

PYTHON PROGRAMMING

About the Authors



Ashok Namdev Kamthane is a retired Associate Professor of the Department of Electronics and Telecommunication Engineering, S. G. G. S. Institute of Engineering and Technology, Nanded, Maharashtra, India. An academic with 37 years of teaching experience, he has authored more than a dozen books and presented several technical papers at national and international conferences. He has earned a first class in ME (Electronics) from S. G. G. S. College of Engineering and Technology. His ME dissertation work from Bhabha Atomic Research Centre, Trombay, Mumbai, was on development of the hardware and software using 8051 (8-bit microcontroller) Acoustic Transceiver System required in submarines.



Amit Ashok Kamthane is a Software Engineer (Python Developer) at HCL, Pune. Recently in 2017, he worked as Research Assistant at National Centre for Aerospace Innovation and Research, IIT Bombay. In the past, he was associated as a lecturer with S. G. G. S. Institute of Engineering and Technology, Nanded and as an Assistant Professor with P. E. S. Modern College, Pune. He completed his ME (Computer Science and Engineering) from M. G. M. College of Engineering and BE (Computer Science and Engineering) in first class from G. H. Raisoni College of Engineering, Pune. A computer programming enthusiast, he also imparts corporate training.

PYTHON PROGRAMMING

Ashok Namdev Kamthane

Retired Associate Professor

*Department of Electronics and Telecommunication Engineering
Shri Guru Gobind Singhji Institute of Engineering and Technology, Nanded
Maharashtra, India*

Amit Ashok Kamthane

Software Engineer

(Python Developer)

HCL, Pune

Maharashtra, India



McGraw Hill Education (India) Private Limited

CHENNAI

McGraw Hill Education Offices

Chennai New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto



McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited
444/1, Sri Ekambara Naicker Industrial Estate, Alapakkam, Porur, Chennai 600 116

Python Programming

Copyright © 2018 by McGraw Hill Education (India) Private Limited.

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,
McGraw Hill Education (India) Private Limited.

1 2 3 4 5 6 7 8 9 22 21 20 19 18

Printed and bound in India.

Print Edition

ISBN (13): 978-93-5316-092-0

ISBN (10): 93-5316-092-8

E-Book Edition

ISBN (13): 978-93-5316-096-8

ISBN (10): 93-5316-096-0

Director—Science & Engineering Portfolio: *Vibha Mahajan*
Senior Portfolio Manager—Science & Engineering: *Hemant K Jha*
Associate Portfolio Manager: *Vaishali Thapliyal*

Production Head: *Satinder S Baveja*

General Manager—Production: *Rajender P Ghansela*
Manager—Production: *Reji Kumar*

Information contained in this work has been obtained by McGraw Hill Education (India), from sources believed to be reliable. However, neither McGraw Hill Education (India) nor its authors guarantee the accuracy or completeness of any information published herein, and neither McGraw Hill Education (India) nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that McGraw Hill Education (India) and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at NuWave eSolutions Pvt. Ltd., New Delhi-110019 and printed at

Cover Design: APSCompugraphics

Cover Image Source: *Shutterstock*

Cover Printer:

Visit us at: www.mheducation.co.in

Write to us at: info.india@mheducation.com

CIN: U22200TN1970PTC111531

Toll Free Number: 1800 103 5875

Dedicated to

*Sow Surekha Ashok Kamthane
(Mother of Amit Ashok Kamthane)*



Preface

It gives us immense pleasure to bring the book '*Python Programming*'. The book is intended for the students in initial years of engineering and mathematics who can use this high-level programming language as an effective tool in mathematical problem solving. Python is used to develop applications of any stream and it is not restricted only to computer science.

We believe that anyone who has basic knowledge of computer and ability of logical thinking can learn programming. With this motivation, we have written this book in a lucid manner. Once you go through the book, you will know how simple the programming language is and at the same time you will learn the basics of Python programming. You will feel motivated enough to develop applications using Python.

Since this book has been written with consideration that reader has no prior knowledge of Python programming, before going through all the chapters, reader should know what are the benefits of learning Python programming. Following are some of the reasons why one should learn Python language.

- Python language is simple and easy to learn. For example, it has simple syntax compared to other programming languages.
- Python is an object-oriented programming language. It is used to develop desktop, standalone and scripting applications.
- Python is also an example of free open-source software. Due to its open nature, one can write programs and can deploy on any of platforms, i.e. (Windows, Linux, Ubuntu and Mac OS), without changing the original program.

Thus, due to the features enlisted above, Python has become the most popular language and is widely used among programmers.

Use of Python in Engineering Domains

Computer Engineering

Python is used in computer engineering.

- To develop web applications
- By data scientists to analyse large amount of data

- In automation testing
- To develop GUI-based applications, cryptography and network security and many more applications

Electronics and Telecommunication Engineering and Electrical Engineering

- Image processing applications can be developed by using Python's 'scikit-image' library
- Widely used in developing embedded applications
- Develop IOT applications using Arduino and Raspberry pi

Python can also be used in **other engineering streams** such as mechanical, chemical, and bioinformatics to perform complex calculations by making use of numpy, scipy and pandas library.

Thus, the end user of this book can be anyone who wants to learn basics of Python programming. To learn the basics, the student can be of any stream/any engineering/Diploma/BCA/MCA background and interested to develop applications using Python.

Organization of the Book

Unit 1 contains two chapters and gives the information on history of Python, Basics of Python programming and installation of the software on different operating systems, such as, Windows and Ubuntu. In Chapters 1 and 2, the keywords and variables are described as supported by Python along with input and output functions, `eval()` function and so on.

Unit 2 comprises three chapters. Chapter 3 explains operators and expressions. Various operators as supported by Python are narrated with programming examples in this chapter. Chapter 4 deals with decision-making statements, such as, `if`, `if-elif-else` and others and Chapter 5 describes the control-loop statements, such as, `for`, `while`, `break`, `continue` and others.

Unit 3 covers two chapters, i.e. Chapter 6 on Lists and Chapter 7 on Tuples, Sets and Dictionaries. The processes of creating lists, slicing lists, the operators to be used in lists, various in-built functions of the lists, comprehension etc. are illustrated with easy programming examples in Chapter 6. Whereas in Chapter 7, it covers the methods of creating tuples, in-built functions for tuples, creating sets and operations on sets and handling dictionaries etc.

Unit 4 consists two chapters, i.e. Chapter 8 dealing with functions and Chapter 9 dealing with modules, packages and introduction to PIP Package Management System.

Unit 5 deals with the object-oriented programming concepts and the exception handling. Chapter 10 explains class, objects and inheritance whereas the Chapter 11 discusses the errors and exception and handling the exceptions with further detail.

Unit 6 contains three chapters. Chapter 12 narrates standard library, Chapter 13 contains graphics and programming and Chapter 14 discusses the concept of unit-testing.

ASHOK NAMDEV KAMTHANE
AMIT ASHOK KAMTHANE

McGraw Hill Education (India) invites suggestions and comments from you, all of which can be sent to info.india@mheducation.com (Kindly mention the title and author name in the subject line). Piracy-related issues may also be reported.



Acknowledgements

We would like to express deep sense of gratitude to Professor B. M. Naik, former Principal of S. G. G. S. College of Engineering and Technology, Nanded, who constantly praised and inspired us to write books on technical subjects and whose enthusiasm and guidance led us to write this book.

Special thanks are also due to **Dr. Y. V. Joshi**, Director, S. G. G. S. Institute of Engineering and Technology, Professor **Dr. U. V. Kulkarni**, HOD, CSE of S. G. G. S. Institute of Engineering and Technology, Nanded for encouraging us to write this book on Python. We are grateful to the Board of Directors of Matoshri Group of Institutes, **Kamaji Pawar, V. V. Chari** and **Dean Dr. Sadhna Chidrawar** who encouraged to write this book. We also acknowledge compliments offered by **Mr. Asrar Khan, Mr. Girish Mulke, Mr. Katakbound, Mr. Amol Patil** and **Mr. Satish Pawar** for authoring first book on Python Programming and that too through International Publisher McGraw Hill Education, Noida.

We are grateful to Professor Dr. Mrs. S. A. Itkar, HOD, CSE and Professor Mrs. Deipali V. Gore of P. E. S's Modern College of Engineering, Pune, for supporting us while writing the book. We are also thankful to the staff members (Santosh Nagargoje, Nilesh Deshmukh, Kunal Khadke, Digvijay Patil and Sujeet Deshpande) of P. E. S's Modern College of Engineering for their valuable suggestions.

Furthermore, we would like to thank our friends—ShriKumar P. Ugale and Navneet Agrawal—for giving valuable inputs while writing the book. Also, we would like to thank our students—Suraj K, Pranav C, and Prajyot Gurav—who offered comments, suggestions and praise while writing the book.

We are thankful to the following reviewers for providing useful feedback and critical suggestions during the development of the manuscript.

