, then both *width* and *precision* should be included within curly braces. Also, using *variable\_name* along with either *width* or *precision* values should be separated by a colon. You can pad or create space around *variable\_name* value element through *width* value. By default, strings are left-justified and numbers are right-justified. *Precision* refers to the total number of digits that will be displayed in a number. This includes the decimal point and all the digits, i.e., before and after the decimal point. For example,

```
1. >>> width = 10
2. >>> precision = 5
3. >>> value = 12.34567
4. >>> f'result: {value:{width}.{precision}}'
```

'result: 12.346'

```
5. >>> f'result: {value:{width}}'
```

'result: 12.34567'

```
6. >>> f'result: {value:.{precision}}'
```

'result: 12.346'

- ①–⑥ Different ways of string formatting in f-strings.

### 5.6.2 Escape Sequences

Escape Sequences are a combination of a backslash (\) followed by either a letter or a combination of letters and digits. Escape sequences are also called as control sequences. The backslash (\) character is used to escape the meaning of characters that follow it by substituting their special meaning with an alternate interpretation. So, all escape sequences consist of two or more characters.

Here is a list of several common escape sequences ([TABLE 5.3](#))

**TABLE 5.3**

List of Escape Sequences

| Escape Sequence | Meaning                                                                      |
|-----------------|------------------------------------------------------------------------------|
| \               | Break a Line into Multiple lines while ensuring the continuation of the line |
| \\              | Inserts a Backslash character in the string                                  |
| \'              | Inserts a Single Quote character in the string                               |
| \"              | Inserts a Double Quote character in the string                               |
| \n              | Inserts a New Line in the string                                             |
| \t              | Inserts a Tab in the string                                                  |
| \r              | Inserts a Carriage Return in the string                                      |
| \b              | Inserts a Backspace in the string                                            |
| \u              | Inserts a Unicode character in the string                                    |
| \ooo            | Inserts a character in the string based on its Octal value                   |
| \xhh            | Inserts a character in the string based on its Hex value                     |

1. >>> print("You can break \
 ... single line to \
 ... multiple lines")
 You can break single line to multiple lines
2. >>> print('print backslash \\ inside a string ')
 print backslash \ inside a string
3. >>> print('print single quote \' within a string')
 print single quote ' within a string
4. >>> print("print double quote \" within a string")
 print double quote " within a string
5. >>> print("First line \nSecond line")
 First line
 Second line
6. >>> print("tab\tspacing")
 tab    spacing

```

7. >>> print("same\\rlike")
 like
8. >>> print("He\\bi")
 Hi
9. >>> print("\\u20B9")
 ₹
10. >>> print("\\046")
 &
11. >>> print("\\x24")
 $

```

By placing a backslash (\) character at the end of the line, you can break a single line to multiple lines while ensuring continuation. It indicates that the next line is also part of the same statement ①. Print backslash by escaping the backslash itself ②. You can use the backslash (\) escape character to add single or double quotation marks in the strings ③–④. The \n escape sequence is used to insert a new line without hitting the enter or return key ⑤. The part of the string after \n escape sequence appears in the next line. A horizontal indentation is provided with the \t escape sequence ⑥. Inserts a carriage return in the string by moving all characters after \r to the beginning of the string by overriding the exact number of characters that were moved ⑦. The \b escape sequence removes the previous character ⑧. A 16-bit hex value Unicode character is inserted in the string as shown in ⑨. A character is inserted in the string based on its Octal ⑩ and Hex values ⑪.

### 5.6.3 Raw Strings

A raw string is created by prefixing the character *r* to the string. In Python, a raw string ignores all types of formatting within a string including the escape characters.

```

1. >>> print(r"Bible Says, \"Taste and see that the LORD is good; blessed is the man
 who takes refuge in him.\")
 Bible Says, \"Taste and see that the LORD is good; blessed is the man who takes
 refuge in him.\"

```

As you can see in the output, by constructing a raw string you can retain quotes, backslashes that are used to escape and other characters, as in ①.

### 5.6.4 Unicodes

Fundamentally, computers just deal with numbers. They store letters and other characters by assigning a number for each one. Before Unicode was invented, there were hundreds of different systems, called character encodings for assigning these numbers. These early character encodings were limited and could not contain enough characters to cover all the world's languages. Even for a simple language like English, no single encoding was adequate for all the letters, punctuation, and technical symbols in common use. The Unicode Standard provides a unique number for every character, no matter what platform, device,

application or language. It has been adopted by all modern software providers and now allows data to be transported through many different platforms, devices and applications without corruption.

Unicode can be implemented by different character encodings. The Unicode Standard defines Unicode Transformation Formats like UTF-8, UTF-16, and UTF-32, and several other encodings are in use. The most commonly used encodings are UTF-8, UTF-16 and UCS-2 (Universal Coded Character Set), a precursor of UTF-16. UTF-8 is dominantly used by websites (over 90%), uses one byte for the first 128 code points and up to 4 bytes for other characters.

Regular Python strings are *not* Unicode: they are just plain bytes. To create a Unicode string, use the 'u' prefix on the string literal. For example,

1. `>>> unicode_string = u'A unicode \u018e string \xf1'`
2. `>>> unicode_string`  
`'A unicode Ě string ñ'`

①–② A Unicode string is a different type of object from regular "str" string type.

---

## 5.7 Summary

- A string is a sequence of characters.
- To access values through slicing, square brackets are used along with the index.
- Various string operations include conversion, comparing strings, padding, finding a substring in an existing string and replace a string in Python.
- Python strings are immutable which means that once created they cannot be changed.

---

## Multiple Choice Questions

1. The arithmetic operator that cannot be used with strings is
  - a. +
  - b. \*
  - c. −
  - d. All of these
2. Judge the output of the following code,  
`print(r"\nWelcome")`
  - a. New line and welcome
  - b. \nWelcome
  - c. The letter r and then welcome
  - d. Error

3. What is the output of the following code snippet?

```
print("Sunday".find("day"))
```

- a. 6
  - b. 5
  - c. 3
  - d. 1
4. The output of the following code is,
- ```
print("apple is a fruit".split("is"))
```
- a. ['is a fruit']
 - b. [fruit]
 - c. ['apple', 'a fruit']
 - d. ['apple']
5. For the given string `s = "nostradamus"`, which of the following statement is used to retrieve the character `t`?
- a. `s[3]`
 - b. `s.getitem(3)`
 - c. `s.__getitem__(3)`
 - d. `s.getItem(3)`
6. The output of the following:
- ```
print("\tapple".lstrip())
```
- a. `\tapple`
  - b. `apple"`
  - c. `apple`
  - d. `""\tapple`
7. Deduce the output of the following code:
- ```
print('hello' 'newline')
```
- a. Hello
 - b. hellonewline
 - c. Error
 - d. Newline
8. What is the output of the following code?
- ```
"tweet"[2:]
```
- a. We
  - b. wee
  - c. eet
  - d. Twee

9. What is the output of the following code?

```
"apple is a fruit"[7:10]
```

- a. Apple
- b. s a
- c. Fruit
- d. None of the above

10. Identify the output of the following code:

```
print("My name is %s" % ('Charles Darwin'))
```

- a. My name is Charles Darwin
- b. Charles
- c. %Charles
- d. %

11. The prefix that is used to create a Unicode string is

- a. u
- b. h
- c. o
- d. c

12. The function that is used to find the length of the string is

- a. len(string)
- b. length(string)
- c. len[string]
- d. length[string]

13. What is the output of the following code?

```
string = "Lion is the king of jungle"
print("%s" %string[4:7])
```

- a. of
- b. king
- c. The
- d. is

14. For the statement given below

```
example = "\t\ntweet\n"
```

The output for the expression `example.strip()` is

- a. `\t\ntweet\n`
- b. `\t\ntweet`
- c. `tweet\n`
- d. `'tweet'`

15. Deduce the output of the following code:

```
print('Data Science'.istitle())
```

- a. True
- b. False
- c. Error
- d. None

16. Predict the output of the following code:

```
print('200.123'.isnumeric())
```

- a. True
- b. False
- c. Error
- d. None

---

## Review Questions

1. What is the use of the *len()* function? Give one example.
2. With the help of an example, explain how we can create string variables in Python.
3. What is slice operation? Explain with an example.
4. List all the escape characters in Python with examples.
5. Explain *in* operator with an example.
6. Write a short note on the format operator.
7. Differentiate between the following.
  - a. *isidentifier()* and *isnumeric()*
  - b. *find()* and *casefold()*
  - c. *split()* and *splitlines()*
8. What would happen if a mischievous user typed in a word when you ask for a number?
9. Write a function called *rotate\_word* that takes a string and an integer as parameters, and that function should return a new string containing the letters from the original string “rotated” by the given amount. For example, “cheer” rotated by 7 is “jolly” and “melon” rotated by -10 is “cubed”.
10. Given that *message* is a string, what does *message[:]* indicate?
11. Write a function that takes a string as an argument and displays the letters backward, one per line.
12. Write a Python program to access the last character of the string with the help of *len()* function.
13. Ask the user for a string, and then for a number. Print out that string, that many times. (For example, if the string is Python and the number is 3 you should print out PythonPythonPython.)

14. Write a program that reads the date in the format (dd/mm/yyyy) and replaces the '/' with a '-' and displays the date in (dd-mm-yyyy) format.
15. Write a function that finds the number of occurrences of a specified character in a string.
16. Write a program that parses a binary number to a decimal integer. For example,  
 $11001 (1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0)$ .
17. Consider the following four string variables, as shown:  
city1 = "London"  
city2 = "Paris"  
city3 = "London"  
city4 = "Sydney"  
What are the results of the following expressions?
  - a. city1 == city2
  - b. city3.count('n')
  - c. city1 <= city4
  - d. city2.upper()
  - e. len(city4)
  - f. city1.lower()
18. Write a program that accepts a string from the user and display the same string after removing vowels from it.
19. Write a function to insert a string in the middle of the string.
20. Write a program to sort a string lexicographically.
21. Write a program to replace a string with another string without using built-in methods.
22. Write a program to concatenate two strings into another string without using the + operator.
23. Write a program to strip a set of characters from a string.
24. Write a program to extract the first  $n$  characters of a string.





**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 6

---

## *Lists*

---

### AIM

Understand the role of lists in Python in storing multiple items of different types and perform manipulations using various methods.

### LEARNING OUTCOMES

At the end of the chapter, you are expected to

- Create and manipulate items in lists.
- Comprehend Indexing and slicing in lists.
- Use methods associated with lists.
- Using lists as arguments in functions.
- Use *for* loop to access individual items in lists.

Most of the times a variable can hold a single value. However, in many cases, you need to assign more than that. Consider a Forest scenario where animals belonging to different types of families live. Some of them like Lion, Tiger, Cheetah are Carnivorous and others like Monkeys, Elephants and Buffalos are Herbivorous. If you want to define this habitat computationally, then you have to create multiple variables to represent each animal belonging to a particular animal family which may not make much sense. Instead, you can combine all the animals belonging to the particular animal family under one variable and then use this variable name for further manipulation.

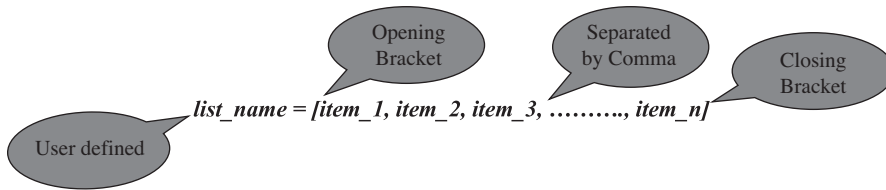
You can think of the list as a container that holds a number of items. Each element or value that is inside a list is called an item. All the items in a list are assigned to a single variable. Lists avoid having a separate variable to store each item which is less efficient and more error prone when you have to perform some operations on these items. Lists can be simple or nested lists with varying types of values. Lists are one of the most flexible data storage formats in Python because they can have values added, removed, and changed.

---

### 6.1 Creating Lists

Lists are constructed using square brackets [ ] wherein you can include a list of items separated by commas.

The syntax for creating list is,



1. `>>> superstore = ["metro", "tesco", "walmart", "kmart", "carrefour"]`
2. `>>> superstore`  
`['metro', 'tesco', 'walmart', 'kmart', 'carrefour']`

In ① each item in the list is a string. The contents of the list variable are displayed by executing the list variable name ②. When you print out the list, the output looks exactly like the list you had created.

You can create an empty list without any items. The syntax is,

**`list_name = []`**

For example,

1. `>>> number_list = [4, 4, 6, 7, 2, 9, 10, 15]`
2. `>>> mixed_list = ['dog', 87.23, 65, [9, 1, 8, 1]]`
3. `>>> type(mixed_list)`  
`<class 'list'>`
4. `>>> empty_list = []`
5. `>>> empty_list`  
`[]`
6. `>>> type(empty_list)`  
`<class 'list'>`

Here, *number\_list* ① contains items of the same type while in *mixed\_list* ② the items are a mix of type string, float, integer and another list itself. You can determine the type of a *mixed\_list* ③ variable by passing the variable name as an argument to *type()* function. In Python, the list type is called as *list*. An empty list can be created as shown in ④–⑤ and the variable *empty\_list* is of *list* type ⑥.



You can store any item in a list like string, number, object, another variable and even another list. You can have a mix of different item types and these item types need not have to be homogeneous. For example, you can have a list which is a mix of type numbers, strings and another list itself.

---

## 6.2 Basic List Operations

In Python, lists can also be concatenated using the `+` sign, and the `*` operator is used to create a repeated sequence of list items. For example,

```
1. >>> list_1 = [1, 3, 5, 7]
2. >>> list_2 = [2, 4, 6, 8]
3. >>> list_1 + list_2
 [1, 3, 5, 7, 2, 4, 6, 8]
4. >>> list_1 * 3
 [1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7]
5. >>> list_1 == list_2
 False
```

Two lists containing numbers as items are created ①–②. The *list\_1* and *list\_2* lists are added to form a new list. The new list has all items of both the lists ③. You can use the multiplication operator on the list. It repeats the items the number of times you specify and in ④ the *list\_1* contents are repeated three times. Contents of the lists *list\_1* and *list\_2* are compared using the `==` operator ⑤ and the result is Boolean *False* since the items in both the lists are different.

You can check for the presence of an item in the list using *in* and *not in* membership operators. It returns a Boolean *True* or *False*. For example,

```
1. >>> list_items = [1,3,5,7]
2. >>> 5 in list_items
 True
3. >>> 10 in list_items
 False
```

If an item is present in the list then using *in* operator results in *True* ② else returns *False* Boolean value ③.

### 6.2.1 The *list()* Function

The built-in *list()* function is used to create a list. The syntax for *list()* function is,

***list([sequence])***

where the sequence can be a string, tuple or list itself. If the optional sequence is not specified then an empty list is created. For example,

```
1. >>> quote = "How you doing?"
2. >>> string_to_list = list(quote)
3. >>> string_to_list
 ['H', 'o', 'w', ' ', 'y', 'o', 'u', ' ', 'd', 'o', 'i', 'n', 'g', '?']
4. >>> friends = ["j", "o", "e", "y"]
```

5. >>> friends + quote

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: can only concatenate list (not "str") to list

6. >>> friends + list(quote)

[', 'o', 'e', 'y', 'H', 'o', 'w', ' ', 'y', 'o', 'u', ' ', 'd', 'o', 'i', 'n', 'g', '?']

The string variable *quote* ① is converted to a list using the *list()* function ②. Now, let's see whether a string can be concatenated with a list ⑤. The result is an Exception which says *TypeError: can only concatenate list (not "str") to list*. That means you cannot concatenate a list with a string value. In order to concatenate a string with the list, you must convert the string value to list type, using Python built-in *list()* function ⑥.

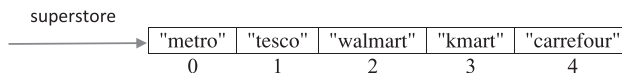
### 6.3 Indexing and Slicing in Lists

As an ordered sequence of elements, each item in a list can be called individually, through indexing. The expression inside the bracket is called the index. Lists use square brackets [ ] to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed.

The syntax for accessing an item in a list is,

*list\_name[index]*

where index should always be an integer value and indicates the item to be selected. For the list *superstore*, the index breakdown is shown below.



1. >>> superstore = ["metro", "tesco", "walmart", "kmart", "carrefour"]

2. >>> superstore[0]

'metro'

3. >>> superstore[1]

'tesco'

4. >>> superstore[2]

'walmart'

5. >>> superstore[3]

'kmart'

6. >>> superstore[4]

'carrefour'

7. >>> superstore[9]

Traceback (most recent call last):

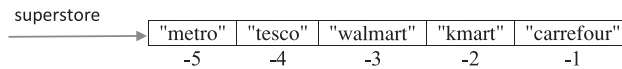
File "<stdin>", line 1, in <module>

IndexError: list index out of range

The *superstore* list has five items. To print the first item in the list use square brackets immediately after list name with an index value of zero ②. The index numbers for this *superstore* list range from 0 to 4 ②–⑥. If the index value is more than the number of items in the list ⑦ then it results in “*IndexError: list index out of range*” error.

In addition to positive index numbers, you can also access items from the list with a negative index number, by counting backwards from the end of the list, starting at  $-1$ . Negative indexing is useful if you have a long list and you want to locate an item towards the end of a list.

For the same list *superstore*, the negative index breakdown is shown below.



```
1. >>> superstore[-3]
 'walmart'
```

If you would like to print out the item 'walmart' by using its negative index number, you can do so as in ①.

### 6.3.1 Modifying Items in Lists

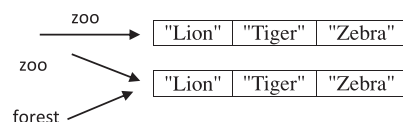
Lists are mutable in nature as the list items can be modified after you have created a list. You can modify a list by replacing the older item with a newer item in its place and without assigning the list to a completely new variable. For example,

```
1. >>> fauna = ["pronghorn", "alligator", "bison"]
2. >>> fauna[0] = "groundhog"
3. >>> fauna
 ['groundhog', 'alligator', 'bison']
4. >>> fauna[2] = "skunk"
5. >>> fauna
 ['groundhog', 'alligator', 'skunk']
6. >>> fauna[-1] = "beaver"
7. >>> fauna
 ['Groundhog', 'alligator', 'beaver']
```

You can change the string item at index 0 from 'pronghorn' to 'groundhog' as shown in ②. Now when you display *fauna*, the list items will be different ③. The item at index 2 is changed from “bison” to “skunk” ④. You can also change the value of an item by using a negative index number  $-1$  ⑥ which corresponds to the positive index number of 2. Now “skunk” is replaced with “beaver” ⑦.

When you assign an existing list variable to a new variable, an assignment ( $=$ ) on lists does not make a new copy. Instead, assignment makes both the variable names point to the same list in memory. For example,

```
1. >>> zoo = ["Lion", "Tiger", "Zebra"]
2. >>> forest = zoo
3. >>> type(zoo)
```



```

<class 'list'>
4. >>> type(forest)
 <class 'list'>
5. >>> forest
 ['Lion', 'Tiger', 'Zebra']
6. >>> zoo[0] = "Fox"
7. >>> zoo
 ['Fox', 'Tiger', 'Zebra']
8. >>> forest
 ['Fox', 'Tiger', 'Zebra']
9. >>> forest[1] = "Deer"
10. >>> forest
 ['Fox', 'Deer', 'Zebra']
11. >>> zoo
 ['Fox', 'Deer', 'Zebra']

```

The above code proves that assigning an existing list variable to a new variable does not create a new copy of the existing list items ①–⑪.

**Slicing** of lists is allowed in Python wherein a part of the list can be extracted by specifying index range along with the colon (:) operator which itself is a list.

The syntax for list slicing is,

***list\_name[start:stop[:step]]***

Colon is used to specify range values

where both *start* and *stop* are integer values (positive or negative values). List slicing returns a part of the list from the *start* index value to *stop* index value which includes the *start* index value but excludes the *stop* index value. *Step* specifies the increment value to slice by and it is optional. For the list *fruits*, the positive and negative index breakdown is shown below.

|        |              |             |               |         |          |
|--------|--------------|-------------|---------------|---------|----------|
| →      | "grapefruit" | "pineapple" | "blueberries" | "mango" | "banana" |
| fruits | 0            | 1           | 2             | 3       | 4        |
|        | -5           | -4          | -3            | -2      | -1       |

```

1. >>> fruits = ["grapefruit", "pineapple", "blueberries", "mango", "banana"]
2. >>> fruits[1:3]
 ['pineapple', 'blueberries']
3. >>> fruits[:3]
 ['grapefruit', 'pineapple', 'blueberries']
4. >>> fruits[2:]
 ['blueberries', 'mango', 'banana']
5. >>> fruits[1:4:2]
 ['pineapple', 'mango']

```

```

6. >>> fruits[:]
 ['grapefruit', 'pineapple', 'blueberries', 'mango', 'banana']
7. >>> fruits[::2]
 ['grapefruit', 'blueberries', 'banana']
8. >>> fruits[::-1]
 ['banana', 'mango', 'blueberries', 'pineapple', 'grapefruit']
9. >>> fruits[-3:-1]
 ['blueberries', 'mango']

```

All the items in the *fruits* list starting from an index value of 1 up to index value of 3 but excluding the index value of 3 is sliced ②. If you want to access the *start* items, then there is no need to specify the index value of zero. You can skip the *start* index value and specify only the *stop* index value ③. Similarly, if you want to access the last *stop* items, then there is no need to specify the *stop* value and you have to mention only the *start* index value ④. When *stop* index value is skipped the range of items accessed extends up to the last item. If you skip both the *start* and *stop* index values ⑥ and specify only the colon operator within the brackets, then the entire items in the list are displayed. The number after the second colon tells Python that you would like to choose your slicing increment. By default, Python sets this increment to 1, but that number after second colon allows you to specify what you want it to be. Using double colon as shown in ⑦ means no value for *start* index and no value for *stop* index and jump the items by two steps. Every second item of the list is extracted starting from an index value of zero. All the items in a list can be displayed in reverse order by specifying a double colon followed by an index value of  $-1$  ⑧. Negative index values can also be used for *start* and *stop* index values ⑨.

---

## 6.4 Built-In Functions Used on Lists

There are many built-in functions for which a list can be passed as an argument (TABLE 6.1).

**TABLE 6.1**

Built-In Functions Used on Lists

| Built-In Functions    | Description                                                                                                                       |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <code>len()</code>    | The <i>len()</i> function returns the numbers of items in a list.                                                                 |
| <code>sum()</code>    | The <i>sum()</i> function returns the sum of numbers in the list.                                                                 |
| <code>any()</code>    | The <i>any()</i> function returns <i>True</i> if any of the Boolean values in the list is <i>True</i> .                           |
| <code>all()</code>    | The <i>all()</i> function returns <i>True</i> if all the Boolean values in the list are <i>True</i> , else returns <i>False</i> . |
| <code>sorted()</code> | The <i>sorted()</i> function returns a modified copy of the list while leaving the original list untouched.                       |

For example,

```

1. >>> lakes = ['superior', 'erie', 'huron', 'ontario', 'powell']
2. >>> len(lakes)
 5

```



```

3. >>> numbers = [1, 2, 3, 4, 5]
4. >>> sum(numbers)
 15
5. >>> max(numbers)
 5
6. >>> min(numbers)
 1
7. >>> any([1, 1, 0, 0, 1, 0])
 True
8. >>> all([1, 1, 1, 1])
 True
9. >>> lakes_sorted_new = sorted(lakes)
10. >>> lakes_sorted_new
 ['erie', 'huron', 'ontario', 'powell', 'superior']

```

You can find the number of items in the list *lakes* using the *len()* function ②. Using the *sum()* function results in adding up all the numbers in the list ④. Maximum and minimum numbers in a list are returned using *max()* ⑤ and *min()* functions ⑥. If any of the items is 1 in the list then *any()* function returns Boolean *True* value ⑦. If all the items in the list are 1 then *all()* function returns *True* else returns *False* Boolean value ⑧. The *sorted()* function returns the sorted list of items without modifying the original list which is assigned to a new list variable ⑨. In the case of string items in the list, they are sorted based on their ASCII values.

---

## 6.5 List Methods

The list size changes dynamically whenever you add or remove the items and there is no need for you to manage it yourself. You can get a list of all the methods (TABLE 6.2) associated with the *list* by passing the list function to *dir()*.

```

1. >>> dir(list)
 ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__', '__hash__',
 '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__sub-
 classhook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']

```

Various methods associated with *list* is displayed ①.

**TABLE 6.2**

Various List Methods

| List Methods | Syntax                   | Description                                                                                                                                                                                                                                                                   |
|--------------|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| append()     | list.append(item)        | The <i>append()</i> method adds a single item to the end of the list. This method does not return new list and it just modifies the original.                                                                                                                                 |
| count()      | list.count(item)         | The <i>count()</i> method counts the number of times the item has occurred in the list and returns it.                                                                                                                                                                        |
| insert()     | list.insert(index, item) | The <i>insert()</i> method inserts the item at the given index, shifting items to the right.                                                                                                                                                                                  |
| extend()     | list.extend(list2)       | The <i>extend()</i> method adds the items in list2 to the end of the list.                                                                                                                                                                                                    |
| index()      | list.index(item)         | The <i>index()</i> method searches for the given item from the start of the list and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the list then <i>ValueError</i> is thrown by this method. |
| remove()     | list.remove(item)        | The <i>remove()</i> method searches for the first instance of the given item in the list and removes it. If the item is not present in the list then <i>ValueError</i> is thrown by this method.                                                                              |
| sort()       | list.sort()              | The <i>sort()</i> method sorts the items <i>in place</i> in the list. This method modifies the original list and it does not return a new list.                                                                                                                               |
| reverse()    | list.reverse()           | The <i>reverse()</i> method reverses the items <i>in place</i> in the list. This method modifies the original list and it does not return a new list.                                                                                                                         |
| pop()        | list.pop([index])        | The <i>pop()</i> method removes and returns the item at the given index. This method returns the rightmost item if the index is omitted.                                                                                                                                      |

*Note:* Replace the word “list” mentioned in the syntax with your *actual list name* in your code.

For example,

```

1. >>> cities = ["oslo", "delhi", "washington", "london", "seattle", "paris", "washington"]
2. >>> cities.count('seattle')
1
3. >>> cities.index('washington')
2
4. >>> cities.reverse()
5. >>> cities
['washington', 'paris', 'seattle', 'london', 'washington', 'delhi', 'oslo']
6. >>> cities.append('brussels')
7. >>> cities
['washington', 'paris', 'seattle', 'london', 'washington', 'delhi', 'oslo', 'brussels']
8. >>> cities.sort()
9. >>> cities
['brussels', 'delhi', 'london', 'oslo', 'paris', 'seattle', 'washington', 'washington']
10. >>> cities.pop()
'washington'

```

```

11. >>> cities
 ['brussels', 'delhi', 'london', 'oslo', 'paris', 'seattle', 'washington']
12. >>> more_cities = ["brussels", "copenhagen"]
13. >>> cities.extend(more_cities)
14. >>> cities
 ['brussels', 'delhi', 'london', 'oslo', 'paris', 'seattle', 'washington', 'brussels', 'copenhagen']
15. >>> cities.remove("brussels")
16. >>> cities
 ['delhi', 'london', 'oslo', 'paris', 'seattle', 'washington', 'brussels', 'copenhagen']

```

Various operations on lists are carried out using list methods ①–⑥.

### 6.5.1 Populating Lists with Items

One of the popular way of populating lists is to start with an empty list [ ], then use the functions *append()* or *extend()* to add items to the list. For example,

```

1. >>> continents = []
2. >>> continents.append("Asia")
3. >>> continents.append("Europe")
4. >>> continents.append("Africa")
5. >>> continents
 ['Asia', 'Europe', 'Africa']

```

Create an empty list *continents* ① and start populating items to the *continents* list using *append()* function ②–④ and finally display the items of the *continents* list ⑤.

#### Program 6.1: Program to Dynamically Build User Input as a List

```

1. list_items = input("Enter list items separated by a space ").split()
2. print(f"List items are {list_items}")

3. items_of_list = []
4. total_items = int(input("Enter the number of items "))
5. for i in range(total_items):
6. item = input("Enter list item: ")
7. items_of_list.append(item)
8. print(f"List items are {items_of_list}")

```

#### OUTPUT

```

Enter list items separated by a space Asia Europe Africa
List items are ['Asia', 'Europe', 'Africa']

```

```
Enter the number of items 2
Enter list item: Australia
Enter list item: Americas
List items are ['Australia', 'Americas']
```

You can build a list from user-entered values in two ways. In the first method ①–② you chain *input()* and *split()* together using dot notation. The user has to enter the items separated by spaces. Even though the user entered input has multiple items separated by spaces, by default the user-entered input is considered a single string. You benefit from the fact that the *split()* method can be used with the string. This user-entered input string value is split based on spaces and the *split()* method returns a list of string items. The advantage of this method is, there is no need for you to specify the total number of items that you are planning to insert into the list. In the second method ③–⑧ you have to specify the total number of items that you are planning to insert into the list beforehand itself ④. Based on this number we iterate through the *for* loop as many times using *range()* function ⑤. During each iteration, you need to append ⑦ the user entered value to the list ⑧.

### 6.5.2 Traversing of Lists

Using a *for* loop you can iterate through each item in a list.

#### Program 6.2: Program to Illustrate Traversing of Lists Using the *for* loop

```
1. fast_food = ["waffles", "sandwich", "burger", "fries"]
2. for each_food_item in fast_food:
3. print(f"I like to eat {each_food_item}")

4. for each_food_item in ["waffles", "sandwich", "burger", "fries"]:
5. print(f"I like to eat {each_food_item}")
```

#### OUTPUT

```
I like to eat waffles
I like to eat sandwich
I like to eat burger
I like to eat fries
I like to eat waffles
I like to eat sandwich
I like to eat burger
I like to eat fries
```

The *for* statement makes it easy to loop over the items in a list. A list variable is created ① and the list variable is specified in the *for* loop ②. Instead of specifying a list variable, you can specify a list directly in the *for* loop ④.

You can obtain the index value of each item in the list by using *range()* along with *len()* function.

**Program 6.3: Program to Display the Index Values of Items in List**

```
1. silicon_valley = ["google", "amd", "yahoo", "cisco", "oracle"]
2. for index_value in range(len(silicon_valley)):
3. print(f"The index value of '{silicon_valley[index_value]}' is {index_value}")
```

**OUTPUT**

```
The index value of 'google' is 0
The index value of 'amd' is 1
The index value of 'yahoo' is 2
The index value of 'cisco' is 3
The index value of 'oracle' is 4
```

You need to pass the list name as an argument to *len()* function and the resulting value will be passed as an argument to *range()* function to obtain the index value ② of each item in the list ①.

**Program 6.4: Write Python Program to Sort Numbers in a List in Ascending Order Using Bubble Sort by Passing the List as an Argument to the Function Call**

```
1. def bubble_sort(list_items):
2. for i in range(len(list_items)):
3. for j in range(len(list_items)-i-1):
4. if list_items[j] > list_items[j+1]:
5. temp = list_items[j]
6. list_items[j] = list_items[j+1]
7. list_items[j+1] = temp
8. print(f"The sorted list using Bubble Sort is {list_items}")

9. def main():
10. items_to_sort = [5, 4, 3, 2, 1]
11. bubble_sort(items_to_sort)

12. if __name__ == "__main__":
13. main()
```

**OUTPUT**

```
The sorted list using Bubble Sort is [1, 2, 3, 4, 5]
```

Bubble sort is an elementary sorting algorithm that repeatedly steps through the items in the list to be sorted by comparing each item with its successor item and swaps them if they are in the wrong order (FIGURE 6.1). The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. Bubble sort requires a maximum of  $n - 1$  passes if there are  $n$  items in the list. To sort the items in the list we require two loops. One for running through the passes ② and another for comparing

| 1 <sup>st</sup> Pass, Index Value 0 | 2 <sup>nd</sup> Pass, Index Value 1 | 3 <sup>rd</sup> Pass, Index Value 2 | 4 <sup>th</sup> Pass, Index Value 3 |
|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| (5,4,3,2,1) → (4,5,3,2,1)           | (4,3,2,1,5) → (3,4,2,1,5)           | (3,2,1,4,5) → (2,3,1,4,5)           | (2,1,3,4,5) → (1,2,3,4,5)           |
| (4,5,3,2,1) → (4,3,5,2,1)           | (3,4,2,1,5) → (3,2,4,1,5)           | (2,3,1,4,5) → (2,1,3,4,5)           |                                     |
| (4,3,5,2,1) → (4,3,2,5,1)           | (3,2,4,1,5) → (3,2,1,4,5)           |                                     |                                     |
| (4,3,2,5,1) → (4,3,2,1,5)           |                                     |                                     |                                     |

**FIGURE 6.1**

Sorting steps in Bubble Sort algorithm.

consecutive items in each pass ③. The indexing value for each item in the list is obtained by `range(len(list_items)-i-1)`. While traversing the list using `for` loop, swap ⑤–⑦ if the current item is found to be greater than the next item ④. You can also pass a list ⑩ as an argument to the function call ⑪ which is assigned to the `list_items` parameter in the function definition ①.

### Program 6.5: Write Python Program to Conduct a Linear Search for a Given Key Number in the List and Report Success or Failure

```

1. def read_key_item():
2. key_item = int(input("Enter the key item to search: "))
3. return key_item

4. def linear_search(search_key):
5. list_of_items = [10, 20, 30, 40, 50]
6. found = False
7. for item_index in range(len(list_of_items)):
8. if list_of_items[item_index] == search_key:
9. found = True
10. break
11. if found:
12. print(f"{search_key} found at position {item_index + 1}")
13. else:
14. print("Item not found in the list")

15. def main():
16. key_in_list = read_key_item()
17. linear_search(key_in_list)

18. if __name__ == "__main__":
19. main()

```

### OUTPUT

Enter the key item to search: 50  
50 found at position 5

Linear search is a method for finding a key item in a list. It sequentially checks each item of the list for the key value until a match is found or until all the items have been searched. The function *read\_key\_item()* is used to read search key from user ①–③. The function *linear\_search()* ④ searches for a key item in the list and reports success or failure. A *for* loop is required to run through all the items in the list ⑦. An *if* condition is used to check for the presence of the key item in the list ⑧. Here the variable *found* keeps track of the presence of the item in the list. Initially *found* is set to False ⑥. If the key item is present, then the variable *found* is assigned with Boolean *True* ⑨. Then the key item along with its position is displayed ⑫.

**Program 6.6: Input Five Integers (+ve and –ve). Find the Sum of Negative Numbers, Positive Numbers and Print Them. Also, Find the Average of All the Numbers and Numbers Above Average**

```

1. def find_sum(list_items):
2. positive_sum = 0
3. negative_sum = 0
4. for item in list_items:
5. if item > 0:
6. positive_sum = positive_sum + item
7. else:
8. negative_sum = negative_sum + item
9. average = (positive_sum + negative_sum) / 5
10. print(f"Sum of Positive numbers in list is {positive_sum}")
11. print(f"Sum of Negative numbers in list is {negative_sum}")
12. print(f"Average of item numbers in list is {average}")
13. print("Items above average are")
14. for item in list_items:
15. if item > average:
16. print(item)
17. def main():
18. find_sum([-1, -2, -3, 4, 5])
19. if __name__ == "__main__":
20. main()

```

**OUTPUT**

```

Sum of Positive numbers in list is 9
Sum of Negative numbers in list is -6
Average of item numbers in list is 0.6
Items above average are
4
5

```

Traverse through the list of five items of integer type ④. If a number is greater than zero then it is positive or if a number is less than zero then it is negative ⑤. Sum of positive

and negative numbers are stored in the variables *positive\_sum* ⑥ and *negative\_sum* ⑧ respectively. Traverse through each item in the list and based on positive and negative numbers, add to the variables *positive\_sum* and *negative\_sum* and print it ⑩–⑪. Average is calculated by  $(\text{positive\_sum} + \text{negative\_sum}) / 5$  ⑨ and print it ⑫. After getting the *average* value, traverse through each item in the list ⑭ and check whether the numbers are above average ⑮. If so, print those numbers ⑯.

**Program 6.7: Check If the Items in the List Are Sorted in Ascending or Descending Order and Print Suitable Messages Accordingly. Otherwise, Print “Items in list are not sorted”**

```

1. def check_for_sort_order(list_items):
2. ascending = descending = True
3. for i in range(len(list_items) - 1):
4. if list_items[i] < list_items[i+1]:
5. descending = False
6. if list_items[i] > list_items[i+1]:
7. ascending = False
8. if ascending:
9. print("Items in list are in Ascending order")
10. elif descending:
11. print("Items in list are in Descending order")
12. else:
13. print("Items in list are not sorted")
14. def main():
15. check_for_sort_order([1, 4, 2, 5, 3])
16. if __name__ == "__main__":
17. main()

```

#### OUTPUT

Items in list are not sorted

In the beginning, it is assumed that the list is sorted either in ascending order or in descending order by setting the variables *ascending* and *descending* to Boolean *True* value ②. Now traverse through each item of the list ③. If the first item is less than the second item, then the list is not in descending order, so set the variable *descending* to *False* Boolean value. If the second item is greater than the third item, then the list is not in ascending order, so set the variable *ascending* to *False* Boolean value ④–⑦. By applying the above logic to each item in the list results in *ascending* variable is set to *True* and *descending* variable is set to a *False* Boolean value for items stored in ascending order. If the list items are in descending order, then the *descending* variable is set to *True* while *ascending* variable is set to a *False* Boolean value. If items are not in any order in the list, then both *ascending* and *descending* variables are set to *False*.



**Program 6.8: Find Mean, Variance and Standard Deviation of List Numbers**

```

1. import math
2. def statistics(list_items):
3. mean = sum(list_items)/len(list_items)
4. print(f"Mean is {mean}")
5. variance = 0
6. for item in list_items:
7. variance += (item-mean)**2
8. variance /= len(list_items)
9. print(f"Variance is {variance}")
10. standard_deviation = math.sqrt(variance)
11. print(f"Standard Deviation is {standard_deviation}")

12. def main():
13. statistics([1, 2, 3, 4])

14. if __name__ == "__main__":
15. main()

```

**OUTPUT**

Mean is 2.5

Variance is 1.25

Standard Deviation is 1.118033988749895

Mean is calculated as the sum of all items in the list / total number of items in the list ③. Variance is defined as the average of the squared differences from the mean. Steps to find the *variance* is to first calculate the *mean* ③–④, then traverse through each element in the list and subtract it with *mean* and square the result which is also called as the squared difference. Finally, find the mean of those squared differences ⑥–⑧ to get the variance ⑨. Standard deviation is the square root of variance ⑩–⑪.

**Program 6.9: Write Python Program to Implement Stack Operations**

```

1. stack = []
2. stack_size = 3

3. def display_stack_items():
4. print("Current stack items are: ")
5. for item in stack:
6. print(item)

7. def push_item_to_stack(item):
8. print(f"Push an item to stack {item}")

```

```
9. if len(stack) < stack_size:
10. stack.append(item)
11. else:
12. print("Stack is full!")

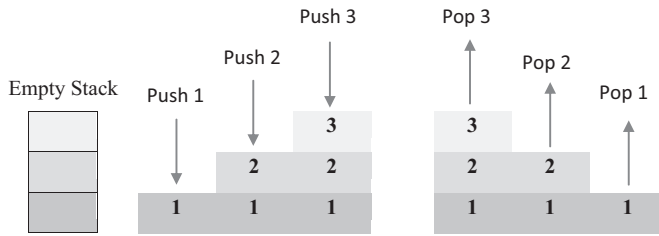
13. def pop_item_from_stack():
14. if len(stack) > 0:
15. print(f"Pop an item from stack {stack.pop()}")
16. else:
17. print("Stack is empty.")

18. def main():
19. push_item_to_stack(1)
20. push_item_to_stack(2)
21. push_item_to_stack(3)
22. display_stack_items()
23. push_item_to_stack(4)
24. pop_item_from_stack()
25. display_stack_items()
26. pop_item_from_stack()
27. pop_item_from_stack()
28. pop_item_from_stack()

29. if __name__ == "__main__":
30. main()
```

## OUTPUT

```
Push an item to stack 1
Push an item to stack 2
Push an item to stack 3
Current stack items are:
1
2
3
Push an item to stack 4
Stack is full!
Pop an item from stack 3
Current stack items are:
1
2
Pop an item from stack 2
Pop an item from stack 1
Stack is empty.
```

**FIGURE 6.2**

Stack push and pop operations.

The list methods make it very easy to use a list as a stack (FIGURE 6.2), where the last item added is the first item retrieved (“last-in, first-out”). To add an item to the top of the stack, use the `append()` method. To retrieve an item from the top of the stack, use `pop()` without an explicit index. Set the stack size to three. The function `display_stack_items()` ③–⑥ displays current items in the stack. The function `push_item_to_stack()` ⑦ is used to push an item to stack ⑩ if the total length of the stack is less than the stack size ⑨ else display “Stack is full” ⑫ message. The function `pop_item_from_stack()` ⑬ pops an item from the stack ⑮ if the stack is not empty ⑭ else display “Stack is empty” message ⑰.

### Program 6.10: Write Python Program to Perform Queue Operations

```

1. from collections import deque
2. def queue_operations():
3. queue = deque(['Eric', 'John', 'Michael'])
4. print(f"Queue items are {queue}")
5. print("Adding few items to Queue")
6. queue.append("Terry")
7. queue.append("Graham")
8. print(f"Queue items are {queue}")
9. print(f"Removed item from Queue is {queue.popleft()}")
10. print(f"Removed item from Queue is {queue.popleft()}")
11. print(f"Queue items are {queue}")
12. def main():
13. queue_operations()
14. if __name__ == "__main__":
15. main()

```

### OUTPUT

```

Queue items are deque(['Eric', 'John', 'Michael'])
Adding few items to Queue
Queue items are deque(['Eric', 'John', 'Michael', 'Terry', 'Graham'])

```

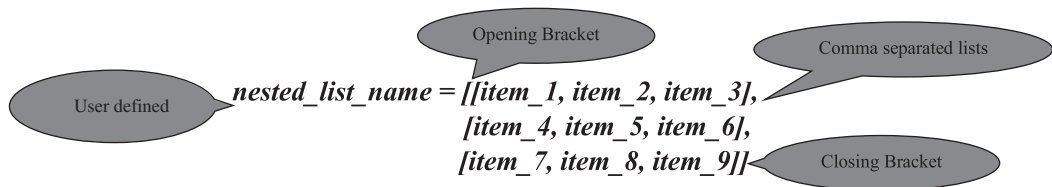
Removed item from Queue is Eric  
 Removed item from Queue is John  
 Queue items are deque(['Michael', 'Terry', 'Graham'])

It is also possible to use a list as a queue, where the first item added is the first item retrieved (“first-in, first-out”). However, lists are not effective for this purpose. While appends and pops are fast from the end of the list, doing inserts or pops from the beginning of a list is slow because all of the other items have to be shifted by one. To implement a queue ③, use *collections.deque* ① which was designed to have fast appends ⑨–⑩ and pops ⑥–⑦ from both ends.

### 6.5.3 Nested Lists

A list inside another list is called a nested list and you can get the behavior of nested lists in Python by storing lists within the elements of another list. You can traverse through the items of nested lists using the *for* loop.

The syntax for nested lists is,



Each list inside another list is separated by a comma. For example,

1. 

```
>>> asia = ["India", "Japan", "Korea",
 ["Srilanka", "Myanmar", "Thailand"],
 ["Cambodia", "Vietnam", "Israel"]]
```
2. 

```
>>> asia[0]
['India', 'Japan', 'Korea']
```
3. 

```
>>> asia[0][1]
'Japan'
```
4. 

```
>>> asia[1][2] = "Philippines"
```
5. 

```
>>> asia
[['India', 'Japan', 'Korea'], ['Srilanka', 'Myanmar', 'Philippines'], ['Cambodia', 'Vietnam', 'Israel']]
```

You can access an item inside a list that is itself inside another list by chaining two sets of square brackets together. For example, in the above list variable *asia* you have three lists ① which represent a  $3 \times 3$  matrix. If you want to display the items of the first list then specify the list variable followed by the index of the list which you need to access within the brackets, like *asia*[0] ②. If you want to access "Japan" item inside the list then you need to specify the index of the list within the list and followed by the index of the item in the list,

like `asia[0][1]` ③. You can even modify the contents of the list within the list. For example, to replace "Thailand" with "Philippines" use the code in ④.

### Program 6.11: Write a Program to Find the Transpose of a Matrix

```

1. matrix = [[10, 20],
 [30, 40],
 [50, 60]]
2. matrix_transpose = [[0, 0, 0],
 [0, 0, 0]]
3. def main():
4. for rows in range(len(matrix)):
5. for columns in range(len(matrix[0])):
6. matrix_transpose[columns][rows] = matrix[rows][columns]
7. print("Transposed Matrix is")
8. for items in matrix_transpose:
9. print(items)
10. if __name__ == "__main__":
11. main()

```

#### OUTPUT

```

Transposed Matrix is
[10, 30, 50]
[20, 40, 60]

```

The transpose of a matrix is a new matrix whose rows are the columns and columns are the rows of the original matrix. To find the transpose of a matrix ① in Python, you need an empty matrix initially to store the transposed matrix ②. This empty matrix should have one column greater and one row less than the original matrix. You need two *for* loops corresponding to rows ④ and columns of the matrix ⑤. Using *for* loops you need to iterate through each row and each column. At each point we place the `matrix[rows][columns]` item into `matrix_transpose[columns][rows]` ⑥. Finally, print the transposed result using *for* loop ⑧–⑨.

### Program 6.12: Write Python Program to Add Two Matrices

```

1. matrix_1 = [[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
2. matrix_2 = [[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]

```

```
3. matrix_result = [[0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]]
4. for rows in range(len(matrix_1)):
5. for columns in range(len(matrix_2[0])):
6. matrix_result[rows][columns] = matrix_1[rows][columns] + matrix_2[rows]
 [columns]
7. print("Addition of two matrices is")
8. for items in matrix_result:
9. print(items)
```

#### OUTPUT

Addition of two matrices is

```
[2, 4, 6]
[8, 10, 12]
[14, 16, 18]
```

To add two matrices in Python, you need to have two matrices which need to be added ①–② and another empty matrix ③. The empty matrix is used to store the addition result of the two matrices. To perform addition of matrices you need two *for* loops corresponding to rows ④ and columns ⑤ of *matrix\_1* and *matrix\_2* respectively. Using *for* loops you need to iterate through each row and each column. At each point we add *matrix\_1[rows][columns]* item with *matrix\_2[rows][columns]* item ⑥. Finally, print the matrix result of adding two matrices using *for* loop ⑧–⑨.

---

## 6.6 The *del* Statement

You can remove an item from a list based on its index rather than its value. The difference between *del* statement and *pop()* function is that the *del* statement does not return any value while the *pop()* function returns a value. The *del* statement can also be used to remove slices from a list or clear the entire list.

```
1. >>> a = [5, -8, 99.99, 432, 108, 213]
2. >>> del a[0]
3. >>> a
 [-8, 99.99, 432, 108, 213]
4. >>> del a[2:4]
5. >>> a
 [-8, 99.99, 213]
6. >>> del a[:]
7. >>> a
 []
```

An item at an index value of zero is removed ②. Now the number of items in the original list is reduced ③. Items starting from an index value of 2 up to 4 but excluding the index value of 4 is removed from the list ④. All the items in the list can be removed by specifying only the colon operator without *start* or *stop* index values ⑥–⑦.

---

## 6.7 Summary

- Lists are a basic and useful data structure built into the Python language.
- Built-in functions include *len()*, which returns the length of the list; *max()*, which returns the maximum element in the list; *min()*, which returns the minimum element in the list and *sum()*, which returns the sum of all the elements in the list.
- An individual elements in the list can be accessed using the index operator `[]`.
- Lists are mutable sequences which can be used to add, delete, sort and even reverse list elements.
- The *sort()* method is used to sort items in the list.
- The *split()* method can be used to split a string into a list.
- Nested list means a list within another list.

---

## Multiple-Choice Questions

1. The statement that creates the list is
  - a. `superstore = list()`
  - b. `superstore = []`
  - c. `superstore = list([1,2,3])`
  - d. All of the above
2. Suppose `continents = [1,2,3,4,5]`, what is the output of `len(continents)`?
  - a. 5
  - b. 4
  - c. None
  - d. error
3. What is the output of the following code snippet?

```
islands = [111,222,300,411,546]
max(islands)
```

  - a. 300
  - b. 222
  - c. 546
  - d. 111

4. Assume the list `superstore` is `[1,2,3,4,5]`, which of the following is correct syntax for slicing operation?
  - a. `print(superstore[0:])`
  - b. `print(superstore[:2])`
  - c. `print(superstore[:-2])`
  - d. All of these
5. If `zoo = ["lion", "tiger"]`, what will be `zoo * 2`?
  - a. `['lion']`
  - b. `['lion', 'lion', 'tiger', 'tiger']`
  - c. `['lion', 'tiger', 'lion', 'tiger']`
  - d. `['tiger']`
6. To add a new element to a list the statement used is?
  - a. `zoo.add(5)`
  - b. `zoo.append("snake")`
  - c. `zoo.addLast(5)`
  - d. `zoo.addend(4)`
7. To insert the string "snake" to the third position in `zoo`, which of the following statement is used?
  - a. `zoo.insert(3, "snake")`
  - b. `zoo.insert(2, "snake")`
  - c. `zoo.add(3, "snake")`
  - d. `zoo.append(3, "snake")`
8. Consider `laptops = [3, 4, 5, 20, 5, 25, 1, 3]`, what will be the output of `laptops.reverse()`?
  - a. `[3, 4, 5, 20, 5, 25, 1, 3]`
  - b. `[1, 3, 3, 4, 5, 5, 20, 25]`
  - c. `[25, 20, 5, 5, 4, 3, 3, 1]`
  - d. `[3, 1, 25, 5, 20, 5, 4, 3]`
9. Assume `quantity = [3, 4, 5, 20, 5, 25, 1, 3]`, then what will be the items of `quantity` list after `quantity.pop(1)`?
  - a. `[3, 4, 5, 20, 5, 25, 1, 3]`
  - b. `[1, 3, 3, 4, 5, 5, 20, 25]`
  - c. `[3, 5, 20, 5, 25, 1, 3]`
  - d. `[1, 3, 4, 5, 20, 5, 25]`
10. What is the output of the following code snippet?

```
letters = ['a', 'b', 'c', 'd', 'e']
letters[::-2]
```

  - a. `['d', 'c', 'b']`
  - b. `['a', 'c', 'e']`
  - c. `['a', 'b', 'd']`
  - d. `['e', 'c', 'a']`



11. Suppose `list_items` is `[3, 4, 5, 20, 5, 25, 1, 3]`, then what is the result of `list_items.remove(4)`?

- a. 3, 5, 29, 5
- b. 3, 5, 20, 5, 25, 1, 3
- c. 5, 20, 1, 3
- d. 1, 3, 25

12. Find the output of the following code.

```
matrix= [[1,2,3],[4,5,6]]
v = matrix[0][0]
for row in range(0, len(matrix)):
 for column in range(0, len(matrix[row])):
 if v < matrix[row][column]:
 v = matrix[row][column]
print(v)
```

- a. 3
- b. 5
- c. 6
- d. 33

13. Gauge the output of the following.

```
matrix = [[1, 2, 3, 4],
 [4, 5, 6, 7],
 [8, 9, 10, 11],
 [12, 13, 14, 15]]
```

```
for i in range(0, 4):
 print(matrix[i][1])
```

- a. 1 2 3 4
- b. 4 5 6 7
- c. 1 3 8 12
- d. 2 5 9 13

14. What will be the output of the following?

```
data = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
print(data[1][0][0])
```

- a. 1
- b. 2
- c. 4
- d. 5

15. The list function that inserts the item at the given index after shifting the items to the right is
    - a. `sort()`
    - b. `index()`
    - c. `insert()`
    - d. `append()`
  16. The method that is used to count the number of times an item has occurred in the list is
    - a. `count()`
    - b. `len()`
    - c. `length()`
    - d. `extend()`
- 

## Review Questions

1. Explain the advantages of the list.
2. Explain the different ways in which the lists can be created.
3. Explain the different list methods with an example.
4. With the help of an example explain the concept of nested lists.
5. Explain the ways of indexing and slicing the list with examples.
6. Differentiate between the following:
  - a. `pop()` and `remove()` methods of list.
  - b. `Del` statement and `pop()` method of list.
  - c. `append()` and `insert()` methods of list.
7. Write a program that creates a list of 10 random integers. Then create two lists by name `odd_list` and `even_list` that have all odd and even values of the list respectively.
8. Write a program to sort the elements in ascending order using insertion sort.
9. Write a Python program to use binary search to find the key element in the list.
10. Make a list of the first eight letters of the alphabet, then using the slice operation do the following operations:
  - a. Print the first three letters of the alphabet.
  - b. Print any three letters from the middle.
  - c. Print the letters from any particular index to the end of the list.
11. Write a program to sort the elements in ascending order using selection sort.
12. Write a program that prints the maximum value of the second half of the list.
13. Write a program that creates a list of numbers 1–100 that are either divisible by 5 or 6.
14. Write a function that prompts the user to enter five numbers, then invoke a function to find the GCD of these numbers.



**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 7

---

## Dictionaries

---

### AIM

Understand the role of dictionaries in Python in storing *key:value* pairs of different data types allowing you to organize your data in controlled ways.

### LEARNING OUTCOMES

At the end of the chapter, you are expected to

- Create and manipulate *key:value* pairs in dictionaries.
- Use methods associated with dictionaries.
- Use *for* loop to access *key:value* pairs in dictionaries.

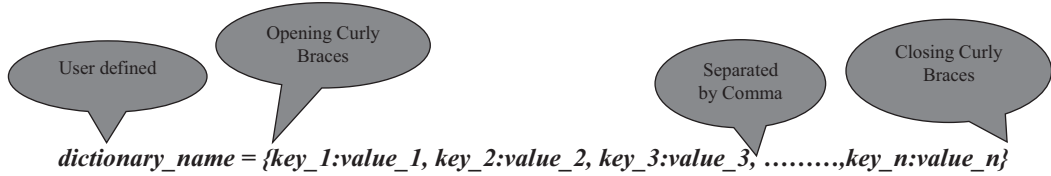
Another useful data type built into Python is the *Dictionary*. In the real world, you have seen your Contacts list in your phone. It is practically impossible to memorize the mobile number of everyone you come across. In the Contacts list, you store the name of the person as well as his number. This allows you to identify the mobile number based on a person's name. You can think of a person's name as the key that retrieves his mobile number, which is the value associated with the key. Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays."

---

### 7.1 Creating Dictionary

A dictionary is a collection of an unordered set of *key:value* pairs, with the requirement that the keys are unique within a dictionary. Dictionaries are constructed using curly braces { }, wherein you include a list of *key:value* pairs separated by commas. Also, there is a colon (:) separating each of these key and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values. Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys. Here a key along with its associated value is called a *key:value* pair. Dictionary keys are case sensitive.

The syntax for creating a dictionary is,



For example,

1. 

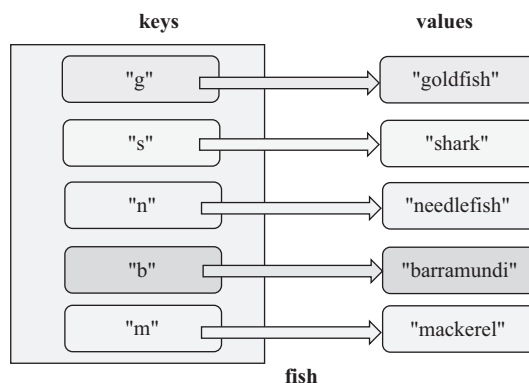
```
>>> fish = {"g": "goldfish", "s": "shark", "n": "needlefish", "b": "barramundi",
 "m": "mackerel"}
```
2. 

```
>>> fish
```

```
{'g': 'goldfish', 's': 'shark', 'n': 'needlefish', 'b': 'barramundi', 'm': 'mackerel'}
```

① Placing a comma-separated list of *key:value* pairs within the curly braces adds initial *key:value* pairs to the dictionary. This is also the way dictionaries are written on output ②. The keys in the dictionary *fish* are "g", "s", "n", "b", and "m" and the values associated with these keys are "goldfish", "shark", "needlefish", "barramundi", and "mackerel" (FIGURE 7.1). Each of the keys and values here is of string type. A value in the dictionary can be of any data type including string, number, list, or dictionary itself.



**FIGURE 7.1**  
Demonstration of *key:value* Pairs.



Dictionary keys are immutable type and can be either a string or a number. Since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`, you cannot use lists as keys. Duplicate keys are not allowed in the dictionary.

In dictionaries, the keys and their associated values can be of different types. For example,

```
1. >>> mixed_dict = {"portable": "laptop", 9: 11, 7: "julius"}
2. >>> mixed_dict
{'portable': 'laptop', 9: 11, 7: 'julius'}
3. >>> type(mixed_dict)
<class 'dict'>
```

In the dictionary, *mixed\_dict*, keys and their associated values are all of different types. For numbers as keys in dictionaries, it need not start from zero and it can be of any number. You can determine the type of *mixed\_dict* by passing the variable name as an argument to *type()* function. In Python, the dictionary type is called as *dict*.

You can create an empty dictionary by specifying a pair of curly braces and without any *key:value* pairs. The syntax is,

```
dictionary_name = { }
```

For example,

```
1. >>> empty_dictionary = {}
2. >>> empty_dictionary
{}
3. >>> type(empty_dictionary)
<class 'dict'>
```

An empty dictionary can be created as shown in ① and *empty\_dictionary* is of *dict* type ③.

In dictionaries, the order of *key:value* pairs does not matter. For example,

```
1. >>> pizza = {"pepperoni": 3, "calzone": 5, "margherita": 4}
2. >>> fav_pizza = {"margherita": 4, "pepperoni": 3, "calzone": 5}
3. >>> pizza == fav_pizza
True
```

The dictionaries *pizza* and *fav\_pizza* have the same *key:value* pairs but in a different order ①–②. If you compare these two dictionaries, it results in Boolean *True* value ③. This indicates that the ordering of *key:value* pairs does not matter in dictionaries. Slicing in dictionaries is not allowed since they are not ordered like lists.



Prior to Python 3.6 version, the execution of dictionary statements resulted in an unordered output of *key:value* pairs. However, starting from Python 3.6 version, the output of dictionary statements is ordered *key:value* pairs. Here, ordered means “insertion ordered”, i.e., dictionaries remember the order in which the *key:value* pairs were inserted.

## 7.2 Accessing and Modifying *key:value* Pairs in Dictionaries

Each individual *key:value* pair in a dictionary can be accessed through keys by specifying it inside square brackets. The key provided within the square brackets indicates the *key:value* pair being accessed.

The syntax for accessing the value for a key in the dictionary is,

*dictionary\_name[key]*

The syntax for modifying the value of an existing key or for adding a new *key:value* pair to a dictionary is,

*dictionary\_name[key] = value*

If the key is already present in the dictionary, then the key gets updated with the new value. If the key is not present then the new *key:value* pair gets added to the dictionary.

For example,

|        | renaissance |             |                |              |          |
|--------|-------------|-------------|----------------|--------------|----------|
| Keys   | "giotto"    | "Donatello" | "Michelangelo" | "Botticelli" | "clouet" |
| Values | 1305        | 1440        | 1511           | 1480         | 1520     |

1. 

```
>>> renaissance = {"giotto":1305, "donatello":1440, "michelangelo":1511, "botticelli":1480, "clouet":1520}
```
2. 

```
>>> renaissance["giotto"] = 1310
```
3. 

```
>>> renaissance
```

  
{'giotto': 1310, 'donatello': 1440, 'michelangelo': 1511, 'botticelli': 1480, 'clouet': 1520}
4. 

```
>>> renaissance["michelangelo"]
```

  
1511
5. 

```
>>> renaissance["leonardo"] = 1503
```
6. 

```
>>> renaissance
```

  
{'giotto': 1310, 'donatello': 1440, 'michelangelo': 1511, 'botticelli': 1480, 'clouet': 1520, 'leonardo': 1503}
7. 

```
>>> renaissance["piero"]
```

  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
KeyError: 'piero'

Since dictionaries are mutable, you can add a new *key:value* pair or change the values for existing keys using the assignment operator. For the dictionary *renaissance* ①, you can modify the existing values for the keys. The value for the key *giotto* is updated from 1305 to 1310 as shown in ② using the assignment operator. The value associated with the key *michelangelo* is displayed in ④. You can add a new *key:value* pair by specifying the name of the dictionary followed by a bracket within where you specify the name of the key and assign a value to it ⑤. If you try to access a non-existing key ⑦ then it results in *KeyError*.

You can check for the presence of a key in the dictionary using *in* and *not in* membership operators. It returns either a Boolean *True* or *False* value. For example,

1. `>>> clothes = {"rainy": "raincoats", "summer": "tees", "winter": "sweaters"}`
2. `>>> "spring" in clothes`  
`False`
3. `>>> "spring" not in clothes`  
`True`

In ② and ③, the presence of the key *spring* is checked in the dictionary *clothes*.

### 7.2.1 The *dict()* Function

The built-in *dict()* function is used to create dictionary. The syntax for *dict()* function when the optional keyword arguments used is,

***dict(\*\*kvarg)***

The function *dict()* returns a new dictionary initialized from an optional keyword arguments and a possibly empty set of keyword arguments. If no keyword argument is given, an empty dictionary is created. If keyword arguments are given, the keyword arguments and their values of the form *kvarg = value* are added to the dictionary as *key:value* pairs. For example,

1. `>>> numbers = dict(one=1, two=2, three=3)`
2. `>>> numbers`  
`{'one': 1, 'two': 2, 'three': 3}`

Keyword arguments of the form *kvarg = value* are converted to *key:value* pairs for *numbers* dictionary ①–②.

The syntax for *dict()* function when iterables used is,

***dict(iterable[, \*\*kvarg])***

You can specify an iterable containing exactly two objects as tuple, the key and value in the *dict()* function. For example

1. `>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])`  
`{'sape': 4139, 'jack': 4098, 'guido': 4127}`

The *dict()* function builds dictionaries directly from sequences of key, value tuple pairs ①.

## 7.3 Built-In Functions Used on Dictionaries

There are many built-in functions for which a dictionary can be passed as an argument (TABLE 7.1). The main operations on a dictionary are storing a value with some key and extracting the value for a given key.



**TABLE 7.1**

Built-In Functions Used on Dictionaries

| Built-in Functions    | Description                                                                                                                                |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <code>len()</code>    | The <i>len()</i> function returns the number of items ( <i>key:value</i> pairs) in a dictionary.                                           |
| <code>all()</code>    | The <i>all()</i> function returns Boolean <i>True</i> value if all the keys in the dictionary are <i>True</i> else returns <i>False</i> .  |
| <code>any()</code>    | The <i>any()</i> function returns Boolean <i>True</i> value if any of the key in the dictionary is <i>True</i> else returns <i>False</i> . |
| <code>sorted()</code> | The <i>sorted()</i> function by default returns a list of items, which are sorted based on dictionary keys.                                |

For example,

1. `>>> presidents = {"washington":1732, "jefferson":1751, "lincoln":1809, "roosevelt":1858, "eisenhower":1890}`
2. `>>> len(presidents)`  
5
3. `>>> all_dict_func = {0:True, 2:False}`
4. `>>> all(all_dict_func)`  
False
5. `>>> all_dict_func = {1:True, 2:False}`
6. `>>> all(all_dict_func)`  
True
7. `>>> any_dict_func = {1:True, 2:False}`
8. `>>> any(any_dict_func)`  
True
9. `>>> sorted(presidents)`  
['eisenhower', 'jefferson', 'lincoln', 'roosevelt', 'washington']
10. `>>> sorted(presidents, reverse = True)`  
['washington', 'roosevelt', 'lincoln', 'jefferson', 'eisenhower']
11. `>>> sorted(presidents.values())`  
[1732, 1751, 1809, 1858, 1890]
12. `>>> sorted(presidents.items())`  
[('eisenhower', 1890), ('jefferson', 1751), ('lincoln', 1809), ('roosevelt', 1858), ('washington', 1732)]

You can find the number of *key:value* pairs in the dictionary *presidents* ① using the *len()* function ②. In Python, any non-zero integer value is *True*, and zero is interpreted as *False* ②–⑥. If all the keys are Boolean *True* values, then the output is *True* else it is *False* Boolean value for *all()* function in dictionaries. If any one of the keys is *True* then it results in a *True* Boolean value else *False* Boolean value for *any()* function in dictionaries ⑦–⑧. The *sorted()* function returns the sorted list of keys by default in ascending order without modifying the original *key:value* pairs ⑨. You can also get a list of keys sorted in descending order by passing the second argument as *reverse=True* ⑩. In the case of dictionary keys being type strings, they are sorted based on their

ASCII values. You can obtain a sorted list of values instead of keys by using the *values()* method along with the dictionary name ①. A sorted list of key, value tuple pairs, which are sorted based on keys, are obtained by using the *items()* method along with the dictionary name ②.

## 7.4 Dictionary Methods

Dictionary allows you to store data in *key:value* format without depending on indexing. You can get a list of all the methods associated with *dict* (TABLE 7.2) by passing the *dict* function to *dir()*.

1. >>> dir(dict)

```
['_class_', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__', '__hash__', '__
init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__',
 '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
 'popitem', 'setdefault', 'update', 'values']
```

Various methods associated with *dict* are displayed ①.

**TABLE 7.2**

Various Dictionary Methods

| Dictionary Methods | Syntax                                             | Description                                                                                                                                                                                                                                                                                                                  |
|--------------------|----------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| clear()            | dictionary_name.<br>clear()                        | The <i>clear()</i> method removes all the <i>key:value</i> pairs from the dictionary.                                                                                                                                                                                                                                        |
| fromkeys()         | dictionary_name.<br>fromkeys(seq<br>[, value])     | The <i>fromkeys()</i> method creates a new dictionary from the given sequence of elements with a value provided by the user.                                                                                                                                                                                                 |
| get()              | dictionary_name.<br>get(key<br>[, default])        | The <i>get()</i> method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to <i>None</i> , so that this method never raises a <i>KeyError</i> .                                                       |
| items()            | dictionary_name.<br>items()                        | The <i>items()</i> method returns a new view of dictionary's key and value pairs as tuples.                                                                                                                                                                                                                                  |
| keys()             | dictionary_name.<br>keys()                         | The <i>keys()</i> method returns a new view consisting of all the keys in the dictionary.                                                                                                                                                                                                                                    |
| pop()              | dictionary_name.<br>pop(key[, default])            | The <i>pop()</i> method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If default is not given and the key is not in the dictionary, then it results in <i>KeyError</i> .                                                                          |
| popitem()          | dictionary_name.<br>popitem()                      | The <i>popitem()</i> method removes and returns an arbitrary (key, value) tuple pair from the dictionary. If the dictionary is empty, then calling <i>popitem()</i> results in <i>KeyError</i> .                                                                                                                             |
| setdefault()       | dictionary_name.<br>setdefault<br>(key[, default]) | The <i>setdefault()</i> method returns a value for the key present in the dictionary. If the key is not present, then insert the key into the dictionary with a default value and return the default value. If key is present, <i>default</i> defaults to <i>None</i> , so that this method never raises a <i>KeyError</i> . |
| update()           | dictionary_name.<br>update([other])                | The <i>update()</i> method updates the dictionary with the <i>key:value</i> pairs from <i>other</i> dictionary object and it returns <i>None</i> .                                                                                                                                                                           |
| values()           | dictionary_name.<br>values()                       | The <i>values()</i> method returns a new view consisting of all the values in the dictionary.                                                                                                                                                                                                                                |

*Note:* Replace the word “*dictionary\_name*” mentioned in the syntax with your *actual dictionary name* in your code.