<i>Vikram Goyal</i>	IIIT Delhi
<i>Partha Pakray</i>	NIT, Mizoram
<i>Harish Sharma</i>	RTU, Kota
<i>Shreedhara K.S.</i>	University BDT College of Engineering, Karnataka
<i>S. Rama Sree</i>	Aditya Engineering College, Andhra Pradesh
<i>Sansar Singh Chauhan</i>	IEC-CET, Greater Noida

We are also thankful to A. Vanathi, Aditya Engineering College, Surampalem, Andhra Pradesh, for reviewing sample chapter of this book.

Lastly, we are indebted to our family members—Mrs. Surekha Kamthane (mother of Amit Kamthane), Amol, Swarupa, Aditya, Santosh Chidrawar, Sangita Chidrawar, Sakshi and Sartak for their love, support and encouragement.

ASHOK NAMDEV KAMTHANE
AMIT ASHOK KAMTHANE

Roadmap to the Syllabus

KAKINADA

PYTHON PROGRAMMING

Unit 1

Introduction: History of Python, Need of Python Programming, Applications Basics of Python Programming Using the REPL(Shell), Running Python Scripts, Variables, Assignment, Keywords, Input-Output, Indentation.

GO TO

Chapter 1: Introduction to Computer and Python Programming, Chapter 2: Basics of Python Programming

Unit 2

Types, Operators and Expressions: Types—Integers, Strings, Booleans; Operators—Arithmetic Operators, Comparison (Relational) Operators, Assignment Operators, Logical Operators, Bitwise Operators, Membership Operators, Identity Operators; Expressions and order of evaluations, Control Flow—if, if-elif-else, for, while, break, continue, pass

GO TO

Chapter 3: Operators and Expressions, Chapter 4: Decision Statements, Chapter 5: Loop Control Statements

Unit 3

Data Structures : Lists—Operations, Slicing, Methods; Tuples, Sets, Dictionaries, Sequences, Comprehensions.

GO TO

Chapter 6: Lists, Chapter 7: Tuples, Sets and Dictionaries

Unit 4

Functions: Defining Functions, Calling Functions, Passing Arguments, Keyword Arguments, Default Arguments, Variable-length arguments, Anonymous Functions, Fruitful Functions (Function Returning Values), Scope of the Variables in a Function—Global and Local Variables.

Modules: Creating modules, import statement, from. Import statement, name spacing

Python packages: Introduction to PIP, Installing Packages via PIP, Using Python Packages

GO TO

Chapter 8: Functions, Chapter 9: Modules, Packages and Introduction to PIP

Unit 5

Object-Oriented Programming OOP in Python: Classes, 'self variable', Methods, Constructor Method, Inheritance, Overriding Methods, Datahiding

Error and Exceptions: Difference between an Error and Exception, Handling Exception, try-except block, Raising Exceptions, User-defined Exceptions

GO TO

Chapter 10: Object-Oriented Programming: Class, Objects and Inheritance,
Chapter 11: Errors and Exceptions

Unit 6

Brief Tour of the Standard Library: Operating System Interface—String Pattern Matching, Mathematics, Internet Access, Dates and Times, Data Compression, Multithreading, GUI Programming, Turtle Graphics

Testing: Why testing is required?, Basic concepts of testing, Unit testing in Python, Writing Test cases, Running Tests.

GO TO

Chapter 12: A Brief Tour of Standard Library, Chapter 13: Graphics Programming:
Drawing with Turtle Graphics, Chapter 14: Unit Testing



Contents

<i>About the Authors</i>	<i>ii</i>
<i>Preface</i>	<i>vii</i>
<i>Acknowledgements</i>	<i>xi</i>
<i>Roadmap to the Syllabus</i>	<i>xii</i>
UNIT 1	1
1. Introduction to Computer and Python Programming	2
<i>Chapter Outline</i>	2
<i>Learning Outcomes</i>	2
1.1 Introduction	2
1.2 What is a Computer?	3
1.2.1 Input/Output (I/O) Unit	3
1.2.2 Central Processing Unit (CPU)	3
1.2.3 Memory Unit	3
1.3 Overview of Programming Languages	4
1.3.1 Machine Language	4
1.3.2 Assembly Language	4
1.3.3 High-level Language	5
1.4 History of Python	6
1.4.1 Why Python?	6
1.4.2 Installing Python in Windows	7
1.4.3 Starting Python in Different Execution Modes	10
1.5 Installing Python in Ubuntu	15
1.6 Executing Python Programs	16
1.6.1 Writing the First Python Program in Script Mode	17
1.7 Commenting in Python	19
1.8 Internal Working of Python	20

- 1.9 Python Implementations 20
 - 1.9.1 Jython 21
 - 1.9.2 IronPython 21
 - 1.9.3 Stackless Python 21
 - 1.9.4 PyPy 21

Summary 21

Key Terms 22

Review Questions 22

A. Multiple Choice Questions 22

B. True or False 23

C. Exercise Questions 23

Programming Assignments 23

2. Basics of Python Programming

24

Chapter Outline 24

Learning Outcomes 24

2.1 Introduction 25

2.2 Python Character Set 25

2.3 Token 25

2.3.1 Literal 26

2.3.2 Value and Type on Literals 26

2.3.3 Keywords 27

2.3.4 Operator 27

2.3.5 Delimiter 27

2.3.6 Identifier/Variable 27

2.4 Python Core Data Type 28

2.4.1 Integer 28

2.4.2 Floating Point Number 30

2.4.3 Complex Number 30

2.4.4 Boolean Type 31

2.4.5 String Type 31

2.5 The `print()` Function 32

2.5.1 The `print()` Function with end Argument 34

2.6 Assigning Value to a Variable 35

2.6.1 More on Assigning Values to Variables 35

2.6.2 Scope of a Variable 36

2.7 Multiple Assignments 36

2.8 Writing Simple Programs in Python 38

2.9 The `input()` Function 39

2.9.1 Reading String from the Console 39

2.10 The `eval()` Function 42

2.10.1 Apply `eval()` to `input()` Function 43

2.11	Formatting Number and Strings	44
2.11.1	Formatting Floating Point Numbers	45
2.11.2	Justifying Format	46
2.11.3	Integer Formatting	46
2.11.4	Formatting String	47
2.11.5	Formatting as a Percentage	47
2.11.6	Formatting Scientific Notation	48
2.12	Python Inbuilt Functions	48
2.12.1	The ord and chr Functions	51
	<i>Summary</i>	52
	<i>Key Terms</i>	52
	<i>Review Questions</i>	53
	A. Multiple Choice Questions	53
	B. True or False	54
	C. Exercise Questions	54
	<i>Programming Assignments</i>	55

UNIT 2

56

3. Operators and Expressions

57

	<i>Chapter Outline</i>	57
	<i>Learning Outcomes</i>	57
3.1	Introduction	57
3.2	Operators and Expressions	58
3.3	Arithmetic Operators	58
3.3.1	Unary Operators	58
3.3.2	Binary Operators	59
3.4	Operator Precedence and Associativity	68
3.4.1	Example of Operator Precedence	69
3.4.2	Associativity	69
3.5	Changing Precedence and Associativity of Arithmetic Operators	70
3.6	Translating Mathematical Formulae into Equivalent Python Expressions	72
3.7	Bitwise Operator	73
3.7.1	The Bitwise AND (&) Operator	74
3.7.2	The Bitwise OR () Operator	75
3.7.3	The Bitwise XOR (^) Operator	76
3.7.4	The Right Shift (>>) Operator	78
3.7.5	The Left Shift (<<) Operator	79
3.8	The Compound Assignment Operator	80
	Mini Project: Goods Service Tax (GST) Calculator	81
	<i>Summary</i>	83
	<i>Key Terms</i>	83

Review Questions 83

A. Multiple Choice Questions 83

B. True or False 84

C. Exercise Questions 85

Programming Assignments 87

4. Decision Statements

88

Chapter Outline 88

Learning Outcomes 88

4.1 Introduction 88

4.2 Boolean Type 89

4.3 Boolean Operators 90

4.3.1 The not Operator 90

4.3.2 The and Operator 90

4.3.3 The or Operator 91

4.4 Using Numbers with Boolean Operators 91

4.5 Using String with Boolean Operators 92

4.6 Boolean Expressions and Relational Operators 92

4.7 Decision Making Statements 94

4.7.1 The if Statements 94

4.7.2 The if-else Statement 96

4.7.3 Nested if Statements 100

4.7.4 Multi-way if-elif-else Statements 101

4.8 Conditional Expressions 105

Mini Project: Finding the Number of Days in a Month 107

Summary 108

Key Terms 109

Review Questions 109

A. Multiple Choice Questions 109

B. True or False 111

C. Exercise Questions 112

Programming Assignments 112

5. Loop Control Statements

113

Chapter Outline 113

Learning Outcomes 113

5.1 Introduction 113

5.2 The while Loop 114

5.2.1 Details of while Loop 114

5.2.2 Flowchart for while Loop 115

5.2.3 Some More Programs on while Loop 117

5.3 The range () Function 119

5.3.1 Examples of range () Function 119

5.4	The for Loop	120
5.4.1	Details of for Loop	120
5.4.2	Some More Programs on for Loop	121
5.5	Nested Loops	125
5.5.1	Some More Programs on Nested Loops	126
5.6	The break Statement	129
5.7	The continue Statement	131
	Mini Project: Generate Prime Numbers using Charles Babbage Function	133
	Summary	135
	Key Terms	135
	Review Questions	135
	A. Multiple Choice Questions	135
	B. True or False	137
	C. Exercise Questions	138
	Programming Assignments	139