The objects returned by *dict.keys()*, *dict.values()*, and *dict.items()* are view objects. They provide a dynamic view of the dictionary's entries, thus when the dictionary changes, the view reflects these changes.

For example,

```

1. >>> box_office_billion = {"avatar":2009, "titanic":1997, "starwars":2015, "harry-
 potter":2011, "avengers":2012}
2. >>> box_office_billion_fromkeys=box_office_billion.fromkeys(box_office_billion)
3. >>> box_office_billion_fromkeys
 {'avatar': None, 'titanic': None, 'starwars': None, 'harrypotter': None, 'avengers': None}
4. >>> box_office_billion_fromkeys = box_office_billion.fromkeys(box_office_
 billion, "billion_dollar")
5. >>> box_office_billion_fromkeys
 {'avatar': 'billion_dollar', 'titanic': 'billion_dollar', 'starwars': 'billion_dollar', 'harry-
 potter': 'billion_dollar', 'avengers': 'billion_dollar'}
6. >>> print(box_office_billion.get("frozen"))
 None
7. >>> box_office_billion.get("frozen",2013)
 2013
8. >>> box_office_billion.keys()
 dict_keys(['avatar', 'titanic', 'starwars', 'harrypotter', 'avengers'])
9. >>> box_office_billion.values()
 dict_values([2009, 1997, 2015, 2011, 2012])
10. >>> box_office_billion.items()
 dict_items([('avatar', 2009), ('titanic', 1997), ('starwars', 2015), ('harrypotter', 2011),
 ('avengers', 2012)])
11. >>> box_office_billion.update({"frozen":2013})
12. >>> box_office_billion
 {'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012, 'frozen': 2013}
13. >>> box_office_billion.setdefault("minions")
14. >>> box_office_billion
 {'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012,
 'frozen': 2013, 'minions': None}
15. >>> box_office_billion.setdefault("ironman", 2013)
16. >>> box_office_billion
 {'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012,
 'minions': None, 'ironman': 2013}

```

```

17. >>> box_office_billion.pop("avatar")
 2009
18. >>> box_office_billion.popitem()
 ('ironman', 2013)
19. >>> box_office_billion.clear()
 {}

```

Various operations on dictionaries are carried out using dictionary methods ①–⑨.

The *dict\_keys*, *dict\_values*, and *dict\_items* data types returned by various methods of dictionary are read-only views and cannot be edited directly. If you want to convert *dict\_keys*, *dict\_values*, and *dict\_items* data types returned from *keys()*, *values()*, and *items()* methods to a true list then pass their list, such as returned values, to *list()* function. Whenever there is a modification to the dictionary, it gets reflected in the *views* of these data types. For example,

```

1. >>> list(box_office_billion.keys())
 ['avatar', 'titanic', 'starwars', 'harrypotter', 'avengers']
2. >>> list(box_office_billion.values())
 [2009, 1997, 2015, 2011, 2012]
3. >>> list(box_office_billion.items())
 [('avatar', 2009), ('titanic', 1997), ('starwars', 2015), ('harrypotter', 2011), ('avengers',
 2012)]

```

The *dict\_keys*, *dict\_values*, and *dict\_items* returned from *keys()*, *values()*, and *items()* are converted to true list using the *list()* function ①–③.

#### 7.4.1 Populating Dictionaries with *key:value* Pairs

One of the common ways of populating dictionaries is to start with an empty dictionary { }, then use the *update()* method to assign a value to the key using assignment operator. If the key does not exist, then the *key:value* pairs will be created automatically and added to the dictionary.

For example,

```

1. >>> countries = {}
2. >>> countries.update({"Asia":"India"})
3. >>> countries.update({"Europe":"Germany"})
4. >>> countries.update({"Africa":"Sudan"})
5. >>> countries
 {'Asia': 'India', 'Europe': 'Germany', 'Africa': 'Sudan'}

```

Create an empty dictionary *countries* ① and start populating *key:value* pairs to the *countries* dictionary using the *update()* function ②–④ and finally display the *key:value* pairs of the *countries* dictionary ⑤. Within the *update()* function the *key:value* pair should be enclosed within the curly braces.

As discussed in [Section 7.2](#), the dictionary can also be built using `dictionary_name[key] = value` syntax by adding `key:value` pairs to the dictionary.

### Program 7.1: Program to Dynamically Build User Input as a List

```
1. def main():
2. print("Method 1: Building Dictionaries")
3. build_dictionary = {}
4. for i in range(0, 2):
5. dic_key = input("Enter key ")
6. dic_val = input("Enter val ")
7. build_dictionary.update({dic_key: dic_val})
8. print(f"Dictionary is {build_dictionary}")

9. print("Method 2: Building Dictionaries")
10. build_dictionary = {}
11. for i in range(0, 2):
12. dic_key = input("Enter key ")
13. dic_val = input("Enter val ")
14. build_dictionary[dic_key] = dic_val
15. print(f"Dictionary is {build_dictionary}")

16. print("Method 3: Building Dictionaries")
17. build_dictionary = {}
18. i = 0
19. while i < 2:
20. dict_key = input("Enter key ")
21. dict_val = input("Enter val ")
22. build_dictionary.update({dict_key: dict_val})
23. i = i + 1
24. print(f"Dictionary is {build_dictionary}")

25. if __name__ == "__main__":
26. main()
```

### OUTPUT

```
Method 1: Building Dictionaries
Enter key microsoft
Enter val windows
Enter key canonical
Enter val ubuntu
Dictionary is {'microsoft': 'windows', 'canonical': 'ubuntu'}
Method 2: Building Dictionaries
```

```

Enter key apple
Enter val macos
Enter key canonical
Enter val ubuntu
Dictionary is {'apple': 'macos', 'canonical': 'ubuntu'}
Method 3: Building Dictionaries
Enter key microsoft
Enter val windows
Enter key apple
Enter val macos
Dictionary is {'microsoft': 'windows', 'apple': 'macos'}

```

You can use the *for* loop to read and build a dictionary in different ways. In the first method ②–⑧, use the *for* loop to iterate the number of times you want to get the *key:value* pairs as input. You have to specify the total number of *key:value* pairs that you are planning to insert into the dictionary beforehand in the *for* loop using the *range()* function. The values that you enter for *key:value* pairs are stored in separate variables. An empty dictionary is created outside the *for* loop. You need to pass the key and value variables as arguments to the *update()* function in the *{key:value}* format to add the *key:value* pair to the dictionary. If the key is already present, then the latest value will be associated with the key or else the *key:value* pair gets added to the dictionary. In the second method, instead of specifying the *update()* function with the dictionary, one can use the *dictionary\_name[key] = value* format to add *key:value* pairs to the dictionary ⑨–⑩. In the third method, *while* loop is used instead of *for* loop to add *key:value* pairs to the dictionary as shown in ⑪–⑭.

#### 7.4.2 Traversing of Dictionary

A *for* loop can be used to iterate over keys or values or *key:value* pairs in dictionaries. If you iterate over a dictionary using a *for* loop, then, by default, you will iterate over the keys. If you want to iterate over the values, use *values()* method and for iterating over the *key:value* pairs, specify the dictionary's *items()* method explicitly. The *dict\_keys*, *dict\_values*, and *dict\_items* data types returned by dictionary methods can be used in *for* loops to iterate over the keys or values or *key:value* pairs.

#### Program 7.2: Program to Illustrate Traversing of *key:value* Pairs in Dictionaries Using *for* Loop

1. `currency = {"India": "Rupee", "USA": "Dollar", "Russia": "Ruble", "Japan": "Yen", "Germany": "Euro"}`
2. `def main():`
3.     `print("List of Countries")`
4.     `for key in currency.keys():`
5.         `print(key)`
6.     `print("List of Currencies in different Countries")`

```

7. for value in currency.values():
8. print(value)
9. for key, value in currency.items():
10. print(f'"{key}" has a currency of type "{value}"')
11. if __name__ == "__main__":
12. main()

```

## OUTPUT

List of Countries

India

USA

Russia

Japan

Germany

List of Currencies in different Countries

Rupee

Dollar

Ruble

Yen

Euro

'India' has a currency of type 'Rupee'

'USA' has a currency of type 'Dollar'

'Russia' has a currency of type 'Ruble'

'Japan' has a currency of type 'Yen'

'Germany' has a currency of type 'Euro'

Using the *keys()* ④–⑤, *values()* ⑦–⑧, and *items()* ⑨–⑩ methods, a *for* loop can iterate over the keys, values, or *key:value* pairs in a dictionary, respectively. By default, a *for* loop iterates over the keys. So the statement *for key in currency.keys():* results in the same output as *for key in currency:.* When looping through dictionaries, the key and its corresponding value can be retrieved simultaneously using the *items()* method. The values in the *dict\_items* type returned by the *items()* method are tuples where the first element in the tuple is the key and the second element is the value. You can use multiple iterating variables in a *for* loop to unpack the two parts of each tuple in the *items()* method by assigning the key and value to separate variables ⑨.

## Program 7.3: Write Python Program to Check for the Presence of a Key in the Dictionary and to Sum All Its Values

```

1. historical_events = {"apollo11": 1969, "great_depression": 1929, "american_revolution":
 1775, "berlin_wall": 1989}
2. def check_key_presence():
3. key = input("Enter the key to check for its presence ")
4. if key in historical_events.keys():
5. print(f'Key "{key}" is present in the dictionary')

```

```

6. else:
7. print(f"Key '{key}' is not present in the dictionary")

8. def sum_dictionary_values():
9. print("Sum of all the values in the dictionary is")

10. print(f"{sum(historical_events.values())}")
11. def main():
12. check_key_presence()

13. sum_dictionary_values()

14. if __name__ == "__main__":
15. main()

```

**OUTPUT**

```

Enter the key to check for its presence apollo11
Key 'apollo11' is present in the dictionary
Sum of all the values in the dictionary is
7662

```

When the function *check\_key\_presence()* ② is called, the user has to enter a *key* ③, which will be checked for its presence in the *historical\_events* dictionary. The presence for a *key* in the dictionary is checked using an *if* statement ④, and, if the *key* is present, then a message saying that the user entered key is present in the dictionary is displayed in the output ⑤. If the *key* is not present, then a message saying that the user entered key is not present in the dictionary is displayed ⑦. In the *sum\_dictionary\_values()* function ⑧, all the values associated with the key is obtained using *values()* method, which, in turn, is summed up using *sum()* function ⑩.

**Program 7.4: Write Python Program to Count the Number of Times an Item Appears in the List**

```

1. novels = ["gone_girl", "davinci_code", "games_of_thrones", "gone_girl",
 "davinci_code"]
2. def main():
3. count_items = dict()
4. for book_name in novels:
5. count_items[book_name] = count_items.get(book_name, 0) + 1
6. print("Number of times a novel appears in the list is")
7. print(count_items)
8. if __name__ == "__main__":
9. main()

```

**OUTPUT**

```

Number of times a novel appears in the list is
{'gone_girl': 2, 'davinci_code': 2, 'games_of_thrones': 1}

```



You can count the number of items in a list. Start with initializing *count\_items* to an empty dictionary ③. Loop through each item of the *novels* list ④ using *for* loop ④. Each unique item in the list is considered as a *key*, and the number of times these items appear will be its associated value. The *get()* dictionary method takes a default value of zero if the *key* is not present in the dictionary to which a value of one is added. This value is then assigned to *count\_items* dictionary key as its associated value ⑤. If the *key* is already present in the *count\_items* dictionary, then *get()* method returns its corresponding value, which is subsequently incremented by one and this latest value is associated with the *key* by overwriting its existing value ⑤.

**Program 7.5: Write Python Program to Count the Number of Times Each Word Appears in a Sentence**

```

1. def main():
2. count_words = dict()
3. sentence = input("Enter a sentence ")
4. words = sentence.split()
5. for each_word in words:
6. count_words[each_word] = count_words.get(each_word, 0) + 1
7. print("The number of times each word appears in a sentence is")
8. print(count_words)
9. if __name__ == "__main__":
10. main()

```

**OUTPUT**

```

Enter a sentence: Everyone needs a little inspiration from time to time
The number of times each word appears in a sentence is
{'Everyone': 1, 'needs': 1, 'a': 1, 'little': 1, 'inspiration': 1, 'from': 1, 'time': 2, 'to': 1}

```

In the *main()* function, you start by initializing *count\_words* to empty dictionary ②. The user enters a sentence ③, which is split into a list of words using the *split()* function ④. Loop through each word in the *words* list ⑤. If the word is not present as a key in the dictionary, then the *get()* method takes a default value of zero to which a value of one is added and assigned to *count\_words* dictionary with the key being the value of the iterating variable. If the word is already present as a key in the dictionary, then the value associated with the key is incremented by one and this latest value is associated with the key by overwriting the existing value ⑥.

**Program 7.6: Write Python Program to Count the Number of Characters in a String Using Dictionaries. Display the Keys and Their Values in Alphabetical Order**

```

1. def construct_character_dict(word):
2. character_count_dict = dict()
3. for each_character in word:
4. character_count_dict[each_character] = character_count_dict.get(each_
 character, 0) + 1

```

```

5. sorted_list_keys = sorted(character_count_dict.keys())
6. for each_key in sorted_list_keys:
7. print(each_key, character_count_dict.get(each_key))
8. def main():
9. word = input("Enter a string ")
10. construct_character_dict(word)
11. if __name__ == "__main__":
12. main()

```

**OUTPUT**

Enter a string: massachussetts

```

a 2
c 1
e 1
h 1
m 1
s 5
t 2
u 1

```

In the *main()* function ⑧, the user enters a string ⑨ that is passed as argument to the *construct\_character\_dict()* ⑩ function. In the *construct\_character\_dict()* function, start by initializing *character\_count\_dict* to empty the dictionary ②. Loop through each character in the *word* string ③. If the character is not present as a key in the dictionary, then the *get()* method takes a default value of zero to which a value of one is added and is assigned to *character\_count\_dict* dictionary with the key being the iterating variable ④. If the character is already present as a key in the dictionary, then the value associated with the key is incremented by one and this latest value is associated with the key by overwriting the existing value ④. The keys are sorted in ascending order using the *sorted()* function, and the sorted list of *key:value* pairs are assigned to *sorted\_list\_keys* ⑤. The number of times a character appearing within the user-entered word is displayed by looping through each key in the *sorted\_list\_keys* dictionary ⑥. The value for each key in the *sorted\_list\_keys* dictionary is displayed using *get()* method ⑦.

**Program 7.7: Write Python Program to Generate a Dictionary That Contains (i: i\*i) Such that i Is a Number Ranging from 1 to n.**

```

1. def main():
2. number = int(input("Enter a number "))
3. create_number_dict = dict()
4. for i in range(1, number+1):
5. create_number_dict[i] = i * i
6. print("The generated dictionary of the form (i: i*i) is")
7. print(create_number_dict)
8. if __name__ == "__main__":
9. main()

```

**OUTPUT**

Enter a number 3

The generated dictionary of the form (i: i\*i) is

{1: 1, 2: 4, 3: 9}

The user enters a number that is stored in *number* variable ②. Then initializes *create\_number\_dict* to empty the dictionary ③. The numbers ranging from 1 to *number + 1* are kept as keys ④, while the squares of these numbers are their values ⑤. Finally, the dictionary is printed ⑦.

**Program 7.8: Write a Program That Accepts a Sentence and Calculate the Number of Digits, Uppercase and Lowercase Letters**

```

1. def main():
2. sentence = input("Enter a sentence ")
3. construct_dictionary = {"digits": 0, "lowercase": 0, "uppercase": 0}
4. for each_character in sentence:
5. if each_character.isdigit():
6. construct_dictionary["digits"] += 1
7. elif each_character.isupper():
8. construct_dictionary["uppercase"] += 1
9. elif each_character.islower():
10. construct_dictionary["lowercase"] += 1
11. print("The number of digits, lowercase and uppercase letters are")
12. print(construct_dictionary)
13. if __name__ == "__main__":
14. main()

```

**OUTPUT**

Enter a sentence I am Time, the great destroyer of the world - Bhagavad Gita 11.32

The number of digits, lowercase and uppercase letters are

{'digits': 4, 'lowercase': 42, 'uppercase': 4}

In the *main()* function ①, the user enters a sentence that is stored in *sentence* variable ②. The *construct\_dictionary* has three keys *digits*, *lowercase* and *uppercase* with each of them having zero as their initial value ③. Loop through each character in the sentence ④. If the character is a digit, then the value for *digits* key in *construct\_dictionary* is incremented by one ⑤–⑥. If the character is in lowercase, then the value for *lowercase* key in *construct\_dictionary* is incremented by one ⑦–⑧. If the character is in uppercase, then the value for *uppercase* key in *construct\_dictionary* is incremented by one ⑨–⑩. Finally, *key:value* pair in the *construct\_dictionary* dictionary is displayed ⑪–⑫.

**Program 7.9: Write a Python Program to Input Information for  $n$  number of Students as Given Below:**

- a. Name**
- b. Registration Number**
- c. Total Marks**

**The user has to specify a value for  $n$  number of students. The Program Should Output the Registration Number and Marks of a Specified Student Given His Name**

```
1. def student_details(number_of_students):
2. student_name = {}
3. for i in range(0, number_of_students):
4. name = input("Enter the Name of the Student ")
5. registration_number = input("Enter student's Registration Number ")
6. total_marks = input("Enter student's Total Marks ")
7. student_name[name] = [registration_number, total_marks]
8. student_search = input('Enter name of the student you want to search ')
9. if student_search not in student_name.keys():
10. print('Student you are searching is not present in the class')
11. else:
12. print("Student you are searching is present in the class")
13. print(f"Student's Registration Number is {student_name[student_search][0]}")
14. print(f"Student's Total Marks is {student_name[student_search][1]}")
15. def main():
16. number_of_students = int(input("Enter the number of students "))
17. student_details(number_of_students)
18. if __name__ == "__main__":
19. main()
```

#### OUTPUT

```
Enter the number of students 2
Enter the Name of the Student jack
Enter student's Registration Number 1AIT18CS05
Enter student's Total Marks 89
Enter the Name of the Student jill
Enter student's Registration Number 1AIT18CS08
Enter student's Total Marks 91
Enter name of the student you want to search jill
Student you are searching is present in the class
Student's Registration Number is 1AIT18CS08
Student's Total Marks is 91
```

In the `main()` function ⑤, the user enters the number of students ⑥, which is then passed as an argument to `student_details()` function ⑦. In the `student_details()` function definition, initialize `student_name` to empty the dictionary ②. Use `for` loop to read and assign student details to `name`, `registration_number`, and `total_marks` variables ③–⑥. The `student_name` dictionary is built by having `name` variable value as the key, while `registration_number` and `total_marks` variables are associated as `list` values to the key ⑦. The name of the student to be searched is assigned to `student_search` variable ⑧. If ⑨ the `student_search` variable value is not present as a key in the `student_name` dictionary, then “Student you are searching is not present in the class” message is displayed ⑩. If the `student_search` variable value is present as a key in the dictionary ⑪, then registration number is retrieved by specifying the `student_search` variable value as a key in `student_name` dictionary followed by the list index value of zero ⑬. An index value of one retrieves the total marks of the student ⑭.

### Program 7.10: Program to Demonstrate Nested Dictionaries

```

1. student_details = {"name": "jasmine", "registration_number": "1AIT18CS05", "sub_
 marks": {"python": 95, "java": 90, ".net": 85}}
2. def nested_dictionary():
3. print(f"Student Name {student_details['name']}")
4. print(f"Registration Number {student_details['registration_number']}")
5. average = sum(student_details["sub_marks"].values()) /
 len(student_details["sub_marks"])
6. print(f"Average of all the subjects is {average}")
7. def main():
8. nested_dictionary()
9. if __name__ == "__main__":
10. main()

```

#### OUTPUT

```

Student Name jasmine
Registration Number 1AIT18CS05
Average of all the subjects 90.0

```

The use of dictionaries within dictionaries is called nesting of dictionaries. You can assign a dictionary as a value to a key inside another dictionary ①. A dictionary is associated as a value to the `sub_marks` dictionary ①. Inside the `nested_dictionary()` ② function, student name, registration number, and average marks are displayed ③–⑥. The value returned from `student_details["sub_marks"]` is of dictionary type i.e.,

```

>>> student_details["sub_marks"]
{'python': 95, 'java': 90, '.net': 85}.

```

To access all the values of the dictionary associated with a `sub_marks` key in the `student_details` dictionary, you need to specify the name of `student_details` dictionary followed by

*sub\_marks* as the key and combine it with *values()* method using dot notation ⑤. Even though a single level of nesting dictionaries may prove useful, having multiple levels of nesting may make the code unreadable.

---

## 7.5 The *del* Statement

To delete the *key:value* pair, use the *del* statement followed by the name of the dictionary along with the *key* you want to delete.

*del dict\_name[key]*

```
1. >>> animals = {'r':"raccoon", "c':"cougar", "m':"moose"}
2. >>> animals
 {'r': 'raccoon', 'c': 'cougar', 'm': 'moose'}
3. >>> del animals["c"]
4. >>> animals
 {'r': 'raccoon', 'm': 'moose'}
```

In the *animals* ①–② dictionary, you can remove the *key:value* pair of "c": "cougar" as shown in ③.

---

## 7.6 Summary

- A dictionary associates a set of keys with values.
- The built-in function *dict()* returns a new dictionary initialized from an optional keyword argument and a possibly empty set of keyword arguments.
- The *for* loop is used to traverse all the keys in the dictionary.
- The *del dictionaryName[key]* statement is used to delete an item for the given key.
- Dictionary methods like *keys()*, *values()*, and *items()* are used to retrieve the values.
- Methods like *pop()* and *update()* are used to manipulate the dictionary *key:value* pairs.

---

## Multiple Choice Questions

1. Which of the following statements create a dictionary?
  - a. `dic = {}`
  - b. `dic = {"charles":40, "peterson":45}`
  - c. `dic = {40: "charles", 45: "peterson"}`
  - d. All of the above

2. Read the code shown below carefully and pick out the keys.

```
dic = {"game":40, "thrones":45}
```

- a. "game", 40, 45, and "thrones"
  - b. "game" and "thrones"
  - c. 40 and 45
  - d. dic = (40: "game", 45: "thrones")
3. Gauge the output of the following code snippet.
- ```
fruit = {"apple":"red", "guava":"green"}  
"apple" in fruit
```
- a. True
 - b. False
 - c. None
 - d. Error
4. Consider phone_book = {"Kalpana":7766554433, "Steffi":4499551100}. To delete the key "Kalpana" the code used is
- a. phone_book.delete("Kalpana":7766554433)
 - b. phone_book.delete("Kalpana")
 - c. del phone_book["Kalpana"]
 - d. del phone_book("Kalpana":7766554433)
5. Assume d = {"Guido":"Python", "Dennis":"C"}. To obtain the number of entries in dictionary the statement used is
- a. d.size()
 - b. len(d)
 - c. size(d)
 - d. d.len()
6. Consider stock_prices = {"IBM":220, "FB":800}. What happens when you try to retrieve a value using the statement stock_prices["IBM"]?
- a. Since "IBM" is not a value in the set, Python raises a KeyError exception.
 - b. It executes fine and no exception is raised
 - c. Since "IBM" is not a key in the set, Python raises a KeyError exception.
 - d. Since "IBM" is not a key in the set, Python raises a syntax error.
7. Which of the following statement is false about the dictionary?
- a. The values of a dictionary can be accessed using keys.
 - b. The keys of a dictionary can be accessed using values.
 - c. Dictionaries are not ordered.
 - d. Dictionaries are mutable.

8. What is the output of the following code?

```
stuff = {"book": "Java", "price": 45}
stuff.get("book")
```

- a. 45
- b. True
- c. Java
- d. price

9. Predict the output of the following code.

```
fish = {'g': "Goldfish", 's': "Shark"}
fish.pop(s)
print(fish)
```

- a. {'g': 'Goldfish', 's': 'Shark'}
- b. {'s': 'Shark'}
- c. {'g': 'Goldfish'}
- d. Error

10. The method that returns the value for the key present in the dictionary and if the key is not present then it inserts the key with default value into the dictionary.

- a. update()
- b. fromkeys()
- c. setdefault()
- d. get()

11. Guess the output of the following code.

```
grades = {90: "S", 80: "A"}
del grades
```

- a. Method *del* doesn't exist for the dictionary.
- b. *del* deletes the values in the dictionary.
- c. *del* deletes the entire dictionary.
- d. *del* deletes the keys in the dictionary.

12. Assume *dic* is a dictionary with some *key:value* pairs. What does *dic.popitem()* do?

- a. Removes an arbitrary *key:value* pair
- b. Removes all the *key:value* pairs
- c. Removes the *key:value* pair for the key given as an argument
- d. Invalid method

13. What will be the output of the following code snippet?

```
numbers = {}
letters = {}
comb = {}
numbers[1] = 56
numbers[3] = 7
letters[4] = "B"
comb["Numbers"] = numbers
comb["Letters"] = letters
print(comb)
```

- a. Nested dictionary cannot occur
- b. 'Numbers': {1: 56, 3: 7}
- c. {'Numbers': {1: 56}, 'Letters': {4: 'B'}}
- d. {'Numbers': {1: 56, 3: 7}, 'Letters': {4: 'B'}}

14. Gauge the output of the following code.

```
demo = {1: 'A', 2: 'B', 3: 'C'}
del demo[1]
demo[1] = 'D'
del demo[2]
print(len(demo))
```

- a. 0
- b. 2
- c. Error
- d. 1

15. Assuming *b* to be a dictionary, what does *any(b)* do?

- a. Returns True if any key of the dictionary is True.
- b. Returns False if dictionary is empty.
- c. Returns True if all keys of the dictionary are True.
- d. Method *any()* doesn't exist for dictionary.

16. Infer the output of the following code.

```
count = {}
count[(1, 2, 4)] = 5
count[(4, 2, 1)] = 7
count[(1, 2)] = 6
count[(4, 2, 1)] = 2
tot = 0
```

```
for i in count:
    tot = tot + count[i]
print(len(count)+tot)
```

- a. 25
 - b. 17
 - c. 16
 - d. Error
17. The _____ function returns Boolean True value if all the keys in the dictionary are True else returns False.
- a. all()
 - b. sorted()
 - c. len()
 - d. any()
18. Predict the output of the following code.
- ```
>>> dic = {}
>>> dic.fromkeys([1,2,3], "check")
```
- a. Syntax error
  - b. {1: 'check', 2: 'check', 3: 'check'}
  - c. 'check'
  - d. {1:None, 2:None, 3:None}
19. For dictionary d = { "plum ":0.66, "pears ":1.25,"oranges ":0.49}, which of the following statement correctly updates the price of oranges to 0.52?
- a. d[2] = 0.52
  - b. d[0.49] = 0.52
  - c. d["oranges "] = 0.52
  - d. d["plum "] = 0.52
20. The syntax that is used to modify or add a new key: value pair to a dictionary is:
- a. dictionary\_name[key] = value
  - b. dictionary\_name[value] = key
  - c. dictionary\_name(key) = value
  - d. dictionary\_name{key} = value
21. Which of the following cannot be used as a key in Python dictionaries?
- a. Strings
  - b. Lists
  - c. Tuples
  - d. Numerical values

22. Guess the output of the following code.

```
week = {1:"sunday", 2:"monday", 3:"tuesday"}
for i,j in week.items():
 print(i, j)
```

- a. 1 sunday 2 monday 3 Tuesday
- b. 1 2 3
- c. sunday monday tuesday
- d. 1:"sunday" 2:"monday" 3:"tuesday"

23. Predict the output of the following code.

```
a = {1: "A", 2: "B", 3: "C"}
b = {4: "D", 5: "E"}
a.update(b)
print(a)
```

- a. {1: 'A', 2: 'B', 3: 'C'}
- b. Error
- c. {4: 'D', 5: 'E'}
- d. {1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E'}

---

## Review Questions

1. Define a dictionary. What are the advantages of using dictionary over lists.
2. Briefly explain how a dictionary is created with an example.
3. Write short notes on the following methods.
  - a. keys()
  - b. values()
  - c. get(key)
  - d. clear()
4. Explain nested dictionaries with an example.
5. Write a function that prompts the user for the average temperature for each day of the week and returns a dictionary containing the entered information.
6. Write a Python program to input information about a few employees as given below:
  - a. Name
  - b. Employee Id
  - c. Salary

The program should output the employee ID and salary of a specified employee, given his name.

7. Write a function named *addfruit*, which is passed with a set of fruit names and their prices and returns a dictionary containing the entered information and raises a *ValueError* exception if the fruit is already present.
8. Write a function to add the air quality index as the value and the date as the key; create the dictionary for the entered information.
9. Create a dictionary that contains usernames as the key and passwords as the associated values. Make up the data for five dictionary entries and demonstrate the use of *clear* and *fromkeys()* methods.
10. Write Pythonic code to create a dictionary that accepts a country name as a key and its capital city as the value. Display the details in sorted order.
11. Write a program that has the dictionary of your friends' names as keys and phone numbers as its values. Print the dictionary in a sorted order. Prompt the user to enter the name and check if it is present in the dictionary. If the name is not present, then enter the details in the dictionary.
12. Write a program to create a dictionary containing the author name as the keys and ISBN number as the value. Make up the data for five dictionary entries and demonstrate the use of *clear()* and *fromkeys()* methods.



**Taylor & Francis**

Taylor & Francis Group

<http://taylorandfrancis.com>

# 8

## Tuples and Sets

### AIM

Understand the role of tuples in Python when storing a finite sequence of homogeneous or heterogeneous data of fixed sizes and performing manipulations using various methods.

### LEARNING OUTCOMES

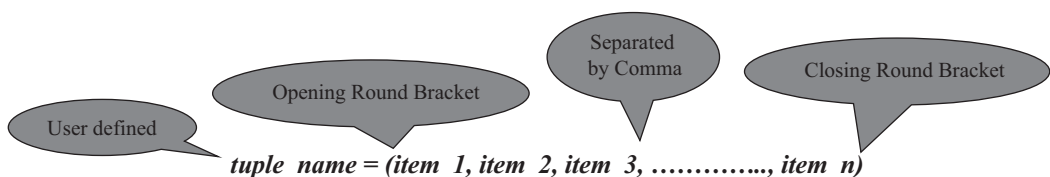
At the end of the chapter, you are expected to

- Create and manipulate items in tuples.
- Use *for* loop to access individual items in tuples.
- Understand the relation between dictionaries and tuples.
- Understand the relation between lists and tuples.
- Apply mathematical operations like union and intersection using sets.

In mathematics, a tuple is a finite ordered list (sequence) of elements. A tuple is defined as a data structure that comprises an ordered, finite sequence of immutable, heterogeneous elements that are of fixed sizes. Often, you may want to return more than one value from a function. Tuples solve this problem. They can also be used to pass multiple values through one function parameter.

### 8.1 Creating Tuples

A tuple is a finite ordered list of values of possibly different types which is used to bundle related values together without having to create a specific type to hold them. Tuples are immutable. Once a tuple is created, you cannot change its values. A tuple is defined by putting a comma-separated list of values inside parentheses ( ). Each value inside a tuple is called an item. The syntax for creating tuples is,



For example,

```

1. >>> internet = ("cern", "timbernerslee", "www", 1980)
2. >>> internet
('cern', 'timbernerslee', 'www', 1980)
3. >>> type(internet)
<class 'tuple'>
4. >>> f1 = "ferrari", "redbull", "mercedes", "williams", "renault"
5. >>> f1
('ferrari', 'redbull', 'mercedes', 'williams', 'renault')
6. >>> type(f1)
<class 'tuple'>

```

In ①, the tuple *internet* is assigned with a sequence of numbers and string types. The contents of the tuple are displayed by executing the tuple name *internet* ②. The contents and order of items in the tuple are the same as they were when the tuple was created. In Python, the tuple type is called *tuple* ③. On output, tuples are always enclosed in parentheses, so that they are interpreted correctly. Tuples can be constructed with or without surrounding parentheses, although often parentheses are used anyway. It is actually the comma that forms a tuple making the commas significant and not the parentheses ④–⑥.

You can create an empty tuple without any values. The syntax is,

```
tuple_name = ()
```

For example,

```

1. >>> empty_tuple = ()
2. >>> empty_tuple
()
3. >>> type(empty_tuple)
<class 'tuple'>

```

An empty tuple can be created as shown in ① and *empty\_tuple* is of *tuple* type ③.



You can store any item of type string, number, object, another variable, and even another tuple itself. You can have a mix of different types of items in tuples, and they need not be homogeneous.

For example,

```

1. >>> air_force = ("f15", "f22a", "f35a")
2. >>> fighter_jets = (1988, 2005, 2016, air_force)
3. >>> fighter_jets
(1988, 2005, 2016, ('f15', 'f22a', 'f35a'))
4. >>> type(fighter_jets)
<class 'tuple'>

```

Here *air\_force* is of tuple type ①. The tuple *fighter\_jets* ④ consists of a sequence of integer types along with a tuple ②. A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. For example,

```
1. >>> empty = ()
2. >>> singleton = 'hello',
3. >>> singleton
('hello')
```

Empty tuples are constructed by an empty pair of parentheses ①. A tuple with one item is constructed by having a value followed by a comma ②. It is not sufficient to enclose a single value in parentheses.

---

## 8.2 Basic Tuple Operations

Like in lists, you can use the + operator to concatenate tuples together and the \* operator to repeat a sequence of tuple items.

For example,

```
1. >>> tuple_1 = (2, 0, 1, 4)
2. >>> tuple_2 = (2, 0, 1, 9)
3. >>> tuple_1 + tuple_2
(2, 0, 1, 4, 2, 0, 1, 9)
4. >>> tuple_1 * 3
(2, 0, 1, 4, 2, 0, 1, 4, 2, 0, 1, 4)
5. >>> tuple_1 == tuple_2
False
```

Two tuples, *tuple\_1* and *tuple\_2*, consisting of integer type are created ①–②. The *tuple\_1* and *tuple\_2* tuples are added to form a new tuple. The new tuple has all items of both the added tuples ③. You can use the multiplication operator \* on tuples. The items in the tuples are repeated the number of times you specify, and in ④ the *tuple\_1* items are repeated three times. In ⑤, the items in tuples are compared using == equal to operator, and the result is Boolean *False* since the items in the tuples are different.

You can check for the presence of an item in a tuple using *in* and *not in* membership operators. It returns a Boolean *True* or *False*. For example,

```
1. >>> tuple_items = (1, 9, 8, 8)
2. >>> 1 in tuple_items
True
3. >>> 25 in tuple_items
False
```

The *in* operator returns Boolean *True* if an item is present in the tuple ②, or else returns *False* Boolean value ③.



Comparison operators like `<`, `<=`, `>`, `>=`, `==` and `!=` are used to compare tuples. For example,

```
1. >>> tuple_1 = (9, 8, 7)
2. >>> tuple_2 = (9, 1, 1)
3. >>> tuple_1 > tuple_2
 True
4. >>> tuple_1 != tuple_2
 True
```

Tuples are compared position by position. The first item of the first tuple is compared to the first item of the second tuple; if they are not equal then this will be the result of the comparison, or else the second item of the first tuple is compared to the second item of the second tuple; if they are not equal then this will be the result of the comparison, else the third item is considered, and so on ①–④.

### 8.2.1 The *tuple()* Function

The built-in *tuple()* function is used to create a tuple. The syntax for the *tuple()* function is,

***tuple([sequence])***

where the sequence can be a number, string or tuple itself. If the optional sequence is not specified, then an empty tuple is created.

```
1. >>> norse = "vikings"
2. >>> string_to_tuple = tuple(norse)
3. >>> string_to_tuple
 ('v', 'i', 'k', 'i', 'n', 'g', 's')
4. >>> zeus = ["g", "o", "d", "o", "f", "s", "k", "y"]
5. >>> list_to_tuple = tuple(zeus)
6. >>> list_to_tuple
 ('g', 'o', 'd', 'o', 'f', 's', 'k', 'y')
7. >>> string_to_tuple + "scandinavia"
 Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 TypeError: can only concatenate tuple (not "str") to tuple
8. >>> string_to_tuple + tuple("scandinavia")
 ('v', 'i', 'k', 'i', 'n', 'g', 's', 's', 'c', 'a', 'n', 'd', 'i', 'n', 'a', 'v', 'i', 'a')
9. >>> list_to_tuple + ["g", "r", "e", "e", "k"]
 Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 TypeError: can only concatenate tuple (not "list") to tuple
10. >>> list_to_tuple + tuple(["g", "r", "e", "e", "k"])
 ('g', 'o', 'd', 'o', 'f', 's', 'k', 'y', 'g', 'r', 'e', 'e', 'k')
11. >>> letters = ("a", "b", "c")
12. >>> numbers = (1, 2, 3)
13. >>> nested_tuples = (letters, numbers)
```

```

14. >>> nested_tuples
 (('a', 'b', 'c'), (1, 2, 3))
15. >>> tuple("wolverine")
 ('w', 'o', 'l', 'v', 'e', 'r', 'i', 'n', 'e')

```

The string variable ① is converted to a tuple using the *tuple()* function ②. Not only strings, but even list variables ④ can be converted to tuples ⑤. If you try to concatenate either a string ⑦ or a list ⑨ with tuples, then it results in an error. You have to convert strings and lists to tuples using *tuple()* function before concatenating with tuple types ⑧ and ⑩. Nesting of tuples is allowed in Python. An output of a tuple operation is always enclosed in parentheses so that nested tuples are interpreted correctly ⑪–⑭. Each individual character in a string is separated by a comma when a string is converted to a tuple ⑮.

### 8.3 Indexing and Slicing in Tuples

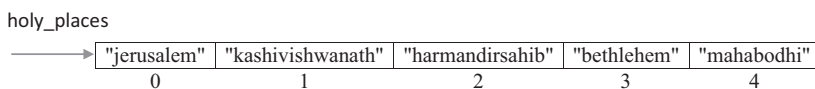
Each item in a tuple can be called individually through indexing. The expression inside the bracket is called the index. Square brackets [ ] are used by tuples to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed.

The syntax for accessing an item in a tuple is,

***tuple\_name[index]***

where index should always be an integer value and indicates the item to be selected.

For the tuple *holy\_places*, the index breakdown is shown below.



For example,

```

1. >>> holy_places = ("jerusalem", "kashivishwanath", "harmandirsahib", "bethle-
 hem", "mahabodhi")
2. >>> holy_places
 ('jerusalem', 'kashivishwanath', 'harmandirsahib', 'bethlehem', 'mahabodhi')
3. >>> holy_places[0]
 'jerusalem'
4. >>> holy_places[1]
 'kashivishwanath'
5. >>> holy_places[2]
 'harmandirsahib'
6. >>> holy_places[3]
 'bethlehem'
7. >>> holy_places[4]
 'mahabodhi'

```

8. >>> holy\_places[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: tuple index out of range

The first item in the tuple is displayed by using square brackets immediately after the tuple name with an index value of zero ①. The range of index numbers in this tuple is 0 to 4. If the index value is more than the number of items in the tuple ⑧, then it results in “*IndexError: tuple index out of range*” error.

In addition to positive index numbers, you can also access tuple items using a negative index number, by counting backwards from the end of the tuple, starting at -1. Negative indexing is useful if you have a large number of items in the tuple and you want to locate an item towards the end of a tuple.

For the same tuple *holy\_places*, the negative index breakdown is shown below.

|             |             |                   |                  |             |             |  |
|-------------|-------------|-------------------|------------------|-------------|-------------|--|
| holy_places |             |                   |                  |             |             |  |
| →           | "jerusalem" | "kashivishwanath" | "harmandirsaheb" | "bethlehem" | "mahabodhi" |  |
|             | -5          | -4                | -3               | -2          | -1          |  |

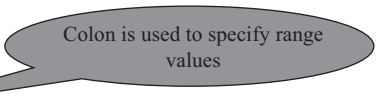
For example,

1. >>> holy\_places[-2]  
'bethlehem'

If you would like to print out the item 'bethlehem' by using its negative index number, you can do so as in ①.

**Slicing** of tuples is allowed in Python wherein a part of the tuple can be extracted by specifying an index range along with the colon (:) operator, which itself results as tuple type.

The syntax for tuple slicing is,


 Colon is used to specify range values  
***tuple\_name[start:stop[:step]]***

where both *start* and *stop* are integer values (positive or negative values). Tuple slicing returns a part of the tuple from the *start* index value to *stop* index value, which includes the *start* index value but excludes the *stop* index value. The *step* specifies the increment value to slice by and it is optional.

For the tuple *colors*, the positive and negative index breakdown is shown below.

|        |     |     |     |     |     |     |     |
|--------|-----|-----|-----|-----|-----|-----|-----|
| colors |     |     |     |     |     |     |     |
| →      | "v" | "i" | "b" | "g" | "y" | "o" | "r" |
|        | 0   | 1   | 2   | 3   | 4   | 5   | 6   |
|        | -7  | -6  | -5  | -4  | -3  | -2  | -1  |

For example,

1. >>> colors = ("v", "i", "b", "g", "y", "o", "r")  
2. >>> colors  
( 'v', 'i', 'b', 'g', 'y', 'o', 'r' )

```
3. >>> colors[1:4]
('i', 'b', 'g')
4. >>> colors[:5]
('v', 'i', 'b', 'g', 'y')
5. >>> colors[3:]
('g', 'y', 'o', 'r')
6. >>> colors[:]
('v', 'i', 'b', 'g', 'y', 'o', 'r')
7. >>> colors[:]
('v', 'i', 'b', 'g', 'y', 'o', 'r')
8. >>> colors[1:5:2]
('i', 'g')
9. >>> colors[::2]
('v', 'b', 'y', 'r')
10. >>> colors[::-1]
('r', 'o', 'y', 'g', 'b', 'i', 'v')
11. >>> colors[-5:-2]
('b', 'g', 'y')
```

The *colors* tuple has seven items of string type. All the items in the *colors* tuple starting from an index value of 1 up to an index value of 4, but excluding the index value of 4 is sliced ③. If you want to access the tuple item from *start* index, then there is no need to specify an index value of zero. You can skip the *start* index value and specify only the *stop* index value ④. Similarly, if you want to access the items in a tuple up to the end of the tuple, then there is no need to specify the *stop* value and you must mention only the *start* index value ⑤. If you skip both the *start* and *stop* index values ⑥ and specify only the colon operator within the brackets, then all the items in the tuple are displayed. The use of double colons instead of a single colon also displays the entire contents of the tuple ⑦. The number after the second colon tells Python that you would like to choose your slicing increment. By default, Python sets this increment to 1, but that number after the second colon allows you to specify what you want it to be ⑧. Using a double colon as shown in ⑨, means no value for *start* index and no value for *stop* index and jumps the items by two steps. Every second item of the tuple is extracted starting from an index value of zero. All the items in a tuple can be displayed in reverse order by specifying a double colon followed by an index value of -1 ⑩. Negative index values can also be used for *start* and *stop* index values ⑪.

---

## 8.4 Built-In Functions Used on Tuples

There are many built-in functions ([TABLE 8.1](#)) for which a tuple can be passed as an argument.

**TABLE 8.1**

Built-In Functions Used on Tuples

| Built-In Functions    | Description                                                                                                           |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>len()</code>    | The <i>len()</i> function returns the numbers of items in a tuple.                                                    |
| <code>sum()</code>    | The <i>sum()</i> function returns the sum of numbers in the tuple.                                                    |
| <code>sorted()</code> | The <i>sorted()</i> function returns a sorted copy of the tuple as a list while leaving the original tuple untouched. |

For example,

```
1. >>> years = (1987, 1985, 1981, 1996)
2. >>> len(years)
4
3. >>> sum(years)
7949
4. >>> sorted_years = sorted(years)
5. >>> sorted_years
[1981, 1985, 1987, 1996]
```

You can find the number of items in the tuple *years* by using the *len()* function ②. Using the *sum()* function results in the adding up of all the number items in the tuple ③. The *sorted()* function returns the sorted list of items without modifying the original tuple which is assigned to new list variable ④. In the case of string items in a tuple, they are sorted based on their ASCII values.

## 8.5 Relation between Tuples and Lists

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually, contain a heterogeneous sequence of elements that are accessed via unpacking or indexing. Lists are mutable, and their items are accessed via indexing. Items cannot be added, removed or replaced in a tuple.

For example,

```
1. >>> coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")
2. >>> coral_reef[0] = "pickles_reef"
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
3. >>> coral_reef_list = list(coral_reef)
4. >>> coral_reef_list
['great_barrier', 'ningaloo_coast', 'amazon_reef', 'pickles_reef']
```

In the tuple *coral\_reef*, if you try to replace “great\_barrier” with a different item, like “pickles\_reef”, then it results in *TypeError* as tuples cannot be modified ①–②. You can convert a tuple to a list by passing the tuple name to the *list()* function ③–④.

If an item within a tuple is mutable, then you can change it. Consider the presence of a list as an item in a tuple, then any changes to the list get reflected on the overall items in the tuple. For example,

```
1. >>> german_cars = ["porsche", "audi", "bmw"]
2. >>> european_cars = ("ferrari", "volvo", "renault", german_cars)
3. >>> european_cars
('ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw'])
4. >>> european_cars[3].append("mercedes")
5. >>> german_cars
['porsche', 'audi', 'bmw', 'mercedes']
6. >>> european_cars
('ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw', 'mercedes'])
```

The tuple "containing" a list seems to change when the underlying list changes. The key insight is that tuples have no way of knowing whether the items inside them are mutable. The only thing that makes an item mutable is to have a method that alters its data. In general, there is no way to detect this. In fact, the tuple did not change. It could not change because it does not have mutating methods. When the list changes, the tuple does not get notified of the change. The list does not know whether it is referred to by a variable, a tuple, or another list ①–⑥. The tuple itself is not mutable as it does not have any methods that can be used for changing its contents. Likewise, the string is immutable because strings do not have any mutating methods.

---

## 8.6 Relation between Tuples and Dictionaries

Tuples can be used as *key:value* pairs to build dictionaries.

For example,

```
1. >>> fish_weight_kg = (("white_shark", 520), ("beluga", 1571), ("greenland_shark",
1400))
2. >>> fish_weight_kg_dict = dict(fish_weight_kg)
3. >>> fish_weight_kg_dict
{'white_shark': 520, 'beluga': 1571, 'greenland_shark': 1400}
```

The tuples can be converted to dictionaries by passing the tuple name to the *dict()* function. This is achieved by nesting tuples within tuples, wherein each nested tuple item should have two items in it ①–②. The first item becomes the key and second item as its value when the tuple gets converted to a dictionary ③.

The method *items()* in a dictionary returns a list of tuples where each tuple corresponds to a *key:value* pair of the dictionary. For example,

```
1. >>> founding_year = {"Google":1996, "Apple":1976, "Sony":1946, "ebay":1995, "IBM":1911}
2. >>> founding_year.items()
dict_items([('Google', 1996), ('Apple', 1976), ('Sony', 1946), ('ebay', 1995), ('IBM', 1911)])
```

```
3. >>> for company, year in founding_year.items():
... print(f"{company} was found in the year {year}")
```

**OUTPUT**

```
Google was found in the year 1996
Apple was found in the year 1976
Sony was found in the year 1946
ebay was found in the year 1995
IBM was found in the year 1911
```

The *for* loop in ③ has two iteration variables—*company* and *year*, because the *items()* method returns a new view of the dictionary's key and value pairs as tuples, which successively iterates through each of the *key:value* pairs in the dictionary.

---

## 8.7 Tuple Methods

You can get a list of all the methods associated with the *tuple* (TABLE 8.2) by passing the *tuple* function to *dir()*.

```
1. >>> dir(tuple)
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
```

Various methods associated with *tuple* are displayed ①.

**TABLE 8.2**

Various Tuple Methods

| Tuple Methods | Syntax                 | Description                                                                                                                                                                                                                                                                      |
|---------------|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| count()       | tuple_name.count(item) | The <i>count()</i> method counts the number of times the item has occurred in the tuple and returns it.                                                                                                                                                                          |
| index()       | tuple_name.index(item) | The <i>index()</i> method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple, then <i>ValueError</i> is thrown by this method. |

*Note:* Replace the word “*tuple\_name*” mentioned in the syntax with your *actual tuple name* in your code.

For example,

```
1. >>> channels = ("ngc", "discovery", "animal_planet", "history", "ngc")
2. >>> channels.count("ngc")
2
3. >>> channels.index("history")
3
```

Various operations on tuples are carried out using tuple methods ①–③.

### 8.7.1 Tuple Packing and Unpacking

The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing.

```
1. >>> t = 12345, 54321, 'hello!'
2. >>> t
(12345, 54321, 'hello!')
```

The values 12345, 54321 and 'hello!' are packed together into a tuple ①–②.

The reverse operation of tuple packing is also possible. For example,

```
1. >>> x, y, z = t
2. >>> x
12345
3. >>> y
54321
4. >>> z
'hello!'
```

This operation is called tuple unpacking and works for any sequence on the right-hand side. Tuple unpacking requires that there are as many variables on the left side of the equals sign as there are items in the tuple ①–④. Note that multiple assignments are really just a combination of tuple packing and unpacking.

### 8.7.2 Traversing of Tuples

You can iterate through each item in tuples using *for* loop.

#### Program 8.1: Program to Iterate Over Items in Tuples Using for Loop

```
1. ocean_animals = ("electric_eel", "jelly_fish", "shrimp", "turtles", "blue_whale")
2. def main():
3. for each_animal in ocean_animals:
4. print(f'{each_animal} is an ocean animal')
5. if __name__ == "__main__":
6. main()
```

#### OUTPUT

```
electric_eel is an ocean animal
jelly_fish is an ocean animal
shrimp is an ocean animal
turtle is an ocean animal
blue_whale is an ocean animal
```

Here `ocean_animals` is of tuple type ①. A *for* ③ loop is used to iterate through each item in the tuple.



### 8.7.3 Populating Tuples with Items

You can populate tuples with items using += operator and also by converting list items to tuple items.

#### Program 8.2: Program to Populate Tuple with User-Entered Items

```
1. tuple_items = ()
2. total_items = int(input("Enter the total number of items: "))
3. for i in range(total_items):
4. user_input = int(input("Enter a number: "))
5. tuple_items += (user_input,)
6. print(f"Items added to tuple are {tuple_items}")

7. list_items = []
8. total_items = int(input("Enter the total number of items: "))
9. for i in range(total_items):
10. item = input("Enter an item to add: ")
11. list_items.append(item)
12. items_of_tuple = tuple(list_items)
13. print(f"Tuple items are {items_of_tuple}")
```

#### OUTPUT

```
Enter the total number of items: 4
Enter a number: 4
Enter a number: 3
Enter a number: 2
Enter a number: 1
Items added to tuple are (4, 3, 2, 1)
Enter the total number of items: 4
Enter an item to add: 1
Enter an item to add: 2
Enter an item to add: 3
Enter an item to add: 4
Tuple items are ('1', '2', '3', '4')
```

Items are inserted into the tuple using two methods: using continuous concatenation += operator ①–⑥ and by converting list items to tuple items ⑦–⑩. In the code, *tuple\_items* is of tuple type. In both the methods, you must specify the total number of items that you are planning to insert to the tuple beforehand ②, ⑧. Based on this number, we iterate through the *for* loop as many times using the *range()* function ③–④, ⑨–⑩. In the first method, we continuously concatenate the user entered items to the tuple using += operator. Tuples are immutable and are not supposed to be changed. During each iteration, each *original\_tuple* is replaced by *original\_tuple* + (*new\_element*), thus creating a new tuple ⑤. Notice a comma after *new\_element*. In the second method, a list is created ⑦.

For each iteration ⑨, the user entered value ⑩ is appended ⑪ to the list variable. This list is then converted to tuple type using *tuple()* function ⑫.

**Program 8.3: Write Python Program to Swap Two Numbers Without Using Intermediate/Temporary Variables. Prompt the User for Input**

```
1. def main():
2. a = int(input("Enter a value for first number "))
3. b = int(input("Enter a value for second number "))
4. b, a = a, b
5. print("After Swapping")
6. print(f"Value for first number {a}")
7. print(f"Value for second number {b}")
8. if __name__ == "__main__":
9. main()
```

**OUTPUT**

```
Enter a value for the first number 5
Enter a value for the second number 10
After Swapping
Value for first number 10
Value for second number 5
```

The contents of variables *a* and *b* are reversed ④. The tuple variables are on the left side of the assignment operator and, on the right side, are the tuple values. The number of variables on the left and the number of values on the right has to be the same. Each value is assigned to its respective variable.

**Program 8.4: Program to Demonstrate the Return of Multiple Values from a Function**

```
1. def return_multiple_items():
2. monument = input("Which is your favorite monument? ")
3. year = input("When was it constructed? ")
4. return monument, year

5. def main():
6. mnt, yr = return_multiple_items()
7. print(f"My favorite monument is {mnt} and it was constructed in {yr}")
8. result = return_multiple_items()
9. print(f"My favorite monument is {result[0]} and it was constructed in {result[1]}")

10. if __name__ == "__main__":
11. main()
```

**OUTPUT**

Which is your favorite monument? Hawa Mahal  
 When was it constructed? 1799  
 My favorite monument is Hawa Mahal and it was constructed in 1799  
 Which is your favorite monument? Big Ben  
 When was it constructed? 1858  
 My favorite monument is Big Ben and it was constructed in 1858

The function `return_multiple_items()` returns multiple values that are actually a tuple ④. For calling functions that return a tuple, it is common to assign the result to multiple variables ⑥. This is simply tuple unpacking. The return value could also have been assigned to a single variable ⑧.

**Program 8.5: Write Python Program to Sort Words in a Sentence in Decreasing Order of Their Length. Display the Sorted Words along with Their Length**

```

1. def sort_words(user_input):
2. list_of_words = user_input.split()
3. words = list()
4. for each_word in list_of_words:
5. words.append((len(each_word), each_word))
6. words.sort(reverse=True)
7. print("After sorting")
8. for length, word in words:
9. print(f"The word \"{word}\" is of {length} characters')
10. def main():
11. sentence = input("Enter a sentence ")
12. sort_words(sentence)
13. if __name__ == "__main__":
14. main()

```

**OUTPUT**

Enter a sentence Everything you can imagine is real  
 After sorting  
 The word "Everything" is of 10 characters  
 The word "imagine" is of 7 characters  
 The word "real" is of 4 characters  
 The word "you" is of 3 characters  
 The word "can" is of 3 characters  
 The word "is" is of 2 characters

The user input *sentence* is split into a list of words ②. A *for* loop ④ is used to append a list of tuples to *words* list ③. Each of these tuples in the list consists of two items:

the length of the word and the word items ⑤. The `sort()` method compares the tuples in the list based on the length of the word that is the first item in the tuple. The second item in the tuple is considered only when the first items are the same and to break the tie. The keyword argument `reverse=True` tells the `sort()` method to sort in decreasing order ⑥. By inserting a list of tuples into the list with a number as the first item in the tuple, you can easily sort the list of tuples. Word length and the word itself are printed using a `for` loop ⑧–⑨.

**Program 8.6: Write Pythonic Code to Sort a Sequence of Names according to Their Alphabetical Order Without Using `sort()` Function**

```

1. def read_list_items():
2. print("Enter names separated by a space")
3. list_items = input().split()
4. return list_items

5. def sort_item_list(items_in_list):
6. n = len(items_in_list)
7. for i in range(n):
8. for j in range(1, n-i):
9. if items_in_list[j-1] > items_in_list[j]:
10. (items_in_list[j-1], items_in_list[j]) = (items_in_list[j], items_in_list[j-1])
11. print("After Sorting")
12. print(items_in_list)

13. def main():
14. all_items = read_list_items()
15. sort_item_list(all_items)

16. if __name__ == "__main__":
17. main()

```

**OUTPUT**

```

Enter names separated by a space
yashmith ava noah isabella emma
After Sorting
['ava', 'emma', 'isabella', 'noah', 'yashmith']

```

Lists of strings is sorted by comparing each string with its successor string and swaps them if they are not in ascending order. To sort the items in the list, we require two loops: one for running through the passes and another for locating and interchanging the strings in each pass ⑦–⑩. The pass through the list is repeated until no swaps are needed, which indicates that the list of strings is sorted.

---

## 8.8 Using `zip()` Function

The `zip()` function makes a sequence that aggregates elements from each of the *iterables* (can be zero or more). The syntax for `zip()` function is,

**`zip(*iterables)`**

An *iterable* can be a list, string, or dictionary. It returns a sequence of tuples, where the *i*-th tuple contains the *i*-th element from each of the *iterables*. The aggregation of elements stops when the shortest input iterable is exhausted. For example, if you pass two iterables with one containing two items and other containing five items, then the `zip()` function returns a sequence of two tuples. With a single iterable argument, it returns an iterator of one tuple. With no arguments, it returns an empty iterator. For example,

```
1. >>> x = [1, 2, 3]
2. >>> y = [4, 5, 6]
3. >>> zipped = zip(x, y)
4. >>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
```

Here `zip()` function is used to zip two iterables of list type ①–④.

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function. For example,

```
1. >>> questions = ('name', 'quest', 'favorite color')
2. >>> answers = ('lancelot', 'the holy grail', 'blue')
3. >>> for q, a in zip(questions, answers):
... print(f'What is your {q}? It is {a}.')
```

### OUTPUT

```
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Since `zip()` function returns a tuple, you can use a *for* loop with multiple iterating variables to print tuple items ①–③.

---

## 8.9 Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate items. Primary uses of sets include membership testing and eliminating duplicate entries. Sets also support mathematical operations, such as union, intersection, difference, and symmetric difference.

Curly braces `{ }` or the `set()` function can be used to create sets with a comma-separated list of items inside curly brackets `{ }`. Note: to create an empty set you have to use `set()` and not `{ }` as the latter creates an empty dictionary.

```

1. >>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2. >>> print(basket)
 {'pear', 'orange', 'banana', 'apple'}
3. >>> 'orange' in basket
 True
4. >>> 'crabgrass' in basket
 False
5. >>> a = set('abracadabra')
6. >>> b = set('alacazam')
7. >>> a
 {'d', 'a', 'b', 'r', 'c'}
8. >>> b
 {'m', 'l', 'c', 'z', 'a'}
9. >>> a - b
 {'b', 'r', 'd'}
10. >>> a | b
 {'l', 'm', 'z', 'd', 'a', 'b', 'r', 'c'}
11. >>> a & b
 {'a', 'c'}
12. >>> a ^ b
 {'l', 'd', 'm', 'b', 'r', 'z'}
13. >>> len(basket)
 4
14. >>> sorted(basket)
 ['apple', 'banana', 'orange', 'pear']

```

A set is a collection of unique items. Duplicate items are removed from the set *basket* ①. Even though we have provided "orange" and "apple" items two times, the set will contain only one item of "orange" and "apple." You can test for the presence of an item in a set using *in* and *not in* membership operators ③–④. Unique letters in sets *a* and *b* are displayed ⑤–⑧. Letters present in set *a*, but not in set *b*, are printed ⑨. Letters present in set *a*, set *b*, or both are printed ⑩. Letters present in both set *a* and set *b* are printed. Letters present in set *a* or set *b*, but not both, are printed ⑪. Total number of items in the set *basket* is found using the *len()* function ⑬. The *sorted()* function returns a new sorted list from items in the set ⑭.



Sets are mutable. Indexing is not possible in sets, since set items are unordered. You cannot access or change an item of the set using indexing or slicing.

## 8.10 Set Methods

You can get a list of all the methods associated with the *set* (TABLE 8.3) by passing the *set* function to *dir()*.

1. `>>> dir(set)`

```
['__and__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__iand__', '__init__', '__init_subclass__', '__ior__', '__isub__', '__iter__', '__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

Various methods associated with *set* are displayed ①.

**TABLE 8.3**

Various Set Methods

| Set Methods                         | Syntax                                            | Description                                                                                                                                                                                |
|-------------------------------------|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>add()</code>                  | <code>set_name.add(item)</code>                   | The <i>add()</i> method adds an <i>item</i> to the set <i>set_name</i> .                                                                                                                   |
| <code>clear()</code>                | <code>set_name.clear()</code>                     | The <i>clear()</i> method removes all the items from the set <i>set_name</i> .                                                                                                             |
| <code>difference()</code>           | <code>set_name.difference(*others)</code>         | The <i>difference()</i> method returns a new set with items in the set <i>set_name</i> that are not in the <i>others</i> sets.                                                             |
| <code>discard()</code>              | <code>set_name.discard(item)</code>               | The <i>discard()</i> method removes an <i>item</i> from the set <i>set_name</i> if it is present.                                                                                          |
| <code>intersection()</code>         | <code>set_name.intersection(*others)</code>       | The <i>intersection()</i> method returns a new set with items common to the set <i>set_name</i> and all <i>others</i> sets.                                                                |
| <code>isdisjoint()</code>           | <code>set_name.isdisjoint(other)</code>           | The <i>isdisjoint()</i> method returns True if the set <i>set_name</i> has no items in common with <i>other</i> set. Sets are disjoint if and only if their intersection is the empty set. |
| <code>issubset()</code>             | <code>set_name.issubset(other)</code>             | The <i>issubset()</i> method returns True if every item in the set <i>set_name</i> is in <i>other</i> set.                                                                                 |
| <code>issuperset()</code>           | <code>set_name.issuperset(other)</code>           | The <i>issuperset()</i> method returns True if every element in <i>other</i> set is in the set <i>set_name</i> .                                                                           |
| <code>pop()</code>                  | <code>set_name.pop()</code>                       | The method <i>pop()</i> removes and returns an arbitrary item from the set <i>set_name</i> . It raises <i>KeyError</i> if the set is empty.                                                |
| <code>remove()</code>               | <code>set_name.remove(item)</code>                | The method <i>remove()</i> removes an <i>item</i> from the set <i>set_name</i> . It raises <i>KeyError</i> if the <i>item</i> is not contained in the set.                                 |
| <code>symmetric_difference()</code> | <code>set_name.symmetric_difference(other)</code> | The method <i>symmetric_difference()</i> returns a new set with items in either the set or <i>other</i> but not both.                                                                      |
| <code>union()</code>                | <code>set_name.union(*others)</code>              | The method <i>union()</i> returns a new set with items from the set <i>set_name</i> and all <i>others</i> sets.                                                                            |
| <code>update()</code>               | <code>set_name.update(*others)</code>             | Update the set <i>set_name</i> by adding items from all <i>others</i> sets.                                                                                                                |

*Note:* Replace the words "*set\_name*", "*other*" and "*others*" mentioned in the syntax with your *actual set names* in your code.

For example,

```

1. >>> european_flowers = {"sunflowers", "roses", "lavender", "tulips", "goldcrest"}
2. >>> american_flowers = {"roses", "tulips", "lilies", "daisies"}
3. >>> american_flowers.add("orchids")
4. >>> american_flowers.difference(european_flowers)
 {'lilies', 'orchids', 'daisies'}
5. >>> american_flowers.intersection(european_flowers)
 {'roses', 'tulips'}
6. >>> american_flowers.isdisjoint(european_flowers)
 False
7. >>> american_flowers.issuperset(european_flowers)
 False
8. >>> american_flowers.issubset(european_flowers)
 False
9. >>> american_flowers.symmetric_difference(european_flowers)
 {'lilies', 'orchids', 'daisies', 'goldcrest', 'sunflowers', 'lavender'}
10. >>> american_flowers.union(european_flowers)
 {'lilies', 'tulips', 'orchids', 'sunflowers', 'lavender', 'roses', 'goldcrest', 'daisies'}
11. >>> american_flowers.update(european_flowers)
12. >>> american_flowers
 {'lilies', 'tulips', 'orchids', 'sunflowers', 'lavender', 'roses', 'goldcrest', 'daisies'}
13. >>> american_flowers.discard("roses")
14. >>> american_flowers
 {'lilies', 'tulips', 'orchids', 'daisies'}
15. >>> european_flowers.pop()
 'tulips'
16. >>> american_flowers.clear()
17. >>> american_flowers
 set()

```

Various operations on sets are carried out using set methods ①–⑰.

### 8.10.1 Traversing of Sets

You can iterate through each item in a set using a *for* loop.

#### Program 8.7: Program to Iterate Over Items in Sets Using *for* Loop

```

1. warships = {"u.s.s._arizona", "hms_beagle", "ins_airavat", "ins_hetz"}
2. def main():

```



```

3. for each_ship in warships:
4. print(f"{each_ship} is a Warship")
5. if __name__ == "__main__":
6. main()

```

**OUTPUT**

```

hms_beagle is a Warship
u.s.s._arizona is a Warship
ins_airavat is a Warship
ins_hetz is a Warship

```

Here *warships* is of a *set* type ①. A *for* ③ loop is used to iterate through each item in the set.

**Program 8.8: Write a Function Which Receives a Variable Number of Strings as Arguments. Find Unique Characters in Each String**

```

1. def find_unique(*all_words):
2. for each_word in all_words:
3. unique_character_list = list(set(each_word))
4. print(f"Unique characters in the word {each_word} are {unique_character_list}")

5. def main():
6. find_unique("egg", "immune", "feed", "vacuum", "goddessship")

7. if __name__ == "__main__":
8. main()

```

**OUTPUT**

```

Unique characters in the word egg are ['e', 'g']
Unique characters in the word immune are ['m', 'n', 'i', 'u', 'e']
Unique characters in the word feed are ['d', 'e', 'f']
Unique characters in the word vacuum are ['m', 'c', 'u', 'a', 'v']
Unique characters in the word goddessship are ['p', 's', 'o', 'h', 'g', 'i', 'd', 'e']

```

The method *find\_unique()* ① accepts a variable number of words as arguments. Iterate through each word using a *for* loop ②. For each word, find unique characters using the *set()* method and convert it to a list ③ and print it ④.

**Program 8.9: Write a Python Program That Accepts a Sentence as Input and Removes All Duplicate Words. Print the Sorted Words**

```

1. def unique_words(user_input):
2. words = user_input.split()

```

```

3. print(f"The unique and sorted words are {sorted(list(set(words)))}")

4. def main():
5. sentence = input("Enter a sentence ")
6. unique_words(sentence)

7. if __name__ == "__main__":
8. main()

```

### OUTPUT

```

Enter a sentence The man we saw saw a saw
The unique and sorted words are ['The', 'a', 'man', 'saw', 'we']

```

In the above program, the user-entered sentence ⑤ is split into a list of words based on space ②. The *words* list is passed as an argument to the *set()* method. The unique set of words returned by the *set()* method is converted to a list and sorted ③.

## 8.11 Frozenset

A frozenset is basically the same as a set, except that it is immutable. Once a frozenset is created, then its items cannot be changed. Since they are immutable, they can be used as members in other sets and as dictionary keys. The frozensets have the same functions as normal sets, except none of the functions that change the contents (update, remove, pop, etc.) are available.

```

1. >>> dir(frozenset)
['_and_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_
subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__
rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__',
 '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'copy',
'difference', 'intersection', 'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference',
'union']

```

List of methods available for *frozenset* ①.

For example,

```

1. >>> fs = frozenset(["g", "o", "o", "d"])
2. >>> fs
frozenset({'d', 'o', 'g'})
3. >>> animals = set([fs, "cattle", "horse"])

```

```

4. >>> animals
 {'cattle', frozenset({'d', 'o', 'g'}), 'horse'}
5. >>> official_languages_world = {"english":59, "french":29, "spanish":21}
6. >>> frozenset(official_languages_world)
 frozenset({'spanish', 'french', 'english'})
7. >>> frs = frozenset(['german'])
8. >>> official_languages_world = {"english":59, "french":29, "spanish":21, frs:6}
9. >>> official_languages_world
 {'english': 59, 'french': 29, 'spanish': 21, frozenset(['german']): 6}

```

The Frozenset type ① is used within a set ③. Keys in a dictionary are returned when a dictionary is passed as an argument to *frozenset()* function ⑥. Frozenset is used as a key in dictionary ⑧.

## 8.12 Summary

- Tuple is an immutable data structure comprising of items that are ordered and heterogeneous.
- Tuples are formed using commas and not the parenthesis.
- Indexing and slicing of items are supported in tuples.
- Tuples support built-in functions such as `len()`, `min()`, and `max()`.
- The set stores a collection of unique values and are not placed in any particular order.
- Add an item to the set using `add()` method and remove an item from the set using the `remove()` method.
- The *for* loop is used to traverse the items in a set.
- The *issubset()* or *issuperset()* method is used to test whether a set is a superset or a subset of another set.
- Sets also provide functions such as `union()`, `intersection()`, `difference()`, and `symmetric_difference()`.

## Multiple Choice Questions

1. Which of the following is a mutable type?
  - a. Strings
  - b. Lists
  - c. Tuples
  - d. Frozenset

2. What will be the output of the following code?

```
t1 = (1, 2, 3, 4)
t1.append((5, 6, 7))
print(len(t1))
```

- a. Error
- b. 2
- c. 1
- d. 5

3. What is the correct syntax for creating a tuple?

- a. ["a","b","c"]
- b. ("a","b","c")
- c. {"a","b","c"}
- d. {}

4. Assume `air_force = ("f15", "f22a", "f35a")`. Which of the following is incorrect?

- a. `print(air_force[2])`
- b. `air_force[2] = 42`
- c. `print(max(air_force))`
- d. `print(len(air_force))`

5. Gauge the output of the following code snippet.

```
bike = ('d','u','c','a','t','i')
bike [1:3]
```

- a. ('u', 'c')
- b. ('u', 'c', 'c')
- c. ('d', 'u', 'c')
- d. ('a', 't', 'i')

6. What is the output of the following code?

```
colors = ("v", "i", "b", "g", "y", "o", "r")
for i in range(0, len(colors),2):
 print(colors[i])
```

- a. ('i', 'b')
- b. ('v', 'i', 'b')
- c. ['v', 'b', 'y', 'r']
- d. ('i', 'g', 'o')

7. What is the output of the following code snippet?

```
colors = ("v", "i", "b", "g", "y", "o", "r")
2 * colors
```

- a. ['v', 'i', 'b', 'g', 'y', 'o', 'r']
- b. ('v', 'i', 'b', 'g', 'y', 'o', 'r')
- c. ('v', 'v', 'i', 'i', 'b', 'b', 'g', 'g', 'y', 'y', 'o', 'o', 'r', 'r')
- d. ('v', 'i', 'b', 'g', 'y', 'o', 'r', 'v', 'i', 'b', 'g', 'y', 'o', 'r')

8. Predict the output of the following code.

```
os = ('w', 'i', 'n', 'd', 'o', 'w', 's')
os1 = ('w', 'i', 'n', 'd', 'w', 's', 'o')
os < os1
```

- a. True
  - b. False
  - c. 1
  - d. 0
9. What is the data type of (3)?
- a. Tuple
  - b. List
  - c. None
  - d. Integer
10. Assume tuple\_1 = (7,8,9,10,11,12,13) then the output of tuple\_1[1:-1] is.
- a. Error
  - b. (8,9,10,11,12)
  - c. [8,9,10,11,12]
  - d. None
11. What might be the output of the following code:
- ```
A = ("hello") * 3  
print(A)
```
- a. Operator Error
 - b. ('hello','hello','hello')
 - c. 'hellohellohello'
 - d. None of these
12. What is the output of the following code:
- ```
number_1 = {1,2,3,4,5}
number_2 = {1,2,3}
number_1.difference(number_2)
```
- a. {4, 5}
  - b. {1, 2, 3}
  - c. (4, 5)
  - d. [4, 5]
13. Judge the output of the following code:
- ```
tuples = (7,8,9)  
sum(tuples, 2)
```
- a. 26
 - b. 20
 - c. 12
 - d. 3

14. `tennis = ('steffi', 'monica', 'serena', 'monica', 'navratilova')`
`tennis.count('monica')`
a. 3
b. 0
c. 2
d. 1
15. A set is an _____ collection with no _____ items.
a. unordered, duplicate
b. ordered, unique
c. unordered, unique
d. ordered, duplicate
16. Judge the output of the following:
`sets_1 = set(['a','b','b','c','c','c','d'])`
`len(sets_1)`
a. 1
b. 4
c. 5
d. 7
17. What is the output of the code shown below?
`s = {1,2,3}`
`s.update(4)`
`print(s)`
a. {1,2,3,4}
b. {1,2}
c. {1,2,3}
d. Error
18. Tuple unpacking requires
a. an equal number of variables on the left side to the number of items in the tuple.
b. greater number of variables on the left side to the number of items in the tuple.
c. less number of variables on the left side to the number of items in the tuple.
d. Does not require any variables.
19. The statement that is used to create an empty set is
a. {}
b. set()
c. []
d. ()

20. The _____ functions removes the first element of the set
- remove()
 - delete()
 - pop()
 - truncate()
21. The method that returns a new set with items common to two sets is
- isdisjoint()
 - intersection()
 - symmetric_difference()
 - union()
22. What is the output of the following code snippet?
- ```
s1 = {'a','b','c'}
s2 = {'d'}
print(s1.union(s2))
```
- {'c', 'd', 'b', 'a'}
  - {'a', 'b', 'c', 'd'}
  - {'b', 'c', 'd', 'a'}
  - {'d', 'a', 'b', 'c'}
23. The function that makes a sequence by aggregating the elements from each of the iterables is
- remove()
  - update()
  - frozenset()
  - zip()
24. Predict the output of the following code:
- ```
even = {'2', '4', '6'}  
odd = {'1', '5', '7'}  
even.isdisjoint(odd)  
odd.isdisjoint(even)
```
- True False
 - False True
 - True True
 - False False
25. Which of the following code snippet returns symmetric difference between two sets
- $x \wedge y$
 - $x \& y$
 - $x \mid y$
 - $x - y$

Review Questions

1. Explain the usage of tuples.
2. Identify the primary differences between a list and a tuple?
3. Explain how to create an empty set.
4. Briefly explain the slice operation of tuples with an example.
5. Illustrate how tuples are used to build dictionaries with an example.
6. Explain `zip()` function with an example.
7. With an example, explain the methods associated with sets.
8. Explain `frozenset` with an example.
9. Explain how to delete or remove elements from tuples and sets.
10. Create a list containing three elements, and then create a tuple from that list.
11. Write a program to unpack a tuple to several variables.
12. Write a program to check whether an item exists within a tuple.
13. Write a program to unzip a list of tuples into individual lists.
14. Write a program to create an intersection, union, set difference, and symmetric difference of sets.
15. Write a program to demonstrate the use of `issubset()` and `issuperset()` methods.
16. Write a program that takes a range and creates a list of tuples within that range with the first element as the number and the second element as the square of the number.
17. Write a program to clear a set.
18. Write a program to find the length of the set.
19. Write a program to store the latitude and longitude of your house as a tuple and display it.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

9

Files

AIM

Understand how to interact with files with the help of built-in Python functions and perform operations such as read, write, and manipulate files.

LEARNING OUTCOMES

After completing this chapter, you should be able to

- Select text files and binary files to read and write data.
- Demonstrate the use of built-in functions to navigate the file system.
- Make use of os module to operate on underlying Operating System tasks.

In everyday life, the term “file” is something of a catchall. It is used for things that are not only written, but also used to describe things that don’t have any words in them at all, like pictures. A file is the common storage unit in a computer, and all programs and data are “written” into a file and “read” from a file. A file extension, sometimes called a file suffix or a filename extension, is the character or group of characters after the period that makes up an entire file name. The file extension helps your computer’s operating system, like Windows, determine which program will be associated with the file. For example, the file *assignments.docx* ends in *docx*, a file extension that is associated with Microsoft Word on your computer. When you attempt to open this file, Windows sees that the file ending in a *docx* extension and knows it should be opened with the Microsoft Word program.

File extensions also often indicate the file type, or file format, of the file but not always. Any file’s extensions can be renamed, but that will not convert the file to another format or change anything about the file other than this portion of its name. File extensions and file formats are often spoken about interchangeably. However, the file extension is just whatever characters are after the period, while the file format illustrates the way in which the data in the file is organized and what sort of file it is. For example, in the file name *pop.csv*, the file extension *csv* indicates that this is a CSV file. You could easily rename that file to *pop.mp3*, but that would not mean that you could play the file on a smartphone. The file itself is still rows of text (a CSV file), not a compressed musical recording (an MP3 file). Some file extensions are classified as executable, meaning that when clicked, they do not just open for viewing or playing, they actually do something all by themselves, like install a program, start a process, or run a script.

All the data on your hard drive consists of files and directories. The fundamental difference between the two is that files store data, while directories store files and other directories. The folders, often referred to as directories, are used to organize files on your computer. The directories themselves take up virtually no space on the hard drive. Files, on the other hand, can range from a few bytes to several gigabytes. Directories are used to organize files on your computer.

[Adapted with kind permission from <https://www.lifewire.com/what-is-a-file-extension-2625879>]

9.1 Types of Files

Python supports two types of files – text files and binary files. These two file types may look the same on the surface but they encode data differently. While both binary and text files contain data stored as a series of bits (binary values of 1s and 0s), the bits in text files represent characters, while the bits in binary files represent custom data.

Binary files typically contain a sequence of bytes or ordered groupings of eight bits. When creating a custom file format for a program, a developer arranges these bytes into a format that stores the necessary information for the application. Binary file formats may include multiple types of data in the same file, such as image, video, and audio data. This data can be interpreted by supporting programs but will show up as garbled text in a text editor. Below is an example of a .JPG image file opened in an image viewer and a text editor (FIGURE 9.1).

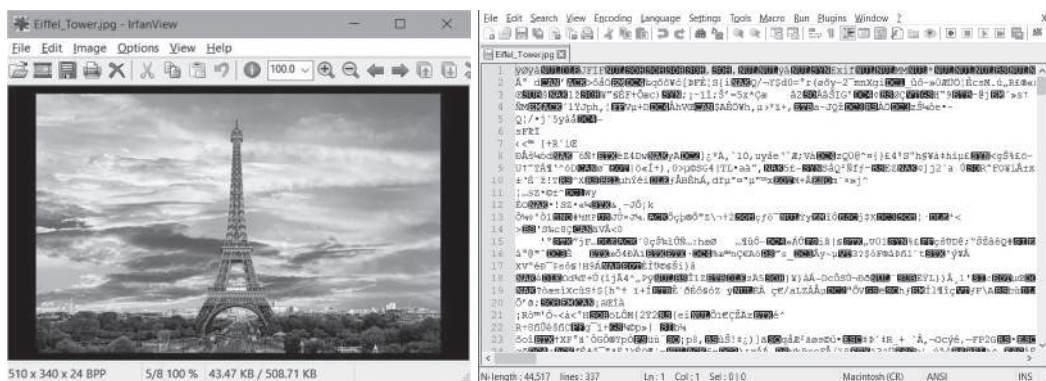


FIGURE 9.1

Image and its binary contents.

As you can see, the image viewer recognizes the binary data and displays the picture. When the image is opened in a text editor, the binary data is converted to unrecognizable text. However, you may notice that some of the text is readable. This is because the JPG format includes small sections for storing textual data. The text editor, while not designed to read this file format, still displays this text when the file is opened. Many other binary file types include sections of readable text as well. Therefore, it may be possible to find out some information about an unknown binary file type by opening it in a text editor. Binary files often contain headers, which are bytes of data at the beginning of a file that identifies the file's contents. Headers often include the file type and other

descriptive information. If a file has an invalid header information, a software program may not open the file or may report that the file is corrupted.

Text files are more restrictive than binary files since they can only contain textual data. However, unlike binary files, they are less likely to become corrupted. While a small error in a binary file may make it unreadable, a small error in a text file may simply show up once the file has been opened. A typical plain text file contains several lines of text that are each followed by an End-of-Line (EOL) character. An End-of-File (EOF) marker is placed after the final character, which signals the end of the file. Text files include a character encoding scheme that determines how the characters are interpreted and what characters can be displayed. Since text files use a simple, standard format, many programs are capable of reading and editing text files. Common text editors include Microsoft Notepad and WordPad, which are bundled with Windows, and Apple TextEdit, which is included with Mac OS X.

We can usually tell if a file is binary or text based on its file extension. This is because by convention the extension reflects the file format, and it is ultimately the file format that dictates whether the file data is binary or text.

Common extensions for binary file formats:

Images: jpg, png, gif, bmp, tiff, psd,...

Videos: mp4, mkv, avi, mov, mpg, vob,...

Audio: mp3, aac, wav, flac, ogg, mka, wma,...

Documents: pdf, doc, xls, ppt, docx, odt,...

Archive: zip, rar, 7z, tar, iso,...

Database: mdb, accde, frm, sqlite,...

Executable: exe, dll, so, class,...

Common extensions for text file formats:

Web standards: html, xml, css, svg, json,...

Source code: c, cpp, h, cs, js, py, java, rb, pl, php, sh,...

Documents: txt, tex, markdown, asciidoc, rtf, ps,...

Configuration: ini, cfg, rc, reg,...

Tabular data: csv, tsv,...

[Adapted with kind permission from https://fileinfo.com/help/binary_vs_text_files]

9.1.1 File Paths

All operating systems follow the same general naming conventions for an individual file: a base file name and an optional extension, separated by a period. Note that a directory is simply a file with a special attribute designating it as a directory, but otherwise must follow all the same naming rules as a regular file. To make use of files, you have to provide a file path, which is basically a route so that the user or the program knows where the file is located. The path to a specified file consists of one or more components, separated by a special character (a backslash for Windows and forward slash for Linux), with each component usually being a directory name or file name, and possibly a volume name or drive name in Windows or root in Linux. If a component of a path is a file name, it must be the last component. It is often critical to the system's interpretation of a path what the

beginning, or prefix, of the path looks like. In the Windows Operating System, the maximum length for a path is 260 characters and in the Linux Operating System the maximum path length is of 4096 characters.

The following fundamental rules enable applications to create and process valid names for files and directories in both Windows and Linux operating systems unless explicitly specified:

- Use a period to separate the base file name from the extension in the file name.
- In Windows use backslash (\) and in Linux use forward slash (/) to separate the components of a path. The backslash (or forward slash) separates one directory name from another directory name in a path and it also divides the file name from the path leading to it. Backslash (\) and forward slash (/) are reserved characters and you cannot use them in the name for the actual file or directory.
- Do not assume case sensitivity. File and Directory names in Windows are not case sensitive while in Linux it is case sensitive. For example, the directory names ORANGE, Orange, and orange are the same in Windows but are different in Linux Operating System.
- In Windows, volume designators (drive letters) are case-insensitive. For example, "D:\\" and "d:\\" refer to the same drive.
- The reserved characters that should not be used in naming files and directories are < (less than), > (greater than), : (colon), " (double quote), / (forward slash), \ (backslash), | (vertical bar or pipe), ? (question mark) and * (asterisk).
- In Windows Operating system reserved words like CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9 should not be used to name files and directories.

9.1.2 Fully Qualified Path and Relative Path

A file path can be a fully qualified path name or relative path. The fully qualified path name is also called an Absolute path. A path is said to be a fully qualified path if it points to the file location, which always contains the root and the complete directory list. The current directory is the directory in which a user is working at a given time. Every user is always working within a directory.

Note that the current directory may or may not be the root directory depending on what it was set to during the most recent “change directory” operation on that disk. The root directory, sometimes just called as root is the “highest” directory in the hierarchy. You can also think of it in general as the start or beginning of a particular directory structure. The root directory contains all other directories in the drive and can of course also contain files. For example, the root directory of the main partition on your Windows system will be C:\ and the root directory on your Linux system will be / (forward slash).

Examples of a fully qualified path are given below.

- "C:\langur.txt" refers to a file named "langur.txt" under root directory C:\.
- "C:\fauna\bison.txt" refers to a file named "bison.txt" in a subdirectory *fauna* under root directory C:\.

A path is also said to be a relative path if it contains “double-dots”; that is, two consecutive periods together as one of the directory components in a path or “single-dot”; that is, one period as one of the directory components in a path. These two consecutive periods are used to denote the directory above the current directory, otherwise known as the “parent directory.” Use a single period as a directory component in a path to represent the current directory.

Examples of the relative path are given below:

- “..\langur.txt” specifies a file named “langur.txt” located in the parent of the current directory *fauna*.
- “\bison.txt” specifies a file named “bison.txt” located in a current directory named *fauna*.
- “..\..\langur.txt” specifies a file that is two directories above the current directory *india*

The following figure shows the structure of sample directories and files (FIGURE 9.2).

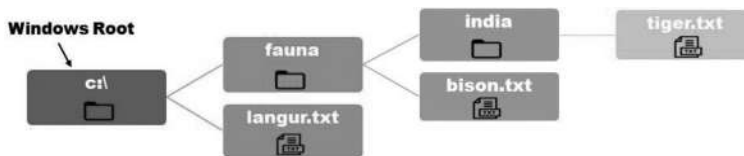


FIGURE 9.2
Structure of files and directories.

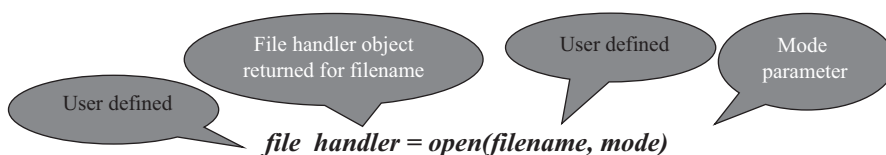
9.2 Creating and Reading Text Data

In all the programs you have executed until now, any output produced during the execution of the program is lost when the program ends. Data has not persisted past the end of execution. Just as programs live on in files, you can write and read data files in Python that persist after your program has finished running. Python provides built-in functions for opening a file, reading from a file, writing to a file, and closing a file.

9.2.1 Creating and Opening Text Files

Files are not very useful unless you can access the information they contain. All files must be opened first before they can be read from or written to using the Python’s built-in *open()* function. When a file is opened using *open()* function, it returns a file object called a file handler that provides methods for accessing the file.

The syntax of *open()* function is given below.



The `open()` function returns a file handler object for the file name. The `open()` function is commonly used with two arguments, where the first argument is a string containing the file name to be opened which can be absolute or relative to the current working directory. The second argument is another string containing a few characters describing the way in which the file will be used as shown in [TABLE 9.1](#). The mode argument is optional; `r` will be used if it is omitted. The file handler itself will not contain any data pertaining to the file.

TABLE 9.1

Access Modes of the Files

Mode	Description
"r"	Opens the file in read only mode and this is the default mode.
"w"	Opens the file for writing. If a file already exists, then it'll get overwritten. If the file does not exist, then it creates a new file.
"a"	Opens the file for appending data at the end of the file automatically. If the file does not exist it creates a new file.
"r+"	Opens the file for both reading and writing.
"w+"	Opens the file for reading and writing. If the file does not exist it creates a new file. If a file already exists then it will get overwritten.
"a+"	Opens the file for reading and appending. If a file already exists, the data is appended. If the file does not exist it creates a new file.
"x"	Creates a new file. If the file already exists, the operation fails.
"rb"	Opens the binary file in read-only mode.
"wb"	Opens the file for writing the data in binary format.
"rb+"	Opens the file for both reading and writing in binary format.

For example,

```

1. >>> file_handler = open("example.txt","x")
2. >>> file_handler = open("moon.txt","r")
3. >>> file_handler = open("C:\langur.txt","r")
4. >>> file_handler = open("C:\prog\example.txt","r")
5. >>> file_handler = open("C:\\fauna\\bison.txt","r")
6. >>> file_handler = open("C:\\network\computer.txt","r")
7. >>> file_handler = open(r"C:\network\computer.txt","r")
8. >>> file_handler = open("titanic.txt","r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'titanic.txt'
9. >>> file_handler = open("titanic.txt","w+")
10. >>> file_handler = open("titanic.txt","a+")

```

The `open()` method returns a file handler object that can be used to read or modify the file. The `open()` method takes two arguments, the name of the file and the mode of operation that you want to perform on that file. In ①, the mode is "x". The file named *example.txt* is created if it is not present. If the file already exists, then the operation fails. The text file *moon.txt* present in the current directory is opened in read mode ②. In ③, the absolute path is given and the text *langur.txt* under root directory C:\ is opened in read-only mode.

The text file *example.txt* is found in the *prog* subdirectory under C:\ root directory and is opened in read-only mode ④. In ⑤, there are two backslashes in the absolute path. Now if the same expression was executed with a single slash, it results in an error.

```
>>> file_handler = open("C:\fauna\bison.txt","r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 22] Invalid argument: 'C:\x0cauna\x08ison.txt'
```

When the Python interpreter reads the path, the characters `\f` and `\b` in the path will be treated as escape sequences and not as part of directory or text name. Python represents backslashes in strings as `\\` because the backslash is an escape character—for instance, `\n` represents a new line, `\t` represents a tab, `\f` represents ASCII Formfeed, and `\b` represents ASCII Backspace. Because of this, there needs to be a way to tell Python you really want two characters of `\f` and `\b` rather than Formfeed and Backspace, and you do that by escaping the backslash itself with another one. In ⑥, to overcome the problem of the characters `\n` being treated as an escape sequence in the path, you have to include another backslash. You can also prefix the absolute path with the *r* character ⑦. If so, there is no need to specify double backslashes in the path to overcome the escape sequence problem. The *r* means that the absolute path string is to be treated as a raw string, which means all escape sequences will be ignored. For example, `'\n'` is a new line character, while `r'\n'` will be treated as the characters `\` followed by `n`. When opening a file for reading, if the file is not present, then the program will terminate with a “no such file or directory” error ⑧. The file is opened in *w+* mode for reading and writing ⑨. If the file does not exist, then the file name *titanic.txt* is created. The file is opened in *a+* mode for reading, writing, and appending. If the file exists, then the content or data is appended. If the file does not exist, then the file is created ⑩.

9.2.2 File *close()* Method

Opening files consume system resources, and, depending on the file mode, other programs may not be able to access them. It is important to close the file once the processing is completed. After the file handler object is closed, you cannot further read or write from the file. Any attempt to use the file handler object after being closed will result in an error.

The syntax for *close()* function is,

```
file_handler.close()
```

For example,

1. `>>> file_handler = open("moon.txt","r")`
2. `>>> file_handler.close()`

You should call *file_handler.close()* to close the file. This immediately frees up any system resources used by it ②. If you do not explicitly close a file, Python’s garbage collector will eventually destroy the object and close the opened file for you, but the file may have stayed open for a while. Another risk is that different Python implementations will do this clean-up at different times.

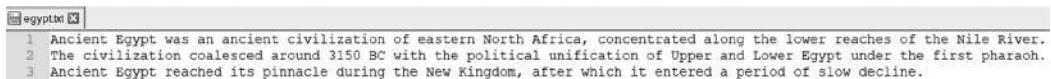
If an exception occurs while performing some operation on the file, then the code exits without closing the file. In order to overcome this problem, you should use a try-except-finally block to handle exceptions. For example,

```

1. try:
2.     f = open("file", "w")
3.     try:
4.         f.write('Hello World!')
5.     finally:
6.         f.close()
7. except IOError:
8.     print('oops!')
```

You should not be writing to the file in the finally block, as any exceptions raised there will not be caught by the except block. The *except* block executes if there is an exception raised by the *try* block ③–④. The *finally* block always executes regardless of whatever happens. The use of the *return* statement in the *except* block will not skip the *finally* block. By its very nature, the *finally* block cannot be skipped no matter what; that is why you want to put your “clean-up” code in there (i.e., closing files) ⑤–⑥. So, even if there is an exception ⑦–⑧, the above code will make sure your file gets appropriately closed.

Program 9.1: Write Python Program to Read and Print Each Line in "egypt.txt" file. Sample Content of "egypt.txt" File is Given Below.



```

1 Ancient Egypt was an ancient civilization of eastern North Africa, concentrated along the lower reaches of the Nile River.
2 The civilization coalesced around 3150 BC with the political unification of Upper and Lower Egypt under the first pharaoh.
3 Ancient Egypt reached its pinnacle during the New Kingdom, after which it entered a period of slow decline.
```

```

1. def read_file():
2.     file_handler = open("egypt.txt")
3.     print("Printing each line in the text file")
4.     for each_line in file_handler:
5.         print(each_line)
6.     file_handler.close()
7. def main():
8.     read_file()
9. if __name__ == "__main__":
10.    main()
```

OUTPUT

Printing each line in the text file

Ancient Egypt was an ancient civilization of eastern North Africa, concentrated along the lower reaches of the Nile River.

The civilization coalesced around 3150 BC with the political unification of Upper and Lower Egypt under the first pharaoh.

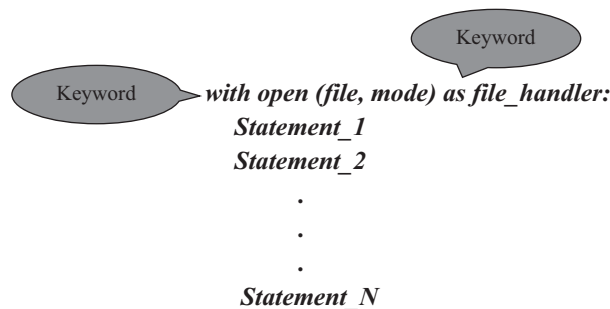
Ancient Egypt reached its pinnacle during the New Kingdom, after which it entered a period of slow decline.

In the `read_file()` function ① definition, you open the file `egypt.txt` and assign the file object to the `file_handler` ②. By default, the file is opened in `read` only mode as no mode is specified explicitly. Use a `for` loop to iterate over `file_handler` and print the lines ④–⑤. Once the file processing operation is over, close the `file_handler` ⑥. In the output, notice a blank space between each line of the file. Understand that at the end of each line, a newline character (`\n`) is present which is invisible and it indicates the end of the line. The `print()` function by default always appends a newline character. This means that if you want to print data that already ends in a newline, we get two newlines, resulting in a blank space between the lines. In order to overcome this problem, pass an `end` argument to the `print()` function and initialize it with an empty string (with no spaces). The `end` argument should always be a string. The value of `end` argument is printed after the thing you want to print. By default, the `end` argument contains a newline ("`\n`") but it can be changed to something else, like an empty string. This means that instead of the usual behavior of placing a newline character after the end of the line by the `print()` function, you can now change it to print an empty string after each line. So, changing line ⑤ as `print(each_line, end="")` removes the blank spaces between the lines in the output.

9.2.3 Use of *with* Statements to Open and Close Files

Instead of using try-except-finally blocks to handle file opening and closing operations, a much cleaner way of doing this in Python is using the *with* statement. You can use a *with* statement in Python such that you do not have to close the file handler object.

The syntax of the *with* statement for the file I/O is,



```

Keyword with open (file, mode) Keyword as file_handler:
    Statement_1
    Statement_2
    .
    .
    .
    Statement_N
  
```

In the syntax, the words *with* and *as* are keywords and the *with* keyword is followed by the `open()` function and ends with a colon. The *as* keyword acts like an alias and is used to assign the returning object from the `open()` function to a new variable `file_handler`. The *with* statement creates a context manager and it will automatically close the file handler object for you when you are done with it, even if an exception is raised on the way, and thus properly managing the resources.



The protocol, such as a class consisting of the `__enter__()` and `__exit__()` methods, is known as the "context management protocol," and the object that implements that protocol is known as the "context manager." The evaluation of the *with* statement results in an object called a "context manager" that supports the "context management protocol". The `__enter__()` method is executed when the control enters the code block inside the *with* statement block context. It returns an object that can be used within the context. When the control leaves the *with* block context, then the `__exit__()` method is called to clean up any resources being used. Thus, the resources are allocated and deallocated when the program requires it.

Program 9.2: Program to Read and Print Each Line in "japan.txt" File Using *with* Statement. Sample Content of "japan.txt" File is Given Below.

```
japan.txt
1 National Treasures of Japan are the most precious of Japan's Tangible Cultural Properties.
2 A Tangible Cultural Property is considered to be of historic or artistic value, classified either as
3 "buildings and structures", or as "fine arts and crafts".
```

```
1. def read_file():
2.     print("Printing each line in text file")
3.     with open("japan.txt") as file_handler:
4.         for each_line in file_handler:
5.             print(each_line, end="")
6. def main():
7.     read_file()
8. if __name__ == "__main__":
9.     main()
```

OUTPUT

Printing each line in text file

National Treasures of Japan are the most precious of Japan's Tangible Cultural Properties.
A Tangible Cultural Property is considered to be of historic or artistic value, classified either as
"buildings and structures", or as "fine arts and crafts".

Using a *with* statement is also much shorter than writing an equivalent try-except-finally block. The *with* statement automatically closes the file after executing its block of statements ③. You can read the contents of the file *japan.txt* line-by-line using a *for* loop without running out of memory ④. This is both efficient and fast.

You can also use a *with* statement to open more than one file. For example,

```
1. with open(in_filename) as in_file, open(out_filename, 'w') as out_file:
2.     for line in in_file:
3.         .
4.         .
5.         .
6.         out_file.write(parsed_line)
```

In the above code snippet, *in_file* and *out_file* are the file handlers ①. The *with* statement in Python is used to open one file for reading ② and another file for writing ③.

9.2.4 File Object Attributes

When the Python *open()* function is called, it returns a file object called a file handler. Using this file handler, you can retrieve information about various file attributes ([TABLE 9.2](#)).

TABLE 9.2

List of File Attributes

Attribute	Description
<code>file_handler.closed</code>	It returns a Boolean True if the file is closed or False otherwise.
<code>file_handler.mode</code>	It returns the access mode with which the file was opened.
<code>file_handler.name</code>	It returns the name of the file.

For example,

1. `>>> file_handler = open("computer.txt", "w")`
 2. `>>> print(f"File Name is {file_handler.name}")`
File Name is computer.txt
 3. `>>> print(f"File State is {file_handler.closed}")`
File State is False
 4. `>>> print(f"File Opening Mode is {file_handler.mode}")`
File Opening Mode is w
- Various file attribute operations are shown in ①–④

9.3 File Methods to Read and Write Data

When you use the *open()* function a file object is created. Here is the list of methods that can be called on this object ([TABLE 9.3](#)).

TABLE 9.3

List of Methods Associated with the File Object

Method	Syntax	Description
<code>read()</code>	<code>file_handler.read([size])</code>	This method is used to read the contents of a file up to a size and return it as a string. The argument <i>size</i> is optional, and, if it is not specified, then the entire contents of the file will be read and returned.
<code>readline()</code>	<code>file_handler.readline()</code>	This method is used to read a single line in file.
<code>readlines()</code>	<code>file_handler.readlines()</code>	This method is used to read all the lines of a file as list items.
<code>write()</code>	<code>file_handler.write(string)</code>	This method will write the contents of the string to the file, returning the number of characters written. If you want to start a new line, you must include the new line character.

(Continued)

TABLE 9.3 (Continued)

List of Methods Associated with File Object

Method	Syntax	Description
writelines()	file_handler. writelines(sequence)	This method will write a sequence of strings to the file.
tell()	file_handler.tell()	This method returns an integer giving the file handler's current position within the file, measured in bytes from the beginning of the file.
seek()	file_handler. seek(offset, from_what)	This method is used to change the file handler's position. The position is computed from adding <i>offset</i> to a reference point. The reference point is selected by the <i>from_what</i> argument. A <i>from_what</i> value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. If the <i>from_what</i> argument is omitted, then a default value of 0 is used, indicating that the beginning of the file itself is the reference point.

For example,

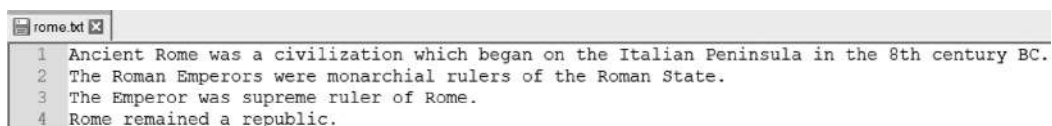
```

1. >>> f = open("example.txt", "w")
2. >>> f.write("abcdefgh")
   8
3. >>> f.close()
4. >>> f = open("example.txt")
5. >>> print(f.read(2))
   ab
6. >>> print(f.read(2))
   cd
7. >>> print(f.read(2))
   ef
8. >>> print(f.read(2))
   gh

```

In the above code, the *size* argument is specified in the *read()* method. In statement ⑤, the first two characters are read. In statement ⑥, next two characters are read. This goes on until the end of the file is reached depending on the size of the argument ⑦–⑧.

Program 9.3: Write Python Program to Read "rome.txt" File Using *read()* Method. Sample Content of "rome.txt" File is Given Below



```

1 Ancient Rome was a civilization which began on the Italian Peninsula in the 8th century BC.
2 The Roman Emperors were monarchical rulers of the Roman State.
3 The Emperor was supreme ruler of Rome.
4 Rome remained a republic.

```

```

1. def main():
2.     with open("rome.txt") as file_handler:
3.         print("Print entire file contents")

```

```

4.     print(file_handler.read(), end="")
5. if __name__ == "__main__":
6.     main()

```

OUTPUT

Print entire file contents

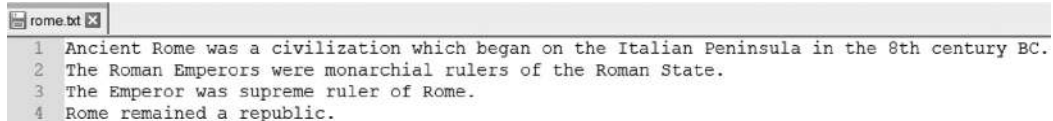
Ancient Rome was a civilization which began on the Italian Peninsula in the 8th century BC.
 The Roman Emperors were monarchial rulers of the Roman State.
 The Emperor was supreme ruler of Rome.
 Rome remained a republic.

The file named *rome.txt* is opened using *with* statement. Even if there are any errors the file handler is closed and the resources are deallocated ②. Above program prints the contents of the entire file ④. If you replace the line in ④ as *print(file_handler.read(13), end="")* then it prints the first 13 characters. Output will be

Ancient Rome

The file name either should be given in as an absolute path if the file is in a different location, or in the relative path if the file is in the same directory as the python source file.

Program 9.4: Consider the "rome.txt" File Specified in Program 9.3. Write Python Program to Read "rome.txt" file Using *readline()* Method



```

rome.txt
1 Ancient Rome was a civilization which began on the Italian Peninsula in the 8th century BC.
2 The Roman Emperors were monarchial rulers of the Roman State.
3 The Emperor was supreme ruler of Rome.
4 Rome remained a republic.

```

```

1. def main():
2.     with open("rome.txt") as file_handler:
3.         print("Print a single line from the file")
4.         print(file_handler.readline(), end="")
5.         print("Print another single line from the file")
6.         print(file_handler.readline(), end="")
7. if __name__ == "__main__":
8.     main()

```

OUTPUT

Print a single line from the file

Ancient Rome was a civilization which began on the Italian Peninsula in the 8th century BC.

Print another single line from the file

The Roman Emperors were monarchial rulers of the Roman State.

The *file_handler.readline()* method reads a single line from the file ③–⑥. If the *file_handler.readline()* returns an empty string, the end of the file has been reached.

Program 9.5: Consider the "rome.txt" File Specified in Program 9.3. Write Python Program to Read "rome.txt" File Using *readlines()* Method

```

1. def main():
2.     with open("rome.txt") as file_handler:
3.         print("Print file contents as a list")
4.         print(file_handler.readlines())
5. if __name__ == "__main__":
6.     main()

```

OUTPUT

[Ancient Rome was a civilization which began on the Italian Peninsula in the 8th century BC.\n', 'The Roman Emperors were monarchical rulers of the Roman State.\n', 'The Emperor was supreme ruler of Rome.\n', 'Rome remained a republic.']

The *readline()* method returns a list of strings with each line being a list item ④. A newline character (\n) is left at the end of each string item of the list indicating that the end of the line has been reached. It is only omitted on the last line of the file if the file does not end in a newline character.

The following code demonstrates the *write()* method.

```

1. >>> file_handler = open("moon.txt","w")
2. >>> file_handler.write("Moon is a natural satellite")
   27
3. >>> file_handler.close()
4. >>> file_handler = open("moon.txt", "a+")
5. >>> file_handler.write("of the earth")
   12
6. >>> file_handler.close()
7. >>> file_handler = open("moon.txt")
8. >>> file_handler.read()
   'Moon is a natural satelliteof the earth'
9. >>> file_handler.close()
10. >>> file_handler = open("moon.txt","w")
11. >>> file_handler.writelines(["Moon is a natural satellite", " ", "of the earth"])
12. >>> file_handler.close()
13. >>> file_handler = open("moon.txt")
14. >>> file_handler.read()
   'Moon is a natural satellite of the earth'
15. >>> file_handler.close()

```

The file *moon.txt* is opened in write mode ①. If the file is not present, then the file is created. The *write()* method is used to write the string to the *moon.txt* file using the *file_handler* object ②. This statement returns the number of characters written. Once the *file_handler*

is closed, you cannot write anymore contents to the file ③. To append data to the existing file, open it with `a+` mode ④. If you try to open the file in `w+` mode, then the contents of the file are overwritten. Read the contents of the file using the `read()` ⑧ method, and close the handler after completing file operations ⑨. Observe that in the output of line ⑧ there is no space between the words *satellite* and *of*. If you have a sequence of strings, then you can write them all using the `writelines()` method. The `writelines(sequence)` expects a list, or tuple, or string as an argument. Each item contained in the list or tuple should be a string ⑩.

The `seek()` method is used to set the file handler's current position. Never forget that when managing files, there'll always be a position inside that file where you are currently working on. When you open a file, that position is the beginning of the file, but as you work with it, you may advance. The `seek()` method will be useful to you when you need to work with that open file.

For example,

```
1. >>> f = open('workfile', 'w')
2. >>> f.write('0123456789abcdef')
   16
3. >>> f.close()
4. >>> f = open('workfile', 'r')
5. >>> f.seek(5)
   5
6. >>> f.read()
   '56789abcdef'
```

The *workfile* file is opened in `'w'` mode ① and some contents are written ② and the file handler is closed ③. The same file is again opened in `'r'` mode ④. The file handler starts reading from the 6th character ⑤, but counting starts from 0th character. It returns the character of the latest position from where the file handler will start to read.

```
1. >>> f = open('workfile', 'w')
2. >>> f.write('0123456789abcdef')
   16
3. >>> f.close()
4. >>> f = open('workfile', 'rb+')
5. >>> f.seek(2)
   2
6. >>> f.seek(2, 1)
   4
7. >>> f.read()
   b'456789abcdef'
8. >>> f.seek(-3, 2)
   13
9. >>> f.read()
   b'def'
```


In the above code, the file is opened in 'rb+' mode. In text files, those opened without a *b* in the mode string, only allow *seeks* relative to the beginning of the file (the exception being seeking to the end of the file with *seek(0, 2)*). Thus, statements in ⑥ and ⑧ only work if they are opened in binary mode. Statement ⑤ moves the file handler to read from the 3rd character. Statement ⑥ moves the file handler further by two characters starting from the current position. Statement ⑧ moves the file handler to the 3rd character before the end.

The *tell()* method returns the file handler's current position. For example,

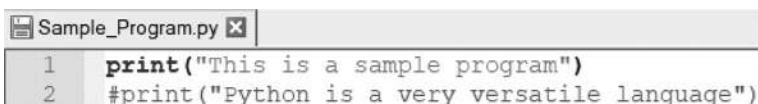
```
1. >>> f = open('workfile', 'w')
2. >>> f.write('0123456789abcdef')
   16
3. >>> f.close()
4. >>> f = open('workfile')
5. >>> s1 = f.read(2)
6. >>> print(s1)
   01
7. >>> f.tell()
   2
8. >>> s2 = f.read(3)
9. >>> print(s2)
   234
10. >>> f.tell()
    5
```

Carriage return means to return to the beginning of the current line without advancing downward. The name comes from a printer's carriage, as monitors were rare when the name was coined. This is commonly escaped as `"\r"` and abbreviated as CR.

Linefeed means to advance downward to the next line; however, it has been repurposed and renamed and used as "newline". This is commonly escaped as `"\n"` and abbreviated LF or NL. CRLF (but not CRNL) is used for the pair `"\r\n"`.

The most common difference (and probably the only one worth worrying about) is lines ending with CRLF in Windows and NL in Linux. In Windows, *tell()* can return illegal values when reading files with Linux-style line-endings. The *tell()* method returns an integer giving the file handler's current position in the file. Use binary mode ("rb") to circumvent this problem. From statement ⑤, the *read()* method returns the first two characters of the text ⑥. The *tell()* method says that the file handler is currently at the 2nd position ⑦. From statement ⑧, the *read()* method returns the next three characters of the text ⑨. The *tell()* method says that the file handler is currently at 5th position ⑩.

Program 9.6: Consider "Sample_Program.py" Python file. Write Python program to remove the comment character from all the lines in a given Python source file. Sample content of "Sample_Program.py" Python file is given below



```
1 print("This is a sample program")
2 #print("Python is a very versatile language")
```

```

1. def main():
2.     with open("Sample_Program.py") as file_handler:
3.         for each_row in file_handler:
4.             each_row = each_row.replace("#", "")
5.             print(each_row, end="")
6. if __name__ == "__main__":
7.     main()

```

OUTPUT

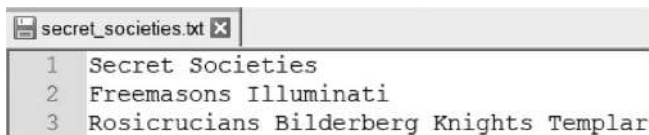
```

print("This is a sample program")
print("Python is a very versatile language")

```

Use a *for* loop to traverse ③ through each row over the file handler ②. Since each row is a string, use the *replace()* function to replace the character "#" with nothing, i.e., ""(without blanks) ④. Then print each row ⑤.

Program 9.7: Write Python Program to Reverse Each Word in "secret_societies.txt" file. Sample Content of "secret_societies.txt" is Given Below



```

secret_societies.txt
1 Secret Societies
2 Freemasons Illuminati
3 Rosicrucians Bilderberg Knights Templar

```

```

1. def main():
2.     reversed_word_list = []
3.     with open("secret_societies.txt") as file_handler:
4.         for each_row in file_handler:
5.             word_list = each_row.rstrip().split(" ")
6.             for each_word in word_list:
7.                 reversed_word_list.append(each_word[::-1])
8.             print(" ".join(reversed_word_list))
9.             reversed_word_list.clear()
10. if __name__ == "__main__":
11.     main()

```

OUTPUT

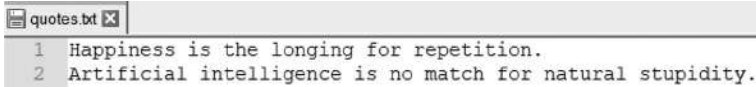
```

terceS seiteicoS
snosameerF itanimullI
snaicurcisoR grebredliB sthgink ralpmeT

```

Define *reversed_word_list* as an empty list ②. Use a *for* loop to traverse through each row ④ over the file handler ③. For each row, use the *rstrip()* function to remove trailing white spaces and split the text to a list of words ⑤. Again, traverse through this list of words ⑥. Reverse each word using *each_word[::-1]* and append the reversed word to *reversed_word_list* ⑦. Join all the reversed words in the list with a space in between them and print it ⑧. Then clear the *reversed_word_list* to make way for a new list of reversed words ⑨.

Program 9.8: Write Python Program to Count the Occurrences of Each Word and Also Count the Number of Words in a "quotes.txt" File. Sample Content of "quotes.txt" File is Given Below



```
quotes.txt
1 Happiness is the longing for repetition.
2 Artificial intelligence is no match for natural stupidity.
```

```
1. def main():
2.     occurrence_of_words = dict()
3.     total_words = 0
4.     with open("quotes.txt") as file_handler:
5.         for each_row in file_handler:
6.             words = each_row.rstrip().split()
7.             total_words += len(words)
8.             for each_word in words:
9.                 occurrence_of_words[each_word] = occurrence_of_words.get(each_word, 0) + 1
10.    print("The number of times each word appears in a sentence is")
11.    print(occurrence_of_words)
12.    print(f"Total number of words in the file are {total_words}")
13. if __name__ == "__main__":
14.    main()
```

OUTPUT

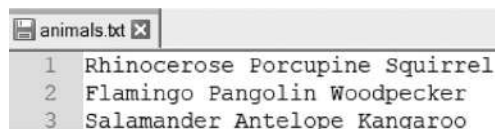
The number of times each word appears in a sentence is

```
{'Happiness': 1, 'is': 2, 'the': 1, 'longing': 1, 'for': 2, 'repetition.': 1, 'Artificial': 1, 'intelligence': 1, 'no': 1, 'match': 1, 'natural': 1, 'stupidity.': 1}
```

Total number of words in the file are 14

Define *occurrence_of_words* as dictionary ② and initialize *total_words* variable to zero ③. Use a *for* loop to traverse through each row over the file handler ⑤. For each row, use *rstrip()* function to remove trailing white spaces and split the text to a list of words ⑥. Calculate the length of the word list for each row and add it to *total_words* ⑦. Get the occurrence of each word in terms of *key:value*, where “key” is the word and “value” is the number of times the word has occurred and assign it to *occurrence_of_words* ⑧–⑨. Finally, print the results ⑩–⑫.

Program 9.9: Write Python Program to Find the Longest Word in a File. Get the File Name from User. (Assume User Enters the File Name as "animals.txt" and its Sample Contents are as Below)



```
animals.txt
1 Rhinoceros Porcupine Squirrel
2 Flamingo Pangolin Woodpecker
3 Salamander Antelope Kangaroo
```

```
1. def read_file(file_name):
2.     with open(file_name) as file_handler:
3.         longest_word = ""
4.         for each_row in file_handler:
5.             word_list = each_row.rstrip().split()
6.             for each_word in word_list:
7.                 if len(each_word) > len(longest_word):
8.                     longest_word = each_word
9.     print(f"The longest word in the file is {longest_word}")
10. def main():
11.     file_name = input("Enter file name: ")
12.     read_file(file_name)
13. if __name__ == "__main__":
14.     main()
```

OUTPUT

Enter file name: animals.txt

The longest word in the file is Rhinoceros

A user enters a file name ⑪, which should include an absolute path if the file is not present in the same directory where the Python source file is saved. Initially, the variable *longest_word* is initialized to an empty the string ③. Use a *for* loop to traverse through each row over the file handler ④. For each row, use *rstrip()* function to remove the trailing white spaces and split the text to a list of words ⑤. Loop through the *word_list* using an iterating variable *each_word* ⑥. Check whether the length of *each_word* is greater than the length of *longest_word* ⑦. If *True*, then assign that word to the *longest_word* variable ⑧. Repeat this for each word. Finally, print the longest word ⑨.

9.4 Reading and Writing Binary Files

We can usually tell whether a file is binary or text based on its file extension. This is because by convention the extension reflects the file format, and it is ultimately the file format that dictates whether the file data is binary or text. The string *'b'* appended to the mode opens the file in binary mode and now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text. Files opened in binary mode (including *'b'* in the mode argument) return contents as bytes objects without any decoding.

Program 9.10: Write Python Program to Create a New Image from an Existing Image

```
1. def main():
2.     with open("rose.jpg", "rb") as existing_image, open("new_rose.jpg", "wb") as
       new_image:
```

3. for each_line_bytes in existing_image:
4. new_image.write(each_line_bytes)
5. if __name__ == "__main__":
6. main()

In the above program, two *open()* functions are used. One to read binary data ("rb" mode) and the other to write binary data ("wb" mode) ②. Use a *for* loop to iterate through each line of bytes in the *existing_image* handler ③ and write those line of bytes using *new_image* handler ④. This behind-the-scenes modification to file data is fine for text files but will corrupt binary data like that in JPEG or EXE files. Be very careful and use binary mode when reading and writing such files.

Program 9.11: Consider a File Called "workfile". Write Python Program to Read and Print Each Byte in the Binary File

1. def main():
2. with open("workfile", "wb") as f:
3. f.write(b'abcdef')
4. with open("workfile", "rb") as f:
5. byte = f.read(1)
6. print("Print each byte in the file")
7. while byte:
8. print(byte)
9. byte = f.read(1)
10. if __name__ == "__main__":
11. main()

OUTPUT

Print each byte in the file

```
b'a'
b'b'
b'c'
b'd'
b'e'
b'f'
```

Write byte strings to *workfile* file using "wb" mode ②–③. Again, open the file in "rb" mode ④. Read the first byte and assign it to a *byte* variable ⑤. Use a *while* loop to traverse through each byte in the file ⑦. Read one byte at a time one after another ⑨ and print it ⑩.

Let's understand *bytes* in detail. Consider the code below.

1. >>> print(b'Hello')
- b'Hello'
2. >>> type(b'Hello')
- <class 'bytes'>

```
3. >>> for i in b'Hello':
...     print(i)
72
101
108
108
111
4. >>> bytes(3)
b'\x00\x00\x00'
5. >>> bytes([70])
b'F'
6. >>> bytes([72, 101, 108, 108, 111])
b'Hello'
7. >>> print(b'\x61')
b'a'
8. >>> bytes('Hi', 'utf-8')
b'Hi'
```

b'Hello' is a byte string literal ①. Bytes literals are always prefixed with 'b' or 'B' and they produce an instance of the *bytes* type instead of the *str* type ②. Python makes a clear distinction between *str* and *bytes* types.

The syntax for *bytes()* class method is,

```
bytes(source[, encoding])
```

where the *source* is used to create a *bytes* object. It can be an integer or a string.

The *bytes()* class method returns a new *bytes* object. While bytes literals and representations are based on ASCII text, *bytes* objects actually behave like immutable sequences of integers, with each value in the sequence ranging from 0 to 255 ③. A zero-filled *bytes* object with a specified length is created as shown in ④. You can construct a *bytes* object from a sequence of list items whose values are integers in the range of 0 to 255 ⑤–⑥. Having a value outside the range of 0 to 255 causes a *ValueError* exception. The *bytes* object with a numeric value of 128 or greater is expressed as an escape sequences ⑦. If the *source* is a string, then the *encoding* of the source has to be specified. In ⑧, the *encoding* type specified is 'utf-8' and it is used to encode a *str* to *bytes* object.

9.5 The Pickle Module

Strings can easily be written to and read from a file. Numbers take a bit more effort since the *read()* method only returns strings that will have to be passed to a function like *int()*, which takes a string like '123' and returns its numeric value 123. However, when you want to save more complex data types like lists, dictionaries, or class instances, things get a lot more complicated.

Rather than having the users to constantly write and debug the code to save complicated data types, Python provides a standard module called *pickle*. This is an amazing module that can take almost any Python object and convert it to a string representation; this process is called *pickling*. Reconstructing the object from the string representation is called *unpickling*. Between pickling and unpickling, the string representing the object may have been stored in a file or data or sent over a network connection to some distant machine.

If you have an object *x* and a file object *f*, which has been opened for writing, the simplest way to pickle the object is,

```
pickle.dump(x, f)
```

The *dump()* method writes a pickled representation of object *x* to the open file object *f*.

If *f* is a file object, which has been opened for reading, then the simplest way to unpickle the object is,

```
x = pickle.load(f)
```

The *load()* method reads a pickled object representation from the open file object *f* and return the reconstituted object hierarchy specified therein.

Pickling is the standard way to make Python objects that can be stored and reused by other programs or by a future invocation of the same program; the technical term for this is “a persistent object.” Because pickle is so widely used, many authors who write Python extensions must ensure that all data types are properly pickled and unpickled.

Program 9.12: Write Python Program to Save Dictionary in Python Pickle

```
1. import pickle
2. def main():
3.     bbt = {'cooper': 'sheldon'}
4.     with open('filename.pickle', 'wb') as handle:
5.         pickle.dump(bbt, handle)
6.     with open('filename.pickle', 'rb') as handle:
7.         bbt = pickle.load(handle)
8.         print(f"Unpickling {bbt}")
9. if __name__ == "__main__":
10.    main()
```

OUTPUT

```
Unpickling {'cooper': 'sheldon'}
```

Pickling is the process whereby a Python object hierarchy is converted into a byte stream ④, and unpickling is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) ⑥ is converted back into an object hierarchy. You have to import a pickle module ①. In this program, the dictionary *key:value* pair ③ is saved in a Python pickle. Pickling is done using the *dump()* method ⑤ to which you are passing the dictionary name and file object as arguments, and unpickling is done using the *load()* method ⑦ to which you have to pass the file object.

9.6 Reading and Writing CSV Files

CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. Since a comma is used to separate the values, this file format is aptly named Comma Separated Values. Consider the "contacts.csv" file, which when opened in a text editor, the CSV file looks like this (FIGURE 9.3):



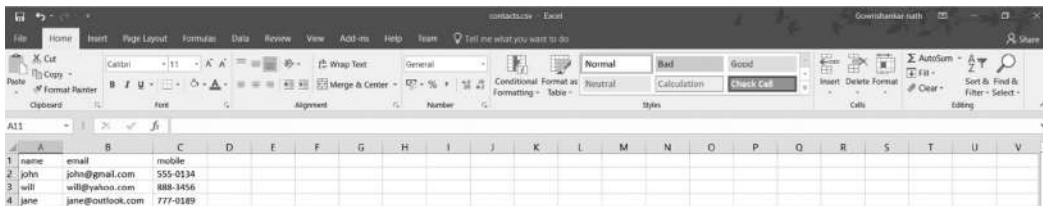
```

1 name, email, mobile
2 john, john@gmail.com, 555-0134
3 will, will@yahoo.com, 888-3456
4 jane, jane@outlook.com, 777-0189
  
```

FIGURE 9.3

CSV file "contacts.csv" opened in notepad.

Columns are separated with commas, and rows are separated by line breaks or the invisible "\n" character. However, the last value is not followed by a comma. Opened in Excel, our example CSV file "contacts.csv" looks like this (FIGURE 9.4):



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	name	email	mobile																			
2	john	john@gmail.com	555-0134																			
3	will	will@yahoo.com	888-3456																			
4	jane	jane@outlook.com	777-0189																			

FIGURE 9.4

CSV file "contacts.csv" opened in Microsoft Excel.

CSV files have .csv extensions. Because a CSV is essentially a text file, it is easy to write data to one with Python. Some advantages of CSV files are:

- It is in a human-readable format and is easier to edit manually.
- It is simple to generate, parse and handle.
- It is having a small footprint and is compact.
- It is a standard format and is supported by many applications.

Some of the characteristics of the CSV format are:

- Each record (also called as row) is located on a separate line, delimited by a line break (CRLF). For example:

ppp,qqq,rrr CRLF

xxx,yyy,zzz CRLF

- A line break may or may not be present at the end of the last record in a file. For example,

```
ppp,qqq,rrr CRLF
xxx,yyy,zzz
```

- An optional header line may appear as the first line of the file with the same format as normal record lines. This header will contain names that give a meaningful representation of the fields in the file and should contain the same number of fields as the records in the rest of the file. A record can be divided into fields. Each record consists of several fields and the fields of all the records form the columns. For example,

```
field_name, field_name, field_name CRLF
ppp,qqq,rrr CRLF
xxx,yyy,zzz CRLF
```

- Commas are used to separate the fields in each record. Each line should contain the same number of fields throughout the file. Spaces are considered to be part of a field and should not be ignored. A comma must not follow the last field in the record. For example,

```
ppp,qqq,rrr
```

- Double quotes may or may not be used to enclose each field; however, some programs, such as Microsoft Excel, do not use double quotes at all. If fields are not enclosed with double quotes, then the double quotes may not appear inside the fields. For example,

```
"ppp","qqq","rrr" CRLF
xxx,yyy,zzz
```

- If the fields are enclosed within double quotes, then a double quote appearing inside a field must be escaped by preceding it with another double quote. For example,

```
"ppp","q""qq""rrr"
```

When working with CSV files in Python, there is a built-in module called *csv*. The *csv* module implements classes to read and write data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file, which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats. The *csv* module’s reader and writer objects read and write sequences.

To read from a CSV file use *csv.reader()* method. The syntax is,

```
csv.reader(csvfile)
```

where *csv* is the module name and *csvfile* is the file object. This method returns a csv reader object, which will iterate over lines in the given *csvfile*. If *csvfile* is a file object, it should be opened with `newline = ''`.

To write to a CSV file, use the *csv.writer()* method. The syntax is,

```
csv.writer(csvfile)
```

where *csv* is the module name and *csvfile* is the file object. This method returns a csv writer object responsible for converting the user’s data into comma separated strings on the given file-like object.

The syntax for *writerow()* method is,

```
csvwriter.writerow(row)
```

where the *csvwriter* is the object returned by the *writer()* method and *writerow()* method will write the *row* argument to the *writer()* method's file object. A row must be an iterable of strings or numbers for writer objects, and a dictionary mapping fieldnames to strings or numbers (by passing them through *str()* first) for *DictWriter* objects.

The syntax for *writerows()* method is,

```
csvwriter.writerows(rows)
```

Here, the *writerows()* method will write all the *rows* argument (a list of row objects) to the *writer()* method's file object.

Programmers can also read and write data in dictionary format using the *DictReader* and *DictWriter* classes, which makes the data much easier to work with. Interacting with your data in this way is much more natural for most Python applications and will be easier to integrate it into your code thanks to the familiarity of dictionaries.

The syntax for *DictReader* is,

```
class csv.DictReader(f, fieldnames=None, restkey = None)
```

This creates an object that operates like a regular reader but maps the information in each row to an *OrderedDict* (prior to Python 3.6 version) or regular dictionary (in Python 3.6 and above versions), whose keys are given by the optional *fieldnames* argument. An *OrderedDict* is a dictionary that remembers the order that keys were first inserted. The optional *fieldnames* keyword argument is a sequence. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the *fieldnames*. If a row has more fields than the *fieldnames*, then the remaining data is put in a list and stored with the *fieldname* specified by *restkey* keyword argument (which by default is *None*). If a non-blank row has fewer fields than *fieldnames*, the missing values are filled-in with *None*.

The syntax for *DictWriter* is,

```
class csv.DictWriter(f, fieldnames, extrasaction='raise')
```

This creates an object that operates like a regular writer but maps dictionaries onto output rows. The *fieldnames* parameter is a sequence of keys that identify the order in which values in the dictionary passed to the *writerow()* method and are written to file *f*. If the dictionary passed to the *writerow()* method contains a key not found in *fieldnames*, then the optional *extrasaction* argument indicates what action to take. If it is set to *'raise'*, the default value, a *ValueError* is raised. If it is set to *'ignore'*, extra values in the dictionary are ignored. Note that unlike the *DictReader* class, the *fieldnames* argument of the *DictWriter* class is not optional.

Program 9.13: Write Python program to read and display each row in "biostats.csv" CSV file. Sample content of "biostats.csv" is given below.

biostats.csv					
1	"Name",	"Sex",	"Age",	"Height (in)",	"Weight (lbs)"
2	"Alex",	"M",	41,	74,	170
3	"Bert",	"M",	42,	68,	166
4	"Elly",	"F",	30,	66,	124
5	"Fran",	"F",	33,	66,	115

```

1. import csv
2. def main():
3.     with open('biostats.csv', newline='') as csvfile:
4.         csv_reader = csv.reader(csvfile)
5.         print("Print each row in CSV file")
6.         for each_row in csv_reader:
7.             print(",".join(each_row))
8. if __name__ == "__main__":
9.     main()

```

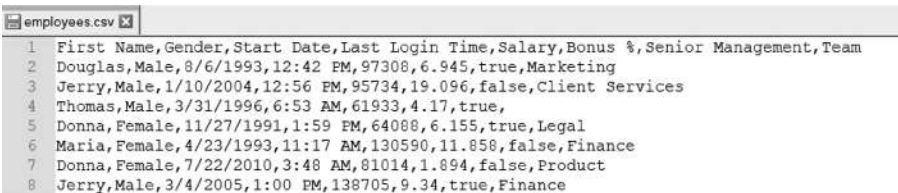
OUTPUT

Print each row in CSV file

Name,	"Sex",	"Age",	"Height (in)",	"Weight (lbs)"
Alex,	"M",	41,	74,	170
Bert,	"M",	42,	68,	166
Elly,	"F",	30,	66,	124
Fran,	"F",	33,	66,	115

You must import a csv module ①. In this code, open *biostats.csv* file as *csvfile* file handler object ③, and then use the *csv.reader()* method to extract the data into the *csv_reader* reader object ④, which you can then iterate over to retrieve each line in csv file ⑥. You have to pass *csvfile* file handler object as an argument to *csv.reader()* method. Here, *each_row* is a list with string items, which are joined with a comma (',') between them. Print each line ⑦.

Program 9.14: Write Python program to read and display rows in "employees.csv" CSV file that start with employee name "Jerry". Sample content of "employees.csv" is given below



1	First Name,Gender,Start Date,Last Login Time,Salary,Bonus %,Senior Management,Team
2	Douglas,Male,8/6/1993,12:42 PM,97308,6.945,true,Marketing
3	Jerry,Male,1/10/2004,12:56 PM,95734,19.096,false,Client Services
4	Thomas,Male,3/31/1996,6:53 AM,61933,4.17,true,
5	Donna,Female,11/27/1991,1:59 PM,64088,6.155,true,Legal
6	Maria,Female,4/23/1993,11:17 AM,130590,11.858,false,Finance
7	Donna,Female,7/22/2010,3:48 AM,81014,1.894,false,Product
8	Jerry,Male,3/4/2005,1:00 PM,138705,9.34,true,Finance

```

1. import csv
2. def main():
3.     with open('employees.csv', newline='') as csvfile:
4.         csv_reader = csv.reader(csvfile)
5.         print("Print rows in CSV file that start with employee name 'Jerry'")
6.         for each_row in csv_reader:
7.             if each_row[0] == "Jerry":
8.                 print(",".join(each_row))
9. if __name__ == "__main__":
10.    main()

```

OUTPUT

Print rows in CSV file that start with employee name 'Jerry'

Jerry,Male,1/10/2004,12:56 PM,95734,19.096,false,Client Services

Jerry,Male,3/4/2005,1:00 PM,138705,9.34,true,Finance

All the employee names are in the first column of the *employees.csv* file. In the code, *each_row* is a list of strings ⑥. Use an *if* condition to check whether the first item in the list of strings is equal to "Jerry" ⑦. If *True*, then print that line by joining all the items in the list with a comma between them ⑧.

Program 9.15: Write Python program to write the data given below to a CSV file.

Category,Winner,Film,Year

Best Picture,Doug Mitchell and George Miller,Mad Max: Fury Road,2015

Visual Effects,Richard Stammers,X-Men:Days of Future Past,2014

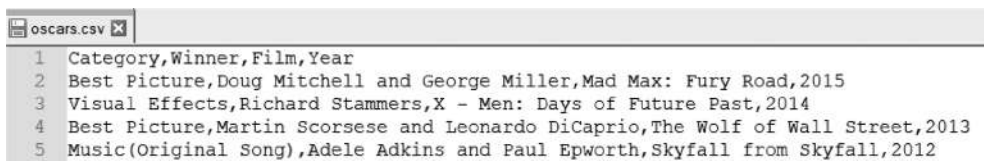
Best Picture,Martin Scorsese and Leonardo DiCaprio,The Wolf of Wall Street,2013

Music(Original Song),Adele Adkins and Paul Epworth,Skyfall from Skyfall,2012

```

1. import csv
2. def main():
3.     csv_header_name = ['Category', 'Winner', 'Film', 'Year']
4.     each_row = [['Best Picture', 'Doug Mitchell and George Miller', 'Mad Max: Fury
        Road', '2015'],
        ['Visual Effects', 'Richard Stammers', 'X - Men: Days of Future Past', '2014'],
        ['Best Picture', 'Martin Scorsese and Leonardo DiCaprio', 'The Wolf of Wall
        Street', '2013'],
        ['Music(Original Song)', 'Adele Adkins and Paul Epworth', 'Skyfall from
        Skyfall', '2012']]
5.     with open('oscars.csv', 'w', newline='') as csvfile:
6.         csv_writer = csv.writer(csvfile)
7.         csv_writer.writerow(csv_header_name)
8.         csv_writer.writerows(each_row)
9. if __name__ == "__main__":
10.     main()

```

OUTPUT


```

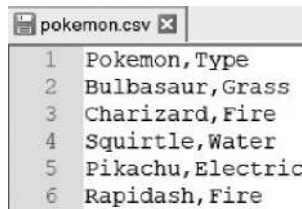
oscars.csv
1 Category,Winner,Film,Year
2 Best Picture,Doug Mitchell and George Miller,Mad Max: Fury Road,2015
3 Visual Effects,Richard Stammers,X - Men: Days of Future Past,2014
4 Best Picture,Martin Scorsese and Leonardo DiCaprio,The Wolf of Wall Street,2013
5 Music(Original Song),Adele Adkins and Paul Epworth,Skyfall from Skyfall,2012

```

Import *csv* module ①. The *csv_header_name* is a list containing all the fieldnames ③, and *each_row* is a nested list consisting of values for each field names in a row ④. The *oscars.csv* file is opened in write mode ⑤. When you have a set of data that you want to store in a

CSV file, use the `writer()` method. The `writer()` method returns an object suitable for writing. The file object `csvfile` is passed as an argument to the `csv.writer()` and this method returns a `csv_writer` object ⑥. Then the `writerow()` method of `csv_writer` object is invoked to write `csv_header_name` fieldnames to the CSV file ⑦. This will be the first row. Use the `writerows()` method of `csv_writer` object to write multiple rows at once to CSV file ⑧.

Program 9.16: Write Python Program to Read Data from "pokemon.csv" csv File Using DictReader. Sample Content of "pokemon.csv" is Given Below



1	Pokemon, Type
2	Bulbasaur, Grass
3	Charizard, Fire
4	Squirtle, Water
5	Pikachu, Electric
6	Rapidash, Fire

```

1. import csv
2. def main():
3.     with open('pokemon.csv', newline='') as csvfile:
4.         reader = csv.DictReader(csvfile)
5.         for row in reader:
6.             print(f"{row['Pokemon']}, {row['Type']}")
7. if __name__ == "__main__":
8.     main()

```

OUTPUT

```

Bulbasaur, Grass
Charizard, Fire
Squirtle, Water
Pikachu, Electric
Rapidash, Fire

```

The first row in this CSV file *pokemon.csv* contains the fieldnames (Pokemon and Type), which provide a label for each column of data. The rows in this file contain pairs of values separated by a comma. These labels are optional but tend to be very helpful, especially when you have to actually look at this data yourself. You can loop through each row of the `reader` object ⑤ but notice how you can now access each row's columns by their label ⑥, which in this case is *Pokemon* and *Type*.

Program 9.17: Write Python program to demonstrate the writing of data to a CSV file using DictWriter class

```

1. import csv
2. def main():
3.     with open('names.csv', 'w', newline='') as csvfile:
4.         field_names = ['first_name', 'last_name']

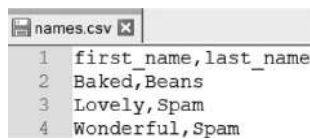
```

```

5.     writer = csv.DictWriter(csvfile, fieldnames=field_names)
6.     writer.writeheader()
7.     writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
8.     writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
9.     writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
10. if __name__ == "__main__":
11.     main()

```

OUTPUT



1	first_name, last_name
2	Baked, Beans
3	Lovely, Spam
4	Wonderful, Spam

You can also create a CSV file using dictionaries. Here in the code, the CSV file *names.csv* is opened in ‘w’ mode and *csvfile* is the CSV file object ③. A *field_names* list is created with *first_name* and *last_name* as items ④. For the DictWriter, the *csvfile* file object is passed as the first argument and the *field_names* list is assigned to *fieldnames* keyword argument ⑤. Here in the code, a dictionary with *first_name* and *last_name* fields as keys are created. The *writer* object uses *writeheader()* and *writerow()* methods to write the data to *names.csv* file. The *writeheader()* method writes a row with the fieldnames ⑥, then values for each fieldnames in the row are written using the *writerow()* method ⑦–⑨.