UNIT 3

140

6. Lists

141

	Chapter Outline	141
	Learning Outcomes	141
6.1	Introduction	142
6.2	Creating Lists	142
6.3	Accessing the Elements of a List	143
6.4	Negative List Indices	143
6.5	List Slicing [Start: end]	144
6.6	List Slicing with Step Size	145
6.6.1	Some More Complex Examples of List Slicing	145
6.7	Python Inbuilt Functions for Lists	145
6.8	The List Operator	147
6.9	List Comprehensions	150
6.9.1	Some More Examples of List Comprehension	151
6.10	List Methods	153
6.11	List and Strings	157
6.12	Splitting a String in List	157
6.13	Passing List to a Function	158
6.14	Returning List from a Function	160
	Summary	168
	Key Terms	168
	Review Questions	168
	A. Multiple Choice Questions	168
	B. True or False	170
	C. Exercise Questions	170
	Programming Assignments	171

7. Tuples, Sets and Dictionaries**173***Chapter Outline* 173*Learning Outcomes* 173**7.1 Introduction** 173

7.1.1 Creating Tuples 174

7.1.2 The `tuple()` Function 174

7.1.3 Inbuilt Functions for Tuples 175

7.1.4 Indexing and Slicing 175

7.1.5 Operations on Tuples 176

7.1.6 Passing Variable Length Arguments to Tuples 176

7.1.7 Lists and Tuples 177

7.1.8 Sort Tuples 178

7.1.9 Traverse Tuples from a List 178

7.1.10 The `zip()` Function 1787.1.11 The Inverse `zip(*)` Function 1807.1.12 More Examples on `zip(*)` Function 180

7.1.13 More Programs on Tuples 181

7.2 Sets 181

7.2.1 Creating Sets 181

7.2.2 The Set `in` and `not in` Operator 182

7.2.3 The Python Set Class 182

7.2.4 Set Operations 184

7.3 Dictionaries 185

7.3.1 Need of Dictionaries 185

7.3.2 Basics of Dictionaries 185

7.3.3 Creating a Dictionary 186

7.3.4 Adding and Replacing Values 187

7.3.5 Retrieving Values 188

7.3.6 Formatting Dictionaries 189

7.3.7 Deleting Items 189

7.3.8 Comparing Two Dictionaries 189

7.3.9 The Methods of Dictionary Class 190

7.3.10 Traversing Dictionaries 191

7.3.11 Nested Dictionaries 192

7.3.12 Traversing Nested Dictionaries 192

7.3.13 Simple Programs on Dictionary 194

7.3.14 Polynomials as Dictionaries 197

Mini Project: Orange Cap Calculator 198

Summary 200*Key Terms* 200

Review Questions	201	
A. Multiple Choice Questions	201	
B. True or False	202	
C. Exercise Questions	203	
Programming Assignments	204	
UNIT 4		205
8. Functions		206
Chapter Outline	206	
Learning Outcomes	206	
8.1 Introduction	206	
8.2 Syntax and Basics of a Function	207	
8.3 Use of a Function	208	
8.4 Parameters and Arguments in a Function	209	
8.4.1 Positional Arguments	211	
8.4.2 Keyword Arguments	212	
8.4.3 Parameter with Default Values	213	
8.5 The Local and Global Scope of a Variable	215	
8.5.1 Reading Global Variables from a Local Scope	216	
8.5.2 Local and Global Variables with the Same Name	217	
8.5.3 The Global Statement	217	
8.6 The return Statement	218	
8.6.1 Returning Multiple Values	221	
8.6.2 Assign Returned Multiple Values to Variable(s)	222	
8.7 Recursive Functions	222	
8.8 The Lambda Function	223	
Mini Project: Calculation of Compound Interest and Yearly Analysis of Interest and Principal Amount	224	
Summary	227	
Key Terms	227	
Review Questions	227	
A. Multiple Choice Questions	227	
B. True or False	230	
C. Exercise Questions	230	
Programming Assignments	231	
9. Modules, Packages and Introduction to PIP		232
Chapter Outline	232	
Learning Outcomes	232	
9.1 Introduction to Modules	232	
9.2 Creating and Importing Modules	233	

9.3	Packages in Python	234
9.3.1	Creating Package	234
9.4	Introduction to PIP	235
9.4.1	PIP Directory Structure	236
9.4.2	Knowing the Version of PIP	236
9.5	Installing and Uninstalling Packages via PIP	239
9.5.1	The PIP Commands and Options	240
9.6	Using Python Packages	241
	<i>Summary</i>	242
	<i>Key Terms</i>	243
	<i>Review Questions</i>	243
	<i>A. Exercise Questions</i>	243
	<i>Programming Assignments</i>	243

UNIT 5

244

10. Object-Oriented Programming: Class, Objects and Inheritance

245

	<i>Chapter Outline</i>	245
	<i>Learning Outcomes</i>	245
10.1	Introduction	246
10.2	Defining Classes	246
10.2.1	Adding Attributes to a Class	247
10.2.2	Accessing Attributes of a Class	248
10.2.3	Assigning Value to an Attribute	248
10.3	The Self-parameter and Adding Methods to a Class	249
10.3.1	Adding Methods to a Class	249
10.3.2	The Self-parameter	249
10.3.3	Defining Self-parameter and Other Parameters in a Class Method	250
10.3.4	The Self-parameter with Instance Variable	251
10.3.5	The Self-parameter with Method	252
10.4	Display Class Attributes and Methods	253
10.5	Special Class Attributes	254
10.6	Accessibility	255
10.7	The <code>__init__</code> Method (Constructor)	256
10.7.1	Attributes and <code>__init__</code> Method	257
10.7.2	More Programs on <code>__init__</code> Method	258
10.8	Passing an Object as Parameter to a Method	258
10.9	<code>__del__</code> () (Destructor Method)	260
10.10	Class Membership Tests	262
10.11	Method Overloading in Python	262
10.12	Operator Overloading	264
10.12.1	Special Methods	265
10.12.2	Special Methods for Arithmetic Operations	265

10.12.3	Special Methods for Comparing Types	266
10.12.4	Reference Equality and Object Equality	267
10.12.5	Special Methods for Overloading Inbuilt Functions	269
10.13	Inheritance	269
10.14	Types of Inheritance	270
10.15	The Object Class	271
10.16	Inheritance in Detail	271
10.17	Subclass Accessing Attributes of Parent Class	273
10.18	Multilevel Inheritance in Detail	274
10.19	Multiple Inheritance in Detail	275
10.19.1	More Practical Examples on Inheritance	276
10.20	Using <code>super()</code>	278
10.20.1	Super to Call Super Class Constructor	279
10.21	Method Overriding	280
10.22	Precaution: Overriding Methods in Multiple Inheritance	282
	Mini Project: Arithmetic Operations on Complex Numbers	283
	Summary	287
	Key Terms	287
	Review Questions	288
	A. Multiple Choice Questions	288
	B. True or False	290
	C. Exercise Questions	291
	Programming Assignments	292
11.	Errors and Exceptions	294
	Chapter Outline	294
	Learning Outcomes	294
11.1	Errors and Exceptions	294
11.1.1	Exception	295
11.2	Python Exception and its Hierarchy	295
11.3	Handling Exception	297
11.3.1	Executing the 'Divide by Zero' Program Without Exception Handling	297
11.3.2	The <code>try</code> and <code>except</code> Blocks to Handle Various Exceptions	299
11.4	Raising Exception	302
	Summary	302
	Key Terms	303
	Review Questions	303
	A. Multiple Choice Questions	303
	B. Exercise Questions	304
	Programming Assignments	304

UNIT 6	305
12. A Brief Tour of Standard Library	306
<i>Chapter Outline</i>	306
<i>Learning Outcomes</i>	306
12.1 Introduction	306
12.2 Operating System Interface—The OS Module	307
12.3 String Pattern Matching	309
12.3.1 The <code>re</code> Module	310
12.3.2 Regular Expression Containing Operators and Symbols	312
12.3.3 Basic Example of Regular Expressions	313
12.4 The Date and Time Module	314
12.4.1 Date Formats	316
12.4.2 Time Module	316
12.5 Mathematics—The Maths Module	317
12.6 Internet Access	319
12.6.1 The <code>urllib</code> module	319
12.7 Data Compression	321
12.7.1 The <code>zlib</code> Library	321
12.7.2 Compressing Large Data Streams	322
<i>Summary</i>	325
<i>Key Terms</i>	325
<i>Review Questions</i>	326
A. Multiple Choice Questions.	326
B. Exercise Questions	326
<i>Programming Assignments</i>	327
13. Graphics Programming: Drawing with Turtle Graphics	328
<i>Chapter Outline</i>	328
<i>Learning Outcomes</i>	328
13.1 Introduction	328
13.2 Getting Started with the Turtle Module	329
13.3 Moving the Turtle in any Direction	330
13.3.1 Programs to Draw Different Shapes	333
13.4 Moving Turtle to Any Location	334
13.5 The <code>color</code> , <code>bgcolor</code> , <code>circle</code> and <code>Speed</code> Method of Turtle	336
13.6 Drawing with Colors	338
13.7 Drawing Basic Shapes using Iterations	339
13.8 Changing Color Dynamically Using List	342
13.9 Turtles to Create Bar Charts	342
Mini Project: Turtle Racing Game	344
<i>Summary</i>	348
<i>Key Terms</i>	348

..	
Review Questions	348	
A. Multiple Choice Questions	348	
B. True or False	349	
C. Exercise Questions	349	
Programming Assignments	350	
14. Unit Testing		351
Chapter Outline	351	
Learning Outcomes	351	
14.1 Why Testing is Required?	351	
14.2 Types of Testing	352	
14.3 Identifying Units	352	
14.4 Unit Testing in Python—The unittest Class	352	
14.5 List of assert Methods	355	
14.6 Writing and Running Multiple Test Cases	356	
Summary	359	
Key Terms	360	
Review Questions	360	
A. Exercise Questions	360	
Programming Assignments	360	
Previous Year's Solved JNTU Examination Questions		361

UNIT 1

- Introduction to Computer and Python Programming
- Basics of Python Programming

Introduction to Computer and Python Programming

1

CHAPTER OUTLINE

1.1 Introduction	1.6 Executing Python Programs
1.2 What is a Computer?	1.7 Commenting in Python
1.3 Overview of Programming Languages	1.8 Internal Working of Python
1.4 History of Python	1.9 Python Implementations
1.5 Installing Python in Ubuntu	

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Identify the functionalities of modern computer systems and various programming languages
- Explain the importance of Python and describe its need as a programming language
- Install Python in various operating systems and write and execute programs in Python

1.1 INTRODUCTION

Nowadays computers have become an integral part of human lives. They are used in diverse sectors to execute a range of everyday tasks such as reservation of tickets, payment of electricity bills, virtual transfer of money, forecasting the weather, diagnosis of diseases and so on. In short, each one of us—directly or indirectly—makes use of computers. So, before learning python programming language, this chapter explains the basics of computers and different types of programming

languages for ease of beginners and then introduces Python in detail, covering installation and execution of Python and Python programs.

1.2 WHAT IS A COMPUTER?

The word computer is derived from 'compute', which means 'to calculate'. A computer is an electronic device which accepts data from a user, processes the data for calculations specified by the user and generates an output. A computer performs these operations with speed and accuracy using certain hardware and software. Hardware is visible physical element of a computer and software consist of a written set of instructions used to control the hardware. Figure 1.1. shows the various components of a modern computer system.

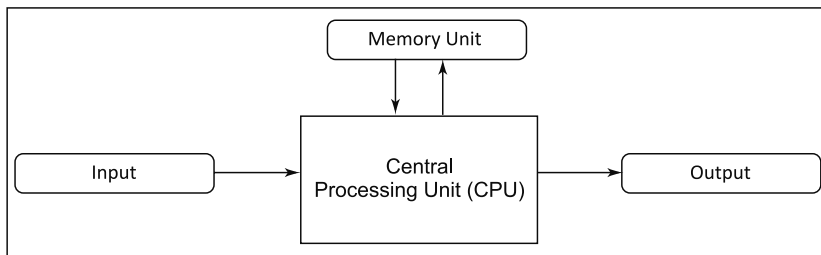


Figure 1.1 Block diagram of a modern computer system

The hardware of a computer system consists of three main components, viz. input/output (I/O) unit, central processing unit and memory unit.

1.2.1 Input/Output (I/O) Unit

Users interact with a computer using various I/O units. Inputs can be given to a computer using input devices, such as a keyboard. The input unit of a computer converts the data that it accepts from a user into a form that is understandable by it. As soon as the computer receives the input, it is processed and sent to its output device. Monitors, printers, etc., are examples of output devices of a computer.

1.2.2 Central Processing Unit (CPU)

The CPU is one of the most important parts of a computer. It handles processing of data and consists of an arithmetic logic unit (ALU) and a control unit. The ALU performs all operations on the input data and the control unit directs the computer memory and input and output devices response to the instructions received from a program.

1.2.3 Memory Unit

The function of the memory unit is to store programs and data. The unit is a compilation of numerous storage cells and each cell can store one bit of information. These cells are processed

in a group of fixed sizes of units called words and they never read or write as individual cells. A computer's memory system can be divided into the following three groups:

1. **Internal memory:** It refers to the set of registers confined to the CPU. These registers hold temporary results when a computation is in progress.
2. **Primary memory:** It is a storage area in which all the programs are executed. All programs and data must be stored in the primary memory for speedy execution.
3. **Secondary memory:** It is known as **external memory** or **storage memory**. Programs and data are stored here for the long term. Hard disk, floppy disk, CDs, DVDs and magnetic tapes are different forms of secondary memory.

1.3 OVERVIEW OF PROGRAMMING LANGUAGES

A computer program is a set of instructions, which performs a specific task when executed by a computer. Computer programs are commonly known as **software**. The instructions in a program tell a computer what to do and these instructions can be written in three types of programming languages described next.

1.3.1 Machine Language

A computer is an electronic machine which can understand any instruction written in binary form, i.e. using only 0s and 1s. A program written in 0s and 1s is called machine language. While a computer easily understands this language, it is difficult for humans to write an instruction in terms of 0's and 1's. Consider the following example.

Example

A series of numbers, such as 0011, 1000, 1010 is an instruction written in machine language. The instruction implies addition of a number stored at location 8 (1000) and another number stored at location 10 (1010) and storing the result at location 8 (1000). Here, the binary code 0011 stands for addition.

1.3.2 Assembly Language

From the above example, we know that it is difficult to write, read, communicate or change a program written in machine language for humans. Hence, the need to create another more convenient language arose. In assembly language, which was developed subsequently, machine operations are represented by mnemonic codes (such as ADD and MUL) and symbolic names that specify the memory address. Consider the following example.

Example

```
MOV X, 10
MOV Y, 20
ADD X, Y
```

Here the mnemonic MOV indicates an operation to store the value of variable X as 10. The mnemonic ADD implies addition of the contents of variable X, Y and finally storing the result in variable X itself.

Since computers cannot understand the assembly language, a program called **assembler** is used to translate assembly language programs into equivalent machine language programs.

1.3.3 High-level Language

High-level languages are much easier to write than low-level languages because programs written in these are similar to instructions written in the English language. Here 'high' does not imply that the language is complicated. It means that the language is more problem oriented. Generally, high-level languages are platform independent. This means that one can write a program in a high-level language and run it on different types of machines. Instructions written in high-level languages are called **statements**.

For example, a statement to calculate the square of a number can be written in a high-level language as

Square = number * number

There are many high-level languages and the selection of a language is based on the purpose it is expected to fulfill. A program written in a high-level language is called *source code* or *source program*. The process of executing programs written in high-level languages is given below.

- ⊙ **STEP 1:** An **interpreter** or **compiler** is used to translate a program written in a high-level language into its equivalent machine code for execution.
- ⊙ **STEP 2:** A **linker** is used to combine the object code and the code stored in libraries into machine language.
- ⊙ **STEP 3:** Finally, the machine language code generated in Step 2 is executed.

Figure 1.2 depicts the steps on how to execute a program written in a high-level language.

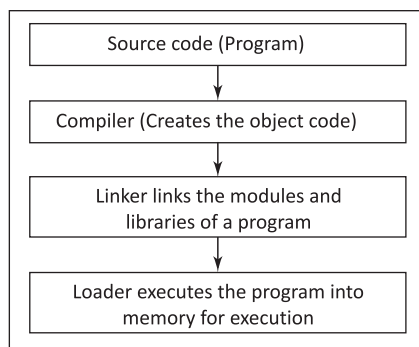


Figure 1.2 Steps to execute a high-level language program

The next section describes compiler, interpreter, linker and loader in detail.

Compiler

It is a software that translates a program written in a high-level language into machine language. This compiled program is called object code. The object code is an executable code which can run as a standalone code, i.e. it does not need the compiler to be present during execution. Every programming language, such as C, C++ and Java has its own compiler.