9.7 Python os and os.path Modules

Python *os* module provides a portable way of using operating system dependent functionality. For accessing the filesystems, use the *os* module. If you want to manipulate paths, use the *os.path* module. Python *os.path* works in a strange way. It looks like *os* should be a package with a submodule *path*, but, in reality, *os* is a normal module that does magic with *sys.modules* to inject *os.path*. Here’s what happens. When Python starts up, it loads a bunch of modules into *sys.modules*. They are not bound to any names in your script, but you can access the already-created modules when you import them. The *sys.modules* is a *dict* in which modules are cached. When you import a module, if it already has been imported somewhere, it gets the instance stored in *sys.modules*. The *os* is among the modules that are loaded when Python starts up. It assigns its *path* attribute to an *os*-specific path module. It injects *sys.modules['os.path'] = path* so that you are able to do “*import os.path*” as though it was a submodule. Think of *os.path* as a module that you want to use rather than a thing in the *os* module. Therefore, even though it is not really a submodule of a package called *os*, you import it sort of like it is one, and always do *import os.path*.

Various methods of the *os* module are shown in [TABLE 9.4](#).

TABLE 9.4Various Methods of *os* Module

Methods	Syntax	Description
<code>chdir()</code>	<code>os.chdir(path)</code>	This method changes the current working directory to <i>path</i> .
<code>getcwd()</code>	<code>os.getcwd()</code>	This method returns a string representing the current working directory.
<code>makedirs()</code>	<code>os.makedirs(path)</code>	This method creates the directory named <i>path</i> . If the directory already exists, <i>FileExistsError</i> is raised.
<code>remove()</code>	<code>os.remove(path)</code>	This method removes (deletes) the file <i>path</i> . If the <i>path</i> is a directory, <i>OSError</i> is raised. Use <i>rmdir()</i> to remove directories. In Windows, attempting to remove a file that is in use causes an exception to be raised; in Linux, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.
<code>rmdir()</code>	<code>os.rmdir(path)</code>	This method removes (deletes) the directory <i>path</i> . It only works when the directory is empty, otherwise, <i>OSError</i> is raised.
<code>walk()</code>	<code>os.walk(top, topdown=True)</code>	This method generates the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory <i>top</i> (including <i>top</i> itself), it yields a 3-tuple (<i>dirpath</i> , <i>dirnames</i> , <i>filenames</i>). The <i>dirpath</i> is a string, the path to the directory. The <i>dirnames</i> is a list of the names of the subdirectories in <i>dirpath</i> (excluding '.' and '..'). The <i>filenames</i> is a list of the names of the non-directory files in <i>dirpath</i> . Note that the names in the lists contain no path components. To get a full path (which begins with <i>top</i>) to a file or directory in <i>dirpath</i> , do <i>os.path.join(dirpath, name)</i> .
<code>rename()</code>	<code>os.rename(old_name, new_name)</code>	This method is used to rename the file from <i>old_name</i> to <i>new_name</i> .
<code>listdir()</code>	<code>os.listdir(path='.')</code>	This method returns a list containing the names of the entries in the directory given by <i>path</i> . The list is in arbitrary order and does not include the special entries '.' and '..' even if they are present in the directory.

Note: The *path* argument can be passed as either strings or bytes.

Various methods of the *os.path* module are shown in [TABLE 9.5](#).

TABLE 9.5Various Methods of *os.path* Module

Methods	Syntax	Description
<code>join()</code>	<code>os.path.join(path, *paths)</code>	This method is used to join one or more path components intelligently. The return value is the concatenation of <i>path</i> and any members of <i>*paths</i> with exactly one directory.
<code>exists()</code>	<code>os.path.exists(path)</code>	This method returns True if <i>path</i> refers to an existing path else returns False for broken links.
<code>isfile()</code>	<code>os.path.isfile(path)</code>	This method returns True if <i>path</i> is an existing regular file.
<code>isdir()</code>	<code>os.path.isdir(path)</code>	This method returns True if <i>path</i> is an existing directory.
<code>getmtime()</code>	<code>os.path.getmtime(path)</code>	This method returns the time of last modification of <i>path</i> .
<code>abspath()</code>	<code>os.path.abspath(path)</code>	This method returns a normalized absolutized version of the pathname <i>path</i> .

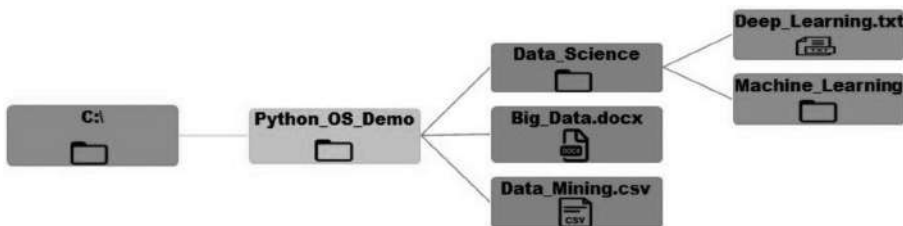
(Continued)

TABLE 9.5 (Continued)Various Methods of *os.path* Module

Methods	Syntax	Description
<code>path.isabs()</code>	<code>os.path.isabs(path)</code>	This method returns True if <i>path</i> is an absolute pathname.
<code>relpath()</code>	<code>os.path.relpath(path, start=os.curdir)</code>	This method returns a relative filepath to <i>path</i> either from the current directory or from an optional start directory.
<code>dirname()</code>	<code>os.path.dirname(path)</code>	This method returns the directory name of the pathname <i>path</i> .
<code>basename()</code>	<code>os.path.basename(path)</code>	This method returns the base name of pathname <i>path</i> .
<code>split()</code>	<code>os.path.split(path)</code>	This method splits the pathname <i>path</i> into a pair, (head, tail) where the tail is the last pathname component and the head is everything leading up to that. The tail part will never contain a slash; if path ends in a slash, the tail will be empty. If there is no slash in the path, the head will be empty. If the path is empty, both head and tail are empty.
<code>splittext()</code>	<code>os.path.splittext(path)</code>	This method splits the pathname <i>path</i> into a pair (root, ext) such that <code>root + ext == path</code> where <i>ext</i> begins with a period and contains at most one period and <i>root</i> is everything leading up to that.
<code>getsize()</code>	<code>os.path.getsize(path)</code>	This method returns the size, in bytes, of <i>path</i> .

Note: The *path* argument can be passed as either strings or bytes.

For example, consider the file structure is shown below (FIGURE 9.5).

**FIGURE 9.5**File Structure to demonstrate *os* and *os.path* modules.

```

1. >>> import os
2. >>> os.getcwd()
   'C:\Python_OS_Demo'
3. >>> os.rename("NLP.csv", "Data_Mining.csv")
4. >>> os.remove("Data_Mining")
5. >>> os.mkdir("Data_Science")
6. >>> os.chdir("Data_Science")
7. >>> os.getcwd()
   'C:\Python_OS_Demo\Data_Science'
8. >>> os.mkdir("Machine_Learning")
9. >>> os.rmdir("Machine_Learning")

```


Demonstration of various methods of the *os* module ①–⑨.

```

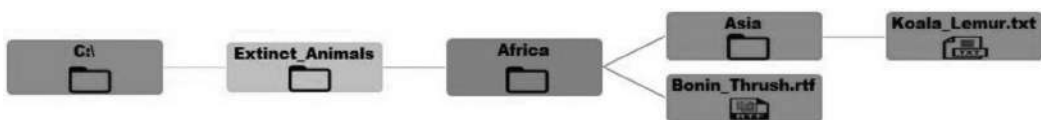
1. >>> os.path.join("C:\\Python_OS_Demo", "Data_Science")
   'C:\\Python_OS_Demo\\Data_Science'
2. >>> os.path.abspath("Big_Data.docx")
   'C:\\Python_OS_Demo\\Big_Data.docx'
3. >>> os.path.getsize("Big_Data.docx")
   12820
4. >>> os.listdir("C:\\Python_OS_Demo")
   ['Data_Mining.csv', 'Data_Science', 'Big_Data.docx']
5. >>> os.path.split("C:\\Python_OS_Demo\\Data_Science\\Deep_Learning.txt")
   ('C:\\Python_OS_Demo\\Data_Science', 'Deep_Learning.txt')
6. >>> os.path.splitext("C:\\Python_OS_Demo\\Data_Science\\Deep_Learning.txt")
   ('C:\\Python_OS_Demo\\Data_Science\\Deep_Learning', '.txt')
7. >>> os.path.basename("C:\\Python_OS_Demo\\Data_Science")
   'Data_Science'
8. >>> os.path.dirname("C:\\Python_OS_Demo\\Data_Science\\Deep_Learning.txt")
   'C:\\Python_OS_Demo\\Data_Science'
9. >>> os.path.relpath("C:\\Python_OS_Demo\\Data_Science")
   'Data_Science'
10. >>> os.chdir(c:\\Thisdirectorydoesnotexist")
      File "<stdin>", line 1
          os.chdir(c:\\Thisdirectorydoesnotexist")
              ^

```

SyntaxError: invalid syntax

Demonstration of various methods of the *os.path* module ①–⑩.

Program 9.18: Consider the File Structure Given Below. Write Python Program to Delete All the Files and Subdirectories from the Extinct_Animals Directory



```

1. import os
2. def delete_files_recursively(directory_path):
3.     for root, dirs, files in os.walk(directory_path):
4.         for file in files:
5.             file_path = os.path.join(root, file)
6.             try:
7.                 print(f"{file_path} is deleted")
8.                 os.remove(file_path)

```

```
9.         except Exception as e:
10.             print(e)
11. def main():
12.     directory_path = input('Enter the directory path from which you want to delete
        files recursively ')
13.     delete_files_recursively(directory_path)
14. if __name__ == "__main__":
15.     main()
```

OUTPUT

Enter the directory path from which you want to delete files recursively C:\Extinct_Animals
C:\Extinct_Animals\Africa\Koala_Lemur.txt is deleted
C:\Extinct_Animals\Africa\Asia\Bonin_Thrush.rtf is deleted

The function *delete_files_recursively()* deletes all the files from a directory and also from its subdirectories ②. The user enters the path for the directory from which files need to be deleted recursively, and the file name is passed as an argument to the *delete_files_recursively()* function ②–③. Use a *for* loop with the *walk()* method to walk through all the subdirectories and files of the user-entered directory. Here, *root* is a string variable, while *dirs* and *files* are list variables. In the initial run, the value of *root* is C:\\Extinct_Animals, and list *dirs* has ['Africa'] as the item and is the only subdirectory found under Extinct_Animals directory. Since there are no files in the Extinct_Animals directory, list *files* is an empty list. In the second run, the value of *root* is 'C:\\Extinct_Animals\\Africa', list *dirs* has ['Asia'] item as Asia, which is the only subdirectory under Africa and list *files* has ['Bonin_Thrush.rtf'] item. Loop through the *files* list and get the absolute path for the file Bonin_Thrush.rtf using *join()* method which in this case is 'C:\\Extinct_Animals\\Africa\\Bonin_Thrush.rtf'. Delete that file. In the next run, the value of *root* is 'C:\\Extinct_Animals\\Africa\\Asia' and *dirs* is an empty list as there are no subdirectories under Asia. The *files* list has ['Koala_Lemur.txt'] item. Get the complete path for the file using *join()* method and delete that file ③–⑧.

9.8 Summary

- Python supports two basic file types, namely text files and binary files.
- File objects can be used to read/write data to/from files. You can open a file with mode 'r' for reading, 'w' for writing, and 'a' for appending.
- The *read()*, *readline()*, and *readlines()* methods are used to read data from a file.
- The *write()* and *writes()* methods are used to write data to a file.
- The file object should be closed after the file is processed to ensure that the content is saved properly.
- The dictionary data can be read and written to a CSV file using *DictReader()* and *DictWriter()* classes.
- The *os* Module methods are used to perform some important processing on files.

Multiple Choice Questions

1. Consider a file named rome.txt, then the statement used to open a file for reading, we use
 - a. `infile = open("c:\rome.txt", "r")`
 - b. `infile = open("c:\\rome.txt", "r")`
 - c. `infile = open(file = "c:\rome.txt", "r")`
 - d. `infile = open(file = "c:\\rome.txt", "r")`
2. Suppose there is a file named rome.txt, then the statement used to open a file for writing, we use
 - a. `outfile = open("c:\rome.txt", "w")`
 - b. `outfile = open("c:\\rome.txt", "w")`
 - c. `outfile = open(file = "c:\rome.txt", "w")`
 - d. `outfile = open(file = "c:\\rome.txt", "w")`
3. Presume a file named rome.txt, then the statement used for appending data is
 - a. `outfile = open("c:\rome.txt", "a")`
 - b. `outfile = open("c:\\rome.txt", "rw")`
 - c. `outfile = open(file = "c:\rome.txt", "w")`
 - d. `outfile = open(file = "c:\\rome.txt", "w")`
4. Which of the following statements are true?
 - a. When you open a file for reading in 'r' mode, if the file does not exist, an error occurs
 - b. When you open a file for writing in 'w' mode, if the file does not exist, a new file is created
 - c. When you open a file for writing in 'w' mode, if the file exists, the existing file is overwritten with the new file
 - d. All of the mentioned
5. The code snippet to read two characters from a file object *infile* is
 - a. `infile.read(2)`
 - b. `infile.read()`
 - c. `infile.readline()`
 - d. `infile.readlines()`
6. If you want to read the entire contents of the file using file object *infile* then
 - a. `infile.read(2)`
 - b. `infile.read()`
 - c. `infile.readline()`
 - d. `infile.readlines()`

7. Predict the output of the following code:

```
for i in range(5):  
    with open("data.txt", "w") as f:  
        if i > 0:  
            break  
    print(f.closed)
```

- a. True
 - b. False
 - c. None
 - d. Error
8. The syntax to write to a CSV file is
- a. CSV.DictWriter(filehandler)
 - b. CSV.reader(filehandler)
 - c. CSV.writer(filehandler)
 - d. CSV.write(filehandler)
9. Which of the following is not a valid mode to open a file
- a. ab
 - b. r+
 - c. w+
 - d. rw
10. The *readline()* method returns
- a. str
 - b. a list of lines
 - c. a list of single characters
 - d. a list of integers
11. Which of the following is not a valid attribute of the file object `file_handler`
- a. `file_handler.size`
 - b. `file_handler.name`
 - c. `file_handler.closed`
 - d. `file_handler.mode`
12. Chose a keyword that is not an attribute of a file.
- a. `closed`
 - b. `softspace`
 - c. `rename`
 - d. `mode`

13. The functionality of `tell()` method in Python is
 - a. tells you the current position within the file
 - b. tells you the end position within the file
 - c. tells you the file is opened or not
 - d. None of the above
14. The syntax for renaming of a file is
 - a. `rename(current_file_name, new_file_name)`
 - b. `rename(new_file_name, current_file_name,)`
 - c. `rename((current_file_name, new_file_name))`
 - d. None of the above
15. To remove a file, the syntax used is,
 - a. `remove(file_name)`
 - b. `(new_file_name, current_file_name,)`
 - c. `remove((), file_name)`
 - d. None of the above
16. An absolute path name begins at the
 - a. leaf
 - b. stem
 - c. root
 - d. current directory
17. The functionality of `seek()` function is
 - a. sets the file's current position at the offset
 - b. sets the file's previous position at the offset
 - c. sets the file's current position within the file
 - d. None of the above
18. What is unpickling?
 - a. It is used for object de-serialization
 - b. It is used for object serialization
 - c. It is used for synchronization
 - d. It is used for converting an object to its string representation
19. Which of the following are basic I/O connections in the file?
 - a. Standard Input
 - b. Standard Output
 - c. Standard errors
 - d. All of the above

20. The mode that is used to refer to binary data is
 - a. r
 - b. w
 - c. +
 - d. b
21. File type is represented by its
 - a. file name
 - b. file extension
 - c. file identifier
 - d. file variable
22. The method that returns the time of last modification of the file is
 - a. getmtime()
 - b. gettime()
 - c. time()
 - d. localtime()
23. Pickling is used for?
 - a. object deserialization
 - b. object serialization
 - c. synchronization
 - d. converting string representation to object

Review Questions

1. Define file and explain the different types of files.
2. Explain the different file mode operations with examples.
3. Describe with an example how to read and write to a text file.
4. Explain with an example how to read and write a binary file.
5. Illustrate with an example how to read and write a csv file.
6. Describe all the methods available in the *os* module.
7. Write a program that prompts the user to enter a text file, reads words from the file, and displays all the non-duplicate words in ascending order.
8. Write a program to get the file size of a plain text file.
9. Write a program that prompts the user to enter a text filename and displays the number of vowels and consonants in the file.

10. Write a program to read the first n lines of a file. Prompt the user to enter the value for n .
11. Write a program that reads the contents of the file and counts the occurrences of each letter. Prompt the user to enter the filename.
12. Write a program to read the last n lines of a file. Prompt the user to enter the value for n .
13. Write a program to combine each line from the first file with the corresponding line in the second file.
14. Write a program to remove newline characters from a file.
15. Write a program to read the random line from a file.
16. Write a program to read and write the contents from one csv file to another.

10

Regular Expression Operations

AIM

Comprehend the rules to construct regular expressions, and apply them to text to search for patterns and make changes.

LEARNING OUTCOMES

After completing this chapter, you should be able to

- Create regular expressions that match text patterns.
- Apply regular expressions to text using methods from *re* module.
- Illustrate the use of metacharacters in building regular expressions.
- Discover commonly used operations involving regular expressions.
- Understand how to use regular expressions for text searching or string replacement.

Regular expressions, also called REs, or regexes, or regex patterns, provide a powerful way to search and manipulate strings. Regular expressions are essentially a tiny, highly specialized programming language embedded inside Python and made available through the *re* module. Regular expressions use a sequence of characters and symbols to define a pattern of text. Such a pattern is used to locate a chunk of text in a string by matching up the pattern against the characters in the string. Regular expressions are useful for finding phone numbers, email addresses, dates, and any other data that has a consistent format.

10.1 Using Special Characters

A regular expression pattern is composed of simple characters, such as *abc*, or a combination of simple and special characters, such as *ab*c*. Simple patterns are constructed of characters for which you want to find a text match. For example, the pattern *abc* matches character combinations in strings only when the characters "abc" occur together and exactly in that order. Such a match would succeed in the strings: "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both the cases, the match is with the substring "abc". There is no match in the string "Grab crab" because, while it contains the substring "ab c", it does not contain the exact substring "abc".

Some characters are metacharacters, also called as special characters, and don't match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the regular expressions by repeating them or changing their meaning. When the search for a match requires something more than a direct match, such as finding one or more b's or finding white space, the pattern includes special characters. For example, the pattern *ab*c* matches any character combination in which a single "a" is followed by zero or more 'b's (* means 0 or more occurrences of the preceding item) and then immediately followed by "c". In the string "cbbabbbbcdebc," the pattern matches the substring "abbbbc".

Below you will find a complete list and description of the special characters that can be used in regular expressions.

Special Character → [xyz]

Description → Square brackets are used to indicate a set of characters. The square brackets [] are used for specifying a character class, also called a "character set," which is a set of characters that you wish to match. Place the characters you want to match between square brackets. This pattern type matches any one of the characters in the brackets, including escape sequences. Special characters like the dot(.) and asterisk(*) are not special inside a character set, so they do not need to be escaped. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a hyphen (-).

Example → The pattern [abc] will match any one of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters. The pattern [akm\$] will match any of the characters 'a', 'k', 'm', or '\$'. The character '\$' is usually a special character, but inside a character class it is stripped of its special nature. The pattern [a-d], which performs the same match as [abcd], matches the 'b' in "brisket" and the 'c' in "city".

Special Character → . (a period)

Description → Matches any single character except newline '\n'.

Example → The pattern .n matches the substrings 'an' and 'on' in the string "nay, an apple is on the tree", but not 'nay'.

Special Character → ^

Description → Matches the start of the string and, in multiline mode, also matches immediately after each newline.

Example → The pattern ^A does not match the character 'A' in the string "an A" but does match the character 'A' in the string "An E". You can match the characters not listed within the class by complementing the character set. That is, if the first character of the character set is '^', all the characters that are not in the character set will be matched. The character '^' has no special meaning if it's not the first character in the character set. You can specify a range of characters by using a hyphen. Everything that works in the normal character set also works here. For example, the pattern [^abc] is the same as [^a-c] pattern. They initially match the character 'r' in the string "brisket" and the character 'h' in the string "chop."

Special Character → \$

Description → Matches the end of the string or just before the newline at the end of the string.

Example → The pattern t\$ does not match the character 't' in the string "eater" but does match it in the string "eat".

Special Character → *

Description → Matches the preceding expression 0 or more times.

Example → The pattern bo* matches the substring 'boooo' in the string "A ghost boooooed" and matches the character 'b' in the string "A bird warbled" but nothing in the string "A goat grunted".

Special Character → +

Description → Matches the preceding expression 1 or more times.

Example → The pattern a+ matches the character 'a' in the string "candy" and all the a's in "caaaaaandy", but nothing in "cndy".

Special Character → ?

Description → Matches the preceding expression 0 or 1 time.

Example → The pattern e?le? matches the substring 'el' in the string "angel" and matches the substring 'le' in the string "angle" and also the character 'l' in the string "oslo". If used immediately after any of the special characters *, +, or {}, makes the special character non-greedy (matching the fewest possible characters), as opposed to the default, which is greedy (matching as many characters as possible). For example, applying the pattern \d+ to the string "123abc" matches the substring "123". But applying the pattern \d+? to that same string matches only the character "1".

Special Character → \d

Description → Matches any decimal digit [0-9]

Example → The pattern \d or [0-9] matches the character '2' in the string "B2 is the suite number."

Special Character → \D

Description → Matches any non-digit character. Equivalent to [^0-9].

Example → The pattern \D or [^0-9] matches the character 'B' in the string "B2 is the suite number."

Special Character → \w

Description → Matches a "word" character and it can be a letter or digit or underscore. It is equivalent to [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word character, not a whole word.

Example → The pattern `\w` matches the character 'a' in the string "apple", the character '5' in the string "\$5.28" and the character '3' in the string "3D."

Special Character → `\W`

Description → Matches any non-word character. Equivalent to `[^A-Za-z0-9_]`.

Example → The pattern `\W` or `[^A-Za-z0-9_]` matches the character '%' in the string "50%."

Special Character → `\s`

Description → Matches a single whitespace character including space, newline, tab, form feed. Equivalent to `[\n\t\f]`.

Example → The pattern `\s\w*` matches the substring 'bar' in the string "foo bar."

Special Character → `\S`

Description → Matches any non-whitespace character. Equivalent to `[^ \n\t\f]`.

Example → The pattern `\S*` matches the substring 'foo' in the string "foo bar."

Special Character → `\b`

Description → Matches a word boundary.

There are three different positions that qualify as word boundaries when the special character `\b` is placed:

- Before the first character in the string and if the first character in the string is a word character.
- After the last character in the string and if the last character in the string is a word character.
- Between two characters in the string, where one is a word character in the string and the other is not a word character.

The special character `\b` allows you to perform a search of a complete word using a regular expression in the form of `\bword\b`; it won't match when it is contained inside another word. Note that a word is defined as a sequence of word characters. The `\b` special character matches the empty string, but only at the beginning or end of a word.

Example → The pattern `\bm` matches the character 'm' in the string "moon".

The pattern `oo\b` does not match the substring 'oo' in the string "moon", because the substring 'oo' is followed by 'n' which is a word character.

The pattern `oon\b` matches the substring 'oon' in the string "moon", because 'oon' is the end of the string, thus not followed by a word character.

The pattern `\bfoo\b` matches the string 'foo', 'foo.', '(foo)', 'bar foo baz' but not 'foobar' or 'foo3'.

The pattern `\w\b\w` will never match anything because `\b` character can never be preceded and followed by a word character.

Special Character → `\B`

Description → Matches a non-word boundary. This matches the following cases when the special character `\B` is placed:

- Before the first character of the word and if the first character is not a word character.
- After the last character of the word and if the last character is not a word character.
- Between two-word characters
- Between two non-word characters

The beginning and end of a string are considered non-word characters. The '\B' special character matches an empty string, only when it is not at the beginning or end of the word.

Example → The pattern \B. matches the substring 'oo' in "noonday", and the pattern y\B. matches the substring 'ye' in the string "possibly yesterday."

Special Character → \

Description → Matches according to the following rules: A backslash that precedes a non-special character indicates that the next character is special and is not to be interpreted literally. A backslash that precedes a special character indicates that the next character is not special and should be interpreted literally.

Example → The pattern 'b' without a preceding '\' generally matches lowercase 'b's wherever they occur. But a '\b' by itself does not match any character; it forms the special word boundary character.

The pattern a* relies on the special character '*' to match 0 or more a's. By contrast, the pattern a* removes the specialness of the '*' to enable matches with strings like 'a*'.

Special Character → {m, n}

Description → Where m and n are positive integers and m ≤ n. Matches at least m and at most n occurrences of the preceding expression. If n is omitted, i.e. {m,}, then it matches at least m occurrences of the preceding expression. Here m must be a positive integer.

Example → The pattern a{1,3} matches nothing in the string "cndy", but matches the character 'a' in the string "candy". The pattern a{1,3} matches the first two a's in the string "caandy," and the first three a's in the string "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it. The pattern a{2,} will match substrings "aa", "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaa" but not "a".

Special Character → {m}

Description → Matches exactly m occurrences of the preceding expression. Here m must be a positive integer.

Example → The pattern a{2} doesn't match the character 'a' in the string "candy," but it does match all of the a's in the string "caandy," and the first two a's in the string "caaandy."

Special Character → |

Description → A | B Matches 'A', or 'B' (if there is no match for 'A'), where A and B are regular expressions.

Example → The pattern `green|red` matches the substring 'green' in the string "green apple" and matches the substring 'red' in the string "red apple." The order of 'A' and 'B' matters. For example, the pattern `a*|b` matches the empty string in the string "b", the pattern `but b|a*` matches character "b" in the same string.

[Adapted with kind permission from MDN <https://developer.mozilla.org/>]

10.1.1 Using *r* Prefix for Regular Expressions

Consider the regular expression, `r'^$',` This regular expression matches an empty line. The '^' indicates the start of a line, and the '\$' indicates the end of a line. Having nothing between the special characters '^' and '\$', therefore, matches an empty line.

The 'r' prefix tells Python that the expression is a raw string and are handy in regular expressions. In a raw string, escape sequences are not parsed. For example, '\n' is a single newline character. But, `r'\n'` would be two characters: a backslash and an 'n'. Using an expression like `r'[\w]'` instead of `'[\w]'` results in easier to read expressions.

10.1.2 Using Parentheses in Regular Expressions

Special Character → (...)

Description → Matches whatever regular expression pattern is inside the parentheses and causes that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use. Parts of a regular expression pattern bounded by parentheses are called groups, and they contain the matched substring. The parentheses are also called as capturing parentheses or capturing group. Parentheses indicate the start '(' and end ')' of a group. Based on the number of parentheses used in a regular expression, the number of groups are created. If your regular expression contains a single pair of parentheses (one capturing group), you only get one group in your match. If there are two pairs of parentheses, then there will be two groups in your match, and so on. If you use a repetition operator on a capturing group (+ or *), the group gets "overwritten" each time the group is repeated, meaning that only the last match is captured. The contents of a group can be retrieved after a match has been performed. Groups are numbered starting from 0, for example, `group(0)` ... up to `group(99)`. To match the literals '(' or ')', use `\(` or `\)`, or enclose them inside a character class: `[([,])`.

Parenthesis not only group substrings but they create backreferences as well.

A backreference in a regular expression identifies a previously matched and remembered group and allows you to specify its contents i.e., backreference matches a substring already found in a group. You simply add a backslash character and the number of the group to match again. For example, to find the content matched by the first group in a regular expression, you would include, `"\1"` in your regular expression pattern. Always represent backreferences as raw strings in regular expressions.

Example → The pattern `Chapter (\d+)\. \d*` illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by a space, followed by one or

more numeric characters (`\d` means any numeric character and `+` means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with `\` means the pattern must look for the literal character `.`), followed by any numeric character 0 or more times (`\d` means numeric character, `*` means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in the string "Open Chapter 4.3, paragraph 6" where '4' is remembered. The pattern is not found in the string "Chapters 3 and 4", because that string does not have a period after the '3'.

To match a substring without causing the matched part to be remembered, within the parentheses preface the pattern with `?:`. For example, `(?:\d+)` matches one or more numeric characters but does not remember the matched characters.

10.2 Regular Expression Methods

Now that we have looked at some simple regular expressions, how do we actually use them in Python? In Python, methods to use and apply regular expressions can be accessed by importing the *re* module. The *re* module provides an interface to the Python regular expression engine.

10.2.1 Compiling Regular Expressions Using *compile()* Method of *re* Module

Regular expressions can be compiled into a pattern object, which has methods for various operations such as searching for pattern matches, finding all pattern matches or performing string substitutions. When you have to use the same regular expression again and again on different strings, then it is an excellent idea to construct a regular expression as a Python object. This can be accomplished through the use of the *re.compile()* method.

re.compile(pattern[,flags])

where *pattern* is the regular expression and the optional *flags* argument is used to enable various special features and syntax variations. For example, specifying the flags *re.A* enables ASCII-only matching, *re.I* enables case-insensitive matching; expressions like `[A-Z]` will also match lowercase letters and *re.M* enables "multi-line matching." When *re.M* flag is enabled, the meaning of `^` and `$` changes. The special character `^` matches at the beginning of the string and also at the beginning of each line (immediately following each newline); and the special character `$` matches at the end of the string and also at the end of each line (immediately preceding each newline). By default, the special character `^` matches only at the beginning of the string, and the special character `$` matches only at the end of the string and immediately before the newline (if any) at the end of the string.

The *compile()* method returns a regular expression as a Python object, which can be used for matching patterns by using its *match()*, *search()*, *sub()*, *findall()* and other methods (TABLE 10.1).

TABLE 10.1

Methods Supported by Compiled Regular Expression Objects

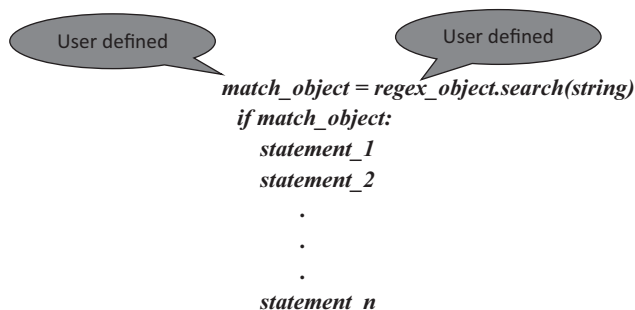
Methods	Syntax	Description
<code>search()</code>	<code>regex_object. search(string[, pos[, endpos]])</code>	This method scans through string looking for the first location where this regular expression produces a match and returns a corresponding match object. Return <i>None</i> if no position in the string matches the pattern.
<code>match()</code>	<code>regex_object. match(string[, pos[, endpos]])</code>	This method returns <i>None</i> if the string does not match the pattern and returns a match object if the method finds a match. This method matches characters at the beginning of the string in accordance with the regular expression pattern. Note that even in MULTILINE mode, the <i>match()</i> method will only match at the beginning of the string and not at the beginning of each line.
<code>findall()</code>	<code>regex_object. findall(string[, pos[, endpos]])</code>	This method returns all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. If the pattern includes two or more parenthesis groups, then instead of returning a list of strings, <i>findall()</i> returns a list of tuples. Each tuple represents one match of the pattern, and inside the tuple is the group(1), group(2)... substrings. Empty matches are included in the result.
<code>sub()</code>	<code>regex_object. sub(pattern, repl, string, count=0, flags=0)</code>	This method returns the string obtained by replacing the leftmost non-overlapping occurrences of the <i>pattern</i> in string by the replacement <i>repl</i> . If the pattern is not found, the string is returned unchanged. Any backslash escapes in <i>repl</i> are processed. That is, <code>\n</code> is converted to a single newline character, <code>\r</code> is converted to a carriage return, and so forth. Unknown escapes such as <code>\&</code> are left alone. Backreferences, such as <code>\2</code> , are replaced with the substring matched by group 2 in the pattern.

Note: The optional parameter *pos* gives an index in the string where the search is to start; it defaults to 0. The optional parameter *endpos* limits how far the string will be searched.

The main difference between *search()* and *match()* methods is *search()* method searches anywhere in the entire string and returns a match object while the *match()* method matches zero or more characters at the beginning of the string and returns a match object.

10.2.2 Match Objects

The *match()* and *search()* methods supported by a compiled regular expression object, returns *None* if no match is found. If they are successful, a match object instance is returned, containing information about the match like the substring it has matched, where the match starts and ends and much more. Since *match()* and *search()* return *None* when there is no match, you can test whether there was a match with a simple *if* statement as shown below.



```

match_object = regex_object.search(string)
if match_object:
    statement_1
    statement_2
    .
    .
    .
    statement_n

```


If Match object is *True*, then execute the statements.

Match object supports several methods and only the most significant ones are covered in [TABLE 10.2](#).

TABLE 10.2

Methods Supported by Match Object

Methods	Syntax	Description
group()	match_object. group([group1,...])	This method returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero and whole match is returned. If a groupN argument is zero, the corresponding return value is the entire matching string. If it is in the inclusive range of [1...99], then it is the string matching the corresponding parenthesized group. If a group number is negative or the larger than the number of groups defined in the pattern, an IndexError exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is <i>None</i> . If a group is contained in a part of the pattern that matched multiple times, the last match is returned.
groups()	match_object. groups(default=None)	This method returns a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to <i>None</i> .
start()	match_object. start([group])	The start() method returns the index of the start and end() method returns the index of the end of the substring matched by <i>group</i> . The default value of the <i>group</i> is zero which means the whole matched substring is returned else a value of -1 is returned if a group exists but did not contribute to the match.
end()	match_object. end([group])	
span()	match_object. span([group])	This method returns a tuple containing the (m.start(group), m.end(group)) positions of the match.

In order to build and use regular expressions, perform the following steps:

Step 1: Import *re* regular expression module.

Step 2: Compile regular expression pattern using *re.compile()* method. This method returns the regular expression pattern as an object.

Step 3: Invoke an appropriate method supported by the compiled regular expression object which returns a matched object instance containing information about matched strings.

Step 4: Call methods (*group()* method is appropriate for most cases) associated with the matched object to display the results.

For example,

1. >>> import re
2. >>> pattern = re.compile(r'(e)g')
3. >>> pattern
re.compile('(e)g')
4. >>> match_object = pattern.match('egg is nutritional food')
5. >>> match_object


```

    <_sre.SRE_Match object; span=(0, 2), match='eg'>
6. >>> match_object.group()
    'eg'
7. >>> match_object.group(0)
    'eg'
8. >>> match_object = pattern.match('brownegg is nutritional food')
9. >>> match_object.group()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'

```

Import *re* module ①. Compile the regular expression pattern *'(e)g'* which matches the characters *eg* found at the beginning of a string ②–③. Pass the string from which you want to extract the regular expression pattern as an argument to *match()* method ④. As you can see in the result of *match_object* ⑤, the matched string is assigned to *match*. To obtain the strings that were matched, use the *group()* method associated with *match_object* ⑥. Groups are always numbered starting with 0. Group 0 is always present and it represents the entire result of the regular expression itself, so *group()* method of match object all have 0 as their default argument ⑦. In ⑧, even though the string has the pattern *eg* in it, the characters *eg* are not found at the beginning of the string. Thus, if you try to use the *group()* method with match object then it results in an error ⑨. The *'r'*, at the start of the pattern string designates a Python “raw” string. It is highly recommended that you make it a habit of writing pattern strings with an *'r'* prefix.

```

1. >>> import re
2. >>> pattern = re.compile(r'(ab)*)
3. >>> match_object = pattern.match('ababababab')
4. >>> match_object.span()
    (0, 10)
5. >>> match_object.start()
    0
6. >>> match_object.end()
    10

```

In the above example, regular expression pattern *(ab)** will match zero or more repetitions of *ab* ②. Pass the string from which you want to extract the regular expression pattern as an argument to the *match()* method ③. Groups indicated with *(', ')* also capture the starting and ending index of the matched substring, and this can be retrieved using *span()* method ④. Also, the starting position of the match can be obtained by the *start()* method ⑤ and ending position of the match is obtained by the *end()* method ⑥. Since the *match()* method only checks if the regular expression matches at the start of a string, *start()* method will always return zero.

```
1. >>> import re
2. >>> pattern = re.compile(r'(a(b)c)d')
3. >>> method_object = pattern.match('abcd')
4. >>> method_object.group(0)
   'abcd'
5. >>> method_object.group(1)
   'abc'
6. >>> method_object.group(2)
   'b'
7. >>> method_object.group(2,1,2)
   ('b', 'abc', 'b')
8. >>> method_object.groups()
   ('abc', 'b')
```

In the above example, the regular expression pattern `'(a(b)c)d'` will match the string `'abcd'` ②. Pass the string from which you want to extract the regular expression pattern as an argument to the `match()` method ③. By passing an integer number argument greater than zero to the `group()` method, you can also extract part of the matched expression instead of entire expression. The `group()` method with integer 0 as argument returns the entire matched text while the `group()` method with greater than zero as argument returns only a part of the matched text. Each of these number arguments corresponds to specific groups. Groups are numbered from left to right, starting from number 1. For example, `group(0)` returns the entire matched string `'abcd'` ④, while `group(1)` returns `'abc'` ⑤ and `group(2)` returns `'b'` ⑥. To determine the integer number, count the number of parentheses pairs from left to right. Also, `group()` method can be passed to multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups ⑦. The `groups()` method returns a tuple `('abc', 'b')` containing all the subgroups of the match ⑧.

```
1. >>> import re
2. >>> pattern = re.compile(r'\d+')
3. >>> match_list = pattern.findall("Everybody think they're famous when they get
   100000 followers on Instagram and 5000 on Twitter")
4. >>> match_list
   ['100000', '5000']
```

In the above example, the regular expression pattern `'\d+'` will match one or more digits of a number ②. Pass the string from which you want to extract the regular expression pattern as an argument to `findall()` method ③. The `findall()` returns a list numbers `['100000', '5000']` as strings with each string representing one match ④.

```
1. >>> import re
2. >>> pattern = re.compile(r'([\w\.\.]+)@([\w\.\.]+)')
```

```

3. >>> matched_email_tuples = pattern.findall('bill_gates@microsoft.com and steve.
   jobs@apple.com are visionaries')
4. >>> print(matched_email_tuples)
   [('bill_gates', 'microsoft.com'), ('steve.jobs', 'apple.com')]
5. >>> for each_mail in matched_email_tuples:
6. ...     print(f"User name is {each_mail[0]}")
7. ...     print(f"Domain name is {each_mail[1]}")
   User name is bill_gates
   Domain name is microsoft.com
   User name is steve.jobs
   Domain name is apple.com

```

In the above example, the regular expression pattern `([w\.\.]+)@([w\.\.]+)` matches user name and the domain name of an email ID which is to the left and right of the `@` symbol. This regular expression pattern has two pairs of parenthesis representing two groups belonging to user name and domain name substrings. The dot (`.`) character is also matched in the user name and domain name substrings ①–③. The `findall()` method returns all the occurrences of the matching pattern as a list of tuples with each tuple having user name and domain name as its string items matching their corresponding parenthesis groups ④. Iterate through each of these tuple items in the list using `for` loop and display user name and domain name ⑤–⑦.

Including parentheses in a regular expression pattern causes the corresponding matched group to be remembered. For example, `/a(b)c/` matches the characters `'abc'` and remembers `'b'`. To recall this matched substring group, use backreference like `\1`.

```

1. >>> import re
2. >>> pattern = re.compile(r'(\w+)\s(\w+)')
3. >>> replaced_string = pattern.sub(r'\2 \1', 'Ken Thompson')
4. >>> replaced_string
   'Thompson Ken'

```

In the above example, regular expression `'(\w+)\s(\w+)'` will match a substring followed by a space and another substring ②. There are two pairs of parenthesis in the above code with each parenthesis matching a substring. The `sub()` method is used to switch the words in the string. For the replacement text, use `r'\2 \1'` where `\1` in the replacement is replaced by a matched substring of the first group and `\2` is replaced by second matched substring of the second group ③–④.

```

1. >>> import re
2. >>> pattern = re.compile(r'.')
3. >>> replaced_string = pattern.sub('$', 'this, is, a, test')
4. >>> replaced_string
   'this$is$a$test'

```

In the above code, comma ',' is replaced with dollar '\$' sign ①–④.

```

1. >>> import re
2. >>> pattern = re.compile(r'tree:\w\w\w')
3. >>> match_object = pattern.search("Example for tree:oak")
4. >>> if match_object:
5. ....     print(f'Matched string is {match_object.group()}')
6. ....     else:
7. ...       print("Match not found")
           Matched string is tree:oak

```

In the above code, the *search()* method searches for the pattern 'tree:' followed by a 3-letter word. The code *pattern.search("Example for tree:oak")* returns the search result as an object and is assigned to *match_object* variable ③. Then use *if* statement to test the *match_object*④. If it evaluates to Boolean *True*, then the search has succeeded and the matched string is displayed using *match_object.group()*. Otherwise, if the match is Boolean *False* (*None* to be more specific), then the search did not succeed, and there is no matching string ⑤–⑦.

Program 10.1: Given an Input File Which Contains a List of Names and Phone Numbers Separated by Spaces in the Following Format:

Alex 80-23425525

Emily 322-56775342

Grace 20-24564555

Anna 194-49611659

Phone Number Contains a 3- or 2-Digit Area Code and a Hyphen Followed By an 8-Digit Number.

Find All Names Having Phone Numbers with a 3-Digit Area Code Using Regular Expressions.

```

1. import re
2. def main():
3.     pattern = re.compile(r'(\w+)\s+\d{3}-\d{8}')
4.     with open("person_details.txt", "r") as file_handler:
5.         print("Names having phone numbers with 3 digit area code")
6.         for each_line in file_handler:
7.             match_object = pattern.search(each_line)
8.             if match_object:
9.                 print(match_object.group(1))
10. if __name__ == "__main__":
11.     main()

```

OUTPUT

Names having phone numbers with 3 digit area code

Emily

Anna

Here the regular expression pattern matches a string, followed by one or more spaces, followed by 3 digits, followed by a hyphen and followed by 8 digits ③. The matched substring within the parentheses can be referenced by passing an integer number of one to *group()* method. The pattern matches three digits to the left and eight digits to the right of the hyphen. The file *person_details.txt* is opened in "r" mode using a *with* statement and the returning object is assigned to *file_handler* ④. Traverse through *each_line* in the *file_handler* using a *for* loop ⑥. In each line search for the pattern ⑦. If the match succeeds, then recall the first group that has the individual names ⑧–⑨.

Program 10.2: Write a Python Program to Check the Validity of a Password Given by User.

The Password Should Satisfy the Following Criteria:

1. Contain at least 1 letter between a and z
2. Contain at least 1 number between 0 and 9
3. Contain at least 1 letter between A and Z
4. Contain at least 1 character from \$, #, @
5. Minimum length of password: 6
6. Maximum length of password: 12

```

1. import re
2. def main():
3.     lower_case_pattern = re.compile(r'[a-z]')
4.     upper_case_pattern = re.compile(r'[A-Z]')
5.     number_pattern = re.compile(r'\d')
6.     special_character_pattern = re.compile(r'[$#@]')
7.     password = input("Enter a Password ")
8.     if len(password) < 6 or len(password) > 12:
9.         print("Invalid Password. Length Not Matching")
10.    elif not lower_case_pattern.search(password):
11.        print("Invalid Password. No Lower-Case Letters")
12.    elif not upper_case_pattern.search(password):
13.        print("Invalid Password. No Upper-Case Letters")
14.    elif not number_pattern.search(password):
15.        print("Invalid Password. No Numbers")
16.    elif not special_character_pattern.search(password):

```

```

17.     print("Invalid Password. No Special Characters")
18.     else:
19.         print("Valid Password")
20. if __name__ == "__main__":
21.     main()

```

OUTPUT

```

Enter a Password DrAit1980@1
Valid Password

```

```

Enter a Password NoSmoking
Invalid Password. No Numbers

```

Pattern `r'[a-z]'` checks for at least one lowercase letter between a and z ③. Pattern `r'[A-Z]'` checks for at least one uppercase letter between A and Z ④. Pattern `r'\d'` checks for at least one number between 0 and 9 ⑤. Pattern `r'[$#@]'` checks for at least one character from \$, #, @ ⑥. Password length for minimum and maximum characters is checked in ⑧–⑨. If all the conditions are satisfied, then a "Valid Password" message is printed ⑩.

Program 10.3 Write Python Program to Validate U.S.-based Social Security Number

```

1. import re
2. def main():
3.     pattern = re.compile(r"\b\d{3}-?\d{2}-?\d{4}\b")
4.     match_object = pattern.search("Social Security Number for James is 916-30-2017")
5.     if match_object:
6.         print(f"Extracted Social Security Number is {match_object.group()}")
7.     else:
8.         print("No Match")
9. if __name__ == "__main__":
10.    main()

```

OUTPUT

```

Extracted Social Security Number is 916-30-2017

```

A US-based Social Security number is a combination of nine numbers, in a sequence of three numbers, two numbers, and four numbers, with or without a hyphen in between. The question mark special character (?) matches zero or exactly one preceding character and in this case it is the hyphen (-). The numbers in a Social Security number can be matched with the digit special character (\d). To look for a set number of digits, you can use the curly brackets surrounding the number of expected digits ③. Pass the text to `search()` method from which the substring matching the pattern is extracted ④. If successful display the social security number else display the message "No Match" ⑤–⑧.

10.3 Named Groups in Python Regular Expressions

Regular expressions use groups to capture strings of interest. As the regular expression becomes complex, it gets difficult to keep track of the number of groups in the regular expression. In order to overcome this problem Python provides named groups. Instead of referring to the groups by numbers, you can reference them by a name.

The syntax for a named group is,

(?P<name>RE)

where the first name character is *?*, followed by letter *P* (uppercase letter) that stands for Python Specific extension, *name* is the name of the group written within angle brackets, and RE is the regular expression. Named groups behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group's name.

```
1. >>> import re
2. >>> pattern = re.compile(r'(?P<word>\b\w+\b)')
3. >>> match_object = pattern.search('laugh out loud')
4. >>> match_object.group('word')
    'laugh'
5. >>> match_object.group(1)
    'laugh'
```

In the above code, the regular expression has a group that matches the pattern of a word boundary followed by one or more alphanumeric characters, that is, a-z, A-Z, 0-9 and `_`, followed by a word boundary. The name given to this group is `<word>` specified within angle brackets ①. Pass the string from which you want to extract the pattern as an argument to the `search()` method ②. By passing the group name 'word' as an argument to the `group()` method, you can extract the matched substring ③. This named group can still be used to retrieve information by the passed integer numbers instead of group name ④.

10.4 Regular Expression with *glob* Module

The `glob` module finds all the file names matching a specified pattern. Starting with Python version 3.5, the `glob` module supports the `"**"` directive (which is parsed only if you pass recursive flag). In earlier Python versions, `glob.glob()` did not list files in subdirectories recursively.

The syntax for `glob` method is,

`glob.glob(pathname, **, recursive=True)`

The `glob()` method of the `glob` module returns a possible list of file names that match a `pathname`, which must be a string containing a path specification. Here `pathname` can be

either absolute (like C:\Anaconda\Python\Python3.6.exe) or relative (like..\Tools**.gif). If recursive is True, the pattern "*" will match any files and zero or more directories and subdirectories.

Program 10.4 Write Python Program to Change the File Extension from .txt to .csv of All the Files (Including from Sub Directories) for a Given Path

```

1. import os
2. import glob
3. def rename_files_recursively(directory_path):
4.     print("File extension changed from .txt to .csv")
5.     for file_path in glob.glob(directory_path + '\**\*.txt', recursive=True):
6.         print(f"File with .txt extension {file_path} changed to", end="")
7.         try:
8.             pre, ext = os.path.splitext(file_path)
9.             print(f" File with .csv extension {pre + '.csv'}")
10.            os.rename(file_path, pre + '.csv')
11.        except Exception as e:
12.            print(e)
13. def main():
14.     directory_path = input("Enter the directory path from which you want to convert
        the files recursively ")
15.     rename_files_recursively(directory_path)
16. if __name__ == "__main__":
17.     main()

```

OUTPUT

```

Enter the directory path from which you want to convert the files recursively C:\Animals
File extension changed from .txt to .csv
File with .txt extension C:\Animals\Mammal\whale .txt changed to File with .csv exten-
sion C:\Animals\Mammal\whale.csv
File with .txt extension C:\Animals\Mammal\Primates\apes .txt changed to File with .csv
extension C:\Animals\Mammal\Primates\apes.csv
File with .txt extension C:\Animals\Reptile\snake.txt changed to File with .csv extension
C:\Animals\Reptile\snake.csv

```

Consider the files whale.txt in C:\Animals\Mammal directory, apes.txt in C:\Animals\Mammal\Primates directory and snake.txt in C:\Animals\Reptile directory. User entered root directory ③ is passed as an argument to *glob.glob()* method along with "*" and *recursive = True* arguments to navigate through all the subdirectories ⑤. While navigating through the subdirectories, recognize the files having a .txt extension and change it to a .csv extension ⑥–⑩.

10.5 Summary

- The term “regular expressions” is also called regexes or regex patterns.
- The module `re` has to be imported to be able to work with regular expressions.
- Use the `compile()` method in `re` module to compile regular expression to match objects.
- Use various methods like `search()`, `match()`, `findall()`, and `sub()` methods to extract substrings matching a pattern.

Multiple Choice Questions

1. The module that supports regular expressions is
 - a. `re`
 - b. `regex`
 - c. `pyregex`
 - d. `strings`
2. The function that creates the pattern object is
 - a. `re.create(str)`
 - b. `re.regex(str)`
 - c. `re.compile(str)`
 - d. `re.assemble(str)`
3. The metacharacter period(`.`) matches any character other than ____.
 - a. `&`
 - b. `^`
 - c. `\b`
 - d. `\n`
4. The functionality of the `regex_pattern.match()`
 - a. matches a pattern at the end of the string
 - b. matches a pattern at the start of the string
 - c. matches a pattern at any position of the string
 - d. None of these
5. The functionality of the `regex_pattern.search()`
 - a. matches a pattern at the end of the string
 - b. matches a pattern at the start of the string
 - c. matches a pattern at any position of the string
 - d. None of these

6. The expression `wood{5,8}` will match how many characters with the regular expression?
 - a. Matches the pattern from five to eight times
 - b. Matches the pattern from four to seven times
 - c. Matches the pattern from zero to five times
 - d. None of these
7. Which special character matches one or more specific characters?
 - a. `*`
 - b. `+`
 - c. `?`
 - d. None of these
8. Which regular expression will match the string `April-4-18`?
 - a. `[a-z]+W[0-9]+W[0-9]+`
 - b. `([a-zA-Z]+)\W([0-9]+)\W([0-9]+)`
 - c. `JUL-w-w`
 - d. `(d+|[a-zA-Z]+)[/](d+)[/](d+)`
9. Consider the following code.

```
pattern = re.compile(r'crying')
replaced_string = pattern.sub('smiling', 'you are crying')
```

The output for above code is
 - a. Crying
 - b. Smiling
 - c. You are crying
 - d. You are smiling
10. In the `match_object.start(group)` and `match_object.end(group)` methods, if the argument *group* is not specified then it defaults to
 - a. 1
 - b. 0
 - c. 2
 - d. 3
11. The metacharacter `\s` matches __ characters
 - a. Word boundary
 - b. Decimal digit
 - c. White space
 - d. Alphabets
12. The __ meta character matches zero or more repetitions of the string
 - a. `+`
 - b. `?`
 - c. `.`
 - d. `*`

13. The characters `__` and `__` matches the start and end of the string, respectively.
- `^` and `.`
 - `*` and `&`
 - `^` and `$`
 - `$` and `$`
14. Consider the statement `pattern = re.compile('inspiring years')`. Guess the output of the following code `pattern.findall('inspiring')`.
- `[years]`
 - `[]`
 - `years`
 - `[inspiring years]`
15. For the statement `pattern = re.compile(r'12*')`, which of the below lines of code does not show a match?
- `pattern.match('1')`
 - `pattern.match('12')`
 - `pattern.match('122')`
 - `pattern.match('21')`
16. The code below validates IP address:
- ```
pattern = re.compile(r'\b(\d{1,3})\b.\b(\d{1,3})\b.\b(\d{1,3})\b.\b(\d{1,3})\b')
```
- Which of the following code matches the pattern?
- `pattern.search("123.111.123.145")`
  - `pattern.search("193.123.2013.45")`
  - `pattern.search("231.56.123")`
  - `pattern.search("123.46.13.3454")`
17. Below code pattern validates a user name:
- ```
pattern = re.compile(r'^[a-z0-9_-]{6,14}$')
```
- Which of the following code matches the pattern?
- `pattern.search("Python.3.superb")`
 - `pattern.search("Python.3_superb")`
 - `pattern.search("python3superb")`
 - `pattern.search("Python_3-superb")`
18. To remove extra spaces from the string `"Lion is King of Jungle"`, the code used is
- `pattern = re.compile(r'\s')`
`pattern.sub(" ", "Lion is King of Jungle")`
 - `pattern = re.compile(r'\s+')`
`pattern.sub(" ", "Lion is King of Jungle")`
 - `pattern = re.compile(r'\S+')`
`pattern.sub(" ", "Lion is King of Jungle")`
 - `pattern = re.compile(r'\S')`
`pattern.sub(" ", "Lion is King of Jungle")`

19. Consider a file 21-12-2016.zip. The regular expression pattern to extract date from filename is
 - a. `([0-9]{1}\-[0-9]{2}\-[0-9]{4})`
 - b. `([0-9]{2}\-[0-9]{2}\-[0-9]{4})`
 - c. `([0-9]{2}\-[0-9]{2}\-[0-9]{2})`
 - d. `([0-9]{2}\-[0-9]{1}\-[0-9]{4})`
 20. Consider the string "October 31". The pattern to extract only the month in the string is
 - a. `([a-zA-Z])`
 - b. `[a-zA-Z]+ \d+`
 - c. `([a-z]+) \d+`
 - d. `([a-zA-Z]+) \d+`
 21. The Indian Aadhar number is a 12-digit unique identification number that is assigned to an individual. The first digit should not be either 0 or 1 while the remaining digits can be between 0 to 9 with no space or hyphen between any of the digits. Pattern matching this criterion is
 - a. `[1-9]{1}[0-9]{11}`
 - b. `[0-9]{1}[0-9]{11}`
 - c. `[2-9]{2}[0-9]{11}`
 - d. `[2-9]{1}[0-9]{11}`
 22. When `findall()` method is used to apply the pattern `r'\d{2, 4}'` for the string `'01, Jan 2015'`, it results in
 - a. `['01', '2015']`
 - b. `['2015']`
 - c. `['01']`
 - d. `['012015']`
-

Review Questions

1. Briefly explain the importance of the raw string notation.
2. Define regular expression and list out all the advantages of the regular expression.
3. Describe any ten metacharacters with examples.
4. Compare and contrast the use of `match()` and `search()` methods with an example.
5. Briefly explain the greedy and non-greedy matching.
6. Describe all the functions available in match objects.
7. Write a regular expression which matches strings which starts with a sequence of digits—at least one digit—followed by a blank and after these arbitrary characters.
8. Write a Python program to find sequences of lower case letters joined with an underscore.

9. Write a Python program that matches a word containing 'z'.
10. Write a Python program to remove all leading zeros' from an IP address
11. Write a Python program to search the numbers (0–9) of length between 1 to 3 in a given string.
12. Write a Python program to find the substrings within a string
13. Write a Python program to extract year, month and date from an url.
14. Write a Python program to read a file and to convert a date of yyyy-mm-dd format to dd-mm-yyyy format.
15. Write a Python program to abbreviate 'Street' as 'St.' in a given string.
16. Write a Python program to find all five characters long word in a string.

11

Object-Oriented Programming

AIM

Understand Object-oriented programming paradigm in controlling the access of data and reducing the duplication of code by employing code reusability techniques.

LEARNING OUTCOMES

At the end of the chapter, you are expected to

- Understand and Create Objects.
- Recognize data attributes and methods for given objects.
- Use the dot notation to access data attributes and methods of an object.
- Demonstrate the implementation of instance variables, methods, and constructors.
- Understand Encapsulation, Polymorphism and Inheritance.

The basic idea of Object-oriented programming (OOP) is that we use objects to model real-world things that we want to represent inside our programs and provide a simple way to access their functionality that would otherwise be hard or impossible to utilize. Large programs are challenging to write. Once a program reaches a certain size, Object-oriented programs are actually easier to program than non-Object-oriented ones. As the programs are not disposable, an Object-oriented program is much easier to modify and maintain than a non-Object-oriented program. So, Object-oriented programs require less work to maintain over time. Object-oriented programming results in code reuse, cleaner code, better architecture, abstraction layers, and fewer programming bugs. Python provides full support for Object-oriented programming, including encapsulation, inheritance, and polymorphism.

11.1 Classes and Objects

In a 1994 *Rolling Stone* interview, Steve Jobs (CEO of Apple) explains Object-oriented programming. His explanation still helps us understand what OOP is in simple terms.

Jeff Goodell: Would you explain, in simple terms, exactly what object-oriented software is?

Steve Jobs: Objects are like people. They are living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here. Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So, I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes." You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you cannot even hail a taxi. You cannot pay for one, you do not have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level.

[Source: <https://www.rollingstone.com/culture/news/steve-jobs-in-1994-the-rolling-stone-interview-20110117>]

Let's look at another example. In the real world, you'll often find many individual objects of all the same kind. There may be thousands of cars in existence, all of the same make and model. Each of these cars were built from the same set of blueprints and, therefore, contains the same components. In object-oriented terms, we say that each of these cars are objects of the class known as Car.

A class is a blueprint from which individual objects are created. An object is a bundle of related state (variables) and behavior (methods). Objects contain *variables*, which represents the *state* of information about the thing you are trying to model, and the *methods* represent the *behavior* or *functionality* that you want it to have (FIGURE 11.1).

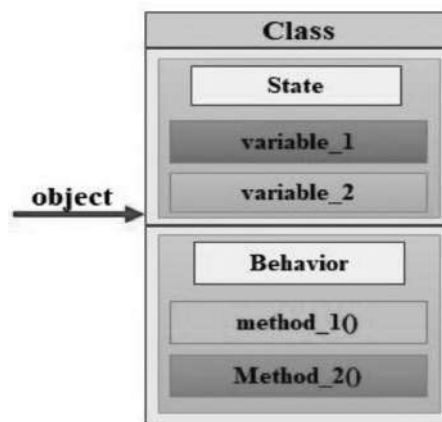


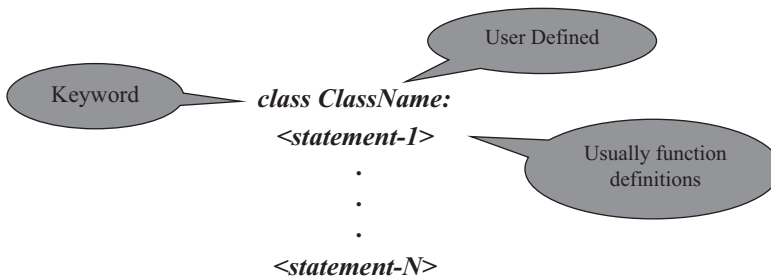
FIGURE 11.1
Generic Object Diagram.

NOTE: *Variables* refers to both Instance variables and Class variables, unless explicitly specified.

Class objects are often used to model the real-world objects that you find in everyday life. Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle. Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

11.2 Creating Classes in Python

Related variables and methods are grouped together in classes. The simplest form of class definition looks like this:



Classes are defined by using the `class` keyword, followed by the `ClassName` and a colon. Class definitions must be executed before they have any effect. In practice, the statements inside a class definition will usually be function definitions, but few other statements are allowed. (We'll discuss this later). Because these functions are indented under a class, they are called methods. *Methods* are a special kind of function that is defined within a class.

Program 11.1: Program to Illustrate Class and Object Creation

1. `class Mobile:`
2. `def __init__(self):`
3. `print("This message is from Constructor Method")`
4. `def receive_message(self):`
5. `print("Receive message using Mobile")`
6. `def send_message(self):`


```
7.     print("Send message using Mobile")
8. def main():
9.     nokia = Mobile()
10.    nokia.receive_message()
11.    nokia.send_message()
12. if __name__ == "__main__":
13.     main()
```

OUTPUT

This message is from Constructor Method
Receive message using Mobile
Send message using Mobile

Let's define a class called *Mobile* ① that has two methods associated with it (FIGURE 11.2), one is *receive_message()* ④ and another is *send_message()* ⑥. The first parameter in each of these methods is the word *self*. When *self* is used, it is just a variable name to which the object that was created based on a class is assigned. In the method definition, *self* doesn't need to be the only parameter and it can have multiple parameters. Creating the *Mobile* class provided us with a blueprint for an object. Just because you have defined a class doesn't mean you have created any *Mobile* objects.

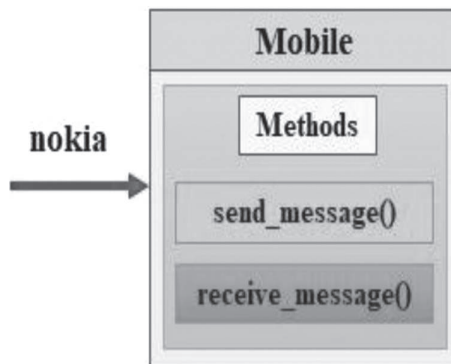


FIGURE 11.2
Object diagram for Mobile Class with Methods only.



Often, the first argument of a method is called *self*. This is nothing more than a convention: the name *self* has absolutely no special meaning to Python. However, by not following this convention, your code may be less readable to other Python programmers.

11.3 Creating Objects in Python

Object refers to a particular instance of a class where the object contains variables and methods defined in the class. Class objects support two kinds of operations: attribute references and instantiation. The term attribute refers to any name (variables or methods) following a dot. This is a syntactic construct. The act of creating an object from a class is called *instantiation*.

The names in a class are referenced by objects and are called *attribute references*. There are two kinds of attribute references, data attributes and method attributes. Variables defined within the methods are called *instance variables* and are used to store data values. New instance variables are associated with each of the objects that are created for a class. These instance variables are also called *data attributes*. Method attributes are methods inside a class and are referenced by objects of a class. Attribute references use the standard dot notation syntax as supported in Python.

The syntax to access data attribute is,

object_name.data_attribute_name

The syntax to assign value to data attribute is,

object_name.data_attribute_name = value

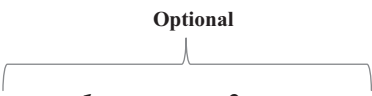
where *value* can be of integer, float, string types, or another object itself.

The syntax to call method attribute is,

object_name.method_attribute_name()

Valid attribute names are all the names that were inside the class when the objects for the class was created. The connection between the attributes with the object is indicated by a "dot" (".") written between them with *object_name* on left and *attribute_name* on right. For example, in the expression *z.real = 10*, *real* is an data attribute of the object *z* and is assigned a value of 10. In the expression *cow.domesticated()*, the *domesticated()* is a method attribute of the *cow* object.

The syntax for Class instantiation is,



object_name = ClassName(argument_1, argument_2,, argument_n)

Class instantiation uses function notation, wherein the class name is followed by parentheses () as if it were a function, *nokia = Mobile()*. The above expression creates a new object for the class *ClassName* and assigns this object to the variable *object_name*. You can specify any number of arguments during instantiation of the class object. An object *nokia* ⑨ for the class *Mobile* is created. The *nokia* object calls the methods *receive_message()* ⑩ and *send_message()* ⑪ using the dot operator. Calling *nokia.receive_message()* and *nokia.send_message()* means that these methods are to be used with a *nokia* instance of the class *Mobile*.

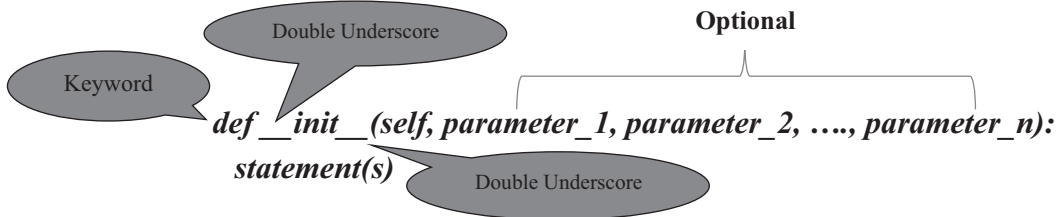
You may have noticed that both of these method definitions have *self* as the first parameter. This *self* variable can also be used inside the method bodies, but you do not appear to pass this as an argument in the method called using the object. This is because *whenever you call a method using an object, the object itself is automatically passed in as the first parameter to the self parameter variable*. The remaining parameter variables must be supplied as arguments in the calling method. The object *nokia* calls two methods in the *main()* function of the program, causing those methods to run. Python raises an exception when a method that requires an argument is called without any, even if the argument is not actually used.



When you create an object for a class, it is called instance of a class. The terms *object* and *instance of a class* are the same thing and are often used interchangeably.

11.4 The Constructor Method

Python uses a special method called a constructor method. Python allows you to define only one constructor per class. Also known as the `__init__()` method, it will be the first method definition of a class and its syntax is,



The `__init__()` method defines and initializes the instance variables. It is invoked as soon as an object of a class is instantiated ⑨. The `__init__()` method for a newly created object is automatically executed with all of its parameters ②–③. The `__init__()` method is indeed a special method as other methods do not receive this treatment. The parameters for `__init__()` method are initialized with the arguments that you had passed during instantiation of the class object. Class methods that begin with a double underscore (`__`) are called special methods as they have special meaning. The number of arguments during the instantiation of the class object should be equivalent to the number of parameters in `__init__()` method (excluding the *self* parameter).



After an object has been created, you should not use the object name itself to call the `__init__()` constructor method directly. For example, the expression `object_name.__init__()` is not recommended. The `__init__()` method never returns a value.

Program 11.2: Program to Illustrate Multiple Parameters in `__init__()` Method

```
1. class Mobile:
2.     def __init__(self, name):
3.         self.mobile_name = name
4.
5.     def receive_message(self):
6.         print(f"Receive message using {self.mobile_name} Mobile")
7.
8.     def send_message(self):
9.         print(f"Send message using {self.mobile_name} Mobile")
10.
11. def main():
12.     nokia = Mobile("Nokia")
13.     nokia.receive_message()
14.     nokia.send_message()
15.
16. if __name__ == "__main__":
17.     main()
```

OUTPUT

```
Receive message using Nokia Mobile
Send message using Nokia Mobile
```

When we call the class object, a new instance of the class is created, and the `__init__()` method for this new object is immediately executed with all the parameters that we passed to the class object ②. As the `__init__()` method is automatically initialized, no need to explicitly call it, instead pass the arguments in the parentheses following the class name when you create a new instance of the class ⑨. Since `self` is the instance of a class, `self.mobile_name = name` is equivalent to saying `nokia.mobile_name = name` ③. Methods have access to all the data attributes contained during the instance of an object. Inside the methods, you can access and modify the values of instance variables that you had previously set on `self` ④–⑦. Because they use `self`, they require an instance of the class in order to be used. For this reason, they are also referred to as “instance methods” ⑩–⑪ (FIGURE 11.3).