Interpreter

While a compiler converts the whole source code into an equivalent object code or machine code, the interpreter reads the source code line by line and converts it into object code (i.e. a code understandable to the machine).

Linker

It is a program that links different program modules and libraries to form a single executable program. A source code of a program is very large. It can consist of hundreds of lines of code. Before the execution of a program, all the modules of the program and the required libraries are linked together using a software called a **linker**. The compiled and linked program is called the **executable code**.

Loader

This software is used to load and relocate an executable program in the main memory during execution. The loader assigns a storage space to a program in the main memory for execution.

1.4 HISTORY OF PYTHON

Python was developed by Guido van Rossum at National Research Institute for Mathematics and Computer Science in Netherlands in 1990. Rossum wanted the name of his new language to be short, unique and mysterious. Inspired by *Monty Python's Flying Circus*, a BBC comedy series, he named the language Python.

Python became a popular programming language, widely used in both industry and academia because of its simple, concise and extensive support of libraries. It is a general purpose, interpreted and object-oriented programming language. Python source code is available under General Public License (GPL) and maintained by a core development team at the same institute.

1.4.1 Why Python?

COBOL, C#, C, C++ and Java are a few of the many programming languages available in information and technology today. One common question that beginners in programming often ask is, '*Why use Python when there are so many programming languages?*' While on one hand it may just be a matter of personal preference, there are some very well-known advantages of Python which make it a popular programming language. These are given below.

1. **Readability:** Developer's **readability** of code is one of the most crucial factors in programming. The longest part of any software's life cycle is its maintenance. Therefore, if a software has a highly readable code, then it is easier to maintain. Readability also helps a programmer to reuse the existing code with ease to maintain and update a software. Python offers more readability of code when compared to other programming languages.
2. **Portability:** Python is platform independent, i.e. its programs run on all platforms. The language is designed for portability.
3. **Vast support of libraries:** Python has a large collection of in-built functionalities known as standard *library functions*. Python also supports various third-party software like NumPy. **NumPy** is an extension, i.e. it provides support for large, multidimensional arrays and matrices.

4. **Software integration:** An important aspect of Python is that it can easily extend, communicate and integrate with several other languages. For example, Python code can easily invoke libraries of C and C++ programming languages. It can also be used to communicate with **Java** and **.net** components. Python can sometimes act as an intermediary or agent between two applications.
5. **Developer productivity:** Compared to other programming languages, Python is a dynamically typed language, which means there is no need to declare variables explicitly. Again, there are various other features of Python due to which the size of code written is typically smaller or half of the code written in some other languages, such as C, C++ or Java.

As the size of code is reduced quite a bit, there is less to type and debug. The amount of time needed to compile and execute is also very less as compared to other programming languages. Python programs run immediately, i.e. without taking much time to link and compile.

These benefits offered by Python make it the topmost choice for programmers to develop application software or projects with Python.

1.4.2 Installing Python in Windows

Python is available for almost all operating systems such as Windows, Mac, Linux/Unix, etc. The complete list of different versions of Python can be found at <http://www.Python.org/downloads>. Step-wise details for installing Python in Windows are given below.

- ◎ **STEP 1:** Open an Internet browser like Internet Browser, Mozilla Firefox or Chrome. Type <http://www.Python.org/> in the address bar and press Enter. Immediately, the following page will appear (Figure 1.3).

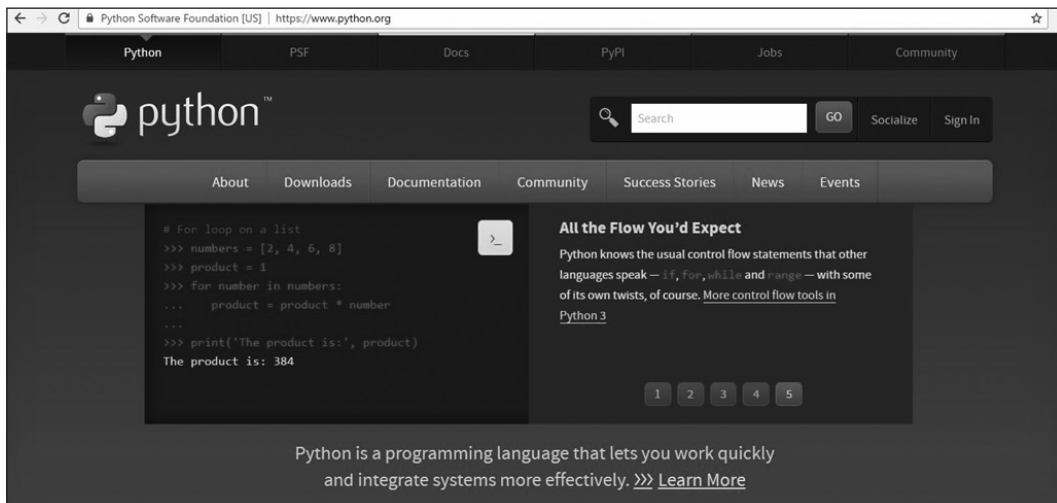


Figure 1.3 Python home page

- ◎ **STEP 2:** Click on Downloads and you will see the latest version of Python. Since all programs in this book are written and executed on **Python 3.4**, download **Python 3.4** version by clicking on **All Releases** under Downloads as shown in Figure 1.4.

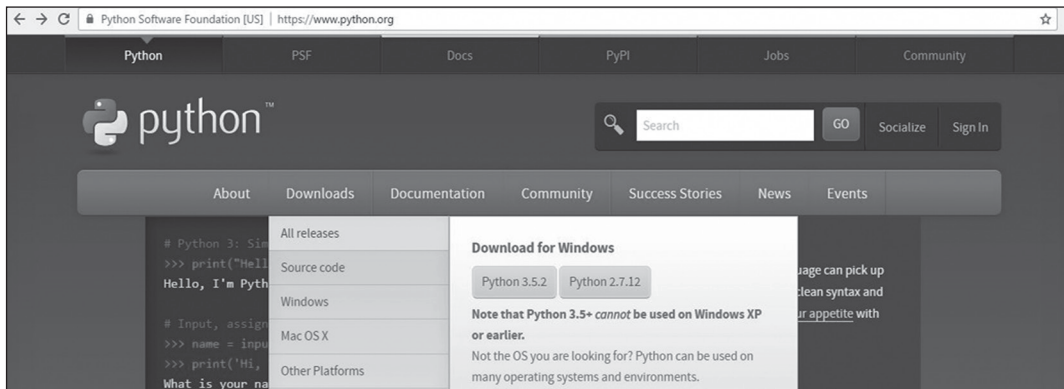


Figure 1.4 Python download page

- ⊙ **STEP 3:** After clicking on All Releases under Downloads browse through the page to the bottom. You will see a list of Python releases as shown in Figure 1.5.

Looking for a specific release?
Python releases by version number:

Release version	Release date		Click for more
Python 3.4.3	2015-02-25	Download	Release Notes
Python 2.7.9	2014-12-10	Download	Release Notes
Python 3.4.2	2014-10-13	Download	Release Notes
Python 3.3.6	2014-10-12	Download	Release Notes
Python 3.2.6	2014-10-12	Download	Release Notes
Python 2.7.8	2014-07-02	Download	Release Notes
Python 2.7.7	2014-06-01	Download	Release Notes
Python 3.4.1	2014-05-19	Download	Release Notes

Figure 1.5 Python release versions

- ⊙ **STEP 4:** Click on Python 3.4.2 and download it.
- ⊙ **STEP 5:** Open the folder where you have downloaded the Python 3.4 version pack and double click on it to start the installation (Figure 1.6).

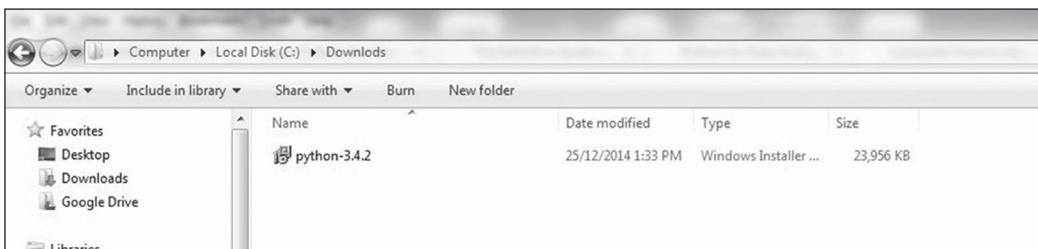


Figure 1.6 Python software

- ⊙ **STEP 6:** After clicking on it you will see the first window to set up Python 3.4.2 (Figure 1.7).



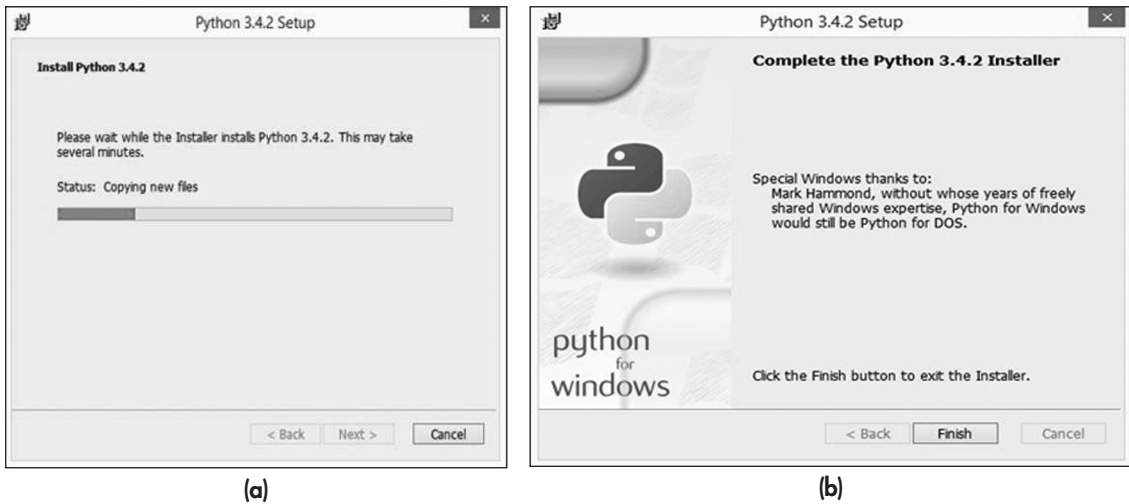
Figure 1.7 Python first setup window

- ⊙ **STEP 7:** Click on Next and you will see a second window which tells you to specify the location where you want to install Python (Figure 1.8).



Figure 1.8 Python second setup window

By default, Python will be installed in C:\. Then click on Next to continue the installation. Just before completing the installation, it will show you the following two windows (Figures 1.9 a and b).



Figures 1.9 a and b Python final setup window

- ⊙ **STEP 8:** Click on Finish to complete the installation.
- ⊙ **STEP 9:** To check if Python is installed successfully just press windows key on Windows 7 or Windows 8 and then in the search bar type Python as shown in Figure 1.10.



Figure 1.10 Windows 8 showing successful installation of Python

1.4.3 Starting Python in Different Execution Modes

After installing Python in Windows, you can start Python in two different modes, viz. Python (Command Line) and Python (IDLE).

Starting Python (Command Line)

Python is an interpreted language. You can directly write the code into the Python interpreter or you can write a sequence of instructions into a file and then run the file.

When you execute Python expressions or statements from the command line then you are in **interactive mode** or **interactive prompt**.

Steps for writing a Python command line in Windows 7 are given as follows:

- ⊙ **STEP 1:** Press the Start button (Figure 1.11).



Figure 1.11

- ⊙ **STEP 2:** Click on All programs and then **Python 3.4**. After clicking on **Python 3.4** you will see a list of options as shown in Figure 1.12.

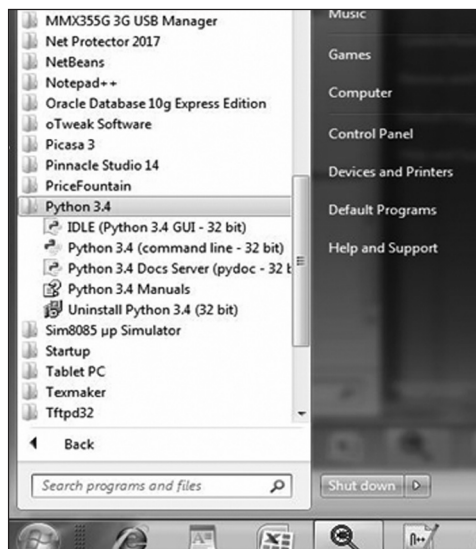


Figure 1.12

- ◎ **STEP 3:** In this list click on **Python (Command Line—32 bit)**. After clicking on it, you will see the Python **interactive prompt** in Python command line as shown in Figure 1.13.

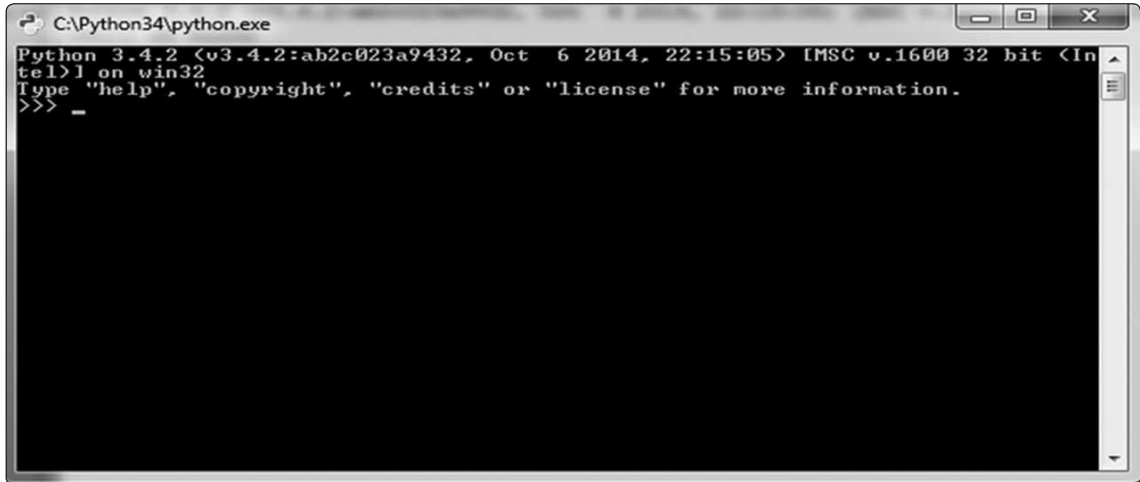


Figure 1.13 Python interactive mode as Python command line window

In Figure 1.13, the Python command prompt contains an opening message `>>>`, called **command prompt**. The cursor at the command prompt waits for you to enter a Python command. A complete command is called a **statement**. Simple commands executed in the interactive mode of Python command line are shown in Figure 1.14.

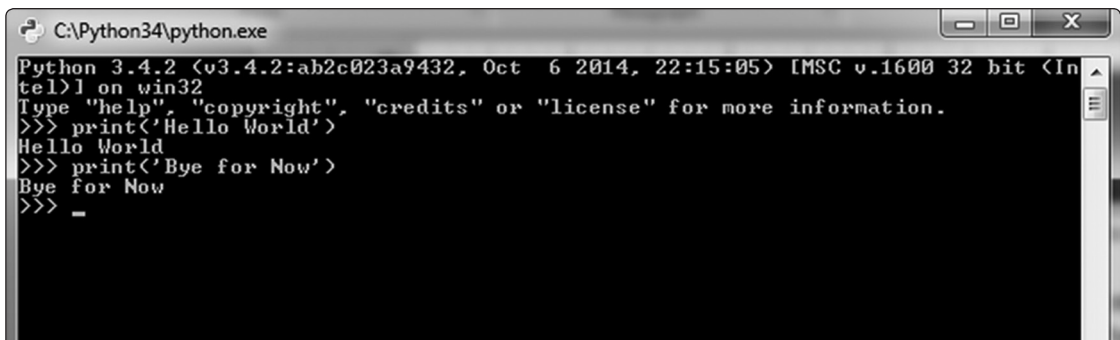


Figure 1.14 Simple commands executed in interactive mode of Python command prompt

We have written two simple commands or statements. The first statement, i.e. `print('Hello World')` when executed in the interactive mode of Python command prompt gives the output as the entered command, i.e. `'Hello World'` for this message. More details about print and its syntax are explained in Chapter 2.

Precautions to be taken while executing commands in the interactive mode of Python with command line are given as follows.

If you try to put an extra space between Python prompt, i.e. >>> and the command, then it will produce an error called **Indentation Error: Unexpected Indent**. A simple example to demonstrate this error is given below.

Example:

```
>>> print('Hello World')
File "<stdin>", line 1
print('Hello World')
^
```

IndentationError: unexpected indent

Thus, due to an extra space between >>> and command, i.e. `print('Hello world')`, the Python interpreter raises an error.

To exit from the command line of **Python 3.4**, press **Ctrl+Z** followed by **Enter**.

Starting Python IDLE

Launching Python **IDLE** is another way to start executing Python statements or commands in the interactive mode of Python IDLE. It is a graphical integrated development environment for Python.

Python statements or commands which run in the interactive mode of Python IDLE are called **shell**. IDLE is downloaded by default while installing Python. Launching Python IDLE is the simplest way to open a Python shell. The steps to launch Python IDLE are similar to those used to start a Python command line and are detailed below.

- ☉ **STEP 1:** Press the Start button.
- ☉ **STEP 2:** Click on All Programs and then **Python 3.4**. After clicking on **Python 3.4** you will see a list of options as shown in Figure 1.15.