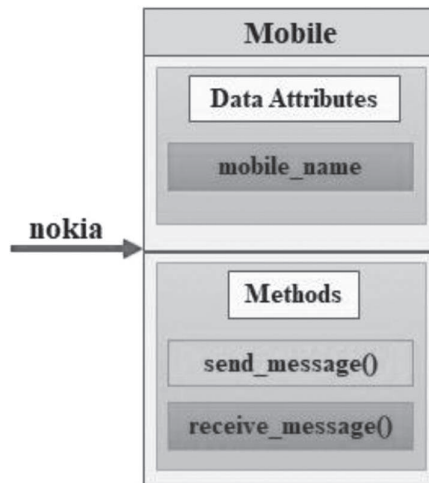


FIGURE 11.3
Object diagram for Mobile class with data attributes and methods.



It's a good programming practice not to introduce new data attributes outside of the `__init__()` method.

Program 11.3: Write Python Program to Calculate the Arc Length of an Angle by Assigning Values to the Radius and Angle Data Attributes of the class ArcLength

```

1. import math
2. class ArcLength:
3.     def __init__(self):
4.         self.radius = 0
5.         self.angle = 0
6.     def calculate_arc_length(self):
7.         result = 2 * math.pi * self.radius * self.angle / 360
8.         print(f"Length of an Arc is {result}")
9. al = ArcLength()
10. al.radius = 9
11. al.angle = 75
12. print(f"Angle is {al.angle}")
13. print(f"Radius is {al.radius}")
14. al.calculate_arc_length()
  
```

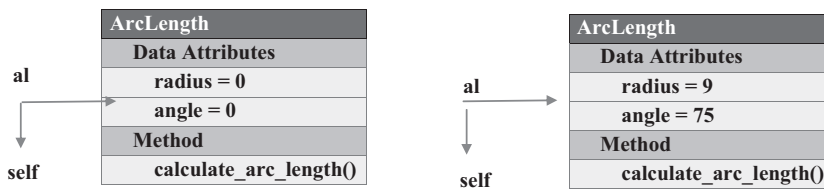


FIGURE 11.4
Object diagram for `ArcLength` class.

OUTPUT

Angle is 75
Radius is 9
Length of an Arc is 11.780972450961725

In Python, you can directly access the data attributes using objects. The data attributes ④–⑤ `radius` and `angle` are added to the `__init__()` method of the `ArcLength` class (FIGURE 11.4). The arc length of an angle is calculated in `calculate_arc_length()` method ⑥–⑧. These data attributes can be referenced outside the class through the `al` object. When the object `al` is created ⑨, each of the data attributes `radius` and `angle` in the `__init__()` method are initialized to zero and the object `al` is passed to the `self` parameter, which essentially becomes `al`. `radius = 0` and `al.angle = 0`. On execution of the expressions `al.radius = 9` and `al.angle = 75` ⑩–⑪, the values of data attributes are changed to the latest value. When the method `calculate_arc_length()` is referenced through `al` object, the latest values are reflected for the data attributes accessed through `self` ⑦. It is good practice to include import statements before defining class itself.

11.5 Classes with Multiple Objects

Multiple objects for a class can be created while attaching a unique copy of data attributes and methods of the class to each of these objects.

Program 11.4: Program to Illustrate the Creation of Multiple Objects for a Class

1. class Birds:
2. def __init__(self, bird_name):
3. self.bird_name = bird_name
4. def flying_birds(self):
5. print(f"{self.bird_name} flies above clouds")
6. def non_flying_birds(self):
7. print(f"{self.bird_name} is the national bird of Australia")
8. def main():

```

9.  vulture = Birds("Griffon Vulture")
10. crane = Birds("Common Crane")
11. emu = Birds("Emu")
12. vulture.flying_birds()
13. crane.flying_birds()
14. emu.non_flying_birds()

15. if __name__ == "__main__":
16.     main()

```

OUTPUT

```

Griffon Vulture flies above clouds
Common Crane flies above clouds
Emu is the national bird of Australia

```

Here, three objects, *vulture*, *crane*, and *emu*, are created for the *Birds* class ⑨–⑪. All of these objects belong to the same class, so they have the same data attribute but different values for each of those data attributes. Objects can also have their own methods to operate on their data attributes. A method is always invoked relative to some object of its class ④–⑦. During object instantiation, each object receives a unique copy of data attribute and method is bundled together. This ensures that correct data attributes and methods are used that are specific to a particular object. The *self* variable is initialized with the particular object of the class that is created during instantiation ②–③ and the parameters of `__init__()` constructor is initialized with the arguments passed on to that class object ⑨–⑪. Now we have three objects whose data attributes have different values. In this case, *vulture.flying_birds()* will output “Griffon Vulture flies above clouds,” *crane.flyingbirds()* will output “Common Crane flies above cloud,” and *emu.non_flying_birds()* will output “Emu is the national bird of Australia” ⑫–⑭. Notice the use of *bird_name* in ② and ③. Even though they have the same name, they are unique. The *bird_name* in `__init__()` method definition header is used as a parameter while the *bird_name* referenced by *self* within the method is an instance variable.

Program 11.5: Write Python Program to Simulate a Bank Account with Support for depositMoney, withdrawMoney and showBalance Operations

```

1. class BankAccount:
2.     def __init__(self, name):
3.         self.user_name = name
4.         self.balance = 0.0

5.     def show_balance(self):
6.         print(f"{self.user_name} has a balance of {self.balance} dollars")

```

```
7. def withdraw_money(self, amount):
8.     if amount > self.balance:
9.         print("You don't have sufficient funds in your account")
10.    else:
11.        self.balance -= amount
12.    print(f"{self.user_name} has withdrawn an amount of {self.balance} dollars")

13. def deposit_money(self, amount):
14.     self.balance += amount
15.     print(f"{self.user_name} has deposited an amount of {self.balance} dollars")

16. def main():
17.     savings_account = BankAccount("Olivia")
18.     savings_account.deposit_money(1000)
19.     savings_account.show_balance()
20.     savings_account.withdraw_money(500)
21.     savings_account.show_balance()

22. if __name__ == "__main__":
23.     main()
```

OUTPUT

Olivia has deposited an amount of 1000.0 dollars
Olivia has a balance of 1000.0 dollars
Olivia has withdrawn an amount of 500.0 dollars
Olivia has a balance of 500.0 dollars

In the `__init__()` method, two data attributes, `user_name` and `balance`, are added ②–④. Also, `show_balance()`, `withdraw_money()`, and `deposit_money()` methods are added to the class. The data attribute `user_name` is initialized with the value of `name` parameter while `balance` data attribute is initialized to zero. This value of data attribute `balance` is changed in methods. The method `show_balance()` displays user name and the balance user has in his account ⑤–⑥. In the `withdraw_money()` method, the user specified amount is compared with the existing balance. If the withdrawal amount is more than the existing balance, then a message is displayed saying, “You don’t have sufficient funds in your account,” or else the amount is subtracted from the balance ⑦–⑩. In the `deposit_money()` method, the user specified amount is added to the existing balance ⑪–⑬. Using the object `savings_account`, various methods are referenced ⑭. Methods `withdraw_money()` and `deposit_money()` have `amount` as the parameter. This `amount` is either added or subtracted to the `balance` data attribute. Inside the methods, the data attribute `balance` is referenced through `self`. This shows that not only the data attributes can be accessed within the methods of the same class, but also the values of data attributes can be manipulated.

Program 11.6: Define a Class Called Cart that Contains Data Attributes Apples and Oranges. Write Methods that Return Appropriate Messages If the Number of Apples is Greater than 5 or When the Number of Oranges are Greater than 10.

```
1. class Cart:
2.     def __init__(self, apples, oranges):
3.         self.apples = apples
4.         self.oranges = oranges
5.
6.     def apple_quantity_check(self):
7.         if self.apples > 5:
8.             return 'Sufficient Quantity'
9.         else:
10.            return 'Insufficient Quantity'
11.
12.    def orange_quantity_check(self):
13.        if self.oranges > 10:
14.            return 'Sufficient Quantity'
15.        else:
16.            return 'Insufficient Quantity'
17.
18.    def main():
19.        fruits = Cart(3, 11)
20.        returned_apple_message = fruits.apple_quantity_check()
21.        returned_orange_message = fruits.orange_quantity_check()
22.        print(f"Apple is in {returned_apple_message}")
23.        print(f"Orange is in {returned_orange_message}")
24.
25.    if __name__ == "__main__":
26.        main()
```

OUTPUT

```
Apple is in Insufficient Quantity
Orange is in Sufficient Quantity
```

The instance variables, *apples* and *oranges*, are added ②–④ to the *Cart* Class. The *apple_quantity_check()* ⑤–⑨ and *orange_quantity_check()* ⑩–⑭ methods check for the quantity of apples and oranges and return a string message. Each of these methods ⑮–⑰ are referenced by *fruits* ⑱ object. In the expressions *returned_apple_message = fruits.apple_quantity_check()* and *returned_orange_message = fruits.orange_quantity_check()*, the left-hand side of the assignment should have a matching number of variables to store the values returned by the *return* statement from the class method. It is imperative to recognize that *apples* and *oranges* parameter variables of *__init__()* method are independent of *apples* and *oranges* of instance variables, as they exist in a different scope.

Program 11.7: Program to Demonstrate the Use of Default Parameters in Methods

```
1. class Dog:
2.     def __init__(self, breed="German Shepherd", color="Tan Black"):
3.         self.breed = breed
4.         self.color = color
5.     def dog_breed(self):
6.         print(f"Dog Breed is {self.breed}")
7.     def dog_color(self):
8.         print(f"Dog Color is {self.color}")
9. def main():
10.     babloo = Dog()
11.     babloo.dog_breed()
12.     babloo.dog_color()
13. if __name__ == "__main__":
14.     main()
```

OUTPUT

```
Dog Breed is German Shepherd
Dog Color is Tan Black
```

In the *Dog* class ①, the parameters of the `__init__()` method have default values ②. If no arguments are specified in *Dog()* ⑩ while creating an instance of the class, then the default values set for the `__init__()` method parameters gets assigned to instance variables ③–④. If you specify arguments in *Dog()*, then the default values assigned to `__init__()` method parameters will be overwritten with the latest values.

11.5.1 Using Objects as Arguments

An object can be passed to a calling function as an argument.

Program 11.8: Program to Demonstrate Passing of an Object as an Argument to a Function Call

```
1. class Track:
2.     def __init__(self, song, artist):
3.         self.song = song
4.         self.artist = artist
5. def print_track_info(vocalist):
```

```

6. print(f"Song is '{vocalist.song}'")
7. print(f"Artist is '{vocalist.artist}'")

8. singer = Track("The First Time Ever I Saw Your Face", "Roberta Flack")
9. print_track_info(singer)

```

OUTPUT

```

Song is "The First Time Ever I Saw Your Face"
Artist is "Roberta Flack"

```

In the class *Track* ①, the `__init__()` method is added with the song and artist data attributes ②–④. The `print_track_info()` function receives an object as parameter ⑤–⑦. The object *singer* ⑧ of *Track* class is passed as an argument to `print_track_info()` function ⑨. (Note: It is a function defined outside the class and not a method.) Since you are passing an object of a class as an argument, you have access to all the data attributes attached to that object.

Program 11.9: Given Three Points (x1, y1), (x2, y2) and (x3, y3), Write a Python Program to Check If they are Collinear

```

1. class Collinear:
2.     def __init__(self, x, y):
3.         self.x_coord = x
4.         self.y_coord = y

5.     def check_for_collinear(self, point_2_obj, point_3_obj):
6.         if (point_3_obj.y_coord - point_2_obj.y_coord)*(point_2_obj.x_coord -
            self.x_coord) == (point_2_obj.y_coord - self.y_coord)*(point_3_obj.x_coord -
            point_2_obj.x_coord):
7.             print("Points are Collinear")
8.         else:
9.             print("Points are not Collinear")

10. def main():
11.     point_1 = Collinear(1, 5)
12.     point_2 = Collinear(2, 5)
13.     point_3 = Collinear(4, 6)
14.     point_1.check_for_collinear(point_2, point_3)

15. if __name__ == "__main__":
16.     main()

```

OUTPUT

```

Points are Collinear

```

Three objects, *point_1*, *point_2* and *point_3*, are created for *Collinear* class ⑩–⑫. Each of these objects have their own unique data attributes with their associated values. The `__init__()` method has two data attributes *x_coord* and *y_coord* ②–④. The method `check_for_collinear()` checks whether the coordinates are collinear or not ⑤–⑨. The method `check_for_collinear()` takes three objects as its parameters. The method `check_for_collinear()` is invoked using *point_1* object while *point_2* and *point_3* objects are passed as arguments. On invoking the `check_for_collinear()` method, the *point_1* object is assigned to *self* parameter, *point_2* object in argument is assigned to *point_2_obj* of the parameter, and *point_3* object in argument is assigned to *point_3_obj* of the parameter in the `check_for_collinear()` method header.

When each of these objects are created, each object gets its own unique copy of data attributes defined by that class (FIGURE 11.5). The *x_coord* and *y_coord* data attributes for *point_1* object have values of 1 and 5, the *x_coord* and *y_coord* data attributes for *point_2* object have values of 2 and 5, and the *x_coord* and *y_coord* data attributes for *point_3* object have values of 4 and 6. Three or more points A, B, C,..., are said to be collinear if they lie on a single straight line. If the line segments AB and BC have the same slope, then A, B, C are necessarily collinear. Collinearity for three points A(a, b), B(m, n), and C(x, y) are checked using the formula $(n - b)(x - m) = (y - n)(m - a)$.

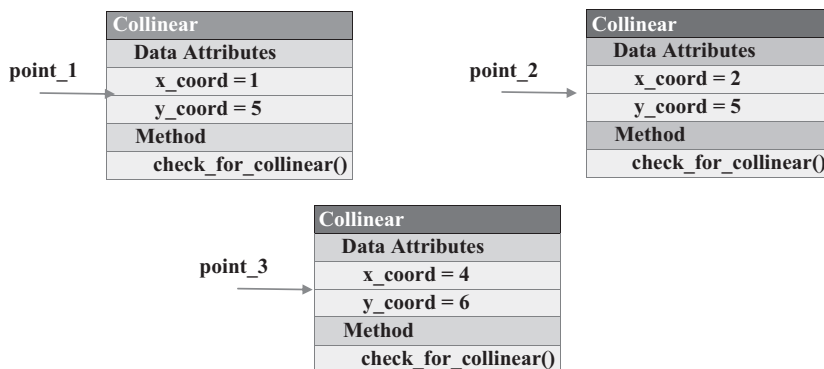


FIGURE 11.5
Object diagram for *Collinear* class with different objects.

11.5.2 Objects as Return Values

It is important to note that everything in Python is an object, including classes. In Python, “everything is an object” (that is, all values are objects) because Python does not include any primitive, unboxed values. Anything that can be used as a value (int, str, float, functions, modules, etc.) is implemented as an object.

The `id()` function is used to find the identity of the location of the object in memory. The syntax for `id()` function is,

id(object)

This function returns the “identity” of an object. This is an integer (or long integer), which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

You can check whether an object is an instance of a given class or not by using the `isinstance()` function. The syntax for `isinstance()` function is,

isinstance (object, classinfo)

where the *object* is an object instance and *classinfo* can be a class, or a tuple containing classes, or other tuples. The *isinstance()* function returns a Boolean stating whether the object is an instance or subclass of another object.

Program 11.10: Given the Coordinates (x, y) of a Center of a Circle and Its Radius, Write Python Program to Determine Whether the Point Lies Inside the Circle, On the Circle or Outside the Circle

```

1. class Circle:
2.     def __init__(self, radius=0, circle_x=0, circle_y=0, point_x=0, point_y=0):
3.         self.radius = radius
4.         self.circle_x_coord = circle_x
5.         self.circle_y_coord = circle_y
6.         self.point_x_coord = point_x
7.         self.point_y_coord = point_y
8.         self.status = ""
9.
10.    def check_point_status(self):
11.        if (self.point_x_coord - self.circle_x_coord) ** 2 + (self.point_y_coord - self.
            circle_y_coord) ** 2 < self.radius ** 2:
12.            self.status = f"Point with coordinates {(self.point_x_coord, self.point_y_
                coord)} is inside the Circle"
13.        elif (self.point_x_coord - self.circle_x_coord) ** 2 + (self.point_y_coord - self.
            circle_y_coord) ** 2 > self.radius ** 2:
14.            self.status = f"Point with coordinates {(self.point_x_coord, self.point_y_
                coord)} is outside the Circle"
15.        else:
16.            self.status = f"Point with coordinates {(self.point_x_coord, self.point_y_
                coord)} is on the Circle"
17.
18.    return self
19.
20. def main():
21.     point = Circle(10, 2, 3, 9, 9)
22.     returned_object = point.check_point_status()
23.     print(returned_object.status)
24.     print(f"Is point an instance of Circle Class? {isinstance(point, Circle)}")
25.     print(f"Is returned_object an instance of Circle Class? {isinstance(returned_
        object, Circle)}")

```

```

23.     print(f"Identity of the location of a point object is {id(point)}")
24.     print(f"Identity of the location of the returned_object object is {id(returned_object)}")

25. if __name__ == "__main__":
26.     main()

```

OUTPUT

```

Point with coordinates (9, 9) is inside the Circle
Is point an instance of Circle Class? True
Is returned_object an instance of Circle Class? True
Identity of the location of a point object is 2351304741216
Identity of the location of the returned_object object is 2351304741216

```

If you have a circle with the center as (*center_x*, *center_y*) and radius as *radius*, then you can test if a given point with coordinates (*x*, *y*) is inside or outside, or on the circle using the formula $(x - \text{center_x})^2 + (y - \text{center_y})^2 < \text{radius}^2$. Please note, the points that satisfy this equation with $<$ operator replaced by $==$ operator are considered to be on the circle, and the points that satisfy this equation with $<$ operator replaced by $>$ operator are considered to be outside the circle.

The *point* ⑧ object is used to invoke *check_point_status()* method. This *check_point_status()* method returns the *self* object itself ⑨–⑯. The *returned_object* variable is used to store the returned object ⑨. Both *point* and *returned_object* are used to store an instance of the same class ⑪–⑫ and they point to the same location in memory, which is why their values are same ⑬–⑭.

Program 11.11: Write Pythonic Program to Compute the End Time of an Opera, While Start Time and Duration are Given

```

1. class Time:
2.     def __init__(self, hours, minutes, seconds):
3.         self.hours = hours
4.         self.minutes = minutes
5.         self.seconds = seconds

6.     def add_time(self, duration):
7.         opera_hours = self.hours + duration.hours
8.         opera_minutes = self.minutes + duration.minutes
9.         opera_seconds = self.seconds + duration.seconds
10.        while opera_seconds >= 60:
11.            opera_seconds = opera_seconds - 60

```

```

12.     opera_minutes = opera_minutes + 1
13.     while opera_minutes >= 60:
14.         opera_minutes = opera_minutes - 60
15.         opera_hours = opera_hours + 1
16.     print(f"Opera ends at {opera_hours}:{opera_minutes}:{opera_seconds}")

17. def main():
18.     opera_start = Time(10, 30, 30)
19.     opera_duration = Time(2, 45, 50)
20.     opera_start.add_time(opera_duration)

21. if __name__ == "__main__":
22.     main()

```

OUTPUT

Opera ends at 13:16:20

In the *Time* class ①, three data attributes, *hours*, *minutes* and *seconds*, are added ②–⑤. The *add_time()* method is invoked using *opera_start* object, while the *opera_duration* object is passed as an argument ⑩–⑫. In the *add_time()* method, *self* parameter is assigned with the *opera_start* object, and duration parameter is assigned with *opera_duration* object. The hours, minutes, and seconds data attributes attached to both of these objects are added and assigned to *opera_hours*, *opera_minutes*, and *opera_seconds* variables. While the total duration of *opera_seconds* variable is greater than sixty, the decrement of *opera_seconds* is done by a value of sixty and increment *opera_minutes* by one. While the total duration of the *opera_minutes* variable is greater than sixty, then decrement *opera_minutes* by a value of sixty and increment the *opera_hours* by one. Finally, print the opera end time ⑬–⑮.

11.6 Class Attributes versus Data Attributes

Generally speaking, Data attributes are instance variables that are unique to each object of a class, and Class attributes are class variables that is shared by all objects of a class.

Program 11.12: Program to Illustrate Class Variables and Instance Variables

```

1. class Dog:
2.     kind = 'canine'
3.     def __init__(self, name):
4.         self.dog_name = name
5. d = Dog('Fido')
6. e = Dog('Buddy')
7. print(f"Value for Shared Variable or Class Variable 'kind' is '{d.kind}'")

```

8. `print(f"Value for Shared Variable or Class Variable 'kind' is '{e.kind}')`
9. `print(f"Value for Unique Variable or Instance Variable 'dog_name' is '{d.dog_name}')`
10. `print(f"Value for Unique Variable or Instance Variable 'dog_name' is '{e.dog_name}')`

OUTPUT

```
Value for Shared Variable or Class Variable 'kind' is 'canine'
Value for Shared Variable or Class Variable 'kind' is 'canine'
Value for Unique Variable or Instance Variable 'dog_name' is 'Fido'
Value for Unique Variable or Instance Variable 'dog_name' is 'Buddy'
```

Here, the variable *kind* is a class variable and is shared by all the objects ②. The instance variable *dog_name* is unique to each of the objects created. You can access both class variables and instance variables using dot notation. The objects *d* and *e* of the class *Dog* share the same class variable *kind*. However, for instance variable *dog_name*, the *self* parameter is replaced with the object created. The instance variable *dog_name* is unique to each of the *d* and *e* objects, which were created resulting in the printing of different values associated with each of these objects.

11.7 Encapsulation

All the programs in this chapter employ the concept of Encapsulation. Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). Encapsulation is the process of combining variables that store data and methods that work on those variables into a single unit called class. In Encapsulation, the variables are not accessed directly; it is accessed through the methods present in the class. Encapsulation ensures that the object's internal representation (its state and behavior) are hidden from the rest of the application. Thus, encapsulation makes the concept of data hiding possible. In order to understand the concept of data hiding, imagine if some programmer is able to access the data stored in a variable from outside the class then there would be a very high danger of them writing their own (not encapsulated) code to deal with your data stored in a variable. This would, at the very least, lead to code duplication (i.e., useless efforts) and to inconsistencies if the implementations are not perfectly compatible. Instead, data hiding means that in order to access data stored in a variable everybody **MUST** use the methods that are provided, so that they are the same for everybody.

An application using a class does not need to know its internal workings or how it is implemented. The program simply creates an object and uses it to invoke the methods of that class. Abstraction is a process where you show only “relevant” variables that are used to access data and “hide” implementation details of an object from the user. Consider your mobile phone (FIGURE 11.6); you just need to know what buttons are to be pressed to send a message or make a call. What happens when you press a button, how your messages are sent, and how your calls are connected are all abstracted away from the user.

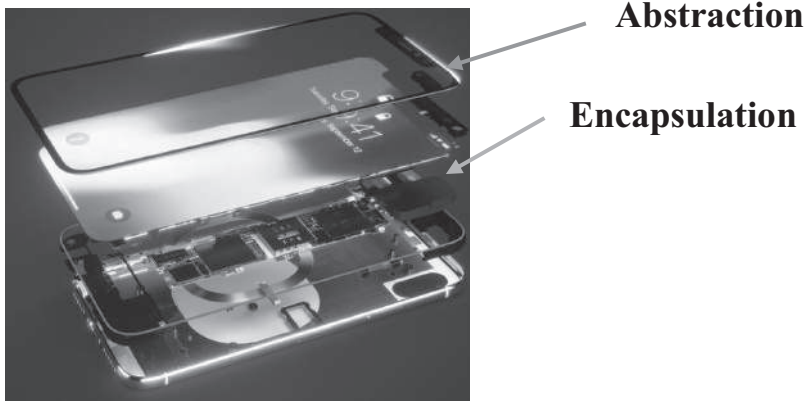
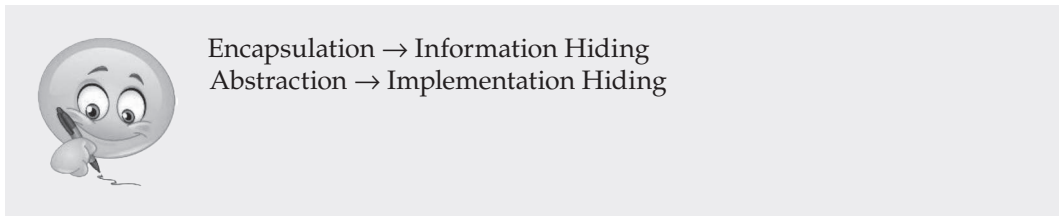


FIGURE 11.6
Demonstration of Abstraction and Encapsulation concept.



Program 11.13: Program to Demonstrate the Difference between Abstraction and Encapsulation

```
1. class foo:
2.     def __init__(self, a, b):
3.         self.a = a
4.         self.b = b
5.
6.     def add(self):
7.         return self.a + self.b
8.
9. foo_object = foo(3,4)
10. foo_object.add()
```

In the above program, the internal representation of an object of *foo* class ①–⑥ is hidden outside the class → Encapsulation. Any accessible member (data/method) of an object of *foo* is restricted and can only be accessed by that object ⑦–⑧. Implementation of *add()* method is hidden → Abstraction.

Program 11.14: Given a point(x, y), Write Python Program to Find Whether it Lies in the First, Second, Third or Fourth Quadrant of x - y Plane

```

1. class Quadrant:
2.     def __init__(self, x, y):
3.         self.x_coord = x
4.         self.y_coord = y
5.
6.     def determine_quadrant(self):
7.         if self.x_coord > 0 and self.y_coord > 0:
8.             print(f'Point with coordinates {(self.x_coord, self.y_coord)} lies in the FIRST
9.             Quadrant')
10.        elif self.x_coord < 0 and self.y_coord < 0:
11.            print(f'Point with coordinates {(self.x_coord, self.y_coord)} lies in the
12.            THIRD Quadrant')
13.        elif self.x_coord > 0 and self.y_coord < 0:
14.            print(f'Point with coordinates {(self.x_coord, self.y_coord)} lies in the
15.            FOURTH Quadrant')
16.        elif self.x_coord < 0 and self.y_coord > 0:
17.            print(f'Point with coordinates {(self.x_coord, self.y_coord)} lies in the
18.            SECOND Quadrant')
19.
20. def main():
21.     point = Quadrant(-180, 180)
22.     point.determine_quadrant()
23.
24. if __name__ == "__main__":
25.     main()

```

OUTPUT

Point with coordinates (-180, 180) lies in the SECOND Quadrant

Depending on the value of *x_coord* and *y_coord* coordinates ②–④, the quadrant in which the point lies is determined. The following conditions characterize the four quadrants: in First Quadrant, both the *x_coord* and *y_coord* coordinates are positive; in Second Quadrant, the *x_coord* coordinate is negative, but the *y_coord* coordinate is positive; in Third Quadrant both *x_coord* and *y_coord* coordinates are negative; and in Fourth Quadrant, *x_coord* coordinate is positive but *y_coord* coordinate is negative ⑤–⑧. Use an *if* statement to determine which quadrant the point under consideration is in.

11.7.1 Using Private Instance Variables and Methods

Instance variables or methods, which can be accessed within the same class and can't be seen outside, are called private instance variables or private methods. Since there is a valid

use-case for class-only private members (namely to avoid name clashes of names with names defined by subclasses), there is support for such a mechanism, which is called name mangling. In Python, an identifier prefixed with a double underscore (e.g., `__spam`) and with no trailing underscores should be treated as private (whether it is a method or an instance variable). Any identifier of the form `__spam` is textually replaced with `__classname__spam`, where *classname* is the current class name with a leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class. Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. Name mangling is intended to give classes an easy way to define “private” instance variables and methods, without having to worry about instance variables defined by derived classes. Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private.

Program 11.15: Program to Demonstrate Private Instance Variables in Python

```
1. class PrivateDemo:
2.     def __init__(self):
3.         self.nonprivateinstance = "I'm not a private instance"
4.         self.__privateinstance = "I'm private instance"
5.     def display_privateinstance(self):
6.         print(f'{self.__privateinstance} used within the method of a class')
7. def main():
8.     demo = PrivateDemo()
9.     print("Invoke Method having private instance")
10.    print(demo.display_privateinstance())
11.    print("Invoke non-private instance variable")
12.    print(demo.nonprivateinstance)
13.    print("Get attributes of the object")
14.    print(demo.__dict__)
15.    print("Trying to access private instance variable outside the class results in an
    error")
16.    print(demo.__privateinstance)
17. if __name__ == "__main__":
18.     main()
```

OUTPUT

```
Invoke Method having private instance
I'm private instance used within the method of a class
Invoke non-private instance variable
I'm not a private instance
Get attributes of the object
```

```
{'nonprivateinstance': 'I'm not private instance', '_PrivateDemo__privateinstance': 'I'm private instance'}
```

Trying to access private instance variable outside the class results in an error

```
AttributeError: 'PrivateDemo' object has no attribute '__privateinstance'
```

In class *PrivateDemo* ①, the `__privateinstance` ④ variable cannot be invoked by an object outside the class but it can be used in a method defined within the class ⑤–⑥. If you try to access private instance variables outside the class, then it results in error ⑮–⑯. You can use `__dict__` to get all the attributes of an object ③–④. As you can see in the output, the `__privateinstance` variable is prefixed with `_PrivateDemo`.

11.8 Inheritance

Inheritance enables new classes to receive or inherit variables and methods of existing classes. Inheritance is a way to express a relationship between classes. If you want to build a new class, which is already similar to one that already exists, then instead of creating a new class from scratch you can reference the existing class and indicate what is different by overriding some of its behavior or by adding some new functionality. A class that is used as the basis for inheritance is called a *superclass* or *base class*. A class that inherits from a base class is called a *subclass* or *derived class*. The terms *parent class* and *child class* are also acceptable terms to use respectively. A derived class inherits variables and methods from its base class while adding additional variables and methods of its own. Inheritance easily enables reusing of existing code.

Class *BaseClass*, on the left, has one variable and one method. Class *DerivedClass*, on the right, is derived from *BaseClass* and contains an additional variable and an additional method (FIGURE 11.7).

If you can describe the relationship between derived classes and base class using the phrase *is-a*, then that relationship is inheritance. For example, a lemon *is-a* citrus fruit, which *is-a* fruit. A shepherd *is-a* dog, which *is-a* animal. A guitar *is-a* steel-stringed instrument, which *is-a* musical instrument. If the *is-a* relationship does not exist between a derived class and base class, you should not use inheritance.

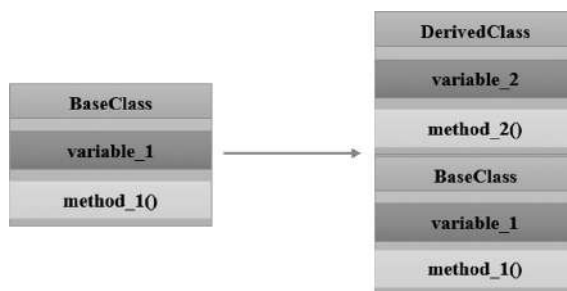
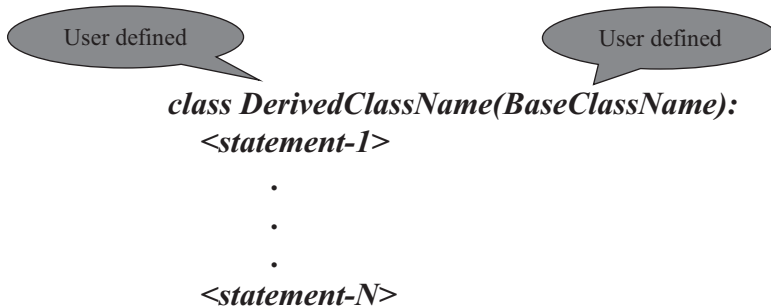


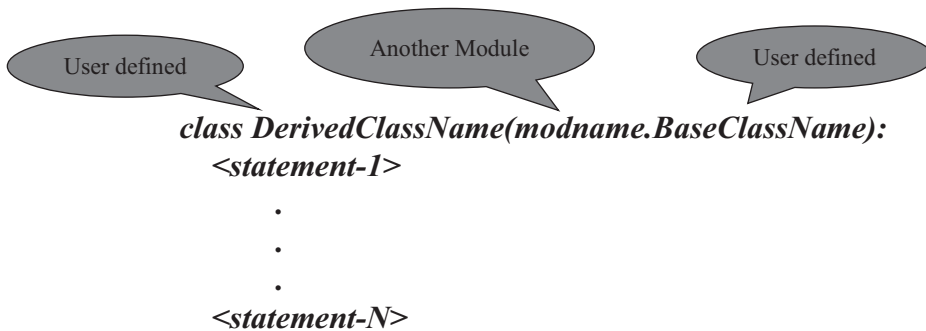
FIGURE 11.7
Relationship between Base Class and Derived Class.

The syntax for a derived class definition looks like this:



To create a derived class, you add a *BaseClassName* after the *DerivedClassName* within the parenthesis followed by a colon. The derived class is said to directly inherit from the listed base class.

In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:



All classes, except the special class *object*, are derived classes, even if they do not have a base class name. The *object* class is the only class that is not derived, since it is the base of the inheritance hierarchy. Classes without a base class name are implicitly derived directly from the class *object*. Leaving off the class *object* from the base class name is just shorthand for specifying that *object* is the base class.

The class definition on the left implicitly derives from the class *object*, while the one on the right explicitly derives from the *object*. The two forms are semantically equivalent, as shown in [FIGURE 11.8](#). When we say derived class, we meant that it is derived from some base class other than the *object* class itself.

<pre>class SomeClass: statement(s)</pre>	<pre>class SomeClass(object): statement(s)</pre>
--	--

FIGURE 11.8
Semantically equivalent class definition.

11.8.1 Accessing the Inherited Variables and Methods

Execution of a derived class definition proceeds the same as for a base class. When the derived class object is constructed, the base class is also remembered. This is used for resolving variable and method attributes. If a requested attribute is not found in the

derived class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class. Inherited variables and methods are accessed just as if they had been created in the derived class itself. (We will discuss inherited constructors later in the chapter which is bit different.)

Program 11.16: Program to Demonstrate Base and Derived Class Relationship Without Using `__init__()` Method in a Derived Class

```
1. class FootBall:
2.     def __init__(self, country, division, no_of_times):
3.         self.country = country
4.         self.division = division
5.         self.no_of_times = no_of_times
6.     def fifa(self):
7.         print(f'{self.country} national football team is placed in {self.division} FIFA
          division")
8. class WorldChampions(FootBall):
9.     def world_championship(self):
10.        print(f'{self.country} national football team is {self.no_of_times} times world
          champions")
11. def main():
12.     germany = WorldChampions("Germany", "UEFA", 4)
13.     germany.fifa()
14.     germany.world_championship()
15. if __name__ == "__main__":
16.     main()
```

OUTPUT

```
Germany national football team is placed in 'UEFA' FIFA division
Germany national football team is 4 times world champions
```

Single inheritance enables a derived class to inherit variables and methods from a single base class. This includes `__init__()` method. So, if you do not define it in a derived class, you will get the one from the base class. Class *FootBall* is the Base Class ①. It has a *country*, *division* and *no_of_times* as the data attributes ③–⑤. The *fifa()* method makes use of two data attributes to print a message ⑥–⑦. The derived class *WorldChampions* ⑧ is derived from *FootBall* as the base class. The derived class has access to all the data attributes and the methods of the base class. In the derived class *WorldChampions()*, there is no mention of the `__init__()` method explicitly, but it has access to the `__init__()` method of base class. The method, *world_championship()* ⑨–⑩, is added to derived class *WorldChampions*. This method can access the data attributes of *FootBall* base class. The instance variable *germany* is created for *WorldChampions* class. Notice that we are passing arguments to *WorldChampions()* ⑫, which in turn assigns the values to the

parameters of `__init__()` method in *FootBall* base class. The *germany* object invokes the methods found in *FootBall* base class and in *WorldChampions* derived class ⑬–⑭.

11.8.2 Using *super()* Function and Overriding Base Class Methods

In a single inheritance, the built-in *super()* function can be used to refer to base classes without naming them explicitly, thus making the code more maintainable. If you need to access the data attributes from the base class in addition to the data attributes being specified in the derived class's `__init__()` method, then you must explicitly call the base class `__init__()` method using *super()* yourself, since that will not happen automatically. However, if you do not need any data attributes from the base class, then no need to use *super()* function to invoke base class `__init__()` method.

The syntax for using *super()* in derived class `__init__()` method definition looks like this:

```
super().__init__(base_class_parameter(s))
```

Its usage is shown below.

```
class DerivedClassName(BaseClassName):
```

```
    def __init__(self, derived_class_parameter(s), base_class_parameter(s))
```

```
        super().__init__(base_class_parameter(s))
```

```
        self.derived_class_instance_variable = derived_class_parameter
```

The derived class `__init__()` method contains its own parameters along with the parameters specified in the `__init__()` method of base class. No need to specify *self* while invoking base class `__init__()` method using *super()*. No need to assign *base_class_parameters* specified in `__init__()` method of the derived class to any data attributes as the same is taken care of in the `__init__()` method of base class.

Sometimes you may want to make use of some of the parent class behaviors but not all of them. Method overriding, in object-oriented programming, is a language feature that allows a derived class to provide its own implementation of a method that is already provided in base class. Derived classes may override methods of their base class. When you change the definition of parent class methods, you override them. These methods have the same name as those in the base class. The method in the derived class and the method in the base class each should have the same method signature. An overriding method in a derived class may, in fact, want to extend rather than simply replace the base class method of the same name. When constructing the base and derived classes, it is important to keep program design in mind so that overriding does not produce unnecessary or redundant code.



Method signature refers to the method name, order and the total number of its parameters. Return types and thrown exceptions are not considered to be a part of the method signature.

Program 11.17: Program to Demonstrate the Use of super() Function

```

1. class Country:
2.     def __init__(self, country_name):
3.         self.country_name = country_name
4.     def country_details(self):
5.         print(f"Happiest Country in the world is {self.country_name}")

6. class HappiestCountry(Country):
7.     def __init__(self, country_name, continent):
8.         super().__init__(country_name)
9.         self.continent = continent
10.    def happy_country_details(self):
11.        print(f"Happiest Country in the world is {self.country_name} and is in {self.
            continent} ")

12. def main():
13.     finland = HappiestCountry("Finland", "Europe")
14.     finland.happy_country_details()
15. if __name__ == "__main__":
16.     main()


```

OUTPUT

Happiest Country in the world is Finland and is in Europe

The `__init__()` method ⑦ for *HappiestCountry* class ⑥ take two parameters *country_name* and *continent*. Within the `__init__()` method of *HappiestCountry* derived class, the `__init__()` method of the *Country* base class is invoked using *super()* function ⑧. When you use *super()* function to invoke base class `__init__()` method, you need to pass the *country_name* parameter as an argument to `__init__()` function to match the method signature. On invoking the base class `__init__()` method, the *country_name* value gets assigned to *country_name* data attribute ②–③ in the *Country* base class ①. Object *finland* is used to invoke the *happy_country_details()* method found in derived class. In the method *happy_country_details()*, the derived class attributes and base class attributes are accessed.

The *super()* function is useful for accessing the base class methods that have been overridden in a derived class without explicitly specifying the base class name. The syntax for using the *super()* function to invoke the base class method is,



User defined

super().invoke_base_class_method(argument(s))

The above expression should be used within a method of the derived class. The main advantage of using *super()* function comes with multiple inheritance.

Program 11.18: Program to Demonstrate the Overriding of the Base Class Method in the Derived Class

```

1. class Book:
2.     def __init__(self, author, title):
3.         self.author = author
4.         self.title = title
5.     def book_info(self):
6.         print(f"{self.title} is authored by {self.author}")

7. class Fiction(Book):
8.     def __init__(self, author, title, publisher):
9.         super().__init__(author, title)
10.        self.publisher = publisher
11.    def book_info(self):
12.        print(f"{self.title} is authored by {self.author} and published by {self.publisher}")
13.    def invoke_base_class_method(self):
14.        super().book_info()

15. def main():
16.     print("Derived Class")
17.     silva_book = Fiction("Daniel Silva", "Prince of Fire", "Berkley")
18.     silva_book.book_info()
19.     silva_book.invoke_base_class_method()
20.     print("-----")
21.     print("Base Class")
22.     reacher_book = Book("Lee Child", "One Shot")
23.     reacher_book.book_info()

24. if __name__ == "__main__":
25.     main()

```

OUTPUT

Derived Class

Prince of Fire is authored by Daniel Silva and published by Berkley

Prince of Fire is authored by Daniel Silva

Base Class

One Shot is authored by Lee Child

The object *silva_book* ⑦ of *Fiction* derived class is used to invoke *book_info()* ⑩ method. When the same method exists in both the base class and the derived class, the method in the derived class will be executed ⑪–⑫. Derived class method overrides the base class method. You can also directly invoke the base class *book_info()* method from within the

derived class by using *super()* function. You need to put *super().book_info()* under another method within the derived class. Use the *silva_book* object to invoke ⑩ *invoke_base_class_method()* ⑬–⑭, which in turn invokes the base class *book_info()* method ⑤–⑥. You can also access the base class methods directly by creating an instance variable for *Book* base class and calling the methods of that class ②–③.

11.8.3 Multiple Inheritances

Python also supports a form of multiple inheritances. A derived class definition with multiple base classes looks like this:

```
class DerivedClassName(Base_1, Base_2, Base_3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Derived class *DerivedClassName* is inherited from multiple base classes, *Base_1*, *Base_2*, *Base_3*. For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in *DerivedClassName*, it is searched for in *Base_1*, then (recursively) in the base classes of *Base_1*, and if it was not found there, it would be searched for in *Base_2*, and so on. Even though multiple inheritances are available in Python programming language, it is not highly encouraged to use it as it is hard and error prone.

You can call the base class method directly using Base Class name itself without using the *super()* function. The syntax is,

```
BaseClassName.methodname(self, arguments)
```

Notice the difference between *super()* function and the base class name in calling the method name.

Use *issubclass()* function to check class inheritance. The syntax is,

```
issubclass(DerivedClassName, BaseClassName)
```

This function returns Boolean True if *DerivedClassName* is a derived class of base class *BaseClassName*. The *DerivedClassName* class is considered a subclass of itself. *BaseClassName* may be a tuple of classes, in which case every entry in *BaseClassName* will be checked. In any other case, a *TypeError* exception is raised.

Program 11.19: Program to Demonstrate Multiple Inheritance

1. class Poissonier:
2. def __init__(self, poissonier_role):

```

3.     self.poissonier_role = poissonier_role
4.     def display_poissonier_chef_info(self):
5.         print(f"Chef is mainly involved in preparing {self.poissonier_role}")

6. class Entremetier:
7.     def __init__(self, entremetier_role):
8.         self.entremetier_role = entremetier_role
9.     def display_entremetier_chef_info(self):
10.        print(f"Chef is mainly involved in preparing {self.entremetier_role}")

11. class Cook(Poissonier, Entremetier):
12.     def __init__(self, poissonier_role, entremetier_role):
13.         Poissonier.__init__(self, poissonier_role)
14.         Entremetier.__init__(self, entremetier_role)
15.     def invoke_base_class_methods(self):
16.         Poissonier.display_poissonier_chef_info(self)
17.         Entremetier.display_entremetier_chef_info(self)

18. def main():
19.     print(f"Is Cook a derived class of Poissonier Base Class? {issubclass(Cook,
        (Entremetier, Poissonier))}")
20.     chef = Cook("SeaFood", "Vegetables")
21.     chef.invoke_base_class_methods()

22. if __name__ == "__main__":
23.     main()

```

OUTPUT

Is Cook a derived class of Poissonier Base Class? True

Chef is mainly involved in preparing SeaFood

Chef is mainly involved in preparing Vegetables

Class *Poissonier* ①–⑤ and *Entremetier* ⑥–⑩ are base classes. For *Poissonier* and *Entremetier* base classes, *poissonier_role* and *entremetier_role* are the data attributes and *display_poissonier_chef_info()* and *display_entremetier_chef_info()* are the methods defined in their respective classes. The class *Cook* ⑪ is the derived class, which inherits from *Poissonier* and *Entremetier* base classes. You can use the base class names *Poissonier* and *Entremetier* to directly invoke their corresponding *__init__()* methods ⑬–⑭. You can also invoke the methods defined in the base classes by using their corresponding base class names, provided that those expressions be used within some method in the derived class ⑮–⑰. You can check whether *Cook* is a derived class of *Poissonier* base class by using *issubclass()* function ⑱.

Program 11.20: Program to Demonstrate Multiple Inheritance with Method Overriding

```
1. class Pet:
2.     def __init__(self, breed):
3.         self.breed = breed
4.     def about(self):
5.         print(f"This is {self.breed} breed")

6. class Insurable:
7.     def __init__(self, amount):
8.         self.amount = amount
9.     def about(self):
10.        print(f"Its insured for an amount of {self.amount}")

11. class Cat(Pet, Insurable):
12.     def __init__(self, weight, breed, amount):
13.         self.weight = weight
14.         Pet.__init__(self, breed)
15.         Insurable.__init__(self, amount)
16.     def get_weight(self):
17.         print(f"{self.breed} Cat weighs around {self.weight} pounds")

18. def main():
19.     cat_obj = Cat(15, "Ragdoll", "$100")
20.     cat_obj.about()
21.     cat_obj.get_weight()
22. if __name__ == "__main__":
23.     main()
```

OUTPUT

```
This is Ragdoll breed
Ragdoll Cat weighs around 15 pounds
```

Base classes *Pet* ①–⑤ and *Insurable* ⑥–⑩ are inherited by the *Cat* derived class ⑪–⑰. Both *Pet* and *Insurable* classes have the *about()* method definition added to them. The *__init__()* method of *Pet* and *Insurable* classes are invoked using their respective base classes in the *Cat* derived class ⑭–⑮. The *cat_obj* object of *Cat* class is used to invoke *about()* method ⑳. Now the question is which *about()* method gets invoked? Is it the *about()* method found in *Pet* base class or *Insurable* base class? The answer to this question lies in Method Resolution Order (MRO).

11.8.4 Method Resolution Order (MRO)

Method Resolution Order, or “MRO” in short, denotes the way Python programming language resolves a method found in multiple base classes. For a single inheritance, the MRO does not come into play but for multiple inheritances MRO matters a lot. When a derived class inherits from multiple classes, Python programming language needs a way to resolve the methods that are found in multiple base classes as well as in the derived class, which is invoked by an object. MRO uses the C3 linearization algorithm to determine the order of the methods to be invoked in multiple inheritances while guaranteeing monotonicity. MRO applies a set of rules to resolve the method order by constructing linearization.

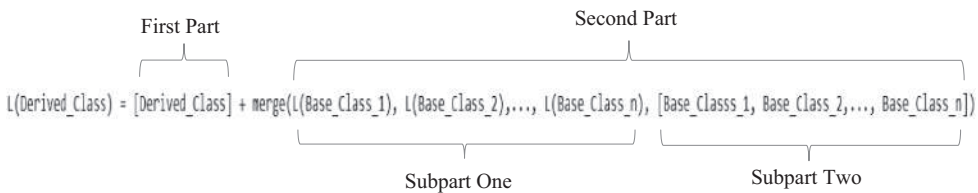
Python provides `mro()` method to get information about Method Resolution Order. The syntax to find MRO is,

`class_name.mro()`

where `class_name` is the name of a class.

Here, we provide you the “behind the scene” details of building an MRO in the case of multiple inheritances. The C3 linearization of a derived class is the sum of the derived class, plus a unique *merge* of linearization of its base classes and a list of base classes itself. The derived class and all the base classes from which the class is derived are represented as list items.

The C3 linearization of a derived class can be represented as below.



The letter *L* stands for Linearization of a class. The above representation consists of two parts, where the first part is the Derived class represented as a list item. The second part consists of *merge*, whose results are added as items to the list of the first part. Again, inside *merge*, it consists of two subparts. Subpart One is the representation of linearization of all the base classes from which the *Derived_Class* was derived, and Subpart Two is a list of all the base classes from which the *Derived_Class* was derived. The ordering of base classes of a derived class from the nearest base class to the furthest base class with the derived class preceding the base classes is called local precedence of classes. The list of base classes as the last argument in the *merge* part preserves the local precedence order of the base classes. An MRO is monotonic when the following is True: if *Base_Class_1* precedes *Base_Class_2* in the linearization of *Derived_Class*, then *Base_Class_1* precedes *Base_Class_2* in the linearization of any derived classes of *Derived_Class* itself. The construction of linearization for MRO should respect local precedence ordering and monotonicity.

The linearization result for a class is replaced by a list sequence. The presence of a class in the first position in multiple list sequences at the same time is called *Head* or *good Head*, but it should not appear in any other position. If a class is in the first position in one of the list sequence and it is present in a different position, other than the first position itself in the other list sequences, or if a class is not present in the first position at all in any of the list sequences, then it is called a *Tail*.

Steps to follow in the construction of C3 linearization MRO for a class is,

Step 1: The linearization result for a base class which derives from *object* class will be represented as a list with the base class itself as the first item and *object* class as the second item.

Step 2: Inside *merge* part, replace L(Base_Class_1), L(Base_Class_2), ..., L(Base_Class_n) with their respective linearization list results.

Step 3: Inside *merge* part, check if the class in the first position in the *first* list is found as a Head or a Tail in all the other list sequences.

Step 4: If the class in the first position in the *first* list is a Head, then remove it from all the list sequences it is present and add it to the end of the list in the first part of linearization.

Step 5: If the class in the first position in the *first* list is a Tail, then skip the *first* list and move to the *next* list sequence.

Step 6: If the class in the first position in the *next* list sequence is a Head, then remove it from all the list sequences it is present and add it to the end of the list in the first part of linearization. If the class in the first position in the *next* list is a Tail, then move on to the *next* list sequence and again check whether the class present in the first position is Head or Tail. If it is Head, then grab it or else move on to the *next* list sequence and so on.

Repeat steps 3 to 6 until all the classes are removed from all the list sequences in the merge part, or it is impossible to find good Heads. In the latter case, it is impossible to construct the linearization for the class.

NOTE: In Step 3, until some class is present (other than the default *object* class) in the first list, it will be considered as the *first* list among all the list sequences present in the merge part of linearization. After all the classes from the first list are removed, the second list will be considered as the *first* list. Once all the classes in the second list are removed, then the third list will be considered as the *first* list and so on. In Step 2, these base classes themselves might have been derived from multiple classes.

Program 11.21: Program to Demonstrate the Construction of Method Resolution Order in Python

```

1. class First:
2.     def my_method(self):
3.         print("You found me in Class First")
4. class Second:
5.     def my_method(self):
6.         print("You found me in Class Second")
7. class Third:
8.     def my_method(self):
9.         print("You found me in Class Third")
10. class Fourth(Third, First):
11.     pass

```

```

12. class Fifth(Third, Second):
13.     pass
14. class Sixth(Fifth, Fourth):
15.     pass
16. def main():
17.     obj = Sixth()
18.     obj.my_method()
19.     print(Sixth.mro())
20. if __name__ == "__main__":
21.     main()

```

OUTPUT

You found me in Class Third

```
[<class '__main__.Sixth'>, <class '__main__.Fifth'>, <class '__main__.Fourth'>, <class '__main__.Third'>, <class '__main__.Second'>, <class '__main__.First'>, <class 'object'>]
```

The method *my_method()* is defined in classes *First* ①–③, *Second* ④–⑥, and *Third* ⑦–⑨. Class *Fourth* ⑩ inherits from classes *Third* and *First*. Class *Fifth* ⑪ inherits from classes *Third* and *Second*, and class *Sixth* ⑫ inherits from classes *Fifth* and *Fourth*. Object *obj* for class *Sixth* ⑫ is created and is used to invoke *my_method()* ⑫ method. The derived classes appear before the base classes. Now the question arises, if you were to call the method *my_method()* through the *obj* object, then from which class would it be called from? Would it be from class *Fifth* or from class *Fourth* or any other class? Use *mro()* method to display the MRO for class *Sixth* ⑫. According to the MRO of class *Sixth*, Python finds the first occurrence of the method *my_method()* in class *Third* and ends up calling the method in that class.

Let's construct an MRO manually for the above multiple inheritance code using C3 Linearization algorithm.

1. L(First) = [First, object]
2. L(Second) = [Second, object]
3. L(Third) = [Third, object]
4. L(Fourth) = [Fourth] + merge(L(Third), L(First), [Third, First])
5. = [Fourth] + merge([Third, object], [First, object], [Third, First])
6. = [Fourth, Third] + merge([object], [First, object], [First])
7. = [Fourth, Third, First] + merge([object], [object])
8. L(Fourth) = [Fourth, Third, First, object]
9. L(Fifth) = [Fifth] + merge(L(Third), L(Second), [Third, Second])
10. = [Fifth] + merge([Third, object], [Second, object], [Third, Second])
11. = [Fifth, Third] + merge([object], [Second, object], [Second])
12. = [Fifth, Third, Second] + merge([object], [object])
13. L(Fifth) = [Fifth, Third, Second, object]

14. $L(\text{Sixth}) = [\text{Sixth}] + \text{merge}(L(\text{Fifth}), L(\text{Fourth}), [\text{Fifth}, \text{Fourth}])$
15. $\quad = [\text{Sixth}] + \text{merge}([\text{Fifth}, \text{Third}, \text{Second}, \text{object}], [\text{Fourth}, \text{Third}, \text{First}, \text{object}], [\text{Fifth}, \text{Fourth}])$
16. $\quad = [\text{Sixth}, \text{Fifth}] + \text{merge}([\text{Third}, \text{Second}, \text{object}], [\text{Fourth}, \text{Third}, \text{First}, \text{object}], [\text{Fourth}])$
17. $\quad = [\text{Sixth}, \text{Fifth}, \text{Fourth}] + \text{merge}([\text{Third}, \text{Second}, \text{object}], [\text{Third}, \text{First}, \text{object}])$
18. $\quad = [\text{Sixth}, \text{Fifth}, \text{Fourth}, \text{Third}] + \text{merge}([\text{Second}, \text{object}], [\text{First}, \text{object}])$
19. $\quad = [\text{Sixth}, \text{Fifth}, \text{Fourth}, \text{Third}, \text{Second}] + \text{merge}([\text{object}], [\text{First}, \text{object}])$
20. $\quad = [\text{Sixth}, \text{Fifth}, \text{Fourth}, \text{Third}, \text{Second}, \text{First}] + \text{merge}([\text{object}], [\text{object}])$
21. $L(\text{Sixth}) = [\text{Sixth}, \text{Fifth}, \text{Fourth}, \text{Third}, \text{Second}, \text{First}, \text{object}]$

Since class *First* ①, *Second* ②, and *Third* ③ are base classes and are derived from the *object* class, their linearization result is represented as a list with the first item being the base class itself and the second item is the *object* class.

Class *Fourth* is a derived class and is derived from classes *Third* and *First*. Let's find the linearization of class *Fourth*, which is represented as in the line ④ and for the last argument of the *merge*. The base classes are represented as lists. For class *Third* and *First*, replace $L(\text{Fifth})$ and $L(\text{Fourth})$ with their linearization results ⑤. According to Step 4 of "construction of linearization MRO for a class," the class *Third* is in the first position of the *first* list, which is a good Head. Class *Third* is also found in the first position of the third list sequence. Remove it from all the list sequences and add it to the end of the list in the first part ⑥. As there are no more classes in the first list, the second list sequence becomes the *first* list. Again, according to Step 3 and Step 4, class *First* is in the first position, which is the good Head. Remove it from all the list sequences and add it to the list in the first part of linearization ⑦. At ⑦, the result of $\text{merge}([\text{object}], [\text{object}])$ gives an *object*, which is added as the last item to the list in the first part of linearization. So, the linearization result for class *Fourth* is $[\text{Fourth}, \text{Third}, \text{First}, \text{object}]$ ⑧. Likewise, the linearization result for class *Fifth* is $[\text{Fifth}, \text{Third}, \text{Second}, \text{object}]$ ⑨–⑩.

Class *Sixth* is derived from classes *Fifth* and *Fourth* and is represented as in the line ⑪. For class *Fifth* and *Fourth*, replace $L(\text{Fifth})$ and $L(\text{Fourth})$ with their linearization results ⑩. According to Step 3 check whether the class in the first position of the *first* list is Head or Tail. According to Step 4, class *Fifth* is a good Head. Remove class *Fifth* from all the list sequences and add it to the end of the list in the first part of linearization ⑫. Next, class *Third* is in the first position of the *first* list. However, class *Third* is a Tail as it is found in a position other than the first position in other list sequences. According to Step 5, move to the *next* list sequence which in our case is the second list. Check whether the class in the first position of the second list, for instance, class *Fourth* is Head or Tail. Since class *Fourth* is Head, remove it from all the list sequences where it appears and add it to the end of the list in the first part of linearization ⑬. Again, start from the *first* list and check for the class in the first position. Since, class *Third* is a good Head, grab it and add it to the list in the first part of linearization ⑭. Continue this process until all the classes are exhausted from all the list sequences ⑮–⑳.

With the construction of C3 linearization MRO for class *Sixth*, the lookup would be in the order:

Class Sixth → Class Fifth → Class Fourth → Class Third → Class Second → Class First
→ Class object ㉑

Program 11.22: Program to Demonstrate the Solving of Diamond Problem in Python

```
1. class First:
2.     def my_method(self):
3.         print("You found me in Class First")
4.
5. class Second(First):
6.     pass
7.
8. class Third(First):
9.     def my_method(self):
10.        print("You found me in Class Third")
11.
12. class Fourth(Second, Third):
13.     pass
14.
15. def main():
16.     obj = Fourth()
17.     obj.my_method()
18.     print(f"Method Resolution Order is {Fourth.mro()}")
19.
20. if __name__ == "__main__":
21.     main()
```

OUTPUT

You found me in Class Third

Method Resolution Order is [<class '__main__.Fourth'>, <class '__main__.Second'>, <class '__main__.Third'>, <class '__main__.First'>, <class 'object'>]

Classes *First*, *Second*, *Third*, and *Fourth* are defined. Class *Fourth* inherits from both *Second* and *Third* classes. Class *Second* and *Third* inherit from class *First*. Class *First* does not inherit from any base classes. This sort of inheritance is called the “Diamond Problem” or the “Deadly Diamond of Death” (FIGURE 11.9).

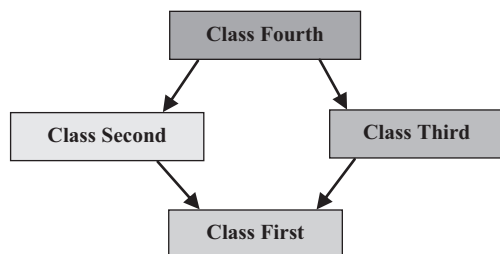
**FIGURE 11.9**

Illustration of diamond problem.

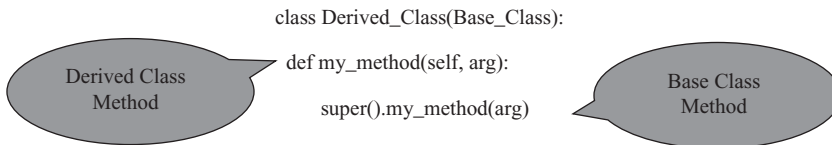
With the construction of C3 linearization MRO for class *Fourth*, the lookup would be in the order:

Class Fourth → Class Second → Class Third → Class First → Class object

Python follows C3 linearization algorithm to build MRO and hence ends up calling the method from class *Third* as it finds the first occurrence of *my_method()* method to be in class *Third* as per the MRO.

Apart from using *super()* function in single inheritance to refer to base classes without naming them explicitly, the *super()* function is also used to support cooperative multiple inheritances in a dynamic execution environment. Cooperative classes are written with multiple inheritances in mind, using a pattern called "cooperative super call". Cooperative multiple inheritances make it possible to handle the "diamond problem," where multiple base classes implement the same method and each method should only be called exactly once. This Good design dictates that this method has the same calling signature in every class because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime. This is unique to Python and is not found in statically compiled languages or languages that only support single inheritance.

A typical *next* base class method call in MRO using *super()* function looks like this:



The *super()* function ensures that the derived class that may be using cooperative multiple inheritances will call the correct *next* class method in the MRO. In multiple inheritances, the *super()* function is not calling the base classes of a derived class, but, instead, it is calling the *next* class in the MRO. The *super()* function finds the method in the *next* class of the MRO.

In program 11.22, if you change the class definition for class *Third* starting from line 6 to 8 as below,

```
class Third(First):
    def my_method(self):
        print("You found me in Class Third")
        super().my_method()
```

THEN THE OUTPUT WILL BE

You found me in Class Third

You found me in Class First

Method Resolution Order is [<class '__main__.Fourth'>, <class '__main__.Second'>, <class '__main__.Third'>, <class '__main__.First'>, <class 'object'>]

From MRO, the method in class *Third* is executed as the first occurrence of the method *my_method()* is in class *Third*. The *super()* function calls the method found in the *next* class of the MRO. Hence, the method *my_method()* in class *First* gets executed.

Let's construct an MRO manually for the above multiple inheritance code using C3 Linearization algorithm.

1. $L(\text{First}) = [\text{First}, \text{object}]$
2. $L(\text{Second}) = [\text{Second}] + \text{merge}(L(\text{First}), [\text{First}])$
3. $\quad = [\text{Second}] + \text{merge}([\text{First}, \text{object}], [\text{First}])$
4. $\quad = [\text{Second}, \text{First}] + \text{merge}([\text{object}])$
5. $L(\text{Second}) = [\text{Second}, \text{First}, \text{object}]$
6. $L(\text{Third}) = [\text{Third}] + \text{merge}(L(\text{First}), [\text{First}])$
7. $\quad = [\text{Third}] + \text{merge}([\text{First}, \text{object}], [\text{First}])$
8. $\quad = [\text{Third}, \text{First}] + \text{merge}([\text{object}])$
9. $L(\text{Third}) = [\text{Third}, \text{First}, \text{object}]$
10. $L(\text{Fourth}) = [\text{Fourth}] + \text{merge}(L(\text{Second}), L(\text{Third}), [\text{Second}, \text{Third}])$
11. $\quad = [\text{Fourth}] + \text{merge}([\text{Second}, \text{First}, \text{object}], [\text{Third}, \text{First}, \text{object}], [\text{Second}, \text{Third}])$
12. $\quad = [\text{Fourth}, \text{Second}] + \text{merge}([\text{First}, \text{object}], [\text{Third}, \text{First}, \text{object}], [\text{Third}])$
13. $\quad = [\text{Fourth}, \text{Second}, \text{Third}] + \text{merge}([\text{First}, \text{object}], [\text{First}, \text{object}])$
14. $\quad = [\text{Fourth}, \text{Second}, \text{Third}, \text{First}] + \text{merge}([\text{object}], [\text{object}])$
15. $L(\text{Fourth}) = [\text{Fourth}, \text{Second}, \text{Third}, \text{First}, \text{object}]$

With the construction of C3 linearization MRO for class *Fourth*, the lookup would be in the order:

Class Fourth \rightarrow Class Second \rightarrow Class Third \rightarrow Class First \rightarrow Class object

Program 11.23: Program to Demonstrate the Use of *super()* Function in Multiple Inheritances

1. class First:
2. def __init__(self):
3. print("In First")
4. super().__init__()
5. class Second:
6. def __init__(self):
7. print("In Second")
8. super().__init__()
9. class Third(First, Second):

```

10. def __init__(self):
11.     print("In Third")
12.     super().__init__()

13. def main():
14.     obj = Third()
15.     print(f"Method Resolution Order is {Third.mro()}")
16. if __name__ == "__main__":
17.     main()

```

OUTPUT

In Third

In First

In Second

Method Resolution Order is [<class '__main__.Third'>, <class '__main__.First'>, <class '__main__.Second'>, <class 'object'>]

The order to resolve `__init__()` method is,

Class Third → Class First → Class Second → Class object

The `__init__()` method of class *Third* is called first. After that "In Third" is printed. Next, according to the MRO, inside the `__init__()` method of class *Third*, the `super().__init__()` calls the `__init__()` method of the next class found in MRO i.e., the `__init__()` method in class *First* gets called. After that "In First" is printed. Here, class *First* derives from the *object* class but in multiple inheritances, if *super()* function is present in the current class then it will call the overridden method found in the *next* class in the MRO. Inside `__init__()` of class *First*, the `super().__init__()` calls the `__init__()` method of the next class found in MRO i.e., the `__init__()` method of class *Second* gets called because that is what the MRO dictates. After that "In Second" is printed. Inside `__init__()` method of class *Second*, the `super().__init__()` calls the `__init__()` method of the *object* class, which amounts to nothing. Whether an overridden method in the *next* class is called or not depends on whether the *super()* function was called from a class preceding it in the MRO.

Let's construct an MRO manually for the above multiple inheritance code using C3 Linearization algorithm.

1. L(First) = [First, object]
2. L(Second) = [Second, object]
3. L(Third) = [Third] + merge(L(First), L(Second), [First, Second])
4. = [Third] + merge([First, object], [Second, object], [First, Second])
5. = [Third, First] + merge([object], [Second, object], [Second])
6. = [Third, First, Second] + merge([object], [object])
7. L(Third) = [Third, First, Second, object]

With the construction of C3 linearization MRO for class *Fourth*, the lookup would be in the order:

Class Third → Class First → Class Second → Class object

11.9 The Polymorphism

Poly means *many* and morphism means *forms*. Polymorphism is one of the tenets of Object Oriented Programming (OOP). Polymorphism means that you can have multiple classes where each class implements the same variables or methods in different ways. Polymorphism takes advantage of inheritance in order to make this happen. A real-world example of polymorphism is suppose when if you are in classroom that time you behave like a student, when you are in market at that time you behave like a customer, when you are at your home at that time you behave like a son or daughter, such that same person is presented as having different behaviors.

Python is a dynamically-typed language and specifically uses duck-typing. The term duck-typing comes from the idiomatic saying, "If it looks like a duck and quacks like a duck, it is probably a duck." Duck-typing in Python allows us to use any object that provides the required methods and variables without forcing it to belong to any particular class. In duck-typing, an object's suitability is determined by the presence of methods and variables rather than the actual type of the object. To elaborate, this means that the expression *some_obj.foo()* will succeed if object *some_obj* has a *foo* method, regardless of to which class *some_obj* object actually belongs to. Difference between inheritance and polymorphism is, while inheritance is implemented on classes, polymorphism is implemented on methods.

Program 11.24: Program to Demonstrate Polymorphism in Python

```
1. class Vehicle:
2.     def __init__(self, model):
3.         self.model = model
4.     def vehicle_model(self):
5.         print(f"Vehicle Model name is {self.model}")

6. class Bike(Vehicle):
7.     def vehicle_model(self):
8.         print(f"Vehicle Model name is {self.model}")

9. class Car(Vehicle):
10.    def vehicle_model(self):
11.        print(f"Vehicle Model name is {self.model}")
```

```
12. class Aeroplane:
13.     pass

14. def vehicle_info(vehicle_obj):
15.     vehicle_obj.vehicle_model()

16. def main():
17.     ducati = Bike("Ducati-Scrambler")
18.     beetle = Car("Volkswagon-Beetle")
19.     boeing = Aeroplane()
20.     for each_obj in [ducati, beetle, boeing]:
21.         try:
22.             vehicle_info(each_obj)
23.         except AttributeError:
24.             print("Expected method not present in the object")
25. if __name__ == "__main__":
26.     main()
```

OUTPUT

```
Vehicle Model name is Ducati-Scrambler
Vehicle Model name is Volkswagon-Beetle
Expected method not present in the object
```

Even though each of these methods in the classes have the same name, their implementation details are different. Polymorphic behavior allows you to specify common methods in a base class and implement them differently in other derived classes. In this program, we defined two derived classes, *Bike* ⑥–⑧ and *Car* ⑨–⑪, inherited from *vehicle* class, and provided their own implementation of *vehicle_model()* method on top of *vehicle_model()* method found in *Vehicle* class ①–⑤. Notice that all the classes have *vehicle_model()* method but they are implemented differently. The method *vehicle_model()* is polymorphic, as these methods have the same name but belong to different classes and are executed depending on the object. The behavior of the same method belonging to different classes is different based on the type of object.

To allow polymorphism, a common interface called *vehicle_info()* ⑭–⑮ function is created that can take an object of any type and call that object's *vehicle_model()* method. When you pass the objects *ducati* ⑰ and *beetle* ⑱ to the *vehicle_info()* function, it executes *vehicle_model()* method effectively. Because of polymorphism, the run-time type of each object is invoked. In the *for* loop, you iterate through each object in the list using *each_obj* as the iterating variable. Depending on what type of object it has, the program decides which methods it should use. If that object does not have the methods that are called, then the function signals a run-time error. If the object does have the methods, then they are executed no matter the type of the object, evoking the quotation and, hence, the name of this form of typing. Since object *boeing* ⑲ has no *vehicle_model()* method associated with it, an exception is raised ⑳.

Program 11.25: Write Python Program to Calculate Area and Perimeter of Different Shapes Using Polymorphism

```
1. import math
2. class Shape:
3.     def area(self):
4.         pass
5.     def perimeter(self):
6.         pass
7. class Rectangle(Shape):
8.     def __init__(self, width, height):
9.         self.width = width
10.        self.height = height
11.    def area(self):
12.        print(f"Area of Rectangle is {self.width * self.height}")
13.    def perimeter(self):
14.        print(f"Perimeter of Rectangle is {2 * (self.width + self.height)}")
15. class Circle(Shape):
16.    def __init__(self, radius):
17.        self.radius = radius
18.    def area(self):
19.        print(f"Area of Circle is {math.pi * self.radius ** 2}")
20.    def perimeter(self):
21.        print(f"Perimeter of Circle is {2 * math.pi * self.radius}")
22. def shape_type(shape_obj):
23.     shape_obj.area()
24.     shape_obj.perimeter()
25. def main():
26.     rectangle_obj = Rectangle(10, 20)
27.     circle_obj = Circle(10)
28.     for each_obj in [rectangle_obj, circle_obj]:
29.         shape_type(each_obj)
30. if __name__ == "__main__":
31.     main()
```

OUTPUT

Area of Rectangle is 200

Perimeter of Rectangle is 60

Area of Circle is 314.1592653589793

Perimeter of Circle is 62.83185307179586

In this program, *Shape* ②–⑥ is the base class while *Rectangle* ⑦–⑭ and *Circle* ⑮–⑳ are the derived classes. All of these classes have common methods *area()* and *perimeter()* added to them but their implementation is different as found in each class. Derived classes *Rectangle* and *Circle* have their own data attributes. Instance variables *rectangle_obj* ㉔ and *circle_obj* ㉕ are created for *Rectangle* and *Circle* classes respectively. The clearest way to express polymorphism is through the function *shape_type()* ㉔–㉕, which takes any object and invokes the methods *area()* and *perimeter()* respectively.

11.9.1 Operator Overloading and Magic Methods

Operator Overloading is a specific case of polymorphism, where different operators have different implementations depending on their arguments. A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names called “Magic Methods” (TABLE 11.1). This is Python’s approach to operator overloading, allowing

TABLE 11.1

Magic Methods for Different Operators and Functions

Binary Operators		
Operator	Method	Description
+	<code>__add__(self, other)</code>	Invoked for Addition Operations
-	<code>__sub__(self, other)</code>	Invoked for Subtraction Operations
*	<code>__mul__(self, other)</code>	Invoked for Multiplication Operations
/	<code>__truediv__(self, other)</code>	Invoked for Division Operations
//	<code>__floordiv__(self, other)</code>	Invoked for Floor Division Operations
%	<code>__mod__(self, other)</code>	Invoked for Modulus Operations
**	<code>__pow__(self, other[, modulo])</code>	Invoked for Power Operations
<<	<code>__lshift__(self, other)</code>	Invoked for Left-Shift Operations
>>	<code>__rshift__(self, other)</code>	Invoked for Right-Shift Operations
&	<code>__and__(self, other)</code>	Invoked for Binary AND Operations
^	<code>__xor__(self, other)</code>	Invoked for Binary Exclusive-OR Operations
	<code>__or__(self, other)</code>	Invoked for Binary OR Operations
Extended Operators		
Operator	Method	Description
+=	<code>__iadd__(self, other)</code>	Invoked for Addition Assignment Operations
-=	<code>__isub__(self, other)</code>	Invoked for Subtraction Assignment Operations
*=	<code>__imul__(self, other)</code>	Invoked for Multiplication Assignment Operations
/=	<code>__idiv__(self, other)</code>	Invoked for Division Assignment Operations
//=	<code>__ifloordiv__(self, other)</code>	Invoked for Floor Division Assignment Operations
%=	<code>__imod__(self, other)</code>	Invoked for Modulus Assignment Operations
**=	<code>__ipow__(self, other[, modulo])</code>	Invoked for Power Assignment Operations
<<=	<code>__ilshift__(self, other)</code>	Invoked for Left-Shift Assignment Operations
>>=	<code>__irshift__(self, other)</code>	Invoked for Right-Shift Assignment Operations
&=	<code>__iand__(self, other)</code>	Invoked for Binary AND Assignment Operations

(Continued)

TABLE 11.1 (Continued)

Magic Methods for Different Operators and Functions

<code>^=</code>	<code>__ixor__(self, other)</code>	Invoked for Binary Exclusive-OR Assignment Operations
<code> =</code>	<code>__ior__(self, other)</code>	Invoked for Binary OR Assignment Operations

Unary Operators		
Operator	Method	Description
<code>-</code>	<code>__neg__(self)</code>	Invoked for Unary Negation Operator
<code>+</code>	<code>__pos__(self)</code>	Invoked for Unary Plus Operator
<code>abs()</code>	<code>__abs__()</code>	Invoked for built-in function <code>abs()</code> . Returns absolute value
<code>~</code>	<code>__invert__(self)</code>	Invoked for Unary Invert Operator

Conversion Operations		
Functions	Method	Description
<code>complex()</code>	<code>__complex__(self)</code>	Invoked for built-in <code>complex()</code> function
<code>int()</code>	<code>__int__(self)</code>	Invoked for built-in <code>int()</code> function
<code>long()</code>	<code>__long__(self)</code>	Invoked for built-in <code>long()</code> function
<code>float()</code>	<code>__float__(self)</code>	Invoked for built-in <code>float()</code> function
<code>oct()</code>	<code>__oct__()</code>	Invoked for built-in <code>oct()</code> function
<code>hex()</code>	<code>__hex__()</code>	Invoked for built-in <code>hex()</code> function

Comparison Operators		
Operator	Method	Description
<code><</code>	<code>__lt__(self, other)</code>	Invoked for Less-Than Operations
<code><=</code>	<code>__le__(self, other)</code>	Invoked for Less-Than or Equal-To Operations
<code>==</code>	<code>__eq__(self, other)</code>	Invoked for Equality Operations
<code>!=</code>	<code>__ne__(self, other)</code>	Invoked for Inequality Operations
<code>>=</code>	<code>__ge__(self, other)</code>	Invoked for Greater Than or Equal-To Operations
<code>></code>	<code>__gt__(self, other)</code>	Invoked for Greater Than Operations

classes to define their own behavior with respect to language operators. Python uses the word “Magic Methods” because these are special methods that you can define to add magic to your program. These magic methods start with double underscores and end with double underscores. One of the biggest advantages of using Python’s magic methods is that they provide a simple way to make objects behave like built-in types. That means you can avoid ugly, counter-intuitive, and nonstandard ways of using basic operators. The basic rule of operator overloading in Python is, *Whenever the meaning of an operator is not obviously clear and undisputed, it should not be overloaded and always stick to the operator’s well-known semantics.* You cannot create new operators and you can’t change the meaning of operators for built-in types in Python programming language. Consider the standard `+` (plus) operator. When this operator is used with operands of different standard types, it will have a different meaning. The `+` operator performs arithmetic addition of two numbers, merges two lists, and concatenates two strings.

Program 11.26: Write Python Program to Create a Class Called as Complex and Implement `__add__()` Method to Add Two Complex Numbers. Display the Result by Overloading the `+` Operator

```

1. class Complex:
2.     def __init__(self, real, imaginary):
3.         self.real = real
4.         self.imaginary = imaginary
5.     def __add__(self, other):
6.         return Complex(self.real + other.real, self.imaginary + other.imaginary)
7.     def __str__(self):
8.         return f"{self.real} + i{self.imaginary}"
9. def main():
10.    complex_number_1 = Complex(4, 5)
11.    complex_number_2 = Complex(2, 3)
12.    complex_number_sum = complex_number_1 + complex_number_2
13.    print(f"Addition of two complex numbers {complex_number_1} and {complex_
        number_2} is {complex_number_sum}")
14. if __name__ == "__main__":
15.    main()

```

OUTPUT

Addition of two complex numbers 4 + i5 and 2 + i3 is 6 + i8

Consider the below code having *Complex* class with *real* and *imaginary* as data attributes ①–④.