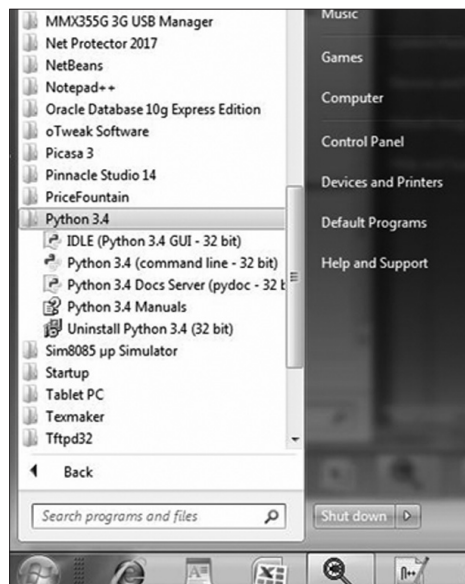


Figure 1.15

- ◎ **STEP 3:** Click on **IDLE (Python 3.4 GUI—32 bit)** and you will see the Python **interactive prompt**, i.e. an **interactive shell** as shown in Figure 1.16.

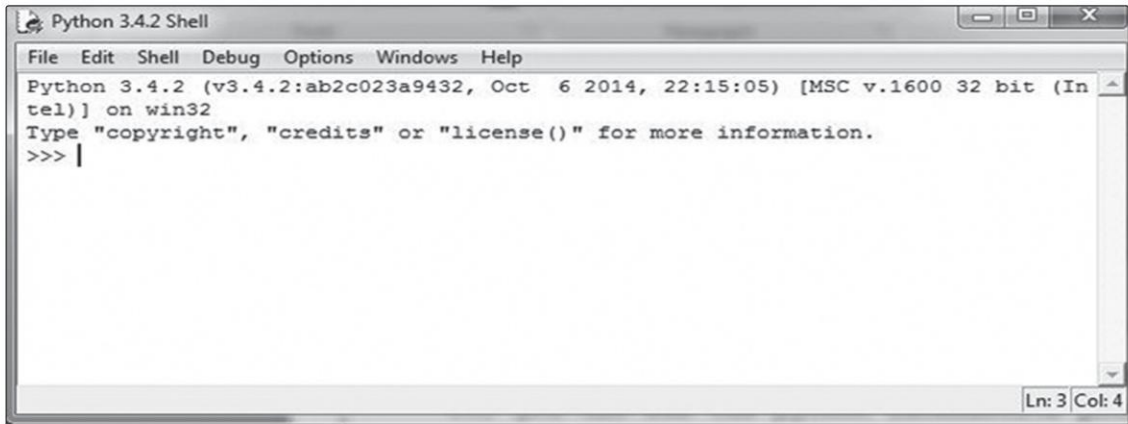


Figure 1.16 Python IDLE—Interactive shell

In Figure 1.16, a Python interactive shell prompt contains an opening message `>>>`, called '**shell prompt**'. The cursor at the shell prompt waits for you to enter a Python command. A complete command is called a **statement**. As soon as you write a command and press Enter, the **Python interpreter** will immediately display the result.

Figure 1.17 shows simple commands which are executed in the interactive mode, i.e. the interactive shell of Python IDLE.

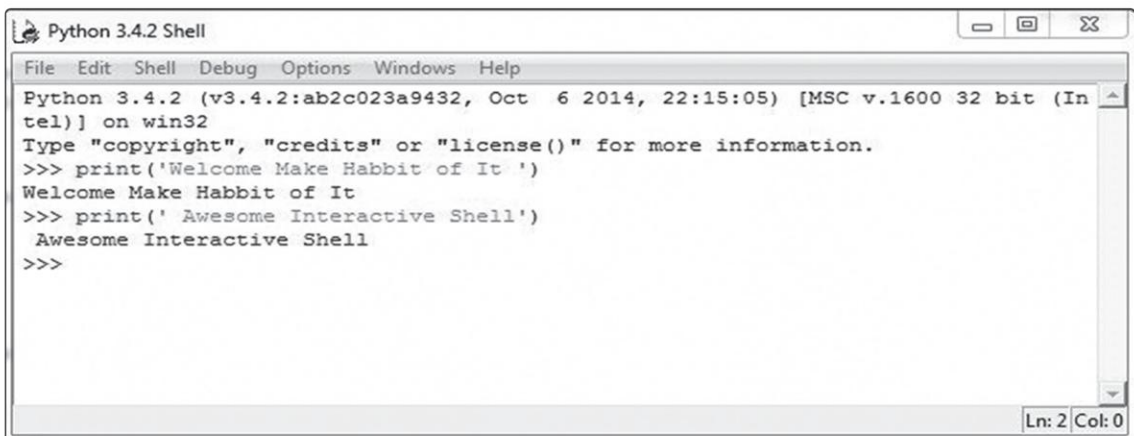


Figure 1.17 Running commands in Python IDLE's interactive shell

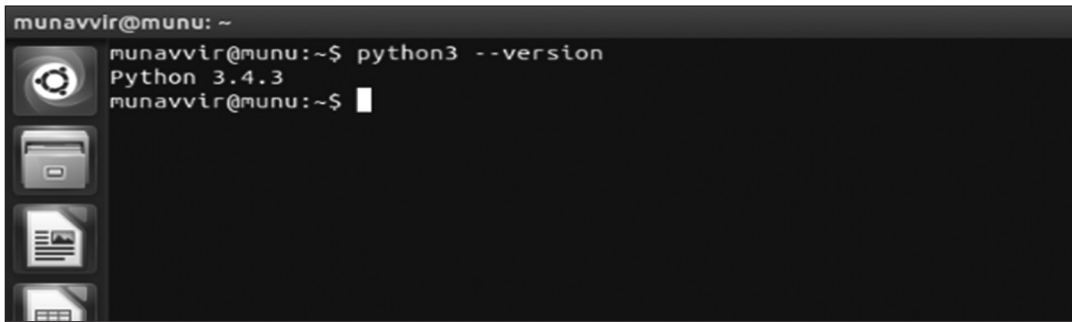


Note: Hereafter all commands given as examples in the forthcoming chapters of this book are executed in Python 3.4 IDLE's interactive mode, i.e. the interactive shell prompt.

1.5 INSTALLING PYTHON IN UBUNTU

Python 2.7 and Python 3.4 are installed by default on Ubuntu 15.0. The following steps can be used to check their presence.

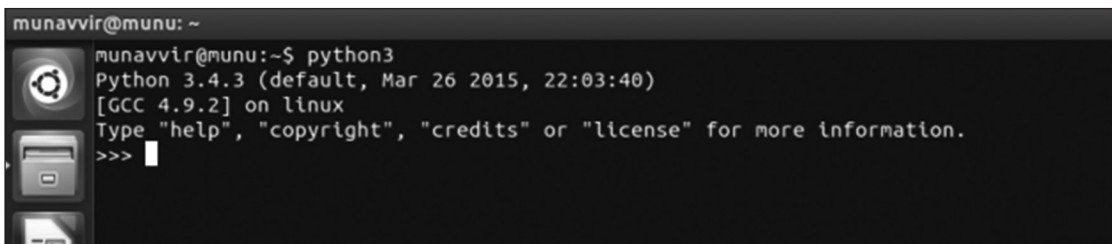
- ⊙ **STEP 1:** Open Ubuntu 15.0.
- ⊙ **STEP 2:** Press the Windows button on the keyboard and type 'terminal' or press the shortcut Ctrl+Alt+T to open the terminal.
- ⊙ **STEP 3:** Once the terminal is open, type **Python3—version** to check if it is installed.



```
munavvir@munu: ~  
munavvir@munu:~$ python3 --version  
Python 3.4.3  
munavvir@munu:~$
```

Figure 1.18 Check default installation of Python

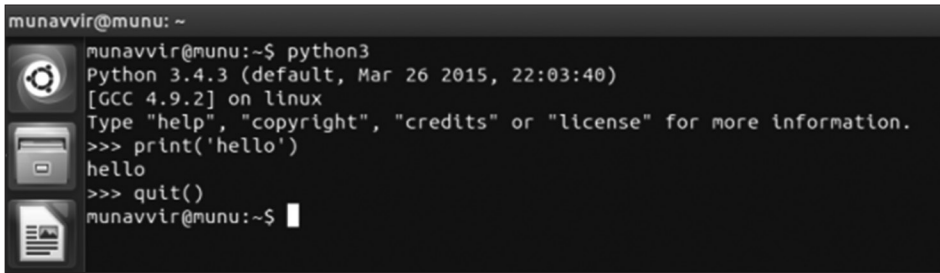
- ⊙ **STEP 4:** From Figure 1.18 we can know that default Python3.X version has been installed successfully.
- ⊙ **STEP 5:** To launch the **command line mode** or **interactive mode** of Python 3.X version in Ubuntu, type Python3 on the terminal.



```
munavvir@munu: ~  
munavvir@munu:~$ python3  
Python 3.4.3 (default, Mar 26 2015, 22:03:40)  
[GCC 4.9.2] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

Figure 1.19 Ubuntu Python3 command line mode

From Figure 1.19 we can see the command line mode of Ubuntu has been started and the programmer is ready to give instructions to Python. The following figure (Figure 1.20) illustrates an example of printing 'hello' in the command line mode of Ubuntu.