```

class Complex:
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

```

Instance variables *complex_number_1* and *complex_number_2* are created for *Complex* class ⑩–⑫.

```

complex_number_1 = Complex(4, 5)
complex_number_2 = Complex(2, 3)
complex_number_sum = complex_number_1 + complex_number_2

```

If you try to add the objects *complex_number_1* and *complex_number_2* then it results in error as shown below.

TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'

Adding the magic method `__add__()` ⑤–⑥ within the *Complex* class resolves this issue. The `__add__()` method takes two objects as arguments with *complex_number_1* object assigned to *self* and *complex_number_2* object assigned to *other* and it is expected to return the result of the computation. When the expression *complex_number_1* + *complex_number_2* is executed, Python will call *complex_number_1.__add__(complex_number_2)*.

```
def __add__(self, other):
    return Complex(self.real + other.real, self.imaginary + other.imaginary)
```

Thus, by adding this method, suddenly magic has happened and the error, which you received earlier, has gone away. This method returns the *Complex* object itself by calling the *Complex* class `__init__()` constructor, with *self.real* + *other.real* value assigned to *real* data attribute and *self.imaginary* + *other.imaginary* value assigned to the *imaginary* data attribute. The `__add__()` method definition has your own implementation. This returning object is assigned to *complex_number_sum* with data attributes *real* and *imaginary*. To print the data attributes associated with *complex_number_sum* object, you have to issue the statements `print(complex_number_sum.real)` and `print(complex_number_sum.imaginary)`.

Instead of issuing the above statements, you can use the object name only within the `print` statement, for example, `print(complex_number_sum)` to print its associated data attributes. This is done by overriding `__str__()` magic method ⑦–⑧. The syntax is `__str__(self)`. The `__str__()` method is called by `str(object)` and the built-in functions `format()` and `print()` to compute the “informal,” or nicely printable string representation of an object. The return value must be a string object. The implementation details of `__str__()` magic method is shown below

```
def __str__(self):
    return f"{self.real} + i{self.imaginary}"
```

The return value of `__str__()` method has to be a string, but it can be any string, including one that contains the string representation of integers. In the implementation of the `__str__()` magic method, you have customized it for your own purpose. The `__str__()` method returns a string with values of *real* and *imaginary* data attributes concatenated together, and the character *i* is prefixed before *imaginary* data attribute value.

Program 11.27: Consider a Rectangle Class and Create Two Rectangle Objects. This Program Should Check Whether the Area of the First Rectangle is Greater than Second by Overloading > Operator

1. class Rectangle:
2. def __init__(self, width, height):
3. self.width = width
4. self.height = height
5. def __gt__(self, other):
6. rectangle_1_area = self.width * self.height

```
7.     rectangle_2_area = other.width * other.height
8.     return rectangle_1_area > rectangle_2_area

9. def main():
10.    rectangle_1_obj = Rectangle(5, 10)
11.    rectangle_2_obj = Rectangle(3, 4)
12.    if rectangle_1_obj > rectangle_2_obj:
13.        print("Rectangle 1 is greater than Rectangle 2")
14.    else:
15.        print("Rectangle 2 is greater than Rectangle 1")
16. if __name__ == "__main__":
17.    main()
```

OUTPUT

Rectangle 1 is greater than Rectangle 2

In the above code, *rectangle_1_obj* ⑩ and *rectangle_2_obj* ⑪ are the objects of Rectangle class ①–⑧. When the expression *if rectangle_1_obj > rectangle_2_obj* ⑫ is executed, the magic method *rectangle_1_obj.__gt__(rectangle_2_obj)* gets invoked. This magic method calculates the area of two rectangles and returns a Boolean *True* value if the area of the first rectangle is greater than the area of second rectangle ⑤–⑧.

11.10 Summary

- Objects are used to model real-world entities that we want to represent inside our programs and an object is an instance of a class.
- A class is a blueprint from which individual objects are created. An object is a bundle of related variables and methods.
- The act of creating an object from a class is called instantiation.
- The `__init__()` method is automatically called and executed when an object of the class is created.
- Class attributes are shared by all the objects of a class.
- An identifier prefixed with a double underscore and with no trailing underscores should be treated as private within the same class.
- Encapsulation is the process of combining variables that store data and methods that work on those variables into a single unit called class.
- Inheritance enables new classes to receive or inherit variables and methods of existing classes and helps to reuse code.
- Inheritances can be Single inheritance or Multiple inheritance.

- Poly means *many* and morphism means *forms*. Polymorphism means that you can have multiple classes where each class implements the same variables or methods in different ways.
- Operator overloading is a specific case of polymorphism, where an operator can have different meaning when used with operands of different types.

Multiple Choice Questions

1. The distinctly identifiable entity in the real world is called as _____
 - a. An object
 - b. A class
 - c. Data attribute
 - d. Method attribute
2. A blueprint that defines the objects of the same type is called as _____
 - a. An object
 - b. A class
 - c. function
 - d. constructor
3. The beginning of the class definition is marked by the keyword _____
 - a. def
 - b. return
 - c. class
 - d. None of the above
4. What is Polymorphism?
 - a. You can have multiple classes where each class implements the same variables or methods in different ways
 - b. Ability of a class to derive members of another as a part of its own definition
 - c. Focuses on variables and passing of variables to functions
 - d. Encapsulating variables and methods to certain classes
5. The correct way of inheriting a derived class from the base class is
 - a. class (Base) Derived:
 - b. class Derived (Base):
 - c. class (Base) Derived:
 - d. class Base (Derived):
6. Identify the function that checks for class inheritance.
 - a. issubclass()
 - b. isobject()

- c. `issuperclass()`
 - d. `isinstance()`
7. Duck-typing in Python is
- a. Makes the program code smaller
 - b. More restriction on the type values that can be passed to a given method.
 - c. No restriction on the type values that can be passed to a given method.
 - d. An object's suitability is determined by the presence of methods and variables rather than the actual type of the object.
8. In Python single inheritance can be defined as
- a. A single class inherits from multiple classes.
 - b. A multiple base class inherits from a single derived class.
 - c. A subclass derives from a class which in turn derives from another class.
 - d. A single subclass derives from a single super class.
9. Which of the following are the fundamental features of OOP?
- a. Inheritance
 - b. Encapsulation
 - c. Polymorphism
 - d. All of the above
10. The `+` operator is overloaded using the method
- a. `__add__()`
 - b. `__plus__()`
 - c. `__sum__()`
 - d. `__total__()`
11. The operator overloaded by `__invert__()` method is
- a. `!`
 - b. `~`
 - c. `^`
 - d. `-`
12. The syntax for using `super()` in derived class `__init__()` method definition looks like
- a. `super().__init__(baseclassparameters)`
 - b. `init__.super()`
 - c. `super().__init__(derivedclassparameters)`
 - d. `super()`
13. MRO stands for
- a. Member Resolution Order
 - b. Member Reverse Order
 - c. Member Resolution Office
 - d. Method Resolute Order

14. Diamond problem in Python is
 - a. It is term used for overloading
 - b. It is term used for an ambiguity that arises when multiple classes of same level are inherited
 - c. It is a term used for polymorphism
 - d. There is no such problem
15. The syntax that is used to get information about Method Resolution Order is
 - a. `mro().class`
 - b. `mro().tuple`
 - c. `<class>.mro()`
 - d. `<class>.diamond()`
16. The function of instantiation is
 - a. Modifying an instance of a class
 - b. Copying an instance of a class
 - c. Deleting an instance of a class
 - d. Creating an instance of a class
17. Identify the type of inheritance that is illustrated in this piece of code?

```
class A()
    pass
class B()
    pass
class C(A,B)
    pass
```

 - a. Single inheritance
 - b. Multilevel inheritance
 - c. Multiple inheritance
 - d. Hierarchical inheritance

Review Questions

1. Explain classes and objects with examples.
2. Describe the need for `__init__()` constructor method.
3. Differentiate between class attributes and data attributes.
4. Briefly explain encapsulation with an example.
5. Examine the different types of inheritances with an example.
6. Demonstrate the use of `super()` function with an example.
7. Discuss polymorphism with an example.

8. Illustrate operator overloading with an example.
9. Create a class named quadratic, where a, b, c are data attributes and the methods are
 - a. `__init__()` to initialize the data attributes
 - b. `roots()` to compute the quadratic equation
10. Define a class called student. Display the marks details of top five students using inheritance.
11. Create a class called library with data attributes like `acc_number`, `publisher`, `title` and `author`. The methods of the class should include
 - a. `read()` – `acc_number`, `title`, `author`.
 - b. `compute()` - to accept the number of days late, calculate and display the fine charged at the rate of \$1.50 per day.
 - c. `display` the data.
12. Create two base classes named `clock` and `calendar`. Based on these two class define a class `calendarclock`, which inherits from both the classes which displays month details, date and time.
13. Write a program to add two polynomials using classes.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

12

Introduction to Data Science

AIM

Realize the power of modules like NumPy, pandas, and Altair in developing solutions to problems related to data science.

LEARNING OUTCOMES

At the end of the chapter, you are expected to

- Understand functional programming.
- Understand serialization and deserialization of JSON objects.
- Demonstrate the application of Numpy and pandas Modules.
- Generate charts using Altair visualization library.

Data Science is currently generating tremendous fascination worldwide. A topic that will strongly influence our everyday life in the next years is data science. The growth of data in the present world has drastically increased, where tons of data comes from a variety of sources, in very large amounts, and often in real-time settings. Due to this enormous growth of data, the value of data has become an important factor in every aspect. The term data science covers the study of raw data to gain insights into data through computation, statistics, and visualization. Data science is a rewarding career that allows you to solve some of the world's most interesting problems. A data scientist can be thought of someone who knows more about statistics than a computer scientist and more computer science than a statistician. Many companies are seeking data scientists who have the skills necessary to analyze and generate business intelligence from their various data sources. This chapter introduces you to various necessary tools that are required to build a successful career in data science.

12.1 Functional Programming

Python supports a form of programming called Functional Programming (FP) that involves programming with functions where functions can be passed, stored, and returned. FP decomposes a problem into a set of functions. The gist of FP is that every function is understood solely in terms of its inputs and its outputs.

12.1.1 Lambda

Small anonymous functions can be created with the *lambda* keyword. Lambda functions are created without using *def* keyword and without a function name. They are syntactically

restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. The syntax for lambda function is,

 ***lambda*** *argument_list: expression*

Here, *lambda* is a keyword, *argument_list* is a comma separated list of arguments, and *expression* is an arithmetic expression using these arguments lists. A colon separates both *argument_list* and *expression*. No need to enclose *argument_list* within brackets. For example,

```
1. >>> addition_operation = lambda a, b: a + b
2. >>> addition_operation(100, 8)
108
```

In the above code, the *lambda* function takes two arguments *a* and *b* and performs an addition operation using these arguments ①. You can assign a *lambda* function to a variable and use this variable as a function name to pass arguments ②. Note, you are not assigning the value of lambda function to the variable; instead you are giving a function name to a lambda expression. A lambda function returns the result of the expression implicitly, and there is no need to specify a *return* keyword.

12.1.2 Iterators

A Python language feature, iterators, is an important foundation for writing functional-style programs. *Iteration* is a general term for taking each item of something, one after another. Any time you use a loop to go over a group of items, that is an iteration. In Python, iterable and iterator have specific meanings.

An *iterable* is an object that has an `__iter__()` method that returns an iterator. So, an iterable is an object that you can get an iterator from. Lists, dictionaries, tuples, and strings are iterable in Python.

An *iterator* is an object with a `__next__()` method. Whenever you use a *for* loop in Python, the `__next__()` method is called automatically to get each item from the iterator, thus going through the process of iteration. Iterators are stateful, meaning once you have consumed an item from them, it's gone.

You can call the `__next__()` method using the *next()* and `__iter__()` method using *iter()* built-in functions. For example,

```
1. >>> phone = "jio"
2. >>> it_object = iter(phone)
3. >>> type(it_object)
<class 'str_iterator'>
4. >>> next(it_object)
'j'
5. >>> next(it_object)
'i'
6. >>> next(it_object)
'o'
```

```
7. >>> next(it_object)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
StopIteration
```

An *iterator* is an object representing a stream of data ③; this object returns the data one element at a time. A Python iterator must support a method called `__next__()` that takes no arguments and always returns the next element of the stream ④–⑥. If there are no more elements in the stream, `__next__()` must raise the *StopIteration* exception ⑦. The built-in *iter()* function takes an arbitrary object and tries to return an iterator that will return the object's contents or elements ②, else raises *TypeError* exception if the object does not support iteration.



Containers are the objects that hold data elements. Containers are iterables. Lists, sets, dictionary, tuple, and strings are all containers.

12.1.3 Generators

Generators are a special class of functions that simplify the task of writing iterators. Regular functions compute a value and return it, but generators return an iterator that returns a stream of values.

When you call a function, its local variables have their own scope. After the *return* statement is executed in a function, the local variables are destroyed and the value is returned to the caller. Later, a call to the same function creates a fresh set of local variables that have their own scope. However, what if the local variables were not thrown away on exiting a function? What if you could later resume the function from where it left off? This is what generators provide; they can be thought of as resumable functions. A generator function does not include a *return* statement. Here's the simplest example of a generator function,

```
1. >>> def generate_ints(N):
2. ...     for i in range(N):
3. ...         yield i
4. >>> gen = generate_ints(3)
5. >>> gen
<generator object generate_ints at 0x00000160E4D26410>
6. >>> next(gen)
0
7. >>> next(gen)
1
8. >>> next(gen)
2
```

9. >>> next(gen)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

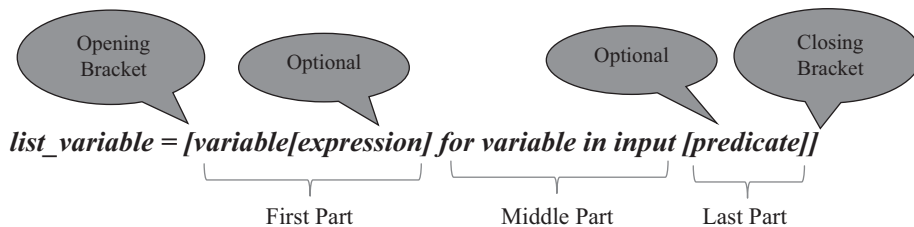
StopIteration

Any function containing a *yield* keyword is a generator function ①. When you call a generator function, it does not return a single value. Instead it returns a generator object that supports the iterator `__next__()` method ④–⑤. Inside the *for* loop ② on executing the *yield* expression ③, the generator outputs the value of *i*, similar to a *return* statement. The big difference between *yield* and a *return* statement is that on reaching a *yield* the generator's state of execution is temporarily suspended and local variables are preserved. On the next call to the generator's `__next__()` method, the function will resume execution from where it left off ⑥–⑨.

In the real world, generator functions are used for calculating large sets of results where you do not know if you are going to need all results.

12.1.4 List Comprehensions

List comprehensions provide a concise way to create lists. Common applications of list comprehensions are to make new lists where each element is the result of some operation applied to each member of another sequence or iterable or to create a subsequence of those elements that satisfy a certain condition.






A list comprehension consists of brackets containing a *variable* or *expression* (First Part) followed by a *for* clause (Middle Part), then predicate *True* or *False* using an *if* clause (Last Part). The components *expression* and *predicate* are optional. The new list resulting from evaluating the *expression* in the context of the *for* and *if* clauses that follow it will be assigned to the *list_variable*. The *variable* represents members of *input*. The order of execution in a list comprehension is (a) If the *if* condition is not specified, then Middle Part and First Part gets executed; (b) If the *if* condition is specified, then the Middle Part, Last Part, and First Part gets executed. For example,


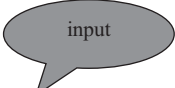
```
1. >>> hardy_ramanujan = []
2. >>> for number in '1729':
3. ...     hardy_ramanujan.append(number)
4. >>> hardy_ramanujan
['1', '7', '2', '9']
```

In the above code, an empty list *hardy_ramanujan* is created ①. Then, you loop through each character in the '1729' string using the *number* iteration variable ②. Each of those characters


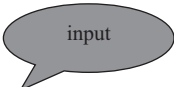
is appended into *hardy_ramanujan* list ③. Finally, print the list ④. For the above code, you can have more readable and concise code through list comprehensions.

1. `>>> hardy_ramanujan = [number for number in '1729']` 
2. `>>> hardy_ramanujan` 
`['1', '7', '2', '9']` 


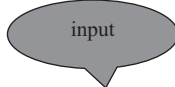
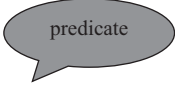



Simple *for* loops can be written as comprehensions offering cleaner and more readable syntax. Comprehensions can be thought of as a compact form of a *for* loop. In the list comprehension, the variable *number* indicates the item that will be inserted into the list *hardy_ramanujan* at each step of the *for* loop. In the *for* loop, the iterating variable *number* iterates through each character of the string '1729' ①. The resulting list is assigned to the *hardy_ramanujan* list variable ① and displayed ②.

1. `>>> display_upper_case = [each_char.upper() for each_char in "farrago"]` 
2. `>>> display_upper_case` 
`['F', 'A', 'R', 'R', 'A', 'G', 'O']`


In the above code, the iterating variable *each_char* iterates through each character of the string "farrago." While iterating through each character using an *each_char* iterating variable, each of those characters is converted to upper case using the *upper()* method and inserted into the *display_upper_case* list ①. Print the items of the *display_upper_case* list ②.

1. `>>> squares = [x**2 for x in range(1, 10)]` 
2. `>>> squares` 
`[1, 4, 9, 16, 25, 36, 49, 64, 81]`

In the above code, numbers from 1 to 9 are generated using the *range()* function. The iterating variable *x* iterates through 1 to 9 and at each iteration, the square of the number is found and assigned to *squares* list ①. Print the items of the *square* list ②.

1. `>>> even_square = [x**2 for x in range(1, 10) if x % 2 == 0]`   
2. `>>> even_square`
`[4, 16, 36, 64]`   

In the above code, numbers from 1 to 9 are generated using the *range()* function. Use a *for* loop to iterate through each number using the iterating variable *x*. While iterating through each number, the *if* condition checks whether the number is even or not using a modulus operator. If the number is even, then that number is squared and inserted into the *even_square* list ①. Print the items of *even_square* list ②.



```

1. >>> words = [each_word for each_word in input().split()]
    petrichor degust jirble flabbergast foppish
2. >>> words.sort()
3. >>> print(" ".join(words))
    degust flabbergast foppish jirble petrichor
  
```

In the above code, the `input()` function requires the user to enter words as input separated by a space. Use the `split()` function on these entered words to get a list of string items. Use a `for` loop to iterate through each of these list items using an `each_word` iterating variable. Insert each of the string items to `words` list ①. Then, sort the `words` list using the `sort()` method in ascending order according to their ASCII values ②. Join the string items in words list using `join()` method and print it ③.



List Comprehensions are an alternate to `filter()`, `reduce()`, and `map()` methods. Guido, the Python BDFL wanted `filter()`, `reduce()`, and `map()` methods removed from the language but, due to severe backlash from the community, these methods were retained. Among these methods `reduce()` was removed from the Python 3.x built-in standard library and was moved to `functools`. List Comprehensions are preferred over the `filter()`, `reduce()`, and `map()` methods as list comprehensions are more Pythonic.

12.2 JSON and XML in Python

JSON (JavaScript Object Notation) and XML (EXtensible Markup Language) standards are commonly used for transmitting data in web applications. The Web is based on a very basic client/server architecture that can be summarized as follows: a client (usually a web browser) sends a request to a server, using the Hypertext Transfer Protocol (HTTP). The server answers the request using the same protocol (FIGURE 12.1).

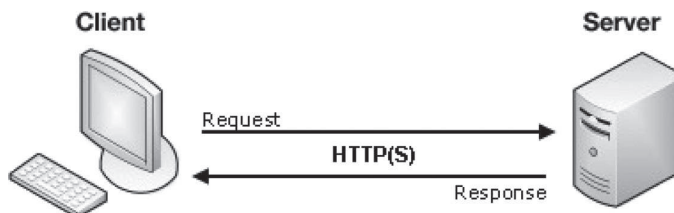


FIGURE 12.1
Client/Server Architecture.

At the most basic level, whenever a browser needs a file, which is hosted on a web server, the browser requests the file via HTTP. When the request reaches the correct web server

(hardware), the HTTP server (software) accepts the request, finds the requested document (if it does not then a 404 response is returned), and sends it back to the browser, also through HTTP.

12.2.1 Using JSON with Python

JSON (JavaScript Object Notation) is a lightweight text-based data-interchange format, which was popularized by Douglas Crockford. It is simple for humans to read and write and easy enough for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition – December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of various languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others and all of these programming languages feature the ability to read (parse) and generate JSON.

The built-in data types in JSON are strings, numbers, booleans (i.e., true and false), null, objects, and arrays. JSON is built on two structures:

- A collection of *string: value* properties. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

The two above mentioned structures are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also is based on these structures (FIGURE 12.2).

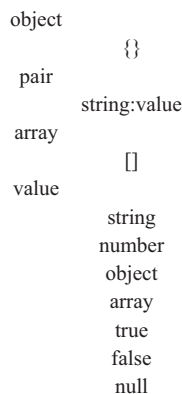


FIGURE 12.2
JSON Structures.

A JSON object can have one or more properties, each of which is a *string: value* pair. An object begins with a left brace ({) and ends with a right brace (}). Each *string* is followed by a colon (:), and the *string: value* pairs are separated by a comma (,). Conventionally, a space is used after the colon. The purpose of this space is to make your code easy for people to read.

In JSON, an array is an ordered collection of values. An array begins with a left bracket ([) and ends with a right bracket (]). Values are separated by a comma (,).

In JSON, a value can be a string in double quotes, a number, true or false or null, an object, or an array. These structures can be nested.

In JSON, it is required to use double quotes around *string* property and single quotes are not valid.

In JSON, a number is very much like a Python number, except that the octal and hexadecimal formats are not used.



Even a single misplaced comma or colon can cause a JSON file to go wrong and not work. You should be careful to validate any data you are attempting to use. Although computer-generated JSON is less likely to include errors, if the generator program is working correctly. You can validate JSON using an application like JSONLint.

You can include the same basic data types inside JSON like strings, numbers, arrays, booleans, and other object literals. This allows you to construct a data hierarchy, like

```

1. { ← Object Starts
2.   "first_name": "Andrew",
3.   "middle_name": "Wood",
4.   "last_name": "Ellis",
5.   "contact": { ← Object Starts
6.       "phone": "1 - 690 - 793 - 4521",
7.       "email": "andrewellis@gmail.com"
8.   }, ← Object Ends
9.   "address": [ ← Array Starts
10.       "address_type": "Office",
11.       "street": "3096 Euclid Avenue",
12.       "city": "Los Angeles", ← Value String
13.       "zip_code": 90017,
14.       "state": "California"
15.   ],
16.   {
17.       "address_type": "Home",
18.       "street": "940 Lewis Street",
19.       "city": "Los Angeles",
20.       "zip_code": 90185, ← Value Number
21.       "state": "California"
22.   }
23. ] ← Array Ends
24. } ← Object Ends

```

In JSON, an object is a list of *string: value* properties separated by commas, with the whole list enclosed in curly brackets. An object begins and ends with curly brackets: {}. The order of *string: value* properties in an object does not matter. A JSON object can have an individual *string: value* properties, each of which is a pairing that includes a string and a value. The *string: value* properties of an object must be separated by commas as in ②–④. The *value* of a property can be an object ⑤–⑦. The *string* property is case sensitive.

In the above example, the string is named *contact*. The value of the *contact* property is an object that consists of an opening curly bracket, two of its own properties (string named *phone* and *email*), and a closing curly bracket. Notice how objects allow you to create a hierarchy of information, in the form of a nested *string: value* properties (object within the object in this case). There must be no comma after the last *string: value* property of an object (as in line ⑦ above). Misuse of commas will break your JSON and make it impossible to parse it. Conventionally, properties of an object are set off using line breaks after the opening curly bracket, after each property, and after the closing curly bracket. Additionally, properties inside an object are indented using either tabs or spaces. The line breaks and indentation make it easy to see which properties are associated to which object.

In the above example, for the string named as *address*, the value is an array of objects separated by commas, with the whole array enclosed in square brackets []. The value of the address string property is an array that consists of an opening square bracket, two values (each of which is an object with five properties of its own), and a closing square bracket. Values in an array must be separated by commas (as in line ⑩ above). A comma cannot be used after the last value in an array (as in line ② above). Conventionally, objects in an array are set off using line breaks after each opening bracket, property, or closing bracket; if a property or closing bracket is followed by a comma, the line break should be after the comma. Additionally, each object is indented within the array, and each object's properties are further indented within that object. The line breaks and indentation make the information hierarchy clear in your JSON.

Within a JSON *string: value* properties, certain characters, known as reserved characters, can be used only if they are preceded by a backslash (\). For example,

```
"text": "The value was \"The Wizard of Oz\""
```

JSON reserved characters are straight quotation marks (") and backslashes (\). To include a straight quotation mark (") or backslash (\) in the string value, escape the reserved character with a preceding backslash (\), as in the example above.

A JSON object can be stored in its own file, which is basically just a text file with a *.json* extension. Alternatively, the JSON object can exist as a string in Python. For example, the above JSON object is stored in a file called *personal_data.json*. Both of these are useful when you want to transmit data across a network. It needs to be converted to a native JavaScript object when you want to access the data. The Python standard module called *json* can take Python data hierarchies and convert them to string representations; this process is called serializing (TABLE 12.1a). The Python *json* module provides methods *dump()* for writing data to JSON file and *dumps()* for writing to a Python string. Reconstructing the data from the string representation is called deserializing (TABLE 12.1b). The Python *json* module provides methods *load()* for turning JSON encoded data into Python objects from a file and *loads()* methods for turning JSON encoded data into Python objects from a string. Between serializing and deserializing, the string representing the object may have been stored in a file or sent over a network connection to some remote machine.

TABLE 12.1

Python Serializing (a) and Deserializing (b) Conversion Table

Python	JSON	JSON	Python
dict	object	object	dict
list, tuple	array	array	list
str	string	string	str
int	number	number (int)	int
float	number	number (float)	float
True	true	true	True
False	false	false	False
None	null	null	None
(a)		(b)	

Program 12.1: Program to Demonstrate Python Deserializing Using JSON load() Method

```

1. import json
2. def main():
3.     with open('personal_data.json', 'r') as f:
4.         json_object_data = json.load(f)
5.         print(f'Type of data returned by json load is {type(json_object_data)}')
6.         print(f'First Name is {json_object_data['first_name']}')
7.         print(f'Middle Name is {json_object_data['middle_name']}')
8.         print(f'Last Name is {json_object_data['last_name']}')
9.         print(f'Phone Number is {json_object_data['contact']['phone']}')
10.        print(f'Email ID is {json_object_data['contact']['email']}')
11.        print("-----*****-----")
12.        for each_json_object in json_object_data['address']:
13.            print(f'Address Type is {each_json_object['address_type']}')
14.            print(f'Street Name is {each_json_object['street']}')
15.            print(f'City Name is {each_json_object['city']}')
16.            print(f'Zip Number is {each_json_object['zip_code']}')
17.            print(f'State Name is {each_json_object['state']}')
18.            print("-----*****-----")
19. if __name__ == "__main__":
20.     main()

```

OUTPUT

```

Type of data returned by json load is <class 'dict'>
First Name is Andrew
Middle Name is Wood

```

Last Name is Ellis
 Phone Number is 1 - 690 - 793 - 4521
 Email ID is andrewellis@gmail.com

-----*****-----
 Address Type is Office
 Street Name is 3096 Euclid Avenue
 City Name is Los Angeles
 Zip Number is 90017
 State Name is California

-----*****-----
 Address Type is Home
 Street Name is 940 Lewis Street
 City Name is Los Angeles
 Zip Number is 90185
 State Name is California

The syntax for *load()* method is,

json.load(fp)

Deserialize *fp* (a *read()* supporting file like object containing a JSON document) to a Python object using the conversion [TABLE 12.1b](#).

Here you import the *json* module in line ①. Open the existing JSON file *personal_data.json* using the *open()* method in read mode where *f* is the file handler, and load that file handler using *load()* method ③–④. The *loads()* method converts a JSON object into Python dictionary and assigns it to the *json_object_data* dictionary. You can display the value associated with a key (JSON string property) by specifying the name of the dictionary *json_object_data* followed by brackets, within which you specify the name of the key (JSON string property) ⑥–⑩. Use a *for* loop to iterate through the array *address* values ⑫–⑰.

Program 12.2: Program to Demonstrate Python Deserializing Using JSON loads() Method

```

1. import json
2. def main():
3.     json_string = ""
4.     {
5.         "title": "Product",
6.         "description": "A product from Patanjali's catalog",
7.         "category": "Ayurvedic",
8.         "item": {
9.             "name": "Aloevera Sun Screen Cream",
10.            "type": "Face Cream"
11.        }
12.    }
```

```

13.     ""
14.     json_object_data = json.loads(json_string)
15.     print(f"Title is {json_object_data['title']}")
16.     print(f"Description is {json_object_data['description']}")
17.     print(f"Category is {json_object_data['category']}")
18.     print(f"Item name is {json_object_data['item']['name']}")
19.     print(f"Item type is {json_object_data['item']['type']}")
20. if __name__ == "__main__":
21.     main()

```

OUTPUT

```

Title is Product
Description is A product from Patanjali's catalog
Category is Ayurvedic
Item name is Aloevera Sun Screen Cream
Item type is Face Cream

```

The syntax for *loads()* method is,

json.loads(s)

Deserialize *s* (a str, bytes, or bytearray instance containing a JSON document) to a Python object using the conversion [TABLE 12.1b](#).

You can load a JSON formatted string ③–⑩ to *loads()* method that returns a dictionary type ⑭. Display the values by using the dictionary name with an appropriate key (JSON string property) ⑮–⑰.

Program 12.3: Program to Demonstrate Python Serializing Using JSON dump() and dumps() Methods

```

1. import json
2. def main():
3.     string_data =[{
4.         "Name": "Debian",
5.         "Owner": "SPI"
6.     },
7.     {
8.         "Name": "Ubuntu",
9.         "Owner": "Canonical"
10.    },
11.    {
12.        "Name": "Fedora",
13.        "Owner": "Red Hat"

```

```

14.     }]
15.     json_data = json.dumps(string_data)
16.     print("Data in JSON format")
17.     print(json_data)
18.     with open('linux_data.json', 'w') as f:
19.         json.dump(json_data, f)
20. if __name__ == "__main__":
21.     main()

```

OUTPUT

Data in JSON format

```
[{"Name": "Debian", "Owner": "SPI"}, {"Name": "Ubuntu", "Owner": "Canonical"}, {"Name": "Fedora", "Owner": "Red Hat"}]
```

The syntax for *dumps()* method is,

json.dumps(obj)

Serialize *obj* to a JSON formatted Python str using the conversion [TABLE 12.1a](#).

The syntax for *dump()* method is,

json.dump(obj, fp)

Serialize *obj* as a JSON formatted stream to *fp* (a *write()* supporting file like object) using the conversion [TABLE 12.1a](#).

You can specify the JSON formatted data object in your program ③–⑭ and use the *dumps()* method to write a data object to a Python string ⑮. A file called *linux_data.json* is created and is opened in write mode. You have to specify two arguments in the *dump()* method, one is the data object to be serialized and the other argument is the file handler object to which data will be written. The *dump()* method writes data to files ⑯–⑲.

12.2.2 Using Requests Module

Requests is an elegant and simple HTTP library for Python, built for human beings. The goal of the project is to make HTTP requests simpler and more human-friendly. Requests make integrating your code with web services seamless.

Installing *requests* is simple with pip:

```
1. C:\> pip install requests
```

Installs *requests* library ①.

Program 12.4: Program to Get Text Response Content Using requests Module

```

1. import requests
2. def main():
3.     response_object = requests.get('https://www.gutenberg.org/cache/epub/419/pg419.txt')

```

```

4. print("Text Contents")
5. print(response_object.text)
6. if __name__ == "__main__":
7.     main()

```

OUTPUT

Text Contents

The Project Gutenberg EBook of The Magic of Oz, by L. Frank Baum



Truncated Output

Making a request with *requests* is very simple. Begin by importing the *requests* module ①. Now, let's try to get a webpage using *get()* method ③. For this example, let's get an ebook from the Project Gutenberg digital library website. The *get()* method requests data from the server. Now, we have a response object called *response_object*. We can get all the information we need from this object. We can read the content of the server's response. Here, *requests* will automatically decode content from the server. Most Unicode character sets are seamlessly decoded. When you make a request, the *requests* makes educated guesses about the encoding of the response based on the HTTP headers. The text encoding guessed by *requests* is used when you access *response_object.text* ⑤. Encoding is the process of translating data between two formats according to a set of rules or a formula required for a number of information processing needs like data transmission. For example, you can encode "abc" to "ABC" using lowercase-to-uppercase rules. Decoding is the inverse process. You can decode "ABC" to "abc" using the same set of rules.

Program 12.5: Program to Get JSON Response Content Using Requests Module

```

1. import requests
2. def main():
3.     r = requests.get('http://date.jsontest.com/')
4.     date_dict = r.json()
5.     print(f"Current Time is {date_dict['time']}")
6.     print(f"Milliseconds Since Epoch is {date_dict['milliseconds_since_epoch']}")
7.     print(f"Today's Date is {date_dict['date']}")
8. if __name__ == "__main__":
9.     main()

```

OUTPUT

Current Time is 05:47:22 PM

Milliseconds Since Epoch is 1525628842530

Today's Date is 05-06-2018

There's also a built-in JSON decoder to deal with JSON data. In case the JSON decoding fails, *r.json()* raises an exception. The URL passed to the *get()* method of *requests* module returns three JSON *string:value* properties which is decoded ③ and assigned to *date_dict* dictionary ④. The values associated with each key (JSON string property) are printed ⑤–⑦.

12.2.3 Using XML with Python

EXtensible Markup Language (XML) document is a simple and flexible text format that is used to exchange wide variety of data on the Web and elsewhere. An XML document is a universal format for data on the Web. XML allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way. XML documents have an *.xml* extension.

Why Use XML?

Developers use XML format for following reasons:

Reuse – Contents are separated from Presentation, which enables rapid creation of documents and content reuse.

Portability – XML is an international, platform-independent standard based on ASCII text, so developers can safely store their documents in XML without being tied to any one vendor.

Interchange – XML is a core data standard that enables XML-aware applications to interoperate and share data seamlessly.

Self-describing – XML is in a human-readable format that users can easily view and understand.

Elements form the backbone of XML documents, creating structures which you can manipulate with programs. *Elements* identify sections of content and are built using tags that identify the name, start, and end of the element. All elements must have names. Element names are case-sensitive and must start with a letter or underscore. An element name can contain letters, digits, hyphens, underscores, and periods. An element generally includes the start and end tags, and everything in between.

Elements provide a way of indicating to which element each sections of content belongs to in XML and this is done by means of *tags*. Tags establish boundaries around the content. A tag consists of the element name between the left-angle bracket (<) and the right-angle bracket (>). A tag is used to identify where a particular element starts, and where the element ends. For an element called *element_name*, the start tag will normally look like <element_name>. The corresponding closing tag for this element is </element_name>. There are no predefined tags in XML language. The tags in an XML document are not part of any XML standard. These tags are thought about by the developer who authors the XML document.

Elements can also contain *attributes*, which have a name and a value and are used to provide additional information about your content. An element's attributes are written inside the start tag for that element and take the form *attribute_name*="attribute_value". Attribute values in XML must be enclosed in either single or double quotes. Double quotes are traditional. Single quotes are useful when the attribute value contains double quotes.

You must follow these syntax rules when you create an XML document:

1. All XML elements must have a closing tag.

It is illegal to omit the closing tag when you are creating XML syntax. XML elements must have a closing tag.

Incorrect:

<movie>Maze Runner.

Correct:

<movie>Maze Runner. </movie>

2. XML tags are case sensitive

When you create XML documents, the tag `<Google>` is different from the tag `<google>`.

Incorrect:

```
<Google>An Alphabet Company. </google>
```

Correct:

```
<google>An Alphabet Company. </google>
```

3. All XML elements must be properly nested.

Improper nesting of tags makes no sense to XML. Here `<country>` and `<state>` are sibling elements.

Incorrect:

```
<country><state>Alaska is the biggest state in USA </country></state>
```

Correct:

```
<country><state>Alaska is the biggest state in USA </state></country>
```

4. All XML documents must have a root element.

All XML documents must contain a single tag pair to define a root element. All other elements must be within this root element. Family metaphors, such as a parent, child and sibling, are used to describe relationships between elements relative to each other. All elements can have sub-elements (child elements). Sub-elements must be correctly nested within their parent element. For example,

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

5. Attribute values must always be quoted.

Omitting quotation marks around attribute values is illegal. The attribute value must always be quoted.

Incorrect:

```
<thor realm=Asgard> God of Thunder </thor>
```

Correct:

```
<thor realm="Asgard"> God of Thunder </thor>
```

6. Writing Comments in XML

Use the following syntax for writing comments in XML:

```
<!-- This is a comment -->
```

Program 12.6: Construct an XML Formatted Data and Write Python Program to Parse that XML Data

```
1. import xml.etree.ElementTree as ET
2. def main():
3.     university_data = ""
4.     <top_universities>
5.         <year_2018>
6.             <university_name location="USA">MIT</university_name>
```

```

7.      <ranking>First</ranking>
8.      </year_2018>
9.      <year_2018>
10.     <university_name location="UK">Oxford</university_name>
11.     <ranking>Sixth</ranking>
12.     </year_2018>
13.     <year_2018>
14.     <university_name location="Singapore">NTU</university_name>
15.     <ranking>Eleventh</ranking>
16.     </year_2018>
17. </top_universities>
18. ""
19. root = ET.fromstring(university_data)
20. for ranking_year in root.findall('year_2018'):
21.     university_name = ranking_year.find('university_name').text
22.     ranking = ranking_year.find('ranking').text
23.     location = ranking_year.find('university_name').get('location')
24.     print(f'{university_name} University has secured {ranking} Worldwide
        ranking and is located in {location}')
25. if __name__ == "__main__":
26.     main()

```

OUTPUT

MIT University has secured First Worldwide ranking and is located in USA
Oxford University has secured Sixth Worldwide ranking and is located in UK
NTU University has secured Eleventh Worldwide ranking and is located in Singapore

The *xml.etree.ElementTree* (ET in short) ① module implements a simple and efficient way for parsing and creating XML data ③–⑯. XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. You need to obtain the root element to easily traverse around a tree.

You can directly read XML data from a string as ⑱,

```
root = ET.fromstring(university_data)
```

The *fromstring()* method parses XML data from a string directly into an element, which is the root element of the parsed tree.

You can also import XML data by reading it from a file. For example, if the XML data is stored in a file named *university_data.xml*, then to read the XML file replace the code in line ⑱ with the below two lines.

```
tree = ET.parse('university_data.xml')
root = tree.getroot()
```

ET has *ElementTree* class to represent the whole XML document as a tree. *Element.findall()* finds only elements with a tag, which are direct children of the current element and returns all the child elements as a list ②. *Element.find()* finds the first child element with a particular tag ②–②. The *Element.text* accesses the element's text content ②–②. *Element.get()* accesses the element's attributes ③.

Program 12.7: Write Python Program to Generate XML Formatted Data and Save it as an XML Document

```

1. import xml.etree.ElementTree as ET
2. def main():
3.     root = ET.Element("catalog")
4.     child = ET.SubElement(root, "book", {"id":"bk101"})
5.     subchild_1 = ET.SubElement(child, "author")
6.     subchild_2 = ET.SubElement(child, "title")
7.     subchild_1.text = "Michael Connelly"
8.     subchild_2.text = "City of Bones"
9.     child = ET.SubElement(root, "book", {"id":"bk102"})
10.    subchild_1 = ET.SubElement(child, "author")
11.    subchild_2 = ET.SubElement(child, "title")
12.    subchild_1.text = "Jeffrey Friedl"
13.    subchild_2.text = "Mastering Regular Expressions"
14.    tree = ET.ElementTree(root)
15.    tree.write("books.xml")
16. if __name__ == "__main__":
17.     main()


```

OUTPUT

```

<?xml version='1.0'>
<catalog>
  <book id="bk101">
    <author>Michael Connelly</author>
    <title>City of Bones</title>
  </book>
  <book id="bk102">
    <author>Jeffrey Friedl</author>
    <title>Mastering Regular Expressions</title>
  </book>
</catalog>

```



books.xml

To construct an XML document, first, you need to create an element that acts as the *root* element. After the root element is created, then you can create its sub-elements using the *SubElement()* method. *Element* class represents a single element in a tree. Interactions with a single XML element and its sub-elements are done on the *Element* level. In ①, you get the *root* element ③. The *SubElement()* method provides a convenient way to create child and subchild elements for a given element. The child element is *book*, and subchild elements are *author* and *title* ④–⑥ and ⑨–⑪. Text can be added to an *Element* object using *Element.text* ⑦–⑧ and ⑫–⑬. In the code, *subchild_1* and *subchild_2* are the *Element* objects and text is added to each of these elements using *text* attribute. The *ElementTree* provides a simple way to build XML documents and write them to files ⑭. The *ElementTree.write()* method serves this purpose ⑮.

12.2.4 JSON versus XML

Since both JSON and XML are widely used as data interchange formats, we will try to draw a comparison between them.

- XML is more expressive than JSON. However, XML suffers from the frequent use of tags, whereas JSON is much more compact.
- XML is more complex than JSON.
- Both JSON and XML can be used with most of the programming languages. But the way the programming languages handle these two format is different. When you are working with XML, its data format does not directly translate to a programming language data structure, thus forcing to work with two systems whose data structures are different. The objects and arrays used in JSON are inherently compatible with most of the programming languages' data structures, which eases the use of JSON format in a programming language.
- XML has XSLT (Extensible Stylesheet Language Transformations) specification, which may be used to apply a style to an XML document. JSON does not have any such thing.

12.3 NumPy with Python

NumPy is the fundamental package for scientific computing with Python. It stands for "Numerical Python." It supports:

- N-dimensional array object
- Broadcasting functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as a multi-dimensional container to store generic data. Arbitrary data types can also be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy's main object is the homogeneous multidimensional array. An array is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers and represented by a single variable. NumPy's array class is called *ndarray*. It is also known by the alias *array*.

In NumPy arrays, the individual data items are called *elements*. All elements of an array should be of the same type. Arrays can be made up of any number of dimensions. In NumPy, dimensions are called axes. Each dimension of an array has a length which is the total number of elements in that direction. The size of an array is the total number of elements contained in an array in all the dimension. The size of NumPy arrays are fixed; once created it cannot be changed again.

For example, [FIGURE 12.3](#) shows the axes (or dimensions) and lengths of two example arrays; 12.3(a) is a one-dimensional array and 12.3(b) is a two-dimensional array. A one-dimensional array has one axis indicated by Axis-0. That axis has five elements in it, so we say it has a length of five. A two-dimensional array is made up of rows and columns.

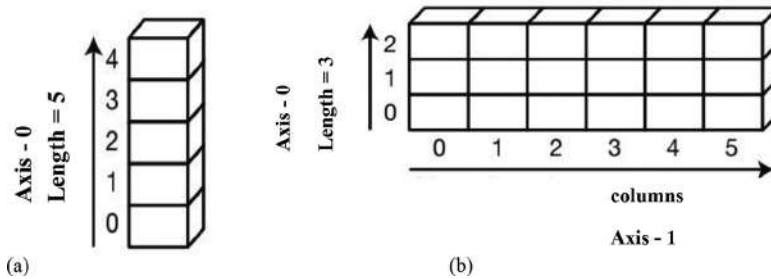


FIGURE 12.3
Dimensions of NumPy Array.

All the rows are indicated by Axis-0 and all the columns are indicated by Axis-1. In a two-dimensional array, Axis-0 has three elements in it, so its length is three and Axis-1 has six elements in it, so its length is six. Notice that for each axis, the indexes range from 0 to length – 1. Array indexes are 0-based. That is, if the length of a dimension is n , the index values range from 0 to $n - 1$.

In order to use NumPy in your program, you need to import NumPy. For example,

```
import numpy as np
```

numpy is usually renamed as np.

12.3.1 NumPy Arrays Creation Using `array()` Function

You can create a NumPy array from a regular Python list or tuple using the `np.array()` function. The type of the resulting array is deduced from the type of the elements. For example,

1. `>>> import numpy as np`
2. `>>> int_number_array = np.array([1,2,3,4])`
3. `>>> int_number_array`
`array([1, 2, 3, 4])`
4. `>>> type(int_number_array)`
`<class 'numpy.ndarray'>`
5. `>>> int_number_array.dtype`
`dtype('int32')`
6. `>>> float_number_array = np.array([9.1, 8.1, 8.8, 3.0])`
7. `>>> float_number_array.dtype`
`dtype('float64')`
8. `>>> two_dimensional_array_list = np.array([[1,2,3], [4,5,6]])`
9. `>>> two_dimensional_array_list`
`array([[1, 2, 3],`
`[4, 5, 6]])`
10. `>>> two_dimensional_array_tuple = np.array(((1,2,3), (4,5,6)))`

```

11. >>> two_dimensional_array_tuple
    array([[1, 2, 3],
           [4, 5, 6]])
12. >>> array_dtype = np.array([1,2,3,4], dtype = np.float64)
13. >>> array_dtype
    array([1., 2., 3., 4.])
14. >>> array_dtype.dtype
    dtype('float64')

```

Import the *numpy* library into your program ①. Pass a list of items to the `np.array()` function and assign the result to *int_number_array* object ②. In the output, you can see all the elements are placed within an iterable object of *array* class and is a one-dimensional array. The *int_number_array* object belongs to *numpy.ndarray* class ④. NumPy provides a large set of numeric datatypes that you can use to construct arrays. NumPy tries to guess a datatype when you create an array, but functions that construct arrays also usually include an optional argument to explicitly specify the datatype. The type of *int_number_array* is `dtype('int32')` ⑤. The type of *float_number_array* ⑥ is `dtype('float64')` ⑦. The `np.array()` function takes either single list or tuple as an argument. If you want to specify multiple lists or tuples, then pass it as nested lists ⑧–⑨ or nested tuples ⑩–⑪. Here, *two_dimensional_array_list* and *two_dimensional_array_tuple* are examples for two-dimensional arrays. You can also explicitly specify the data type of array by assigning type values like `np.float64`, `np.int32`, and others to a *dtype* attribute and pass it as a second argument to `np.array()` function ⑫–⑭.

12.3.2 Array Attributes

The contents of *ndarray* can be accessed and modified by indexing or slicing the array and via the methods and attributes of the *ndarray*. The more important attributes of *ndarray* object are (TABLE 12.2).

TABLE 12.2

Array Attributes

ndarray Attributes	Description
<code>ndarray.ndim</code>	Gives the number of axes or dimensions in the array
<code>ndarray.shape</code>	Gives the dimensions of the array. For an array with n rows and m columns, shape will be a tuple of integers (n, m).
<code>ndarray.size</code>	Gives the total number of elements of the array.
<code>ndarray.dtype</code>	Gives an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally, NumPy provides its own types like <code>np.int32</code> , <code>np.int16</code> , <code>np.float64</code> , and others.
<code>ndarray.itemsize</code>	Gives the size of each element of the array in bytes.
<code>ndarray.data</code>	Gives the buffer containing the actual elements of the array. Normally, we will not use this attribute because we will access the elements in an array using indexing facilities.

Note: In your code, replace *ndarray* with you Python NumPy ndarray object name.

For example,

```

1. >>> import numpy as np
2. >>> array_attributes = np.array([[10, 20, 30], [14, 12, 16]])
3. >>> array_attributes.ndim
    2
4. >>> array_attributes.shape
    (2, 3)
5. >> array_attributes.size
    6
6. >>> array_attributes.dtype
    dtype('int32')
7. >>> array_attributes.itemsize
    4
8. >>> array_attributes.data
    <memory at 0x000001E61DB963A8>

```

Various ndarray attributes ①–⑧.

12.3.3 NumPy Arrays Creation with Initial Placeholder Content

Often, the elements of an array are initially unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content ([TABLE 12.3](#)). These minimize the necessity of growing arrays, an expensive operation.

TABLE 12.3

NumPy Arrays Creation Functions

Function Name	Description
<code>np.zeros()</code>	Creates an array of zeros
<code>np.ones()</code>	Creates an array of ones
<code>np.empty()</code>	Creates an empty array
<code>np.full()</code>	Creates a full array
<code>np.eye()</code>	Creates an identity matrix
<code>np.random.random()</code>	Creates an array with random values
<code>np.arange()</code>	The syntax for <code>arange()</code> is, <code>np.arange([start,]stop, [step,][dtype=None])</code> Returns evenly spaced values within a given interval where <i>start</i> (a number and optional) is the start of interval and its default value is zero, <i>stop</i> (a number) is the end of interval, and <i>step</i> (a number and is optional) is the spacing between the values and <i>dtype</i> is the type of output array.

(Continued)

TABLE 12.3 (Continued)

NumPy Arrays Creation Functions

Function Name	Description
<code>np.linspace()</code>	<p>The syntax for <code>linspace</code> is,</p> <p><code>numpy.linspace(start, stop, num=50, dtype=None)</code></p> <p>Returns evenly spaced numbers over a specified interval where <i>start</i> is the starting value of the sequence, <i>stop</i> is the end value of the sequence, and <i>num</i> (an integer and optional) is the number of samples to generate. Default is 50. Must be non-negative. The optional <i>dtype</i> is the type of the output array.</p>

For example,

1. `>>> import numpy as np`
2. `>>> np.zeros((2,3))`
`array([[0., 0., 0.],`
`[0., 0., 0.]])`
3. `>>> np.ones((3,4))`
`array([[1., 1., 1., 1.],`
`[1., 1., 1., 1.],`
`[1., 1., 1., 1.]])`
4. `>>> np.empty((2,3))`
`array([[0., 0., 0.],`
`[0., 0., 0.]])`
5. `>>> np.full((3,3),2)`
`array([[2, 2, 2],`
`[2, 2, 2],`
`[2, 2, 2]])`
6. `>>> np.eye(2,2)`
`array([[1., 0.],`
`[0., 1.]])`
7. `>>> np.random.random((2,2))`
`array([[0.95022839, 0.23253555],`
`[0.843828 , 0.57976282]])`
8. `>>> np.arange(10, 30, 5)`
`array([10, 15, 20, 25])`
9. `>>> np.arange(0, 2, 0.3)`
`array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])`
10. `>>> np.linspace(0, 2, 9)`
`array([0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2.])`

Various NumPy functions to create arrays ①–⑩. When the *arange()* function is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision ⑨. For this reason, it is usually better to use the *linspace()* function to which you can pass an argument specifying the number of elements you want to generate instead of the *step*. In ⑩, the *linspace()* function produces nine data elements between zero and two.

12.3.4 Integer Indexing, Array Indexing, Boolean Array Indexing, Slicing and Iterating in Arrays

One-dimensional arrays can be indexed, sliced, and iterated over, much like lists and other Python sequences.

```

1. >>> import numpy as np
2. >>> a = np.arange(5)
3. >>> a
    array([0, 1, 2, 3, 4])
4. >>> a[2]
    2
5. >>> a[2:4]
    array([2, 3])
6. >>> a[4:2] = -999
7. >>> a
    array([-999, 1, -999, 3, 4])
8. >>> a[::-1]
    array([ 4, 3, -999, 1, -999])
9. >>> for each_element in a:
10. ...     print(each_element)
    -999
    1
    -999
    3
    4

```

Indexing, slicing, and iterating operations on one-dimensional NumPy arrays ①–⑩.

For multi-dimensional arrays you can specify an index or slice per axis. For example,

```

1. >>> import numpy as np
2. >>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

```

		COLUMNS				
		0	1	2	3	
ROWS	0	1	2	3	4	● a[0, 0]
	1	5	6	7	8	● a[1, 3]
	2	9	10	11	12	● a[2, 1]

3.

```
>>> a[1, 3]
```


8
4.

```
>>> a[:2, 1:3]
```


array([[2, 3],
 [6, 7]])
5.

```
>>> lower_axes = a[1, :]
```
6.

```
>>> lower_axes
```


array([5, 6, 7, 8])
7.

```
>>> lower_axes.ndim
```


1
8.

```
>>> same_axes = a[1:2, :]
```
9.

```
>>> same_axes
```


array([[5, 6, 7, 8]])
10.

```
>>> same_axes.ndim
```


2
11.

```
>>> a[:, 1]
```


array([2, 6, 10])
12.

```
>>> a[:, 1:2]
```


array([[2],
 [6],
 [10]])
13.

```
>>> for row in a:
```
14.

```
...     print(row)
```


[1 2 3 4]
[5 6 7 8]
[9 10 11 12]
15.

```
>>> for each_element in a.flat:
```

```

16. ...     print(each_element)
1
2
3
4
5
6
7
8
9
10
11
12

```

Use integer indexing to pull out the data elements present in row 1 and column 3 ③. You can also mix integer indexing with slice indexing. In ④, pull out the subarray consisting of row 0 and row 1 and column 1 and column 2 having a shape of (2, 2). Display the elements present in all the columns of row 1 ⑤–⑥, ⑧–⑨. Mixing integer indexing with slices yields an array of lower axes ⑦. Using slices in all the axes of an array yields an array of the same axes as the original array ⑩. Display all the elements present in all the rows of column 1 ⑪–⑫. Iterating over multi-dimensional arrays is done with respect to the first axis ⑬–⑭. However, if one wants to perform an operation on each element in the array, one can use the *flat* attribute, which is an iterator over all the elements of the array ⑮–⑯.

```

1. >>> import numpy as np
2. >>> a = np.array([[1, 2], [3, 4], [5, 6]])
3. >>> a
   array([[1, 2],
          [3, 4],
          [5, 6]])
4. >>> a[[0, 1, 2], [0, 1, 0]]
   array([1, 4, 5])

```

Integer array indexing allows you to construct arbitrary arrays using the data from another array ②. Here in code line ④, it is used to construct an array from another array ③.

```

a[[0, 1, 2], [0, 1, 0]]

```

Elements found in (0, 0), (1, 1), and (2, 0) index are pulled out and displayed ④.

```

1. >>> import numpy as np
2. >>> a = np.array([[11, 12], [13, 14], [15, 16]])

```

```

3. >>> a
   array([[11, 12],
          [13, 14],
          [15, 16]])
4. >>> a[a > 13]
   array([14, 15, 16])

```

Boolean array indexing lets you select the elements of an array that satisfy some condition. With Boolean array indexing, you explicitly choose which items in the array you want and which ones you do not want. With Boolean array indexing of $a[a > 13]$ ④, elements greater than 13 in the array a are displayed as a one-dimensional array.

12.3.5 Basic Arithmetic Operations on NumPy Arrays

Basic mathematical functions perform element-wise operation on arrays and are available both as operator overloads and as functions in the NumPy module. For example,

```

1. >>> import numpy as np
2. >>> a = np.array( [20, 30, 40, 50] )
3. >>> b = np.arange(4)
4. >>> b
   array([0, 1, 2, 3])
5. >>> a + b
   array([20, 31, 42, 53])
6. >>> np.add(a, b)
   array([20, 31, 42, 53])
7. >>> a - b
   array([20, 29, 38, 47])
8. >>> np.subtract(a, b)
   array([20, 29, 38, 47])
9. >>> A = np.array( [[1, 1], [6, 1]] )
10. >>> B = np.array( [[2, 8], [3, 4]] )
11. >>> A * B
   array([[2, 8],
          [18, 4]])
12. >>> np.multiply(A, B)
   array([[ 2, 8],
          [18, 4]])
13. >>> A / B
   array([[0.5 , 0.125],
          [2. , 0.25 ]])

```

```

14. >>> np.divide(A, B)
    array([[0.5 , 0.125],
           [2. , 0.25 ]])
15. >>> np.dot(A, B)
    array([[ 5, 12],
           [15, 52]])
16. >>> B**2
    array([[ 4, 64],
           [ 9, 16]], dtype=int32)

```

Element-wise sum, subtract, multiply, and divide operations are performed resulting in an array ⑤–⑭. Matrix product is carried out in ⑮. Every element is squared in array *B* as shown in ⑯.

12.3.6 Mathematical Functions in NumPy

Various mathematical functions are supported in NumPy. A few frequently used mathematical functions are shown below.

```

1. >>> import numpy as np
2. >>> a = np.array( [20, 30, 40, 50] )
3. >>> np.sin(a)
    array([ 0.91294525, -0.98803162, 0.74511316, -0.26237485])
4. >>> np.cos(a)
    array([ 0.40808206, 0.15425145, -0.66693806, 0.96496603])
5. >>> np.tan(a)
    array([ 2.23716094, -6.4053312 , -1.11721493, -0.27190061])
6. >>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
7. >>> np.floor(a)
    array([-2., -2., -1., 0., 1., 1., 2.])
8. >>> np.ceil(a)
    array([-1., -1., -0., 1., 2., 2., 2.])
9. >>> np.sqrt([1,4,9])
    array([ 1., 2., 3.])
10. >>> np.maximum([2, 3, 4], [1, 5, 2])
    array([2, 5, 4])
11. >>> np.minimum([2, 3, 4], [1, 5, 2])
    array([1, 3, 2])
12. >>> np.sum([0.5, 1.5])
    2.0

```

```

13. >>> np.sum([[0, 1], [0, 5]], axis=0)
      array([0, 6])
14. >>> np.sum([[0, 1], [0, 5]], axis=1)
      array([1, 5])

```

Trigonometric operations like *sin()*, *cos()*, and *tan()* are supported ③–⑤. The *floor()* function having syntax as *floor(x)*, returns the largest integer value less than or equal to *x*, element-wise ⑦. The *ceil()* function having a syntax as *ceil(x)*, returns the smallest integer value greater than or equal to *x*, element-wise ⑧. The *sqrt()* function returns the positive square-root of an array, element-wise ⑨. The *maximum()* function compares two arrays and returns a new array containing element-wise maximum of array elements ⑩. The *minimum()* function compares two arrays and returns a new array containing element-wise minimum of array elements ⑪. The *sum()* function returns the sum of array elements over a given axis ⑫–⑭. All of the above functions return a new array.

12.3.7 Changing the Shape of an Array

You can change the shape of an array. For example,

```

1. import numpy as np
2. >>> a = np.floor(10*np.random.random((3,4)))
3. >>> a
      array([[2., 3., 2., 5.],
             [3., 3., 8., 7.],
             [8., 6., 5., 0.]])
4. >>> a.shape
      (3, 4)
5. >>> a.ravel()
      array([2., 3., 2., 5., 3., 3., 8., 7., 8., 6., 5., 0.])
6. >>> a.reshape(6,2)
      array([[2., 3.],
             [2., 5.],
             [3., 3.],
             [8., 7.],
             [8., 6.],
             [5., 0.]])

```

An array has a shape given by the number of elements along each axis ④. The shape of an array can be changed with *ravel()* ⑤ and *reshape()* ⑥ functions. Both *ravel()* and *reshape()* return a modified array but do not change the original array. The function *ravel()* returns a

flattened array, such as a one-dimensional array, containing the elements of the input. The function *reshape()* gives a new shape to an array without changing its data.

12.3.8 Stacking and Splitting of Arrays

You can stack several arrays together or split an array to several arrays. For example,

```

1. >>> import numpy as np
2. >>> a = np.array([[3, 1], [8, 7]])
3. >>> b = np.array([[2, 4], [4, 8]])
4. >>> np.vstack((a, b))
   array([[3, 1],
          [8, 7],
          [2, 4],
          [4, 8]])
5. >>> np.hstack((a, b))
   array([[3, 1, 2, 4],
          [8, 7, 4, 8]])
6. >>> a = np.floor(10*np.random.random((2, 12)))
7. >>> a
   array([[8., 3., 6., 3., 5., 5., 5., 8., 9., 7., 6., 8.],
          [8., 3., 1., 2., 9., 0., 5., 5., 0., 3., 3., 8.]])
8. >>> np.hsplit(a, 3)
   [array([[8., 3., 6., 3.],
          [8., 3., 1., 2.]]), array([[5., 5., 5., 8.],
          [9., 0., 5., 5.]]), array([[9., 7., 6., 8.],
          [0., 3., 3., 8.]])]
9. >>> np.hsplit(a, (3, 4))
   [array([[8., 3., 6.],
          [8., 3., 1.]]), array([[3.],
          [2.]]), array([[5., 5., 5., 8., 9., 7., 6., 8.],
          [9., 0., 5., 5., 0., 3., 3., 8.]])]
10. >>> np.vsplit(a, 2)
     [array([[2., 9., 4., 4., 3., 0., 2., 9., 1., 2., 0., 1.]]), array([[3., 4., 8., 2., 5., 8., 5., 5., 7., 7.,
     7., 8.]])]

```

Several arrays can be stacked together along different dimensions using *vstack()* and *hstack()* functions. With *vstack()* ④ and *hstack()* ⑤ functions you are stacking the arrays *a* and *b* together in row-wise and column-wise fashion. The number of columns when stacking with *vstack()* and the number of rows when stacking with *hstack()* should be

the same. Use `hsplit()` function to split an array along its horizontal axis. You can either specify the number of equally shaped arrays to return or specify the columns after which the division should occur. In ⑧, you split array *a* into three subarrays. In ⑨, you split array *a* after the third and the fourth column. Use `vsplit()` function to split an array along the vertical axis ⑩.

12.3.9 Broadcasting in Arrays

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Broadcasting allows NumPy functions to deal in a meaningful way with input arrays that do not have exactly the same shape. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes and occurs automatically whenever possible. The rules of broadcasting are:

- Rule 1 → If two input arrays do not have the same number of dimensions, a “1” will repeatedly be padded to the shape of the smaller array on its left side by NumPy so both the arrays have the same number of dimensions.
- Rule 2 → If the shape of two input arrays does not match, then the array with a shape of “1” along a particular dimension is stretched by NumPy to match the shape of the array having the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the “broadcast” array. After application of the broadcasting rules, the sizes of all arrays must match.
- Rule 3 → If the above two rules are not met, a *ValueError: frames are not aligned* exception is thrown, indicating that the arrays have incompatible shapes.

NOTE: The Above Rules can be applied to arrays with any number of dimensions.

Example-1.

1. `>>> import numpy as np`
2. `>>> array_1 = np.ones([4, 5])`
3. `>>> array_2 = np.arange(5)`
4. `>>> array_1`
`array([[1., 1., 1., 1., 1.],`
 `[1., 1., 1., 1., 1.],`
 `[1., 1., 1., 1., 1.],`
 `[1., 1., 1., 1., 1.]])`
5. `>>> array_2`
`array([0, 1, 2, 3, 4])`
6. `>>> array_1.shape`
`(4, 5)`
7. `>>> array_2.shape`
`(5,)`


```
8. >>> array_1 + array_2
array([[1., 2., 3., 4., 5.],
       [1., 2., 3., 4., 5.],
       [1., 2., 3., 4., 5.],
       [1., 2., 3., 4., 5.]])
```

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, two arrays must have exactly the same shape. NumPy's broadcasting rule relaxes this constraint when the arrays' shapes meet certain rules. In the above code, two arrays, *array_1* ② and *array_2* ③, with different dimensions are added. Elements of *array_1* is displayed in line ④ and *array_2* is displayed in line ⑤. The shape of *array_1* is (4, 5) ⑥ and *array_2* is (5,) ⑦.

```
array_1.shape → (4, 5)
array_2.shape → (5,)
```

Since *array_2* has less dimension compared to *array_1*, according to Rule 1, *array_2* is padded with 1's on its left. Now the shape of *array_2* becomes (1, 5). NumPy automatically handles this step.

```
array_1.shape → (4, 5)
array_2.shape → (1, 5)
```

Next, according to Rule 2, the shape of *array_2* having "1" in the first dimension is stretched to match the highest shape along that dimension of *array_1*. The shape of *array_2* becomes (4, 5). NumPy automatically handles this step.

```
array_1.shape → (4, 5)
array_2.shape → (4, 5)
```

After stretching, the elements of *array_2* seems to be stacked upon themselves for four times along the first dimension. The elements of *array_2* appear to be the copies of the original array.

```
array_2 → array([[0, 1, 2, 3, 4],
                  [0, 1, 2, 3, 4],
                  [0, 1, 2, 3, 4],
                  [0, 1, 2, 3, 4]])
```

This stretching of the array elements is purely conceptual and does not actually happen as NumPy is smart enough not to make duplicate copies of the original array elements. Also, the original array is not affected.

The final shape of *array_2* becomes (4, 5) matching the shape of *array_1*, thus paving the way for NumPy to perform an addition operation on these two arrays ⑧.

Example-2.

```
1. >>> import numpy as np
2. >>> array_1 = np.random.random(4).reshape([4,1])
3. >>> array_2 = np.arange(4)
```

```

4. >>> array_1.shape
   (4, 1)
5. >>> array_2.shape
   (4,)
6. >>> array_1 + array_2
   array([[0.20188425, 1.20188425, 2.20188425, 3.20188425],
          [0.51342227, 1.51342227, 2.51342227, 3.51342227],
          [0.03364189, 1.03364189, 2.03364189, 3.03364189],
          [0.6176858 , 1.6176858 , 2.6176858 , 3.6176858 ]])

```

In the above code, the shape of *array_1* ② is (4, 1) ④ and *array_2* ③ is (4,) ⑤.

```

array_1.shape → (4, 1)
array_2.shape → (4,)

```

Since *array_2* has less dimension compared to *array_1*, according to Rule 1, *array_2* is padded with 1's on its left. Now the shape of *array_2* becomes (1, 4). NumPy automatically handles this step.

```

array_1.shape → (4, 1)
array_2.shape → (1, 4)

```

Next, according to Rule 2, the shape of *array_1* having "1" in the second dimension is stretched to match the highest shape along that dimension of *array_2*. Thus, the shape of *array_1* becomes (4, 4). The shape of *array_2* having "1" in the first dimension is stretched to match the highest shape along that dimension of *array_1*. Thus, the shape of *array_2* becomes (4, 4). NumPy automatically handles this step.

```

array_1.shape → (4, 4)
array_2.shape → (4, 4)

```

With equal shapes, NumPy performs an addition operation on these two arrays ⑥.

Example-3.

```

1. >>> import numpy as np
2. >>> array_1 = np.random.random([2, 3])
3. >>> array_2 = np.ones(5)
4. >>> array_1.shape
   (2, 3)
5. >>> array_2.shape
   (5,)
6. >>> array_1 + array_2
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   ValueError: operands could not be broadcast together with shapes (2,3) (5,)

```

In the above code, the shape of *array_1* ② is (2, 3) ④ and *array_2* ③ is (5,) ⑤.

```
array_1.shape → (2, 3)
```

```
array_2.shape → (5,)
```

Since *array_2* has less dimension compared to *array_1*, according to Rule 1, *array_2* is padded with 1's on its left. Now the shape of *array_2* becomes (1, 5).

```
array_1.shape → (2, 3)
```

```
array_2.shape → (1, 5)
```

Next, according to Rule 2, the shape of *array_2* having “1” in the first dimension is stretched to match the highest shape along that dimension of *array_1*. Thus, the shape of *array_2* becomes (2, 5).

```
array_1.shape → (2, 3)
```

```
array_2.shape → (2, 5)
```

But the shapes of both the arrays differ and according to Rule 3 the addition operation fails in this case ⑥.

12.4 Pandas

pandas is a Python library that provides fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

The two primary data structures of *pandas*, *Series* (one-dimensional) and *DataFrame* (two-dimensional), handle the vast majority of typical-use cases in finance, statistics, social science, and many areas of engineering. *pandas* is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other third-party libraries.

pandas is well suited for inserting and deleting columns from *DataFrame*, for easy handling of missing data (represented as NaN), explicitly aligning data to a set of labels, converting data in other Python and NumPy data structures into *DataFrame* objects, intelligent label-based slicing, indexing, and subsetting of large data sets, merging and joining of data sets, and flexible reshaping. Additionally, it has robust input/output tools for loading data from CSV files, Excel files, databases, and other formats. You have to import a *pandas* library to make use of various functions and data structures defined in *pandas*.

```
import pandas as pd
```

pandas is usually renamed as *pd*.

12.4.1 Pandas Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the *index*. Pandas Series is created using *series()* method and its syntax is,

```
s = pd.Series(data, index=None)
```

Here, *s* is the Pandas Series, *data* can be a Python *dict*, a *ndarray*, or a scalar value (like 5). The passed index is a list of axis labels. Both integer and label-based indexing are supported. If the index is not provided, then the index will default to *range(n)* where *n* is the length of *data*. For example,

Create Series from ndarrays

```
1. >>> import numpy as np
2. >>> import pandas as pd
3. >>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
4. >>> type(s)
    <class 'pandas.core.series.Series'>
5. >>> s
      a  -0.367740
      b   0.855453
      c  -0.518004
      d  -0.060861
      e  -0.277982
      dtype: float64
6. >>> s.index
    Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
7. >>> s.values
    array([-0.367740,  0.855453, -0.518004, -0.060861, -0.277982])
8. >>> pd.Series(np.random.randn(5))
    0    0.334947
    1   -2.184006
    2   -0.209440
    3   -0.492398
    4   -1.507088
      dtype: float64
```

Import NumPy and pandas libraries ①–②. Create a series using *ndarray* which is NumPy's array class using *Series()* method ③ which returns a Pandas Series type *s* ④. You can also specify axis labels for *index*, i.e., *index=['a', 'b', 'c', 'd', 'e']* ⑤. When data is a *ndarray*, the *index* must be the same length as data. In series *s* ⑥, by default the type of values of all the elements is *dtype: float64*. You can find out the index for a series using *index* attribute ⑦.

The *values* attribute returns a *ndarray* ⑦ containing only values, while the axis labels are removed. If no labels for the index is passed, one will be created having a range of index values [0,..., len(data) - 1] ⑧.

Create Series from Dictionaries

```
1. >>> import numpy as np
2. >>> import pandas as pd
3. >>> d = {'a' : 0., 'b' : 1., 'c' : 2.}
4. >>> pd.Series(d)
   a    0.0
   b    1.0
   c    2.0
dtype: float64
5. >>> pd.Series(d, index=['b', 'c', 'd', 'a'])
   b    1.0
   c    2.0
   d   NaN
   a    0.0
dtype: float64
```

Series can be created from the dictionary. Create a dictionary ③ and pass it to *Series()* method ④. When a series is created using dictionaries, by default the keys will be index labels. While creating series using a dictionary, if labels are passed for the *index*, the values corresponding to the labels in the index will be pulled out ⑤. The order of index labels will be preserved. If a value is not associated for a label, then *NaN* is printed. *NaN* (not a number) is the standard missing data marker used in pandas.

Create Series from Scalar data

```
1. >>> import numpy as np
2. >>> import pandas as pd
3. >>> pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
   a    5.0
   b    5.0
   c    5.0
   d    5.0
   e    5.0
dtype: float64
```

You can create a Pandas Series from scalar value. Here scalar value is five ③. If data is a scalar value, an index must be provided. The value will be repeated to match the length of the index.

Series Indexing and Slicing

```
1. >>> import numpy as np
2. >>> import pandas as pd
```

```
3. >>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
4. >>> s
a    0.481557
b    2.053330
c   -1.799993
d   -0.396880
e   -1.270751
dtype: float64
5. >>> s[0]
0.48155677569897515
6. >>> s[1:3]
b    2.053330
c   -1.799993
dtype: float64
7. >>> s[:3]
a    0.481557
b    2.053330
c   -1.799993
dtype: float64
8. >>> s[s > .5]
b    2.05333
dtype: float64
9. >>> s[[4, 3, 1]]
e   -1.270751
d   -0.396880
b    2.053330
dtype: float64
10. >>> s['a']
0.48155677569897515
11. >>> s['e']
-1.270750548062543
12. >>> 'e' in s
True
13. >>> 'f' in s
False
```

You can provide index ⑤ or slice data by index numbers ⑥–⑦ in a Pandas Series ③–④. You can also specify a Boolean array indexing for Pandas Series ⑧. Multiple indices are specified as a list in ⑨. The index can be an integer value or a label ⑩. Values associated with labeled index are extracted and displayed ⑩–⑪. Check for the presence of a label in Series using *in* operator ⑫–⑬.

Working with Text Data

The Pandas Series supports a set of string processing methods that make it easy to operate on each element of the array. These methods are accessible via the *str* attribute and they generally have the same name as that of the built-in Python string methods.

```
1. >>> import numpy as np
2. >>> import pandas as pd
3. >>> empires_ds = pd.Series(["Vijayanagara", "Roman", "Chola", "Mongol",
    "Akkadian"])
4. >>> empires_ds.str.lower()
0 vijayanagara
1 roman
2 chola
3 mongol
4 akkadian
dtype: object
5. >>> empires_ds.str.upper()
0 VIJAYANAGARA
1 ROMAN
2 CHOLA
3 MONGOL
4 AKKADIAN
dtype: object
6. >>> empires_ds.str.len()
0 11
1 5
2 5
3 6
4 8
dtype: int64
7. >>> tennis_ds = pd.Series(['Seles ', ' Graph ', ' Williams '])
8. >>> tennis_ds.str.strip()
0 Seles
1 Graph
2 Williams
dtype: object
9. >>> tennis_ds.str.contains(' ')
0 True
1 True
```

```
2 True
dtype: bool
10. >>> marvel_ds = pd.Series(['Thor_loki', 'Thor_Hulk', 'Gamora_Storm'])
11. >>> marvel_ds.str.split('_')
0 [Thor, loki]
1 [Thor, Hulk]
2 [Gamora, Storm]
dtype: object
12. >>> planets = pd.Series(["Venus", "Earth", "Saturn"])
13. >>> planets.str.replace("Earth", "Mars")
0 Venus
1 Mars
2 Saturn
dtype: object
14. >>> letters_ds = pd.Series(['a', 'b', 'c', 'd'])
15. >> letters_ds.str.cat(sep=',')
'a,b,c,d'
16. >>> names_ds = pd.Series(['Jahnavi', 'Adelmo', 'Pietro', 'Alejandro'])
17. >>> names_ds.str.count('e')
0 0
1 1
2 1
3 1
dtype: int64
18. >>> names_ds.str.startswith('A')
0 False
1 True
2 False
3 True
dtype: bool
19. >>> names_ds.str.endswith('O')
0 False
1 False
2 False
3 False
dtype: bool
20. >>> names_ds.str.find('J')
0 0
1 -1
```



```
2 -1
3 -1
dtype: int64
```

Various string methods to operate with Pandas Series is discussed ①–②①.

12.4.2 Pandas DataFrame

DataFrame is a two-dimensional, labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or database table, or a *dict* of Series objects. It is generally the most commonly used pandas object. DataFrame accepts many different kinds of input like *Dict* of one-dimensional *ndarrays*, lists, *dicts*, or Series, two-dimensional *ndarrays*, structured or record *ndarray*, a dictionary of Series, or another DataFrame.

```
df = pd.DataFrame(data=None, index=None, columns=None)
```

Here, *df* is the DataFrame and *data* can be NumPy *ndarray*, *dict*, or DataFrame. Along with the *data*, you can optionally pass an *index* (row labels) and *columns* (column labels) attributes as arguments. If you pass an *index* and/or *columns*, you are guaranteeing the *index* and/or *columns* of the resulting DataFrame. Both *index* and *columns* will default to *range(n)* where *n* is the length of *data*, if they are not provided. When the *data* is a dictionary and *columns* are not specified, then the DataFrame column labels will be dictionary's *keys*.

Create DataFrame from Dictionary of Series/Dictionaries

```
1. >>> import pandas as pd
2. >>> dict_series = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
...   'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
3. >>> df = pd.DataFrame(dict_series)
4. >>> df
      

|   | one | two |
|---|-----|-----|
| a | 1.0 | 1.0 |
| b | 2.0 | 2.0 |
| c | 3.0 | 3.0 |
| d | NaN | 4.0 |


5. >>> df.shape
(4, 2)
6. >>> df.index
Index(['a', 'b', 'c', 'd'], dtype='object')
7. >>> df.columns
Index(['one', 'two'], dtype='object')
8. >>> list(df.columns)
['one', 'two']
9. >>> dicts_only = {'a':[1,2,3], 'b':[4,5,6]}
```

```

10. >>> dict_df = pd.DataFrame(dict_series)
11. >>> dict_df
      a  b
0  1  4
1  2  5
2  3  6
12. >>> dict_df.index
RangeIndex(start=0, stop=3, step=1)

```

You need to import pandas library ①. Create a dictionary whose values are a series of a one-dimensional arrays ②. You can create a DataFrame from a dictionary of series. Pass *dict_series* dictionary as argument to *DataFrame()* class which returns a DataFrame object ③. Index labels are passed as a list to index attribute. If the number of labels specified in the various series are not the same, then the resulting index will be the union of all the index labels of various series. In DataFrame *df* under column "one", there is no data element associated with index label "d", so NaN will be inserted at that position ④. A number of rows and columns in a DataFrame is obtained using *shape* attribute ⑤. Get the index labels for the DataFrame using *index* attribute ⑥. With *columns* attribute, you get all the columns of the DataFrame ⑦. Get the columns of DataFrame as a list by passing the *columns* attribute as an argument to the *list()* function ⑧. You can create a DataFrame from a dictionary ⑨ without using *index* and *columns* attributes ⑩. For *dict_df* DataFrame ⑪, *a* and *b* are columns and index labels are integers ranging from zero to two ⑫.

Create DataFrame from ndarrays/lists/list of dictionaries

```

1. >>> import numpy as np
2. >>> import pandas as pd
3. >>> dict_ndarrays = {'one': np.random.random(5), 'two': np.random.random(5)}
4. >>> pd.DataFrame(dict_ndarrays)
      one      two
0  0.346580  0.827881
1  0.738850  0.577504
2  0.969715  0.781170
3  0.668432  0.746535
4  0.709333  0.440675
5. >>> pd.DataFrame([1,2,3,4,5], [6,7,8,9,10])
      0  1  2  3  4
0  1  2  3  4  5
1  6  7  8  9 10

```

↙ 0 1 2 3 4 ← columns
rows 0 1 2 3 4 5
1 6 7 8 9 10

```

6. >>> dict_lists = {'one': [1, 2, 3, 4, 5], 'two': [5, 4, 3, 2, 1]}
7. >>> pd.DataFrame(dict_lists)

```

```

    one  two
0  1    5
1  2    4
2  3    3
3  4    2
4  5    1
8. >>> pd.DataFrame(dict_lists, index=['a', 'b', 'c', 'd', 'e'])
    one  two
a  1    5
b  2    4
c  3    3
d  4    2
e  5    1
9. >>> lists_dicts = [{'a':1, 'b':2}, {'a':5, 'b':10, 'c':20}]
10. >>> pd.DataFrame(lists_dicts)
    a  b    c
0  1  2  NaN
1  5 10 20.0

```

The pandas ② library is built on top of NumPy ①. Here, *dict_ndarrays* ③ is a dictionary of *ndarrays* from which you can create a DataFrame ④. Also, nested lists can be used to create a DataFrame ⑤. If no index and columns are specified, then both index and columns will have integer labels. *Keys* are considered as *column* labels ⑦ when a DataFrame is created using dictionaries ⑥. The DataFrame columns will be preserved in the same order as specified by dictionary *keys*. In ⑧, index labels are specified for a DataFrame created from the dictionary. The DataFrame can also be created from a list of dictionaries ⑨. Since DataFrame columns will be a union of all the keys in the list of dictionaries, elements for missing columns will be NaN ⑩.

DataFrame Column Selection, Addition and Deletion

```

1. >>> import pandas as pd
2. >>> la_liga = {"Ranking":[1,2,3], "Team": ["Barcelona", "Atletico Madrid", "Real Madrid"]}
3. >>> df = pd.DataFrame(la_liga)
4. >>> df
    Ranking      Team
0         1  Barcelona
1         2  Atletico Madrid
2         3   Real Madrid
5. >>> df["Team"]

```

```

0      Barcelona
1   Atletico Madrid
2      Real Madrid
Name: Team, dtype: object
6. >>> df['Played'] = [34, 36, 38]
7. >>> df['Won'] = [27, 23, 22]
8. >>> df[['Played', 'Won']]
   Played  Won
0     34    27
1     36    23
2     38    22
9. >>> df['Points'] = df['Won'] * 2
10. >>> df
   Ranking  Team      Played  Won  Points
0     1     Barcelona      34   27     54
1     2  Atletico Madrid      36   23     46
2     3    Real Madrid      38   22     44
11. >>> df['Lost'] = [1, 5, 6]
12. >>> df
   Ranking  Team      Played  Won  Points  Lost
0     1     Barcelona      34   27     54     1
1     2  Atletico Madrid      36   23     46     5
2     3    Real Madrid      38   22     44     6
13. >>> df['Drawn'] = df['Played'] - df['Won'] - df['Lost']
14. >>> df
   Ranking  Team      Played  Won  Points  Lost  Drawn
0     1     Barcelona      34   27     54     1     6
1     2  Atletico Madrid      36   23     46     5     8
2     3    Real Madrid      38   22     44     6    10
15. >>> df['Year'] = 2018
16. >>> df
   Ranking  Team      Played  Won  Points  Lost  Drawn  year
0     1     Barcelona      34   27     54     1     6    2018
1     2  Atletico Madrid      36   23     46     5     8    2018
2     3    Real Madrid      38   22     44     6    10    2018
17. >>> del df['Year']

```

```
18. >>> df.pop('Drawn')
```

```
0    6
1    8
2   10
```

```
Name: Drawn, dtype: int64
```

```
19. >>> df.insert(5, 'Goal Difference', [63, 38, 42])
```

```
20. >>> df
```

	Ranking	Team	Played	Won	Points	Goal Difference	Lost
0	1	Barcelona	34	27	54	63	1
1	2	Atletico Madrid	36	23	46	38	5
2	3	Real Madrid	38	22	44	42	6

```
21. >>> df.rename(columns = {'Team':'Club Team'})
```

	Ranking	Club Team	Played	Won	Points	Goal Difference	Lost
0	1	Barcelona	34	27	54	63	1
1	2	Atletico Madrid	36	23	46	38	5
2	3	Real Madrid	38	22	44	42	6

Create DataFrame *df* ③ from *la_liga* dictionary ②. You can select a particular column in a DataFrame by specifying the column name within quotes inside a bracket of a DataFrame ⑤.

You can add a new column to the DataFrame by specifying the column label within the bracket of DataFrame and assign data elements to it ⑥–⑦. Grab multiple columns from a DataFrame by passing a list of columns ⑧. You can also create a new column by making use of the data elements found in existing columns. Column “Points” is inserted to the DataFrame *df* after multiplying all the data elements in column “Won” by 2 ⑨. You can perform basic arithmetic operations on DataFrame columns ⑨. When inserting a scalar value, it will naturally be propagated to fill the column ⑮–⑯. Columns can be deleted ⑰ or popped ⑱. By default, columns get inserted at the end. The insert function is available to insert at a particular location in the columns ⑨. You can rename the column label using the *rename()* method. The *columns* attribute has to be passed to the *rename()* method and assign it with a dictionary where the old column label will be key and new column label will be a value of ⑳. All the above operations have a direct impact on the DataFrame.

Displaying Data in DataFrame

```
1. >>> import pandas as pd
```

```
2. >>> df = pd.DataFrame({'WorldCup_Winner':['Brazil', 'Germany', 'Argentina',  
      "Brazil", "Spain"], 'Year':[1962, 1974, 1986, 2002, 2010]})
```

```
3. >>> df.columns
```

```
Index(['WorldCup_Winner', 'Year'], dtype='object')
```

```
4. >>> df.head(2)
```

	WorldCup_Winner	Year
0	Brazil	1962
1	Germany	1974

```

5. >>> df.tail(2)
      WorldCup_Winner Year
3             Brazil 2002
4             Spain 2010
6. >>> df['WorldCup_Winner'].unique()
array(['Brazil', 'Germany', 'Argentina', 'Spain'], dtype=object)
7. >>> df['WorldCup_Winner'].unique().tolist()
['Brazil', 'Germany', 'Argentina', 'Spain']
8. >>> df.transpose()
      0      1      2      3      4
WorldCup_Winner  Brazil  Germany  Argentina  Brazil  Spain
Year            1962    1974    1986    2002    2010
9. >>> df.sort_values(by=['Year'], ascending = False)
      WorldCup_Winner Year
4             Spain 2010
3             Brazil 2002
2          Argentina 1986
1             Germany 1974
0             Brazil 1962
10. >>> df.sort_index(ascending = False)
      WorldCup_Winner Year
4             Spain 2010
3             Brazil 2002
2          Argentina 1986
1             Germany 1974
0             Brazil 1962
11. >>> df['WorldCup_Winner'].value_counts()
Brazil      2
Argentina   1
Germany     1
Spain       1
Name: WorldCup_Winner, dtype: int64
12. >>> df['WorldCup_Winner'].value_counts().index.tolist()
['Brazil', 'Argentina', 'Germany', 'Spain']
13. >>> df['WorldCup_Winner'].value_counts().values.tolist()
[2, 1, 1, 1]

```

DataFrame *head(n)* ④ method returns first *n* rows and *tail(n)* ⑤ method returns last *n* rows. You can find unique data elements in a column by chaining *unique()* method with a DataFrame column using dot notation. The *unique()* ⑥ method returns a one-dimensional array-like object, which can be converted to a list using *tolist()* method ⑦. The *transpose()* method flips the DataFrame over its main diagonal by writing rows as columns and vice versa ⑧. The syntax for *sort_values()* method is,

df.sort_values(by, axis=0, ascending=True)

where the *by* parameter can be a string, list of strings, index label, column label, list of index labels, or list of column labels to sort by. If the value of the *axis* is 0 then *by* may contain column labels. If the value of the *axis* is 1, then *by* may contain index labels. By default, the value of the *axis* parameter is 0. The default value of *ascending* parameter is *True*, if so then the data elements will be sorted in ascending order. A *False* value leads to sorting the data elements in descending order ⑨. By default, the *sort_index()* method, performs sorting on row labels in ascending order and returns a copy of the DataFrame. If the *ascending* parameter is set to Boolean *False*, then the *sort_index()* method performs sorting in descending order ⑩. The *value_counts()* method when chained with a DataFrame, returns a Series object containing counts of unique values ⑪. The resulting object will be in descending order so that the first element is the most frequently-occurring element. The *NA* values are excluded by default. The *index* attribute returns the index or row labels of the Series ⑫. The *values* attribute returns a NumPy representation of the Series ⑬.

Using DataFrame *assign()* method

```
1. >>> import pandas as pd
2. >>> df_mountain = pd.DataFrame({"Mountain": ['Mount Everest', 'K2',
'Kangchenjunga'], "Length": [8848, 8611, 8586]})
3. >>> df_mountain.assign(Ranking = [1, 2, 3])
```

	Length	Mountain	Ranking
0	8848	Mount Everest	1
1	8611	K2	2
2	8586	Kangchenjunga	3

```
4. >>> df = pd.DataFrame({'A': [2, 4, 6], 'B': [3, 6, 9]})
5. >>> df.assign(C = lambda x: x['A'] ** 2)
```

	A	B	C
0	2	3	4
1	4	6	16****
2	6	9	36

DataFrame has an *assign()* method that allows you to easily create new columns that are potentially derived from existing columns ⑭. The *assign()* method always returns a copy of the data, leaving the original DataFrame untouched.

DataFrame Indexing and Selecting Data

The Python and NumPy indexing operators `[]` and dot operator `.` provide quick and easy access to select a subset of data elements in a pandas DataFrame across a wide

range of use cases. However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits. For production code, it's highly recommended that you take advantage of the optimized pandas data access methods, like `.loc[]` and `.iloc[]`, which are used to retrieve rows. Note that `.loc[]` and `.iloc[]` methods are followed by square brackets `[]`, not parentheses `()` and are called as indexers.

The `.loc[]` method is primarily label based, but may also be used with a Boolean array. The `.loc[]` method will raise `KeyError` when the items are not found. Inputs accepted by `.loc[]` method are a single label, e.g. 5 or 'a' (note that 5 is interpreted as a label of the index/row; this use is not an integer position along the index), a list or array of labels ['a', 'b', 'c'], a slice object with labels 'a':'f' (note that contrary to usual Python slices, both the start and the stop are included, when present in the index!) and Boolean array.

The `.iloc[]` method is primarily an integer position based (from 0 to length-1 of the axis), but may also be used with a Boolean array. The `.iloc[]` method will raise an `IndexError` if a requested indexer is out-of-bounds, except in the case of slice indexers, which allow out-of-bounds indexing (this conforms with Python/NumPy slice semantics). Allowed inputs for `.iloc[]` method are an integer, such as 5, a list or array of integers [4, 3, 0], a slice object with ints 1:7, and a Boolean array. For example,

```
1. >>> import numpy as np
2. >>> import pandas as pd
3. >>> df = pd.DataFrame(np.random.rand(5,5), index = ['row_1', 'row_2', 'row_3',
    'row_4', 'row_5'], columns = ['col_1', 'col_2', 'col_3', 'col_4', 'col_5'])
4. >>> df
      col_1    col_2    col_3    col_4    col_5
row_1  0.302179  0.067154  0.848890  0.291533  0.710989
row_2  0.668777  0.246157  0.339020  0.232109  0.390328
row_3  0.787487  0.703837  0.542948  0.839311  0.050887
row_4  0.905814  0.026933  0.381502  0.754635  0.399242
row_5  0.244861  0.343171  0.992433  0.058433  0.266207
5. >>> df.loc['row_1']
col_1    0.302179
col_2    0.067154
col_3    0.848890
col_4    0.291533
col_5    0.710989
Name: row_1, dtype: float64
6. >>> df.loc['row_2', 'col_3']
0.339020
7. >>> df.loc[['row_1', 'row_2'], ['col_2', 'col_3']]
      col_2    col_3
row_1  0.067154  0.84889
row_2  0.246157  0.33902
```


8. >>> df.loc[:, ['col_2', 'col_3']]

	col_2	col_3
row_1	0.067154	0.848890
row_2	0.246157	0.339020
row_3	0.703837	0.542948
row_4	0.026933	0.381502
row_5	0.343171	0.992433

9. >>> df.iloc[1]

col_1	0.668777
col_2	0.246157
col_3	0.339020
col_4	0.232109
col_5	0.390328

Name: row_2, dtype: float64

10. >>> df.iloc[3:5, 0:2]

	col_1	col_2
row_4	0.905814	0.026933
row_5	0.244861	0.343171

11. >>> df.iloc[3, :]

	col_1	col_2	col_3	col_4	col_5
row_1	0.302179	0.067154	0.848890	0.291533	0.710989
row_2	0.668777	0.246157	0.339020	0.232109	0.390328
row_3	0.787487	0.703837	0.542948	0.839311	0.050887

12. >>> df.iloc[:, :]

	col_1	col_2	col_3	col_4	col_5
row_1	0.302179	0.067154	0.848890	0.291533	0.710989
row_2	0.668777	0.246157	0.339020	0.232109	0.390328
row_3	0.787487	0.703837	0.542948	0.839311	0.050887
row_4	0.905814	0.026933	0.381502	0.754635	0.399242
row_5	0.244861	0.343171	0.992433	0.058433	0.266207

13. >>> df.iloc[2:, 2:]

	col_3	col_4	col_5
row_3	0.542948	0.839311	0.050887
row_4	0.381502	0.754635	0.399242
row_5	0.992433	0.058433	0.266207

14. >>> df.iloc[:, 1]

row_1	0.067154
row_2	0.246157
row_3	0.703837

```

row_4    0.026933
row_5    0.343171
Name: col_2, dtype: float64
15. >>> df[df > 0.2]

```

	col_1	col_2	col_3	col_4	col_5
row_1	0.302179	NaN	0.848890	0.291533	0.710989
row_2	0.668777	0.246157	0.339020	0.232109	0.390328
row_3	0.787487	0.703837	0.542948	0.839311	NaN
row_4	0.905814	NaN	0.381502	0.754635	0.399242
row_5	0.244861	0.343171	0.992433	NaN	0.266207

For `.loc[row_label_indexing, col_label_indexing]` and `.iloc[row_integer_indexing, col_integer_indexing]` methods, a single argument always refers to selecting data elements from row indices in the DataFrame and not the column indices. When `col_label_indexing` or `col_integer_indexing` is absent, it means all the columns for that particular row will be selected.

For example, `.loc['a']` is equivalent to `.loc['a',:]`. This example applies to `iloc` as well. With `df.loc[indexer]` you know automatically that `df.loc[]` is selecting rows. In contrast, it is not clear if `df[indexer]` will select rows or columns (or raise `ValueError`) without knowing details about `indexer` and `df`. In ⑤, select the first row labeled as `row_1` and all the columns of that row. Line ⑥ selects data elements present in the second row labeled as `row_2` and third column labeled as `col_3`. ⑦ Selects values present in the first row, `row_1` and second row, `row_2` along with their corresponding columns. In ⑧, you slice via labels and select all the rows under column 2 and 3.

You can grab data based on position instead of labels using `.iloc` method. The `.iloc[]` method provides integer-based indexing. The semantics follow Python and NumPy slicing closely. These are zero-based indexing. When slicing, the start bound is included, while the upper bound is excluded. In ⑨, select all the data elements from the second row along the entire columns. Even though we have labeled these rows and columns, still their integer indices range from 0 to $n - 1$, where n is the length of the data. Slicing returns a subset of data elements present in DataFrame along with their corresponding labels. In ⑩, even though the row index is out of range, still the existing rows will be selected and out-of-range indexes are handled gracefully. All data elements starting from first to the third row along their entire column are selected ⑪. The entire DataFrame is selected in ⑫. Rows from position three onwards and columns from position three onwards are selected ⑬. All the rows in the second column are selected ⑭. An important feature of pandas is conditional selection using bracket notation, very similar to *numpy*. Data elements greater than 0.2 in DataFrame are displayed while the lower values are treated as NaN ⑮. Note, none of the above operations change the original data elements of DataFrame.

Group By: *split-apply-combine*

Here, “group by” refers to a process involving one or more of the following steps:

- Splitting the data into groups based on some criteria.
- Applying a function to each group independently.
- Combining the results into a data structure.

1. Out of these, the *split* step is the most straightforward. In fact, in many situations, we may wish to split the data set into groups and do something with those groups.
2. In the *apply* step, we might wish to do one of the following:

Aggregation: compute a summary statistic (or statistics) for each group. For example,

Compute group sums or means.

Compute group sizes/counts.

Transformation: perform some group-specific computations and return a like-indexed object. For example,

Standardize data (zscore) within a group.

Filling NAs within groups with a value derived from each group.

Filtration: discard some groups, according to a group-wise computation that evaluates True or False. For example,

Discard data that belong to groups with only a few members.

Filter out data based on the group sum or mean.

3. Some *combination* of the above: GroupBy will examine the results of the apply step and try to return a sensibly combined result if it does not fit into either of the above two categories.

For example,

1.

```
>>> import pandas as pd
```
2.

```
>>> cars_data = {'Company':['General Motors','Ford', 'Toyota', 'General Motors',  
'Ford', 'Toyota'], 'Model': ['Camaro', 'Mustang', 'Prius', 'Malibu', 'Fiesta', 'Camry'],  
'Sold':[12285, 35273, 34287, 29325, 27459, 17621]}
```
3.

```
>>> cars_df = pd.DataFrame(cars_data)
```
4.

```
>>> cars_df
```

	Company	Model	Sold
0	General Motors	Camaro	12285
1	Ford	Mustang	35273
2	Toyota	Prius	34287
3	General Motors	Malibu	29325
4	Ford	Fiesta	27459
5	Toyota	Camry	17621
5.

```
>>> cars_df.groupby('Company').mean()
```

	Company	Sold
	Ford	31366
	General Motors	20805
	Toyota	25954
6.

```
>>> cars_df.groupby('Company').std()
```

	Company	Sold
	Ford	5525.332388

```

General Motors    12049.099551
Toyota            11784.641615
7. >>> cars_df.groupby('Company').min()
Company    Model    Sold
Ford       Fiesta    27459
General Motors    Camaro    12285
Toyota      Camry    17621
8. >>> cars_df.groupby('Company').max()
Company    Model    Sold
Ford       Mustang    35273
General Motors    Malibu    29325
Toyota      Prius    34287
9. >>> cars_df.groupby('Company').sum()
Company    Sold
Ford       62732
General Motors    41610
Toyota      51908
10. >>> cars_df.groupby('Company').describe()
Company    Sold count mean      std      min      25%      50%      75%      max
Ford       2.0  31366.0  5525.332388  27459.0  29412.5  31366.0  33319.5  35273.0
General Motors  2.0  20805.0  12049.099551  12285.0  16545.0  20805.0  25065.0  29325.0
Toyota       2.0  25954.0  11784.641615  17621.0  21787.5  25954.0  30120.5  34287.0
11. >>> cars_df.groupby('Company').count()
Company    Model    Sold
Ford       2      2
General Motors    2      2
Toyota      2      2
12. >>> cars_df.groupby('Company')['Company'].count()
Company
Ford      2
General Motors    2
Toyota     2
Name: Company, dtype: int64
13. >>> cars_df.groupby('Company')['Company'].count().tolist()
[2, 2, 2]
14. >>> cars_df.groupby('Company')['Company'].count().index.tolist()
['Ford', 'General Motors', 'Toyota']

```

```

15. >>> cars_df.groupby(['Company','Model']).groups
{'Ford', 'Fiesta': Int64Index([4], dtype='int64'), ('Ford', 'Mustang'):
Int64Index([1], dtype='int64'), ('General Motors', 'Camaro'): Int64Index([0],
dtype='int64'), ('General Motors', 'Malibu'): Int64Index([3], dtype='int64'),
('Toyota', 'Camry'): Int64Index([5], dtype='int64'), ('Toyota', 'Prius'):
Int64Index([2], dtype='int64')}

16. >>> grp_by_company = cars_df.groupby('Company')
17. >>> for label, group in grp_by_company:
...     print(label)
...     print(group)
...
Ford
   Company  Model  Sold
1  Ford      Mustang  35273
4  Ford      Fiesta  27459
General Motors
   Company  Model  Sold
0  General Motors  Camaro  12285
3  General Motors  Malibu  29325
Toyota
   Company  Model  Sold
2  Toyota  Prius  34287
5  Toyota  Camry  17621

```

In the above code, `cars_df` ③, is the `DataFrame` on which the `groupby()` method will be applied. The `groupby()` method allows you to group rows of data together based on a column name and call aggregate functions on them. For instance, let's group based on "Company" column using the `groupby()` method. This will return a `DataFrameGroupBy` object upon which you can call aggregate methods ⑤–⑪. If you need to count only one column then specify the name of the column within brackets as shown in ⑫ for which you can get values ⑬ and index labels ⑭ as a list by chaining `tolist()` method. Various aggregate functions are listed below (TABLE 12.4).

TABLE 12.4

Aggregate Functions and Their Description

Function	Description
<code>mean()</code>	Compute mean of groups
<code>sum()</code>	Compute sum of group values
<code>size()</code>	Compute group sizes
<code>count()</code>	Compute count of group
<code>std()</code>	Standard Deviation of groups
<code>describe()</code>	Generate Descriptive Statistics
<code>min()</code>	Compute minimum of group values
<code>max()</code>	Compute maximum of group values

The `groups` ⑥ attribute is a dictionary whose keys are the computed unique groups and corresponding values are the index labels belonging to each group. Assign the `DataFrameGroupBy` object returned by `groupby()` method to a `grp_by_company` variable ⑥. With a `grp_by_company` object, you can iterate through the grouped data by specifying two iterating variables ⑦. Here `label` variable returns the data elements of the grouped column `company` and `group` variable returns the grouped data.

Concatenate, Append and Merge

The pandas library provides various facilities for easily *combining/concatenating* together Series as well as DataFrame objects. The pandas library also has support for full-featured, high performance in-memory *merge* operations, also called *join* operations. The pandas library provides a single function, `merge()`, as the entry point for all standard merge operations between different DataFrame objects.

1.

```
>>> import pandas as pd
```
2.

```
>>> left_df = pd.DataFrame({'Ranking':[1, 2, 3, 4, 5],
...   'University':['MIT', 'Stanford', 'Harvard', 'UCB', 'Princeton'],
...   'Student':['Liam', 'William', 'Sofia', 'Logan', 'Olivia']})
```
3.

```
>>> right_df = pd.DataFrame({'Ranking':[1, 2, 3, 4, 5],
...   'University':['Oxford', 'ETH', 'Cambridge', 'Utrecht', 'Humboldt'],
...   'Student':['Charles', 'Liam', 'Sofia', 'Rafael', 'Hannah']})
```
4.

```
>>> left_df
```

	Ranking	University	Student
0	1	MIT	Liam
1	2	Stanford	William
2	3	Harvard	Sofia
3	4	UCB	Logan
4	5	Princeton	Olivia
5.

```
>>> right_df
```

	Ranking	University	Student
0	1	Oxford	Charles
1	2	ETH	Liam
2	3	Cambridge	Sofia
3	4	Utrecht	Rafael
4	5	Humboldt	Hannah
6.

```
>>> concatenate_df = pd.concat([left_df, right_df])
```
7.

```
>>> concatenate_df
```

	Ranking	University	Student
0	1	MIT	Liam

1	2	Stanford	William
2	3	Harvard	Sofia
3	4	UCB	Logan
4	5	Princeton	Olivia
0	1	Oxford	Charles
1	2	ETH	Liam
2	3	Cambridge	Sofia
3	4	Utrecht	Rafael
4	5	Humboldt	Hannah

```

8. >>> concatenate_df = pd.concat([left_df, right_df], keys = ['Universities_Americas',
'Universities_Europe'])
9. >>> concatenate_df.loc['Universities_Americas']
   Ranking  University  Student
0        1      MIT      Liam
1        2    Stanford  William
2        3    Harvard   Sofia
3        4      UCB     Logan
4        5    Princeton  Olivia
10. >>> append_df = left_df.append(right_df)
11. >>> append_df
   Ranking  University  Student
0        1      MIT      Liam
1        2    Stanford  William
2        3    Harvard   Sofia
3        4      UCB     Logan
4        5    Princeton  Olivia
0        1    Oxford   Charles
1        2      ETH     Liam
2        3    Cambridge  Sofia
3        4    Utrecht   Rafael
4        5    Humboldt  Hannah
12. >>> pd.merge(left_df, right_df, on = 'Student')
   Ranking_x  University_x  Student  Ranking_y  University_y
0         1         MIT      Liam         2         ETH
1         3         Harvard   Sofia         3         Cambridge
13. >>> pd.merge(left_df, right_df, on = ['Ranking', 'Student'])
   Ranking  University_x  Student  University_y
0         3         Harvard   Sofia         Cambridge

```

14. >>> pd.merge(left_df, right_df, on = 'Student', how = 'left')

	Ranking_x	University_x	Student	Ranking_y	University_y
0	1	MIT	Liam	2.0	ETH
1	2	Stanford	William	NaN	NaN
2	3	Harvard	Sofia	3.0	Cambridge
3	4	UCB	Logan	NaN	NaN
4	5	Princeton	Olivia	NaN	NaN

15. >>> pd.merge(left_df, right_df, on = 'Student', how = 'right')

	Ranking_x	University_x	Student	Ranking_y	University_y
0	1.0	MIT	Liam	2	ETH
1	3.0	Harvard	Sofia	3	Cambridge
2	NaN	NaN	Charles	1	Oxford
3	NaN	NaN	Rafael	4	Utrecht
4	NaN	NaN	Hannah	5	Humboldt

16. >>> pd.merge(left_df, right_df, on = 'Student', how = 'outer')

	Ranking_x	University_x	Student	Ranking_y	University_y
0	1.0	MIT	Liam	2.0	ETH
1	2.0	Stanford	William	NaN	NaN
2	3.0	Harvard	Sofia	3.0	Cambridge
3	4.0	UCB	Logan	NaN	NaN
4	5.0	Princeton	Olivia	NaN	NaN
5	NaN	NaN	Charles	1.0	Oxford
6	NaN	NaN	Rafael	4.0	Utrecht
7	NaN	NaN	Hannah	5.0	Humboldt

17. >>> pd.merge(left_df, right_df, on = 'Student', how = 'inner')

	Ranking_x	University_x	Student	Ranking_y	University_y
0	1	MIT	Liam	2	ETH
1	3	Harvard	Sofia	3	Cambridge

The *concat()* function does all of the heavy lifting of performing concatenation operations. The syntax for *concat()* function is,

pd.concat(objs, keys=None)

Here *objs* can be a Series, DataFrame, List, or Dictionary. The *keys* are a list of labels and its default value is *None*. This function returns a DataFrame object, if DataFrames are concatenated, and returns a Series object if Series are concatenated.

Two DataFrames *left_df* ② and *right_df* ③ are created. Both DataFrames have *Ranking*, *University*, and *Student* as column labels ④–⑤. You can concatenate multiple DataFrames using the *concat()* function by passing multiple DataFrames as list items to the *concat()* function ⑥. Observe that the integer index labels of different DataFrames that were

concatenated are retained as it is in the concatenated DataFrame ⑦. If you want to associate specific keys with each of the pieces of the chopped up DataFrame, then you can do this by using the *keys* argument ⑧. You can extract each chunk of DataFrame by passing the key associated with it as an index to *.loc[]* method ⑨. A useful shortcut to the *concat()* function is the *append()* instance methods on Series and DataFrame ⑩.

A *merge()* function combines columns from multiple DataFrames and returns a new DataFrame object. These columns must be found in all the DataFrames that are to be merged. The syntax for *merge()* function is,

```
df_obj = pd.merge(left_df, right_df, how='inner', on=None)
```

where *left_df* is a DataFrame object, *right_df* is another DataFrame object, *on* is the names of column labels that you want to merge. The column labels that you assign to an *on* argument are called as *keys*. For a *how* argument, you can assign any one of the values 'left', 'right', 'outer', or 'inner'. The default value is 'inner'. The *merge()* function returns a DataFrame object.

A *merge()* function is a means for combining columns from a *left_df* DataFrame and *right_df* DataFrame by using data elements that are common to each other. The *how* argument in the *merge()* function specifies how to determine which *keys* are to be included in the resulting DataFrame. The data elements of *keys* may or may not be found in both the *left_df* and *right_df* DataFrame objects while carrying out the merge operation; if not then NaN will be assigned in the resulting merged DataFrame.

The pandas *merge()* method specifies four types of merge operations: 'left', 'right', 'outer' and 'inner'. Let's understand each of these merge operations in detail.

Left Merge → In the left merge operation, preference is given to the *key* columns of *left_df* DataFrame. All the rows in the *key* columns of the *left_df* DataFrame are retained in the resulting DataFrame. If any of the data elements in the *left_df* DataFrame *key* columns are present in the *right_df* DataFrame *key* columns, then those rows are also retained. But for the rows of the data elements of the *right_df* DataFrame *key* columns, which are not same as that of the data elements of *left_df* DataFrame *key* columns, NaN is assigned (FIGURE 12.4a).

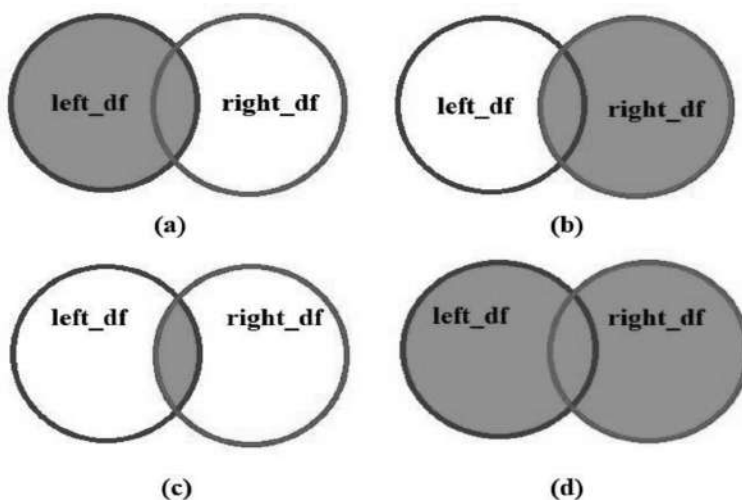


FIGURE 12.4

Pictorial representation of various types of merge operation. (a) Left merge; (b) Right merge; (c) Inner merge and (d) Outer merge.

Right Merge → In the right merge operation, preference is given to the *key* columns of *right_df* DataFrame. All the rows in *key* columns of the *right_df* DataFrame are retained in the resulting DataFrame. If any of the data elements in the *right_df* DataFrame *key* columns are present in the *left_df* DataFrame *key* columns, then those rows are also retained. However, for the rows of the data elements of the *left_df* DataFrame *key* columns, which are not same as that of the data elements of *right_df* DataFrame *key* columns, NaN is assigned (FIGURE 12.4b).

Inner Merge → In the inner merge operation, if the data elements are found in the *key* columns of both *left_df* and *right_df* DataFrames, then only those rows are retained in the resulting DataFrame. This is the default behavior of *merge()* function (FIGURE 12.4c).

Outer Merge → In the outer merge operation, all the rows from *left_df* and all the rows from *right_df* DataFrames are retained in the resulting DataFrame. Rows for all the data elements found in the *key* columns of both *left_df* and *right_df* DataFrames are matched and retained in the resulting DataFrame, and for missing rows of *left_df* and *right_df* DataFrames NaN is assigned (FIGURE 12.4d).

In ②, an inner merge operation is carried out by merging *left_df* and *right_df* DataFrames on 'Student' key column. Notice *_x* and *_y* appended to *Ranking* and *University* column labels. These suffixes are added for any clashes in column names that are not involved in the merge operation. You can also perform a merge operation on multiple key columns by assigning them as list items to on argument ③. Left merge ④, right merge ⑤, outer merge ⑥, and inner merge ⑦ operations are carried out as shown in the above code. If you want to merge more than two DataFrames, then the syntax is *df_1.merge(df_2, on = 'column_name').merge(df_3, on = 'column_name')*

Handling Missing Data

In the real world, the dataset you encounter will contain lots of missing data. Hence, pandas offer different methods to handle missing data elements.

1. >>> import pandas as pd
2. >>> df = pd.DataFrame({'a':pd.Series([1, 2]), 'b':pd.Series([10, 20, 30, 40, 50]), 'c':pd.Series([100, 200, 300])})
3. >>> df

	a	b	c
0	1.0	10	100.0
1	2.0	20	200.0
2	NaN	30	300.0
3	NaN	40	NaN
4	NaN	50	NaN
4. >>> df.dropna()

	a	b	c
0	1.0	10	100.0
1	2.0	20	200.0
5. >>> df.fillna(value = '0')

	a	b	c
0	1	10	100
1	2	20	200

```

2  0  30  300
3  0  40    0
4  0  50    0
6. >>> df['c'].fillna(value = df['c'].mean())
0    100.0
1    200.0
2    300.0
3    200.0
4    200.0
Name: c, dtype: float64

```

The DataFrame *df* ② consists of a few missing data elements ③. You have the option of dropping labels with missing data via the *dropna()* function ④. The *fillna()* method fills the missing values with specified scalar value ⑤–⑥. Note: None of these methods change the original data elements of DataFrame.

DataFrame Data Input and Output

You can read from CSV and Excel files using *read_csv()* and *read_excel()* methods. Also, you can write to CSV and Excel files using *to_csv()* and *to_excel()* methods. For example,

```

1. >>> df_csv = pd.read_csv("foo.csv")
2. >>> df_excel = pd.read_excel("foo.xlsx")
3. >>> df.to_csv('foo.csv')
4. >>> df.to_excel('foo.xlsx', sheet_name='Sheet1')

```

Both *read_csv()* ① and *read_excel()* ② methods return DataFrame. DataFrame is written to CSV ③ and Excel files ④.

12.5 Altair

Altair is a declarative statistical visualization library for Python and it is based on Vega-Lite. What is Vega-Lite? Vega-Lite is a high-level grammar of interactive graphics. It provides a concise JSON syntax for rapidly generating visualizations to support data analysis. The Vega-Lite compiler automatically produces visualization components including axes, legends, and scales.

Altair's API is simple, friendly and consistent and built on top of the powerful Vega-Lite visualization grammar. With Altair, you can spend more time understanding your data and its meaning.

The key idea of Altair is that you are declaring links between DataFrame columns and visual encoding channels, such as the x-axis, y-axis, color, etc. The rest of the plot details are handled automatically. This elegant simplicity produces beautiful and compelling

visualizations with a minimal amount of code. Building on this *declarative* plotting idea, a surprising range of simple to sophisticated plots and visualizations can be created using a relatively simple grammar.

Install Altair using *pip* command as shown below:

1. C:\>pip install altair.

Installs *altair* library ①.

Follow the steps mentioned below to generate an Altair chart.

- Create an Altair *Chart object* with a pandas DataFrame.
- Choose a suitable *mark* relevant to your Dataset.
- *Encode* the X and Y values with appropriate columns in the DataFrame.
- *Save* the data emitted by Altair Chart object as a file with a *.json* extension.
- Navigate to <https://vega.github.io/editor/> an online Vega-Lite editor and paste the contents of the JSON file in the left pane of the editor. You can see a Chart generated in the right pane of the editor.

Specifying Data in Altair

Data in Altair is built around the pandas DataFrame. In Altair, every dataset should be provided as a DataFrame. This makes the encoding quite simple, as Altair uses the data type information provided by pandas to determine the data types required in the encoding automatically.

Chart

The fundamental object in Altair is the *Chart*. It takes the DataFrame as a single argument. A Chart is an object that knows how to emit a JSON dictionary representing the data and visualization encodings, which is visually rendered by the online Vega-Lite editor using the Vega-Lite JavaScript library.

Chart Marks

Now you have data that is not yet defined on how it should be visualized. You need to decide what sort of *mark* you would like to use to represent the data. Basic graphical elements in Vega-Lite are marks. Marks provide basic shapes whose properties (such as size, opacity, and color) can be used to visually encode data from pandas DataFrame columns. For example, you can choose a bar mark to plot your data as bar chart. Some of the more commonly used *mark_*()* methods supported in Altair are *mark_line()* → a Line plot, *mark_bar()* → a Bar plot, *mark_area()* → a filled Area plot, *mark_rect()* a filled rectangle used for Heat maps, *mark_point()* a Scatterplot and others.

The encode() method

The next step is to add visual encodings (or encodings for short) to the chart. Encodings can be created with the *encode()* method of the Chart object. The encoding object is a key-value mapping between encoding channels (such as X, Y) and DataFrame columns.

The keys in the encoding object are encoding channels. Altair supports the following group of encoding channels:

- $X \rightarrow$ x-axis value
- $Y \rightarrow$ y-axis value
- Color \rightarrow color of the mark
- Row and Column \rightarrow Row and Column are special encoding channels that facets single plots into small multiples plots within a grid of facet plots.

The details of any mapping depend on the type of the data. Altair is able to automatically determine the type of data using built-in heuristics. Altair supports four primitive data types ([TABLE 12.5](#)).

TABLE 12.5

Primitive Data Types Supported by Altair

Data Type	Code	Description
Quantitative	Q	Numerical Quantity (Real-Valued)
Nominal	N	Name/Unordered Categorical
Ordinal	O	Ordered Categorical
Temporal	T	Date/Time

You can set the data type of a DataFrame column explicitly using a one-letter code attached to the DataFrame column name with a colon (:). If types are not specified for DataFrame data input, Altair defaults to quantitative for any numeric data, temporal for date/time data and nominal for string data, but be aware that these defaults are by no means always the correct choice!

Data Aggregation

The process of gathering and expressing information in a summary form is called Data aggregation. The X and Y encodings in Altair accepts different aggregate functions. Some of the commonly used aggregate functions in Altair are shown in [TABLE 12.6](#).

TABLE 12.6

Aggregate Functions Built into Altair

Aggregate Function	Description
count()	The total count of data values in a group
max()	The maximum data value
min()	The minimum data value
median()	The median data value
mean()	The mean (or average) data value
sum()	The sum of all the data values

Saving Altair Charts

The fundamental chart representation output by Altair is a JSON string format. You can save the JSON data produced by Chart object to a JSON file using the `Chart.save()` method and by passing a filename with a `.json` extension as an argument. For example,

```
1. >>> chart.save('chart_name.json')
```

Altair chart objects have a `Chart.save()` method, which allows charts to be saved in *json* format ①.

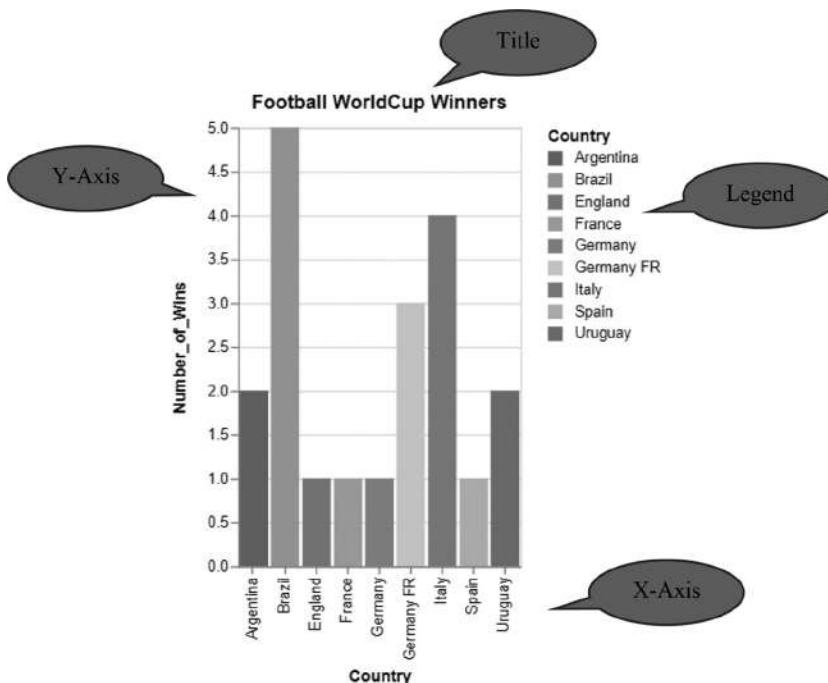
Program 12.8: Write Python Program to Read ‘WorldCups.csv’ File. Sample Contents of ‘WorldCups.csv’ File Is Given Below. Plot Bar Chart to Display the Number of Times a Country Has Won Football WorldCup

Year	Country	Winner	Runners-Up
1930	Uruguay	Uruguay	Argentina
1934	Italy	Italy	Czechoslovakia
1938	France	Italy	Hungary
1950	Brazil	Uruguay	Brazil
1954	Switzerland	Germany FR	Hungary
1958	Sweden	Brazil	Sweden

```

1. import pandas as pd
2. import altair as alt
3. def main():
4.     worldcup_df = pd.read_csv('WorldCups.csv')
5.     winning_countries = worldcup_df['Winner'].value_counts()
6.     winners_total_df = pd.DataFrame({'Country': winning_countries.index.tolist(),
                                     'Number_of_Wins': winning_countries.values.tolist()})
7.     chart = alt.Chart(winners_total_df).mark_bar().encode(
8.         alt.X('Country:N'),
9.         alt.Y('Number_of_Wins:Q'),
10.         alt.Color('Country:N')).properties(
11.             title="Football WorldCup Winners")
12.     chart.save('WorldCup_Winners.json')
13. if __name__ == "__main__":
14.     main()
```

OUTPUT



The above chart is an example of plotting Bar chart. A Bar chart displays the categorical data as rectangular bars with lengths proportional to the values that they represent. The rectangular bars in a bar chart can be plotted either vertically or horizontally. Categorical data represents the type of data that can be divided into groups or collected in groups. For example, gender, race and genre of books; 5 boys and 12 girls in a class represent categorical data. Ensure that your data is converted to a DataFrame format with the desired structuring for easy access and analysis ④–⑥. The *winners_total_df* ⑥ DataFrame has two columns; the *Country* column, which is a list of countries that have won Football WorldCup, and *Number_of_Wins* column, which is a list of number of times each country in the *Country* column have won the WorldCup. The key to creating meaningful visualizations is to map DataFrame columns to encoding channels through the *encode()* method ⑦. Here *X* ⑧ and *Y* ⑨ class represent the x-axis and y-axis of the chart, which takes the column names as arguments. Each of these column names are attached with single characters and separated by a colon. Even though Altair has the ability to decide the type of data for each of the DataFrame columns, it is better you explicitly specify the type of data so that you get the chart you were expecting. The *Color* class generates a legend that describes each unique data element in the column that makes up the chart ⑩. The above chart shows a legend explaining the colors of each country that won the WorldCup. Again, note the column name *Country* attached with N character, separated by a colon. The *X*, *Y*, *Column* and *Color* classes are specified within *encode()* method. To specify a title for the Chart use *title* attribute ⑪ within *properties()* method. Use the *save()* method to save the JSON data emitted by the Chart object as a file ⑫. Copy and paste the JSON data

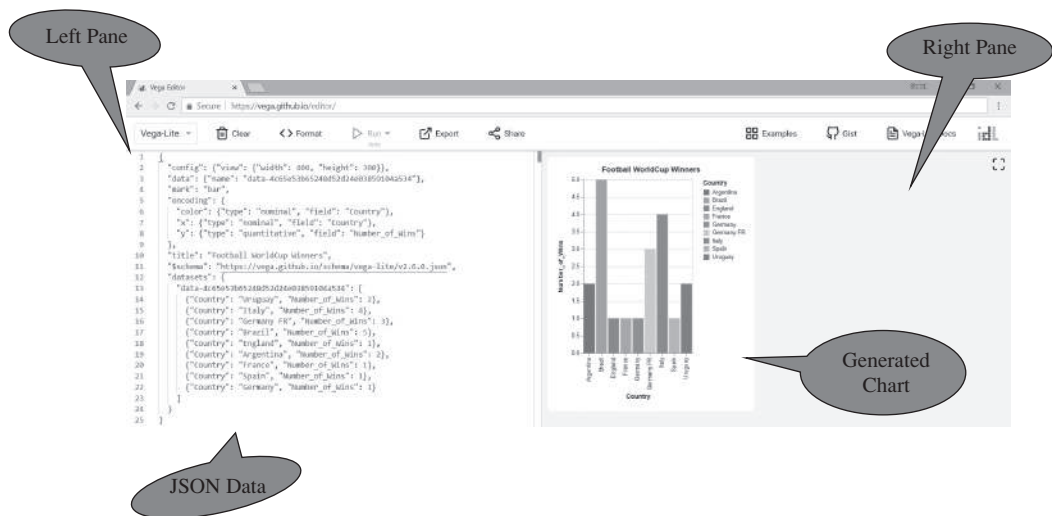


FIGURE 12.5
Online Vega-Lite Editor in action.

from the file into a Vega-Lite online editor and see the generated chart. Here is a snapshot of the Vega-Lite online editor in action ([FIGURE 12.5](#)).

Program 12.9: Write Python Program to Read 'Endangered_Species.csv' File. Sample Contents of 'Endangered_Species.csv' File Is Given Below. Plot Grouped Bar Chart to Display the Population Growth of These Endangered Species

Species	Population	Year
Bengal Tiger	1201	2000
Bengal Tiger	1411	2005
Polar Bear	8832	2000
Polar Bear	14753	2005
African Lion	21690	2000
African Lion	28431	2005

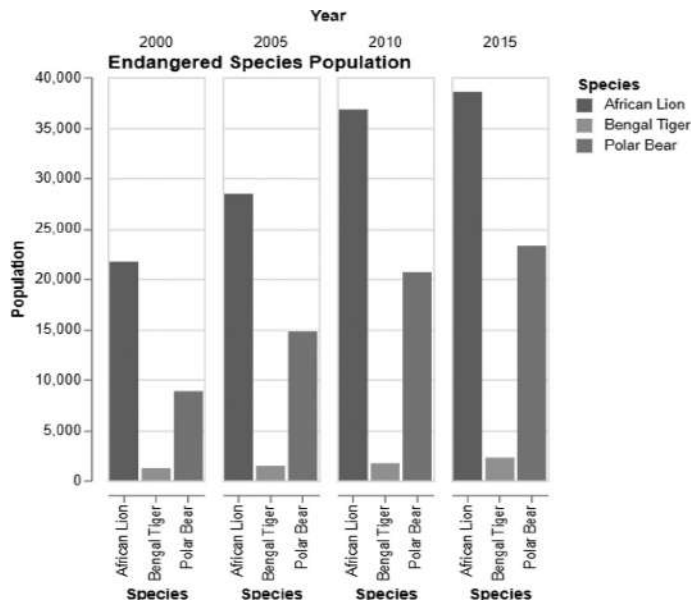
1. import altair as alt
2. import pandas as pd
3. def main():
4. df = pd.read_csv('Endangered_Species.csv')
5. chart = alt.Chart(df).mark_bar().encode(
6. alt.X('Species:N'),
7. alt.Y('Population:Q'),
8. alt.Column('Year'),


```

9.     alt.Color('Species:N')).properties(
10.     title="Endangered Species Population")
11.     chart.save('Endangered_Species_Population.json')
12. if __name__ == "__main__":
13.     main()

```

OUTPUT



The above chart is an example of plotting Grouped Bar chart. A Grouped Bar chart has two or more rectangular bars for each categorical group. The rectangular bars are color coded to represent a particular grouping. The above chart conveys information about the population of endangered species broken out by species type and year. Altair is smart enough to decide that a Grouped Bar chart has to be generated for the dataset ④–⑪.

Program 12.10: Write Python Program to Read 'Company_Annual_Net_Income.csv' File. Sample Contents of 'Company_Annual_Net_Income.csv' File is Given Below. Plot Line Chart to Display the Annual Net Income of These Companies

Year	Company	Profit
2010	Microsoft	18.76
2011	Microsoft	23.15
2010	Alphabet	8.372
2011	Alphabet	9.706
2010	Amazon	1.152
2011	Amazon	0.631

```

1. import pandas as pd
2. import altair as alt
3. def main():

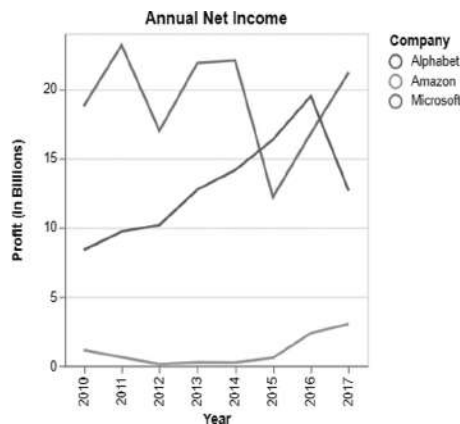
```

```

4. company_net_income_df = pd.read_csv('Company_Annual_Net_Income.csv')
5. chart = alt.Chart(company_net_income_df).mark_line().encode(
6.     alt.X('Year:N', axis=alt.Axis(title='Year')),
7.     alt.Y('Profit:Q', axis=alt.Axis(title='Profit (in Billions)')),
8.     alt.Color('Company:N')).properties(
9.     title="Annual Net Income",
10.    width=250, height=250)
11. chart.save('Company_Net_Income.json')
12. if __name__ == "__main__":
13.     main()

```

OUTPUT



The above chart is an example of plotting Line chart. The line chart displays information by connecting a series of data points (also called markers) through a straight line. Line charts are ideal for viewing data that changes over time. Use *Axis* class to add labels to X ⑥ and Y ⑦ axis overriding the labels inherited from DataFrame columns. You can set the chart properties, such as *title*, *width*, and *height*, using the *properties()* method ⑧–⑨. This is a simple way to add some more information to a Line chart. This Chart conveys information about the profit earned by each company.

Program 12.11: Write Python Program to read 'Height_Weight_Ratio.csv' file. Sample contents of 'Height_Weight_Ratio.csv' file is given below. Using the Scatterplot, display the relation between height and weight in adult male and female

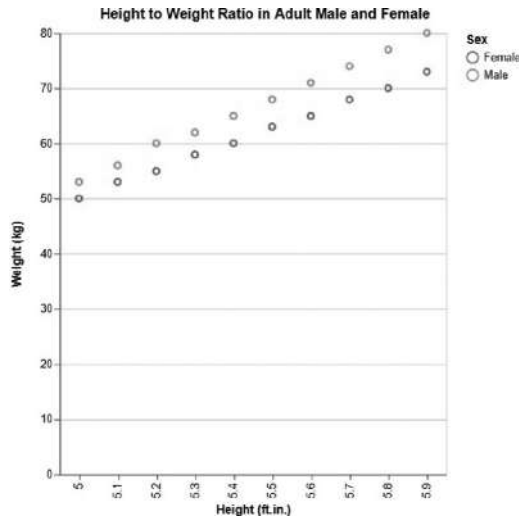
Height_Weight_Ratio.csv		
1	Sex, Height, Weight	
2	Female, 5.0, 50	
3	Female, 5.1, 53	
4	Female, 5.2, 55	
5	Male, 5.0, 53	
6	Male, 5.1, 56	
7	Male, 5.2, 60	

```

1. import altair as alt
2. import pandas as pd
3. def main():
4.     df = pd.read_csv('Height_Weight_Ratio.csv')
5.     chart = alt.Chart(df).mark_point().encode(
6.         alt.X('Height:N', axis=alt.Axis(title='Height (ft.in.)')),
7.         alt.Y('Weight:Q', axis=alt.Axis(title='Weight (kg)')),
8.         alt.Color('Sex:N').properties(
9.             title="Height to Weight Ratio in Adult Male and Female",
10.            width=350, height=400)
11.     chart.save('Height_Weight_Ratio.json')
12. if __name__ == "__main__":
13.     main()

```

OUTPUT



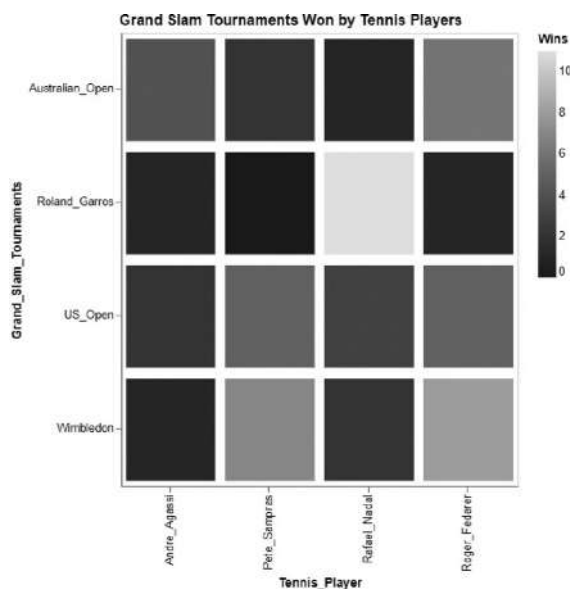
The above chart is an example of plotting data using Scatterplot. A scatterplot consists of a set of individual dots that represent the values dependent on two different variables with values of one variable plotted against the x-axis and values of another variable plotted against y-axis. Scatter plot shows how the change in one variable affects the other variable. The relation between these two variables is called correlation. A variable is any characteristics, number, or quantity that can be measured or counted. The `mark_point()` method is used to plot Scatterplot for the given data. This Chart conveys information about the relation between height and weight in adult male and female ④–⑩.

Program 12.12: Write Python Program to read 'Tennis_Summary.csv' file. Sample contents of 'Tennis_Summary.csv' file is given below. Using the Heatmap, display the number of Grand Slam Tournaments won by different players

Tennis_Summary.csv			
1	Tennis_Player,Grand_Slam_Tournaments,Wins		
2	Roger_Federer,Australian_Open,6		
3	Roger_Federer,Roland_Garros,1		
4	Rafael_Nadal,Wimbledon,2		
5	Rafael_Nadal,US_Open,3		
6	Pete_Sampras,Wimbledon,7		
7	Pete_Sampras,US_Open,5		
8	Andre_Agassi,Australian_Open,4		
9	Andre_Agassi,Roland_Garros,1		

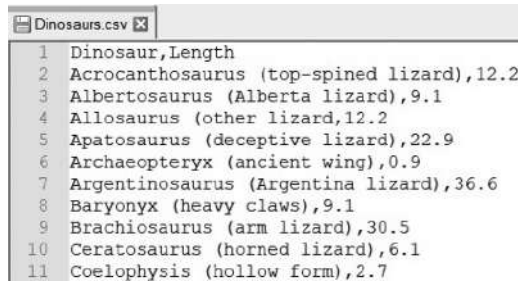
1. import altair as alt
2. import pandas as pd
3. def main():
4. df = pd.read_csv('Tennis_Summary.csv')
5. chart = alt.Chart(df).mark_rect().encode(
6. alt.X('Tennis_Player:N'),
7. alt.Y('Grand_Slam_Tournaments:N'),
8. alt.Color('Wins:Q')).properties(
9. title="Grand Slam Tournaments Won by Tennis Players",
10. width=350, height=400)
11. chart.save('Tennis.json')
12. if __name__ == "__main__":
13. main()

OUTPUT



The above chart is an example of plotting data using Heatmap. In Heatmaps, the data values are represented as colors of varying degrees allowing the users to visualize data information. You can think of Heatmap as a spreadsheet like data table wherein each individual data value is represented as different gradient colors. The color bar represents the relation between the color and the data values and is placed to the right of the Heatmap by default. The `mark_rect()` method is used to plot Heatmap for the given data. The number of times different players who have won each of the Grand Slam tournaments is conveyed through the above Heatmap ④–⑩.

Program 12.13: Write Python Program to read 'Dinosaurs.csv' file. Sample contents of 'Dinosaurs.csv' file is given below. Create a Histogram displaying the length of different Dinosaurs

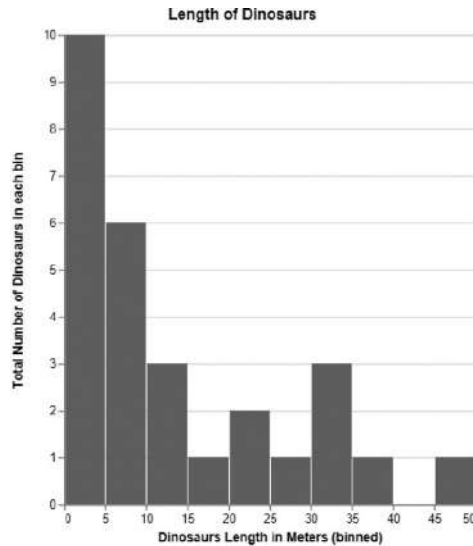


Dinosaur	Length
Acrocanthosaurus (top-spined lizard)	12.2
Albertosaurus (Alberta lizard)	9.1
Allosaurus (other lizard)	12.2
Apatosaurus (deceptive lizard)	22.9
Archaeopteryx (ancient wing)	0.9
Argentinosaurus (Argentina lizard)	36.6
Baryonyx (heavy claws)	9.1
Brachiosaurus (arm lizard)	30.5
Ceratosaurus (horned lizard)	6.1
Coelophysis (hollow form)	2.7

```

1. import altair as alt
2. import pandas as pd
3. def main():
4.     df = pd.read_csv('Dinosaurs.csv')
5.     chart = alt.Chart(df).mark_bar().encode(
6.         alt.X('Length:Q', bin=True, axis=alt.Axis(title='Dinosaurs Length in Meters
7.             (binned)'),
8.         alt.Y('count():Q', axis=alt.Axis(title='Total Number of Dinosaurs in each
9.             bin'))).properties(
10.         title="Length of Dinosaurs",
11.         width=350, height=400)
12.     chart.save("Dinosaurs.json")
13. if __name__ == "__main__":
14.     main()

```

OUTPUT

The above chart is an example of creating a Histogram. A histogram is a chart that groups numeric data into bins and displays the bins as bars. To construct a histogram from the numeric data you need to split the data into a series of intervals called the bins. A histogram is used to depict the frequency distribution of numeric data values in a dataset by counting how many of these values fall into each bin. In Altair, the `mark_bar()` method is used to plot Histogram for the given data. When the `bin` flag is set to `True`, then the number of bins is automatically chosen for you ④. All bins are of equal width and have a height proportional to the number of records or numeric data values present in the bin. Data values are split based on the bin it falls in and the results are aggregated within each bin using the `count()` function ⑦. The above histogram tells us that most of the Dinosaurs have a length within 5 meters range ④–⑩.

12.6 Summary

- Lambdas, Iterators, Generators, and List Comprehensions are used for functional programming in Python.
- In Python, Deserializing is done using `JSON load()` and `loads()` methods, and Serializing is done using `JSON dump()` and `dumps()` methods.
- HTTP requests can be made using a Requests library, which eases the integration of Python programs with web services.

- The `xml.etree.ElementTree` built-in Python library provides various methods to perform different operations on XML files.
- The fundamental package for carrying out scientific computing in Python scientific community is NumPy, which stands for “Numerical Python.”
- pandas library is seen as the reason for the tremendous adaption of Python in the field of data science.
- Altair is a declarative statistical visualization library for Python, based on Vega-Lite, which can be used to generate different charts with minimal code.

Multiple Choice Questions

- The full form of abbreviation XML is
 - Extensible Markup Language
 - Excisable Markup Language
 - Executive Markup Language
 - Extensible Managing Language
- Guess the correct syntax of the declaration which defines the XML version.
 - `<xml version="1.0"/>`
 - `<?xml version="1.0"/>`
 - `<?xml version="1.0"?>`
 - `</xml version="1.0"/>`
- Comments in XML is identified by
 - `<?----- >`
 - `</----- />`
 - `<!----- >`
 - `</----- >`
- Consider the following XML code and identify the root node.


```
<?xml version= "1.0" encoding ="UTF-8"?>
<fullname>
  <firstname>Alex</firstname>
  <lastname>Stanley</lastname>
  <employeeecode>EC123</employeeecode>
</fullname>
```

 - `<fullname>`
 - `<firstname>`
 - `<lastname>`
 - `<employeeecode>`

5. JSON stands for _____.
 - a. JavaScript Object Notation
 - b. Java Object Notation
 - c. JSON Object Notation
 - d. All of these
6. The extension for JSON files is
 - a. .json
 - b. .js
 - c. .jn
 - d. .jsn
7. JSON string value pair is written as
 - a. string = 'value'
 - b. "string": "value"
 - c. name = "value"
 - d. name: 'value'
8. Which of the following syntax is correct for a JSON array?
 - a. {"digits": ["1", "2", "3"]}
 - b. {"digits": {"1", "2", "3"}}
 - c. {"digits": [1, 2, 3]}
 - d. {"digits": ["1", "2", "3"]}
9. JSON elements are separated by
 - a. semi-colon
 - b. line break
 - c. comma
 - d. white space
10. Which of the following can be data in panda?
 - a. dictionary
 - b. An ndarray
 - c. A scalar value
 - d. All of these
11. Identify the correct syntax to import the pandas library.
 - a. import pandas as pd
 - b. import panda as py
 - c. import pandaspy as py
 - d. None of the above
12. Which of the following is the standard data missing marker used in pandas?
 - a. NaN
 - b. Null
 - c. None
 - d. All of the above

13. The object that is returned after reading CSV file in pandas is _____
 - a. Character Vector
 - b. DataFrame
 - c. Panel
 - d. None of the above
14. Point out the correct statement.
 - a. NumPy's main object is the homogeneous multidimensional array
 - b. In NumPy, dimensions are called axes
 - c. NumPy's array class is called ndarray
 - d. All of these
15. The function that returns its arguments with a modified shape and the method that modifies the array itself respectively in NumPy are
 - a. resize, reshape
 - b. reshape, resize
 - c. reshape2, resize
 - d. reshape2, resize2
16. The declarative statistical visualization library available in Python is
 - a. Altair
 - b. Matplotlib
 - c. Seaborn
 - d. Bokeh
17. Input Data in Altair is primarily based on
 - a. Pandas DataFrame
 - b. Strings
 - c. Lists
 - d. Dictionaries
18. If the type of Data is not specified in Altair, then nominal data defaults to
 - a. Tuple
 - b. Dictionary
 - c. String
 - d. List

Review Questions

1. Explain the use of Lambdas in Python with an example.
2. Describe iterators and generators in Python.
3. Illustrate the use of List Comprehensions with an example.
4. State the need for requests library in Python.
5. Write Pythonic code to parse the XML code shown below and calculate the total number of students in College.

```
<College>
  <Department>
    <DepartmentName>CSE</DepartmentName>
    <TotalStudents>200</TotalStudents>
  </Department>
  <Department>
    <DepartmentName>ISE</DepartmentName>
    <TotalStudents>60</TotalStudents>
  </Department>
  <Department>
    <DepartmentName>ECE</DepartmentName>
    <TotalStudents>200</TotalStudents>
  </Department>
</College>
```

6. Define JSON. Construct a simple JSON document and write Pythonic code to parse JSON document.
7. Elaborate on the differences between XML and JSON.
8. Define XML. Construct a simple XML document and write Python code to loop through XML nodes in the document.
9. Explain NumPy array creation functions with examples.
10. Explain NumPy integer indexing, array indexing, Boolean array indexing and slicing with examples.
11. Write Python program to create and display a one-dimensional array-like object containing an array of data using pandas library.
12. Write Python program to add, subtract, multiply and divide two Pandas Series.

13. Write Python program to create and display a DataFrame from a dictionary data which has the index labels.
14. Explain the steps involved in generating an Altair chart in detail.
15. Plot Altair Line chart for below data to display company performance.

Year	Sales	Expenses
2010	1000	400
2011	1170	460
2012	660	1120
2013	1030	540
2014	2193	1052
2015	1168	843

16. Plot Altair Bar chart for below data to display the density of precious metals in g/cm³.

Element	Density
Copper	8.94
Silver	10.49
Gold	19.30
Platinum	21.45

Appendix-A: Debugging Python Code

AIM

To understand the process of debugging Python program using the inbuilt debugger of PyCharm IDE.

Debugging in computer programming is a multistep process that involves identifying a problem, isolating the source of the problem, and then correcting the problem. Software which assists in this process is known as a debugger. Using a debugger, a software developer can step through a program's code and analyze its variable values, searching for errors. Debugging helps in preventing incorrect operation of a software and operate according to a set of specifications.

Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program does not catch the exception, the interpreter prints a stack trace. The debugger in PyCharm allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. PyCharm debugger assists a software developer in becoming more productive.

A1 A Python Program to Debug

Consider the Python program to solve quadratic equation $ax^2 + bx + c = 0$ (FIGURE A1).

As you see, there is the *main* clause here. It means that execution will begin with it, let you enter the desired values of the variables *a*, *b* and *c*, and then enter the method *demo*.

```

1      import math
2
3
4      class Solver:
5          def demo(self, a, b, c):
6              d = b ** 2 - 4 * a * c
7              if d > 0:
8                  disc = math.sqrt(d)
9                  root1 = (-b + disc) / (2 * a)
10                 root2 = (-b - disc) / (2 * a)
11                 return root1, root2
12             elif d == 0:
13                 return -b / (2 * a)
14             else:
15                 return "This equation has no roots"
16
17
18  ▶   if __name__ == '__main__':
19       solver = Solver()
20       while True:
21           a = int(input("a: "))
22           b = int(input("b: "))
23           c = int(input("c: "))
24           result = solver.demo(a, b, c)
25           print(result)

```

FIGURE A1

Program to solve Quadratic equation.

A2 Placing Breakpoints

In software development, a breakpoint is an intentional stopping or pausing place in a program, put in place for debugging purposes, perhaps to see the state of code variables. It is also sometimes simply referred to as a pause.

Breakpoints are triggered when the program reaches the specified line of source code before it is executed. The line of code that contains a set breakpoint, is marked with a red stripe; once such line of code is reached, the marking stripe changes to blue.

To place breakpoints, just click the left gutter next to the line you want your application to suspend at ([FIGURE A2](#)):

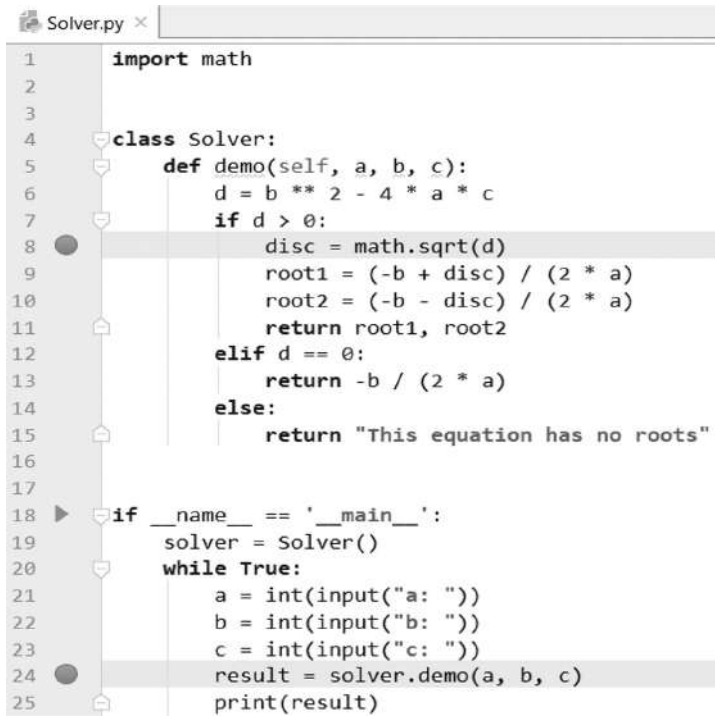


FIGURE A2
Placing breakpoints.

A3 Starting the Debugger Session

OK, now, as we have added breakpoints, everything is ready for debugging. PyCharm allows starting the debugger session in several ways. Let's choose one: click ► in the left gutter, and then select the command *Debug 'Solver'* in the pop-up menu that opens (FIGURE A3):



FIGURE A3
Starting the debugger session.

The debugger starts, shows the Console tab of the Debug tool window, and lets you enter the desired values ([FIGURE A4](#)):

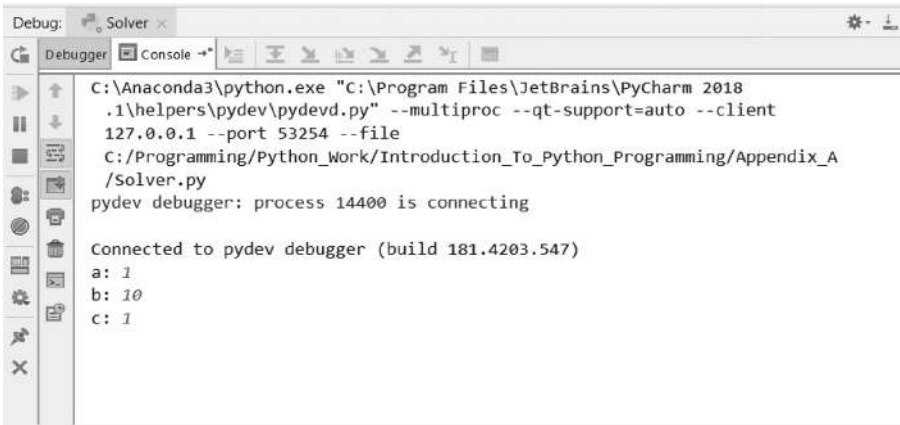



FIGURE A4

Enter values in the Console tab.

By the way, in the Console, you can show a Python prompt and enter the Python commands. To do that, click  ([FIGURE A5](#)):

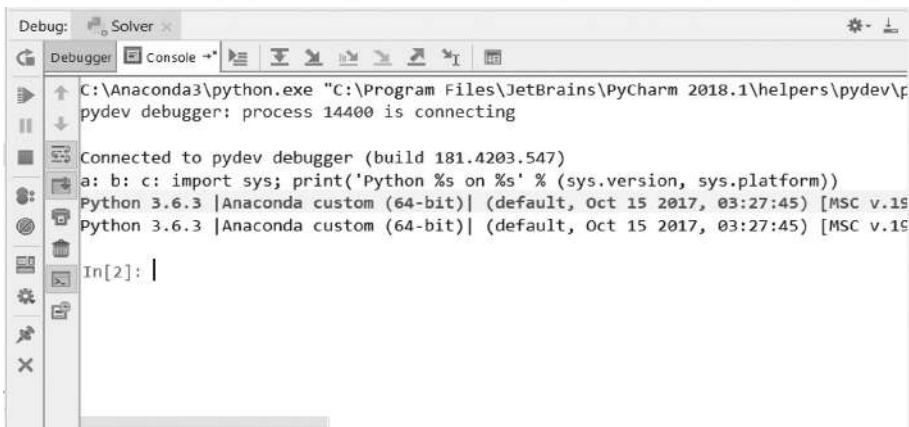


FIGURE A5

Invoke Command prompt during debugging.

Then the debugger suspends the program at the first breakpoint. It means that the line with the breakpoint is not yet executed. The line becomes blue ([FIGURE A6](#)):

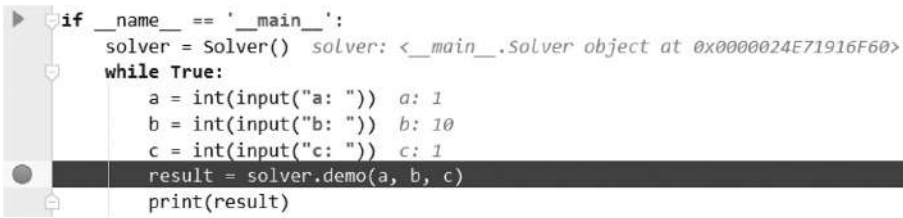


FIGURE A6

Blue marker during debugging.

A4 Inline Debugging

In the editor, you see the grey text next to the lines of code (FIGURE A7):

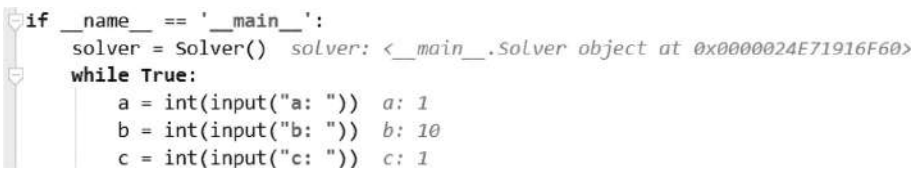


FIGURE A7

Inline debugging.

What does it mean?

This is the result of the so-called inline debugging. The first lines show the address of the *Solver* object and the values of the variables *a*, *b* and *c* you've entered.


The inline values functionality simplifies the debugging procedure, as it lets you view the values of variables used in your source code right next to their usage, without having to switch to the Variables pane of the Debug tool window.


If this option is enabled, when you launch a debug session and step through the program, the values of variables are displayed at the end of the lines where these variables are used.

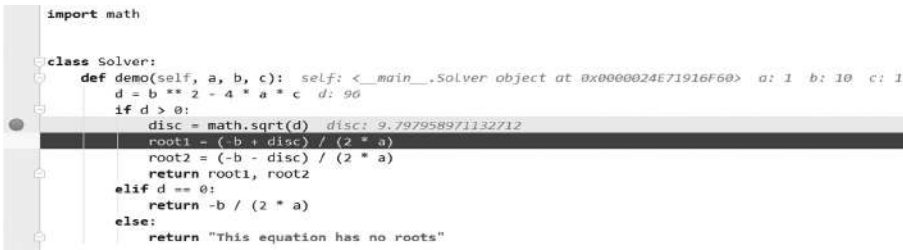
Inline debugging can be turned off. To switch them off, do one of the following:

In the Debug tool window toolbar, click the *Settings* icon  and deselect the *Show Values Inline* option from the pop-up menu.

A5 Let's Step!

So, you've clicked the button , and now see that the blue marker moves to the next line with the breakpoint.

If you use the stepping toolbar buttons, you'll move to the next line. For example, click the button . Since the inline debugging is enabled, the values of the variables show in italic in the editor (FIGURE A8).



```

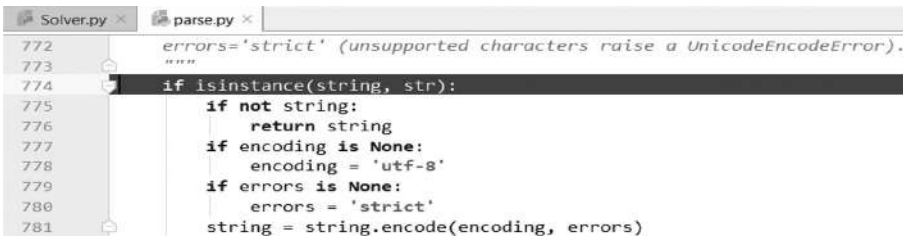
import math

class Solver:
    def demo(self, a, b, c):
        self: <__main__.Solver object at 0x0000024E71916F60> a: 1 b: 10 c: 1
        d = b ** 2 - 4 * a * c d: 96
        if d > 0:
            disc = math.sqrt(d) disc: 9.797958971132712
            root1 = (-b + disc) / (2 * a)
            root2 = (-b - disc) / (2 * a)
            return root1, root2
        elif d == 0:
            return -b / (2 * a)
        else:
            return "This equation has no roots"

```

FIGURE A8
Step Into the code.

If you click the button , you will see that after the line `a = int(input("a: "))` the debugger goes into the file `parse.py` (FIGURE A9):





```

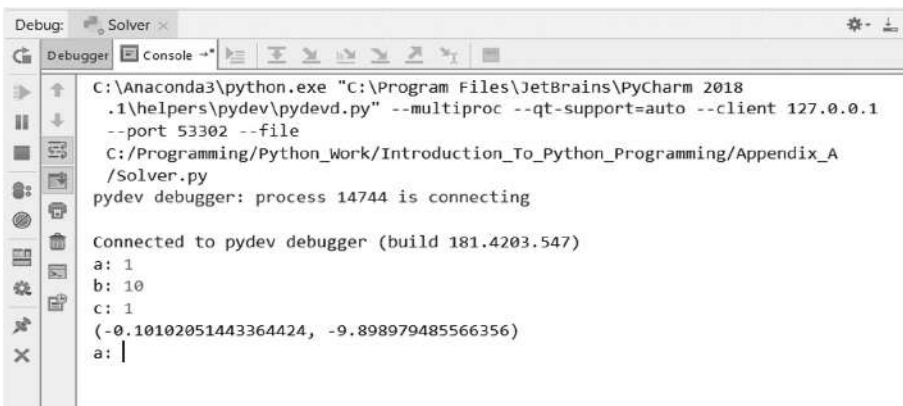
772 errors='strict' (unsupported characters raise a UnicodeEncodeError).
773
774 if isinstance(string, str):
775     if not string:
776         return string
777     if encoding is None:
778         encoding = 'utf-8'
779     if errors is None:
780         errors = 'strict'
781     string = string.encode(encoding, errors)

```

FIGURE A9
Step Into other files.

However, if you continue using the button , you'll see that your application just passes to the next loop and you will see the end result (FIGURE A10):

If you want to concentrate on your own code, use the button *Step Into My Code* () – thus You will avoid stepping into library classes.



```


Debug: Solver
Debugger Console
C:\Anaconda3\python.exe "C:\Program Files\JetBrains\PyCharm 2018
.1\helpers\pydev\pydevd.py" --multiproc --qt-support=auto --client 127.0.0.1
--port 53302 --file
C:/Programming/Python_Work/Introduction_To_Python_Programming/Appendix_A
/Solver.py
pydev debugger: process 14744 is connecting

Connected to pydev debugger (build 181.4203.547)
a: 1
b: 10
c: 1
(-0.10102051443364424, -9.898979485566356)
a: |

```

FIGURE A10
Program result.

A6 Watching

PyCharm allows you to watch a variable. Just click  on the toolbar of the Variables tab, and type the name of the variable you want to watch. Note that code completion is available (FIGURE A11):

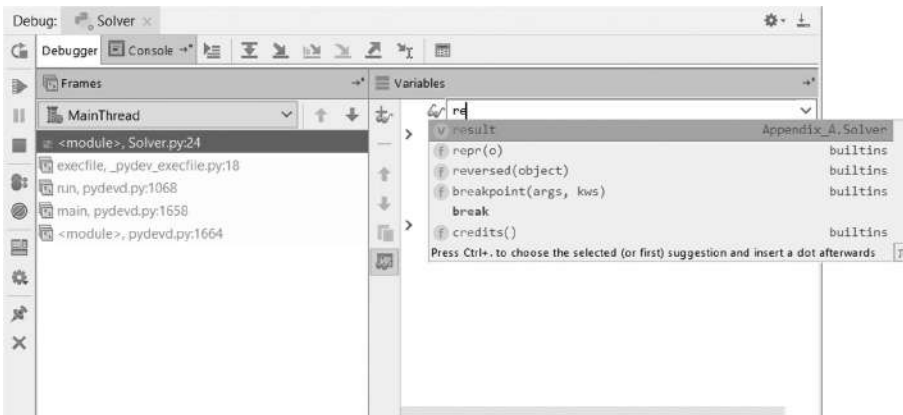


FIGURE A11
Watch program variables.

At first, you see an error – it means that the variable is not yet defined (FIGURE A12):

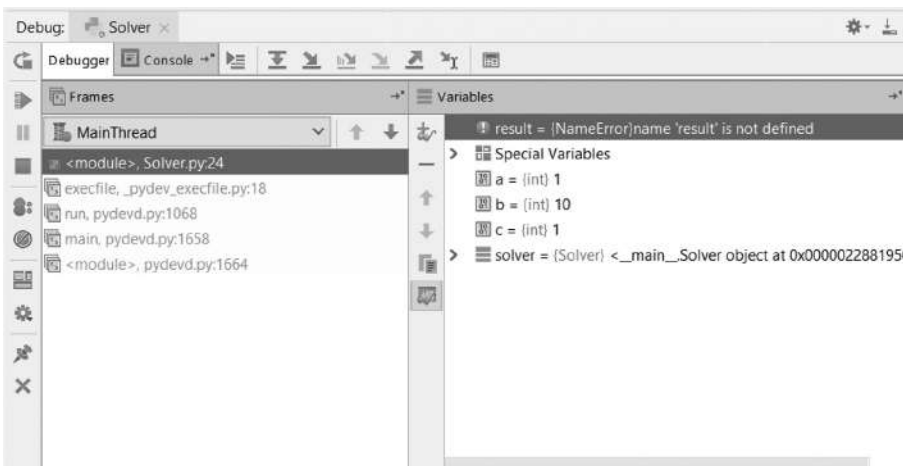


FIGURE A12
Program variable under watch not yet defined.

However, when the program execution continues to the scope that defines the variable, the watch gets the following view (FIGURE A13):

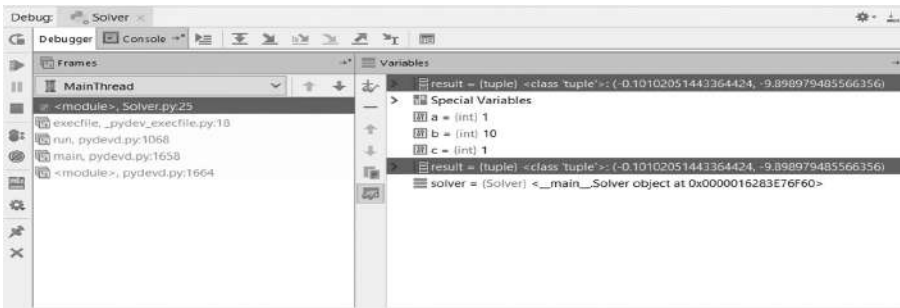



FIGURE A13
View when watch variable gets its value.

A7 Evaluating Expressions

Finally, you can evaluate any expression at any time. For example, if you want to see the value of the variable, click the button , and then in the dialog box that opens, click *Evaluate* (FIGURE A14):

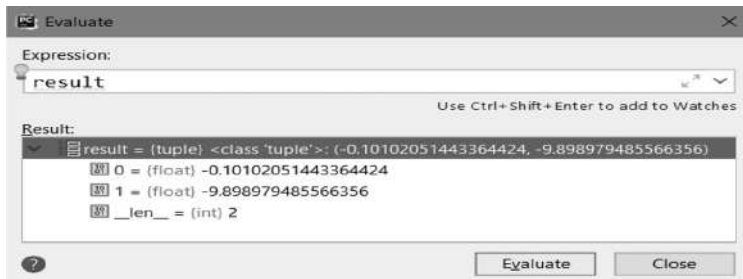


FIGURE A14
Evaluate window to see the value of a variable.

PyCharm gives you the possibility to evaluate any expression. For example (FIGURE A15):

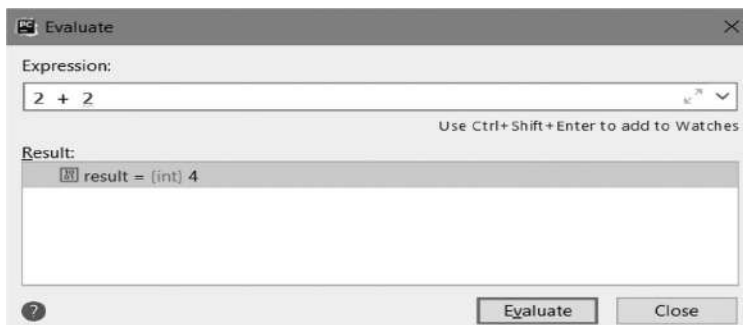

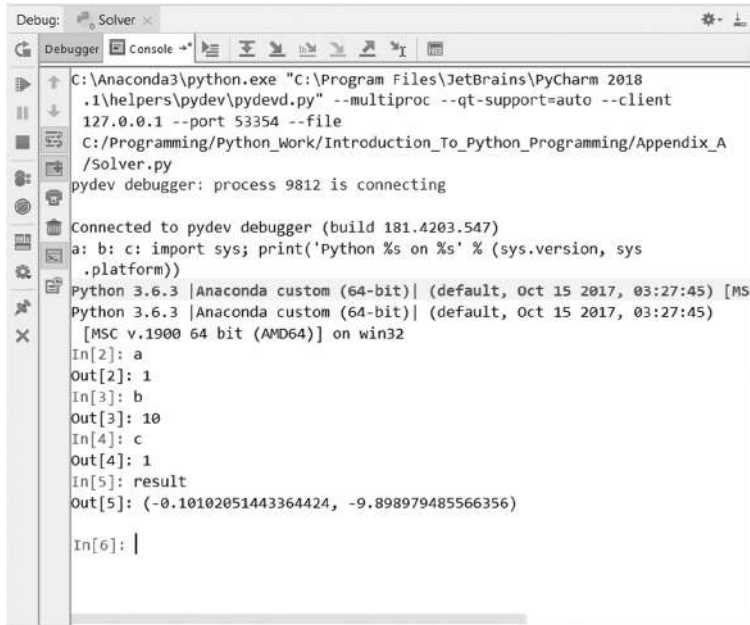


FIGURE A15
Use Evaluate window to evaluate any expression.

You can also click the button  in the Debug console and enter some commands that show the variables values. For example, with Jupyter installed, you can easily get an expression value (FIGURE A16):



```
Debug: Solver
Debugger Console
C:\Anaconda3\python.exe "c:\Program Files\JetBrains\PyCharm 2018
.1\helpers\pydev\pydevd.py" --multiproc --qt-support=auto --client
127.0.0.1 --port 53354 --file
C:/Programming/Python_Work/Introduction_To_Python_Programming/Appendix_A
/Solver.py
pydev debugger: process 9812 is connecting

Connected to pydev debugger (build 181.4203.547)
a: b: c: import sys; print('Python %s on %s' % (sys.version, sys
.platform))
Python 3.6.3 [Anaconda custom (64-bit)] (default, Oct 15 2017, 03:27:45) [MSC
Python 3.6.3 [Anaconda custom (64-bit)] (default, Oct 15 2017, 03:27:45)
[MSC v.1900 64 bit (AMD64)] on win32
In[2]: a
Out[2]: 1
In[3]: b
Out[3]: 10
In[4]: c
Out[4]: 1
In[5]: result
Out[5]: (-0.10102051443364424, -9.898979485566356)

In[6]: |
```

FIGURE A16

Invoke Python prompt to inspect variable values.
(Adapted with kind permission from JetBrains.com.)

A8 Summary

- You have learned how to place breakpoints.
- You have learned how to begin the debugger session, and how to show the Python prompt in the debugger console.
- You have understood the advantage of in-line debugging.
- You have tried hands-on stepping, watches and evaluating expressions.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Bibliography

<https://www.python.org/>
<http://www.numpy.org/>
<https://www.scipy.org/>
<https://pandas.pydata.org/>
<https://altair-viz.github.io/>
<https://matplotlib.org/>
<https://www.statsmodels.org/>
<https://www.jetbrains.com/pycharm/documentation/>
<https://jupyter.org/>
<https://pytorch.org/>
<http://docs.python-requests.org/en/master/>
<http://scikit-learn.org/>
<https://www.numfocus.org/>
<https://spacy.io/>
<https://www.nltk.org/>
<https://micropython.org/>
<http://flask.pocoo.org/>
<https://www.djangoproject.com/>
<https://www.pythonweekly.com/>
<https://www.reddit.com/r/Python/>
<https://developer.mozilla.org/>
<https://msdn.microsoft.com/>
<https://docs.oracle.com/>
<https://developer.ibm.com/>
<https://developers.google.com/>
<https://www.kaggle.com/>
<https://www.techopedia.com>
<http://www.linfo.org>



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Solutions

Chapter 1

Self-Assessment Questions	
Question No.	Answer
1	b
2	a
3	d
4	b
5	b
6	a
7	b
8	a
9	b
10	c

Chapter 2

Self-Assessment Questions	
Question No.	Answer
1	a
2	a
3	a
4	a
5	b
6	a
7	d
8	c
9	c
10	b
11	a
12	b
13	c
14	c

(Continued)

Self-Assessment Questions	
Question No.	Answer
15	b
16	b
17	d
18	b
19	a
20	b
21	a
22	d
23	c
24	a
25	a
26	c
27	d
28	b
29	b
30	a
31	a
32	d
33	d
34	b
35	c
36	d
37	a
38	a

Activity



Activity Type: Offline Duration: 10 Minutes

- Evaluate the expression below by assuming the values given.
 - $x + y / z > 5 * z \ || \ x - y < z \ \&\& \ z >> 5$; Assume $x = 5, y = 4, z = 6$
 - $x * a * a + y * a - z / b > = \&\&c! = 15.0$; Assume $x = 2, y = 3, z = 5$ and $a = 3, b = 1, c = 5$.
- Write a program to find the ASCII code for 1,A, B,a and b using the *ord* function. Use the *chr* function to find the character for the decimal codes 40,59,79,85 and 90.

Chapter 3

Self-Assessment Questions	
Question No.	Answer
1	a
2	b
3	b
4	c
5	c
6	b
7	c
8	a
9	a
10	b
11	d
12	c
13	d
14	c
15	a
16	c
17	a
18	b
19	b
20	b
21	a
22	b

Chapter 4

Self-Assessment Questions	
Question No.	Answer
1	c
2	d
3	c
4	d
5	a
6	a
7	b

(Continued)

Self-Assessment Questions	
Question No.	Answer
8	c
9	d
10	b
11	c
12	c
13	a
14	b
15	a
16	d
17	b
18	a
19	a
20	d
21	b
22	a
23	a

Chapter 5

Self-Assessment Questions	
Question No.	Answer
1	c
2	b
3	c
4	c
5	a
6	c
7	b
8	c
9	b
10	a
11	a
12	a
13	d
14	d
15	a
16	b

Chapter 6

Self-Assessment Questions	
Question No.	Answer
1	d
2	a
3	c
4	d
5	c
6	b
7	b
8	d
9	c
10	d
11	b
12	c
13	d
14	d
15	c
16	a

Chapter 7

Self-Assessment Questions	
Question No.	Answer
1	d
2	b
3	a
4	c
5	b
6	b
7	b
8	c
9	c
10	c
11	c
12	a
13	c

(Continued)

Self-Assessment Questions	
Question No.	Answer
14	b
15	a
16	c
17	a
18	b
19	c
20	a
21	b
22	a
23	d

Chapter 8

Self-Assessment Questions	
Question No.	Answer
1	b
2	a
3	b
4	b
5	a
6	c
7	d
8	a
9	d
10	b
11	c
12	a
13	a
14	c
15	a
16	b
17	d
18	a
19	b
20	c
21	b
22	a
23	d
24	c
25	a

Chapter 9

Self-Assessment Questions	
Question No.	Answer
1	b
2	b
3	a
4	d
5	a
6	b
7	a
8	c
9	d
10	b
11	a
12	c
13	a
14	a
15	a
16	c
17	a
18	a
19	d
20	d
21	b
22	a
23	b

Chapter 10

Self-Assessment Questions	
Question No.	Answer
1	a
2	c
3	d
4	b
5	c
6	a

(Continued)

Self-Assessment Questions	
Question No.	Answer
7	b
8	b
9	d
10	b
11	c
12	d
13	c
14	b
15	d
16	a
17	c
18	b
19	b
20	d
21	d
22	a

Chapter 11

Self-Assessment Questions	
Question No.	Answer
1	a
2	b
3	c
4	a
5	d
6	a
7	d
8	d
9	d
10	a
11	b
12	a
13	d
14	b
15	c
16	d
17	c

Chapter 12

Self-Assessment Questions	
Question No.	Answer
1	a
2	c
3	d
4	a
5	a
6	a
7	b
8	d
9	c
10	d
11	a
12	a
13	b
14	d
15	b
16	a
17	a
18	c



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Index

Note: Page numbers in *italic* and **bold** refer to figures and tables respectively.

- abs()* function **96**
- absolute path, file **232**
- abstraction **307, 308**
- addition assignment operator (+)= **41**
- addition operator (+) **39**
- add()* method **218**
- aggregate functions **393, 400**
- all()* function **155, 180**
- Altair **398–409**
- “a+” mode, file **234**
- “a” mode, file **234**
- Anaconda (Python distribution) **13–15**
- AND operator: binary (&) **45**; logical (and) **43**
- any()* function **155, 180**
- Apache License **29–30**
- Apache Software Foundation **29**
- append()* method **157**
- application development **5**
- application software **2**
- *args **110–12**
- arithmetic operators **39, 39–40**
- array **348**
- array, NumPy **359–60, 360**; arithmetic
 - operations on **367–8**; attributes **361, 361–2**; broadcasting in **371–4**; creation functions **362–3**; indexing/slicing/iterating operations **364–7**; with initial placeholder content **362–4**; shape changing **369–70**; stacking and splitting **370–1**; using *array()* function **360–1**
- arrow* module **99**
- as* keyword **237**
- assembler **3**
- assembly language **3–4**
- assignment operators **37, 40–2, 41**
- associative memories/arrays **175**
- associativity, operator **47–8**
- attribute references **293**
- attributes **355**
- axes (NumPy arrays) **359**
- backslash (\) character **141**
- bar chart **402**
- base class **311, 311–12**
- Benevolent Dictator For Life (BDFL) **7**
- binary **45**; left shift operator (<<) **45, 46**; ones complement operator (~) **45**; AND operator (&) **45**; OR operator (|) **45**; right shift operator (>>) **45**; XOR operator (^) **45**
- binary files **230**; reading and writing **247–9**
- bit **45**
- bitwise logical operators **46**
- bitwise operators **44–7, 45**
- bitwise truth table **45**
- Boolean **48**; array indexing **364–7**; logic truth table **43**
- breakpoints **416, 417**
- break* statements **81–4**
- broadcasting, NumPy array **371–4**
- BSD license **30**
- bubble sort **160–1, 161**
- built-in functions **95–6, 96**; *dict()* **179**; for dictionaries **179–81, 180**; *iter()* **342–3**; *list()* **151–2**; for list **155, 155–6**; for string **122, 122–3**; *super()* **314**; *tuple()* **204–5**; for tuple **207–8, 208**
- bytes()* class method **249**
- C3 linearization algorithm **320–1**
- calling function **100**; objects as arguments in **301–3**
- capitalize()* method **132**
- capturing parentheses/group **272**
- carriage return (CR or \r) **244**
- casefold()* method **132**
- center()* method **132**
- chain comparison operation **130–1**
- character: encoding **142**; reserved **349**
- chart **399**
- child class **311**
- chr()* function **56**
- class(es) **291, 307**; attributes *versus* data
 - attributes **306–7**; creation in Python **291–2**; instance of **294**; local precedence of **320**; with multiple objects **297–306**; OOP **289–91**
- clear()* method **181, 218**
- client/server architecture **346, 346–7**

- `close()` method 235–6
- cloud computing 11–12
- command line arguments 112–13
- Comma Separated Values (CSV) files 251, 251–7; characteristics 251–2
- comments 50
- community, Python 12
- comparison operators 42, 42–3, 204, 332
- compilation 3
- compiled regular expression objects 274
- compiler 3–4
- `complex()` function 56
- complex numbers 48
- compound assignment operator 40
- `concat()` function 395
- constructor method 294–7
- containers 343
- context management protocol 238
- context manager 238
- `continue` statements 81–4
- control flow statements 67; *break* statements 81–4; *continue* statements 81–4; decision 67; forms of 68; loop 67; sequential 67
- control sequences 141
- conversion operations 332
- cooperative multiple inheritances 325
- `count()` method 132, 157, 210
- Creative Commons licenses 31–2
- Crockford, Douglas 347
- cross compiler 4
- CSV files *see* Comma Separated Values (CSV) files
- csv module 252
- `csv.reader()` method 252
- `csv.writer()` method 252
- current directory 232
- data attributes 293; class attributes *versus* 306–7
- database connectors 11
- DataFrame, pandas 380; *assign()* method 386; column selection/addition/deletion 382–4; concatenate, append and merge 392–7; creation 380–2; data, displaying 384–6; data input and output 398; “group by” process 389–92; indexing and data selection 386–9; missing data 397–8
- data science 341
- data types 48–9
- debugger 415
- debugging 415; blue marker during 419; breakpoints, placing 416, 417; debugger session 417–19; expressions, evaluating 422–3; inline 419, 419
- decision control flow statements 67; *if* 68; *if...elif...else* 71–3; *if...else* 69–71; nested *if* 73–4
- default parameters 108–9
- `def` keyword 99–100
- `del` statement 169–70, 193
- derived class 311, 311–12; C3 linearization 320
- deserializing, Python 349, 350
- `dict()` function 179, 209
- dictionaries 175; built-in functions for 179–81, 180; creation 175–7; defined 175; keys 176; *key:value* pairs, accessing/modifying 178–9; methods 181, 181–93; nested 192; populating with *key:value* pairs 183–5; structure 233; traversing 185–6; *versus* tuple 209–10
- DictReader 253
- dict type 177
- DictWriter 253
- `difference()` method 218
- directories 230
- `dir()` function 98, 131, 156, 181, 210, 218
- `discard()` method 218
- division assignment operator (*/=*) 41
- division operator (*/*) 39
- `divmod()` function 96
- Django 11
- `.iloc[]` method 387, 389
- `.loc[]` method 387
- duck-typing 328
- `dump()` method 250, 353
- dynamic language 58–9
- elements: NumPy arrays 359; XML 355
- empty dictionary 177
- empty list 150
- empty set 216
- empty strings 120
- empty tuple 202
- encapsulation 307–11, 308
- End-of-Line (EOL) character 231
- `endswith()` method 132
- equal sign (*=*) 37
- “equal to” operator (*==*) 42
- Escape Sequences 141–2
- exception handling 84–5; *try...except...finally* statements 85–9
- exponentiation assignment operator (***=*) 41

- exponent operator (**) **39**
- expressions, Python **36**
- extend()* method **157**
- EXtensible Markup Language (XML) **346**, **355–8**; *versus* JSON **359**
- Extensible Stylesheet Language Transformations (XSLT) **359**
- file(s): access modes **234**; *close()* method **235–6**; defined **229**; extension/suffix **229**; fully qualified path **232–3**; handler **233**, **239**; methods **239–40**, **239–47**; object attributes **239**; paths **231–2**; relative path **232–3**; structure **233**; types of **230–3**; *with* statements **237**
- filename extension **229**
- find()* method **132**
- Flask (web framework) **11**
- float()* function **55**
- floating point number **48**
- floor division assignment operator (*//*=) **41**
- floor division operator (*//*) **39**
- for* loop **79–81**
- format()* method **51–2**
- format specifiers **140**
- formatted strings *see* **f-strings (formatted strings)**
- FP *see* **Functional Programming (FP)**
- fromkeys()* method **181**
- frozenset **221–2**
- f-strings (formatted strings) **53–4**, **138–40**; escape sequences **141–2**; format specifiers **140**; raw string **142**; unicodes **142–3**
- fully qualified path, file **232–3**
- function(s) **51**, **95**; *see also* **method(s)**; *abs()* **96**; aggregate **393**, **400**; *all()* **155**, **180**; *any()* **155**, **180**; **args* and ***kwargs* in **110–12**; built-in *see* **built-in functions**; call/calling **100**; *chr()* **56**; *complex()* **56**; *concat()* **395**; definition **99–103**; *dir()* **98**, **131**, **156**, **181**, **210**, **218**; *divmod()* **96**; *float()* **55**; generator **343–4**; header **100**; *help()* **98**; *hex()* **57**; *id()* **303**; *input()* **50–2**; *int()* **54–5**; *isinstance()* **303–4**; *issubclass()* **317**; *lambda* **341–2**; *len()* **96**, **122**, **155**, **180**, **208**; Magic Methods for **331–2**; *main()* **101**; *max()* **96**; *merge()* **396**; *min()* **96**; *np.array()* **360–1**; *oct()* **57–8**; *open()* **233–4**, **239**; *ord()* **57**; *pow()* **96**; *print()* **51–4**; *random()* **98**; *random.randint()* **98**; *range()* **79**, **345**; *sorted()* **155**, **180**, **208**; *str()* **55–6**, **120**; *sum()* **155**, **208**; *type()* **58**, **120**; user-defined **99**; *void* **103–6**; *zip()* **216**
- Functional Programming (FP) **341**; generators **343–4**; iterators **342–3**; *lambda* function **341–2**; languages **5**; list comprehensions **344–6**
- generators **343–4**
- get()* method **181**, **188**
- global variables **106–8**
- glob module **282–3**
- GNU General Public License (GNU GPL) **30–1**
- “greater than” operator (*>*) **42**
- “greater than or equal to” operator (*>=*) **42**
- “group by” process **389–92**
- grouped bar chart **404**
- hash (#) symbol **50**
- heatmaps **408**
- help()* function **98**
- hex()* function **57**
- high-level language **3**; advantages **3–4**; functional **5**; imperative **4**; logical **4–5**; object-oriented **5**
- histogram **409**
- Hypertext Transfer Protocol (HTTP) **11**, **346**
- identifiers **35–6**
- id()* function **303**
- if...elif...else* statement **71–3**
- if...else* statement **69–71**
- if* statement **68–9**; *see also* **nested if statement**
- immutable string **128**
- imperative programming **4**
- import* statement **97**
- indentation **49**, **49–50**; in tuple **205–7**
- IndentationError* **50**
- index **152**, **205**, **375**
- indexing: array **364–7**; in lists **152–3**; pandas DataFrame **386–9**; in string **123–4**
- index()* method **157**, **210**
- index number, string **123–4**
- inheritance **311–12**; inherited variables/methods, accessing **312–14**; MRO **320–8**; multiple **317–19**; *super()* function in **314–17**
- __init__()* method **294**
- inline debugging **419**, **419**
- inner merge operation, DataFrame **396**, **397**
- input()* function **50–2**
- insert()* method **157**
- instance of class **294**

- instance variables 293, 309–11
- instantiation 293
- integer indexing 364–7
- integers 48
- Integrated Development Environments (IDEs) 12; PyCharm 12, 16–19
- interpreter 4
- intersection()* method 218
- int()* function 54–5
- isalnum()* method 132
- isalpha()* method 132
- isdecimal()* method 132
- isdigit()* method 132
- isdisjoint()* method 218
- isidentifier()* method 132
- isinstance()* function 303–4
- islower()* method 132
- isnumeric()* method 132
- isspace()* method 132
- issubclass()* function 317
- issubset()* method 218
- issuperset()* method 218
- istitle()* method 133
- isupper()* method 133
- item 149, 153–5, 201
- items()* method 181, 209
- iterable 342
- iteration: in arrays 364–7; statements *see* loop control flow statements
- iterators 342–3
- iter()* function 342–3
- JavaScript Object Notation (JSON) 346–53, 347; *versus* XML 359
- join()* method 127
- json* module 349
- Jupyter 10; dashboard 25
- Jupyter Notebook 23–7; in edit mode 27; Python code in 27; renaming 26; starting command to 24
- KeyError* 181
- keys()* method 181
- key:value* pairs 175, 176, 209; accessing/modifying 178–9
- keyword arguments 52, 109–10
- keywords 36, 36
- **kwargs** 110–12
- lambda function 341–2
- left merge operation, DataFrame 396, 396
- left shift operator (<<) 45, 46
- len()* function 96, 122, 155, 180, 208
- “lesser than” operator (<) 42
- “lesser than or equal to” operator (<=) 42
- linear search 162
- line chart 405
- linefeed (LF or NL or \n) 244
- list 149; built-in functions for 155, 155–6; comprehensions 344–6; creation 149–50; indexing and slicing in 152–5; items modifying in 153–5; methods 156–69, 157; mutable 153; nested 167–9; operations 151–2; populating with items 158–9; slicing of 154; traversing of 159; *versus* tuple 208–9
- list()* function 151–2
- literals 49
- ljust()* method 133
- load()* method 250, 352
- local precedence of classes 320
- local variables 106–8
- logical AND operator (and) 43
- logical NOT operator (not) 43
- logical operators 43, 43–4
- logical OR operator (or) 43
- logical paradigm 4–5
- loop control flow statements 67; iteration of 75; *for* loop 79–81; *while* loop 74–8
- lower()* method 133
- lstrip()* method 133
- machine code 2
- machine language 2
- Machine Learning 10
- Magic Methods 331–2, 331–2
- main()* function 101
- match objects 274–84, 275
- math* module 97
- Matplotlib 10
- max()* function 96, 122
- merge()* function 396
- metacharacters *see* special characters
- method(s) 291; *see also* function(s); attributes 293; *bytes()* class 249; *close()* 235–6; constructor 294–7; *csv.reader()* 252; *csv.writer()* 252; dictionaries 181, 181–93; *.iloc[]* 387, 389; *.loc[]* 387; *dump()* 250, 353; files 239–40, 239–47; *format()* 51–2; *__init__()* 294; *join()* 127; list 156–69, 157; *load()* 250, 352; Magic Methods 331–2; *mro()* 320; overriding 314; *re.compile()* 273–4; sets 218, 218–21; signature 314; *split()* 127–8, 159; *str.format()* 51–3, 139; string 131–8, 132–3; tuple 210, 210–15; *writerow()* 253; *writerows()* 253

- Method Resolution Order (MRO) 320–8
- min()* function 96, 122
- MIT license 31
- mnemonics 3
- modules 97–9
- modulus operator (%) 39
- MRO (Method Resolution Order) 320–8
- mro()* method 320
- multiline comment 50
- multiple inheritance 317–19
- multiplication assignment operator (*=) 41
- multiplication operator (*) 39
- multi-way decision control statement 71
- name mangling 309–10
- native-code compiler 4
- Natural Language Toolkit (NLTK) 10
- ndarray* 359
- negative indexing 124–5, 206
- nested dictionaries 192
- nested function 107–8
- nested *if* statement 73–4; *see also if* statement
- nested list 167–9
- NLTK (Natural Language Toolkit) 10
- none (data type) 49, 103
- “not equal to” operator (!=) 42
- NOT operator (not), logical 43
- np.array()* function 360–1
- numbers (data types) 48
- NumPy (Numerical Python) 10, 359–60; array
 - see array, NumPy*; mathematical functions in 368–9
- object(s) 293; as arguments 301–3; class 312;
 - creation in Python 293–4; in JSON 349;
 - OOP 289–91; as return values 303–6
- object-oriented programming (OOP) 5, 289;
 - classes 289–91; encapsulation 307–11;
 - inheritance *see inheritance*; method overriding in 314; objects 289–91;
 - polymorphism 328–35
- Object Role Modeling (ORM) 11
- oct()* function 57–8
- ones complement operator () 45
- OOP *see object-oriented programming (OOP)*
- open()* function 233–4, 239
- open source software 27–8; Apache License
 - 29–30; BSD license 30; Creative Commons licenses 31–2; GNU GPL 30–1; licenses 29–32; MIT license 31;
 - purpose of 28–9
- OpenStack 11–12
- operands 38
- operations: chain comparison 130–1; list 151–2;
 - string 120–3
- operator(s) 38–9; arithmetic 39, 39–40;
 - assignment 40–2, 41; associativity 47–8; bitwise 44–7, 45; comparison 42, 42–3, 204; logical 43, 43–4; Magic Methods for 331–2; precedence 47, 47–8; subscript 123
- Operator Overloading 331–5
- OrderedDict 253
- ord()* function 57
- ORM (Object Role Modeling) 11
- OR operator: binary (|) 45; logical (or) 43
- os* module 257–61, 258, 259
- os.path* module 257–61, 258–9, 259
- outer merge operation, DataFrame 396, 397
- packing, tuple 211
- pandas 10, 374; DataFrame 380–98; Series 375–80
- parent class 311
- parent directory 233
- parentheses in REs 272–3
- parsing errors 84
- pass* statement 136
- %-formatting 139
- pickle module 249–50
- pickling 250
- pip* command 98, 399
- polymorphism 328–31; Operator Overloading 331–5
- popitem()* method 181
- pop()* method 157, 169, 181, 218
- populating: dictionaries with *key:value* pairs
 - 183–5; list with items 158–9; tuple with items 212–13
- positional arguments 52
- pow()* function 96
- precedence, operator 47, 47–8
- print()* function 51; f-strings 53–4; *str.format()* method 51–3
- private instance variables/methods 309–11
- procedural programming 4
- program 1–2
- programming languages 2; assembly language 3; high-level language 3–5; machine language 2; ranking of 9
- programming paradigm 4
- programming software 2
- prompt* statement 51
- proprietary software 28
- PSF (Python Software Foundation) 1, 12
- PTVS (Python Tools for Visual Studio) 12

PyCharm 12, 16–19, 415

Python: for academia 9–10; Anaconda 13–15; classes creation in 291–2; cloud computing 11–12; code in Jupyter Notebook 27; comments 50; community 12; creating and running 19–23; data analysis 10; database connectors/ORM/NoSQL connectors 11; data types 48–9; debugging in 415; defined 1, 8; distributions 12; duck-typing in 328; as dynamic language 58; expressions 36; history of 7–8; HTTP library 11; IDE available 12; identifier 35–6; indentation 49, 49–50; *input()* function 50–1; JSON and XML 346–59; keywords 36, 36; logo 7; Machine Learning 10; *versus* Matlab 9; NLTK 10; objects creation in 293–4; operators *see* *operator(s)*; *print()* function 51–4; PyCharm 12, 16–19; scientific tools 10; stack in industry 12–13; statements 36; *see also* *statements*, *Python*; statistics 11; as strongly typed language 58–9; thrust areas of 8–13; type conversions 54–8; variables 37–8; versions 8; web frameworks 11

Python 3.6 version 13, 53, 177

Python Software Foundation (PSF) 1, 12

Python Tools for Visual Studio (PTVS) 12

random() function 98

random.randint() function 98

range() function 79, 345

Raspberry Pi 9–10

“rb+” mode, file 234

“rb” mode, file 234

readline() method 239

readlines() method 239

read() method 239, 240–1

re.compile() method 273–4

regexes/regex patterns 267

regular expressions (REs): backreference in 272; compiling using *compile()* method 273–4; defined 267; with *glob* module 282–3; match objects 274–84, 275; methods 273–81; named groups in 282; parentheses in 272–3; pattern 267; *r* prefix for 272; special characters in 267–73

relative path, file 232–3

remainder assignment operator (%) 41

remove() method 157, 218

repetition statements *see* *loop control flow statements*

replace() method 133

Requests HTTP library 11

requests module 353–4

REs *see* *regular expressions (REs)*

reserved characters 349

return statement 103–6

reverse() method 157

right merge operation, DataFrame 396, 397

right shift operator (>>) 45

rjust() method 133

“r+” mode, file 234

“r” mode, file 234

root directory 232

r prefix for REs 272

rstrip() method 133

run-time errors 85

scatterplot 406

scientific tools 10

Scikit-Learn 10

SciPy library 10

scope of variables 106–8

SDLC (Software Development Life Cycle) 5, 5–6

seek() method 240, 243–4

separator 128

sequential control flow statements 67

serializing, Python 349, 350

Series, pandas 375–80

setdefault() method 181

sets 216–17; methods 218, 218–21; traversing of 219–20

simple assignment operator 40

single line comment 50

\b escape sequence 142

\n escape sequence 142

\t escape sequence 142

slicing: array 364–7; DataFrame 389; list 154; string 124–7; in tuple 205–7

software 1–2; development 5–6; open source 27–32

Software Development Life Cycle (SDLC) 5, 5–6

sorted() function 155, 180, 208

sort() method 157

source program/code 3, 27

special characters 267–73

splitlines() method 133

split() method 127–8, 159

startswith() method 133

statements, Python 36; *break* 81–4; *continue* 81–4; control flow 67; decision control flow 67; *del* 169–70, 193; *if* 68–9; *if...elif...else* 71–3; *if...else* 69–71; loop control flow 67; nested *if* 73–4; *pass* 136; *prompt* 51; *return* 103–6; sequential control flow 67; *with* 237

Statsmodels 11

str.format() method 51–3, 139

str() function 55–6, 120

string 48–9, 119; built-in functions for 122, 122–3; comparison 122; concatenation 120–1; creating and storing 119–20; empty 120; formatting 138–43; immutable 128; index number, characters accessing by 123–4; literal 48; methods 131–8, 132–3; operations 120–3; slicing and joining 124–7; splitting 127–8; traversing 128–9

strongly typed language 58–9

structured programming 4

subclass 311

subscript operator 123

substring 125

subtraction assignment operator (*=*) 41

subtraction operator (*-*) 39

sum() function 155, 208

superclass 311

super() function 314–17, 325

swapcase() method 133

Symbolic Programming Language 3

symmetric_difference() method 218

SymPy library 10

syntax: base class 312; *bytes()* class method 249; class instantiation 293; *close()* function 235; compiled regular expression objects 274; *concat()* function 395; *data* attribute 293; *derived* class 312; *dict()* function 179; dictionary creation 176; dictionary methods 181; *dump()* method 353; *dumps()* method 353; empty dictionary 177; empty list 150; empty tuple 202; errors 84; file methods 239–40; *format()* method 52; f-string formatting 140; function call/calling function 100; function definition 99; glob method 282; *id()* function 303; *if...elif...else* statement 71; *if...else* statement 70; *if* statement 68; *import* statement 97; *input()* function 50; *isinstance()* function 303–4; *issubclass()* function 317; *join()* method 127; *key:value* pairs,

accessing/modifying 178–9; lambda function 342; list creation 150; list item, accessing 152; list methods 157; list slicing 154; *loads()* method 352; *for* loop 79; *merge()* function 396; method attribute 293; *mro()* method 320; named groups in REs 282; nested *if* statement 73; nested lists 167; *open()* function 233; *os* module 258; *os.path* module 258–9; *random.randint()* function 98; *range()* function 79; *return* statement 103; set methods 218; *split()* method 127–8; *str()* function 120; string characters accessing 123; string methods 132–3; string slicing 124; *super()* function 314; *try...except...finally* statement 86; *tuple()* function 204; tuple item, accessing 205; tuple methods 210; tuples creation 201; tuple slicing 206; *type()* function 58; *while* loop 74; *with* statements 237; XML document 355–6

sys.argv 112

sys.modules 257

system software 2

tags, XML 355

tell() method 240, 244

text data 233–9

text files 230–1; creating and opening 233–5

TIOBE Programming Community Index 9

title() method 133

transpose of matrix 168

traversing: dictionaries 185–6; list 159; sets 219–20; string 128–9; tuple 211

try...except...finally statements 85–9

tuple: built-in functions for 207–8, 208; comparison operators, use of 204; creation 201–3; defined 201; *versus* dictionaries 209–10; indexing and slicing 205–7; *versus* lists 208–9; methods 210, 210–15; operations 203–5; packing 211; populating with items 212–13; traversing 211; unpacking 104, 211

tuple() function 204–5

type conversion functions 54; *chr()* 56; *complex()* 56; *float()* 55; *hex()* 57; *int()* 54–5; *oct()* 57–8; *ord()* 57; *str()* 55–6

type() function 58, 120

unary operators 332

unicodes 142–3

Unicode Standard 142–3

Unicode Transformation Formats [143](#)

union() method [218](#)

unpacking, tuple [211](#)

unpickling [250](#)

update() method [181](#), [183](#), [218](#)

upper() method [133](#)

urllib.request [11](#)

user-defined functions [99](#)

value [36](#)

ValueError exception [87](#)

values() method [181](#)

van Rossum, Guido [7](#), [7](#)

variables, OOP [290–1](#)

variables, Python [37](#); legal variable names [37](#);
scope and lifetime [106–8](#); values to,
assigning [37–8](#)

Vega-Lite [398](#); online editor [403](#)

void function [103–6](#)

“wb” mode, file [234](#)

web frameworks [11](#)

while loop [74–8](#)

Windows [10](#) OS: Anaconda installation in
[13–15](#); PyCharm installation in
[16–19](#)

with statements [237–9](#)

“w+” mode, file [234](#)

“w” mode, file [234](#)

writelines() method [240](#)

write() method [239](#), [242–3](#)

writerow() method [253](#)

writerows() method [253](#)

XML *see* [EXtensible Markup Language \(XML\)](#)

“x” mode, file [234](#)

XOR operator (^) [45](#)

XSLT (Extensible Stylesheet Language
Transformations) [359](#)

The Zen of Python [8](#)

ZeroDivisionError exception [87–8](#)

zfill() method [133](#)

zip() function [216](#)