```
munavvir@munu: ~
munavvir@munu:~$ python3
Python 3.4.3 (default, Mar 26 2015, 22:03:40)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello')
hello
>>> quit()
munavvir@munu:~$
```

Figure 1.20 Executing instructions of Python3 in Ubuntu command line mode

- ☉ **STEP 6:** A programmer can launch **Python IDLE mode** in Ubuntu in the same manner. To launch the IDLE mode of Python, type the command given below on the terminal.

Python -m idlelib



Note: If IDLE is not installed then the programmer can install it by typing the command given below on the terminal.

`sudo apt-get install idle3`

1.6 EXECUTING PYTHON PROGRAMS

The previous section explained the installation of Python3 in Windows and Ubuntu. This section describes how to execute Python programs in script mode on Windows. All the programs written in this book are written and executed on Windows. Once IDLE is launched in Ubuntu, a programmer can write programs in script mode in the same manner as done in Windows.

Running Python programs from a script file is known as running Python in **script mode**. You can write a sequence of instructions in one file and execute them. The steps required to write Python programs in Python IDLE's script mode are given as follows.

- ☉ **STEP 1:** In Python IDLE's - Shell window, click on **File** and then on **New File** or just click **CTRL+N** (Figure 1.21).

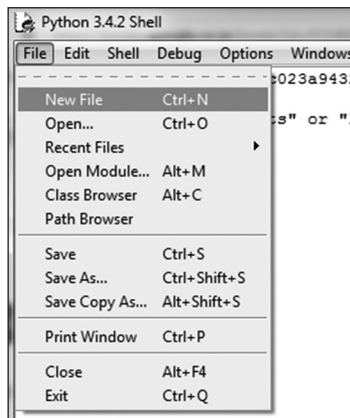


Figure 1.21 Python IDLE file menu bar

As soon as you click on New File, the window shown below will open (Figure 1.22).

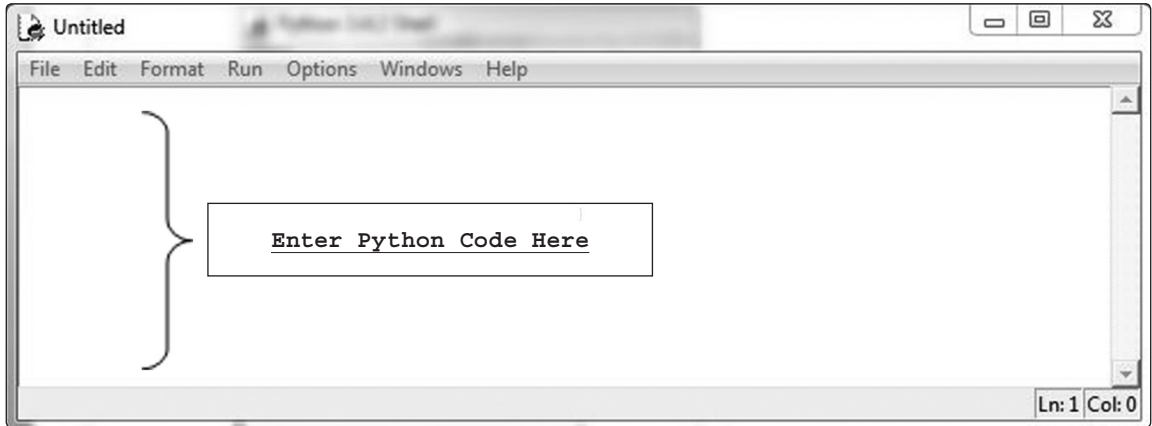


Figure 1.22 Python script mode

You can write a series of instructions in this window and then run it to view the output.

1.6.1 Writing the First Python Program in Script Mode

Use the following steps to create and run your first Python program.

◎ **STEP 1:** Writing Python code in script mode

Let us consider a simple program to print the messages “Hello Welcome to Python”, “Awesome Python!” and “Bye” on the console. The statements needed to print these are

```
print('Hello Welcome to Python')
print('Awesome Python!')
```

Once you write the above statements in Python script mode, they will look like as given in Figure 1.23.

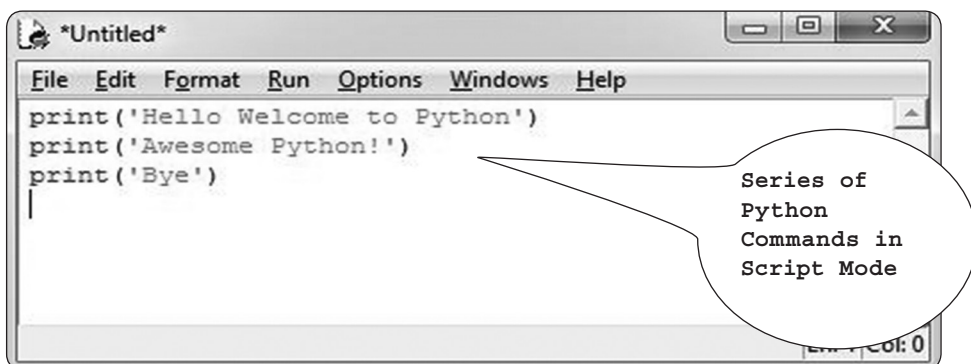


Figure 1.23 Writing program in Python script mode

- ⊙ **STEP 2:** Save the above code written in script mode by some name.

In Figure 1.23 we can see the name ***Untitled**. If you don't save the above code by some specific name, then by default the Python interpreter will save it using the name **Untitled.py**. In this name, **py** indicates that the code is written in Python. The ***** in front of **Untitled** indicates that the program has not been saved. To identify the purpose of a program, you should give it a proper name. Follow the steps given below to save the above program.

- ⊙ **STEP 1:** Click on File and then click on Save or press Ctrl+S. Then you will see the default installation folder (Python34) to save the file (Figure 1.24).

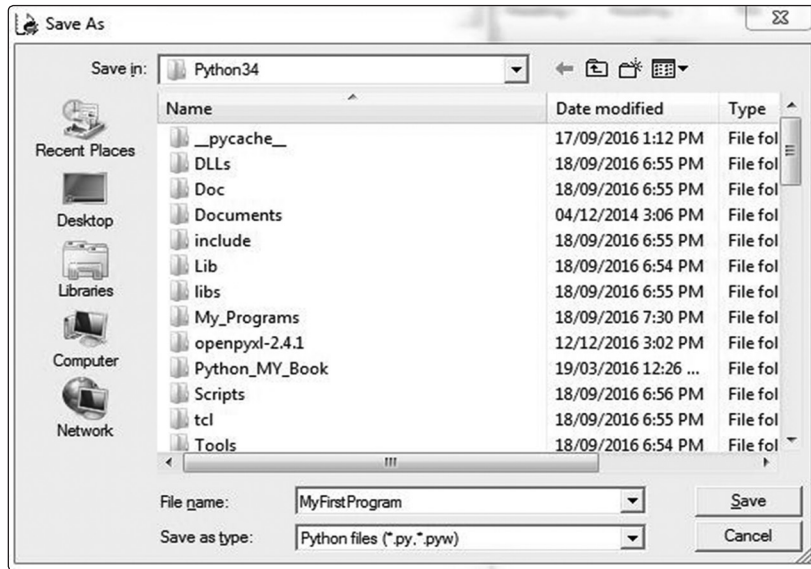


Figure 1.24 Saving a Python program

- ⊙ **STEP 2:** Write the name of your Python program. As it is your first Python program, you can save it as **MyFirstProgram**. Once you write the name of the file, click on Save. After the name is saved, it will get displayed on title bar of the Python script window as shown in Figure 1.25.



Figure 1.25 File name appearing on the title bar

- ◎ **STEP 3:** Executing a Python program: A Python program is executed only after it is saved with a specific file name. Thus, to run the above Python program, click on **Run** and then **Run Module** as shown in Figure 1.26. Alternatively, you can also press **Ctrl+F5** to run the program.

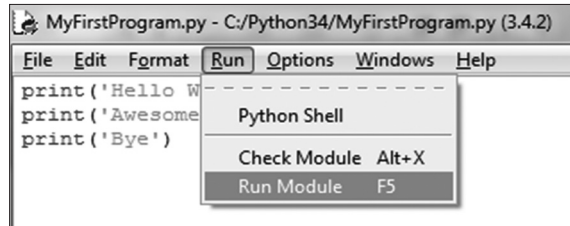


Figure 1.26 Executing a Python program

After clicking on **Run Module** you will see the output of the program if it is written correctly (Figure 1.27).

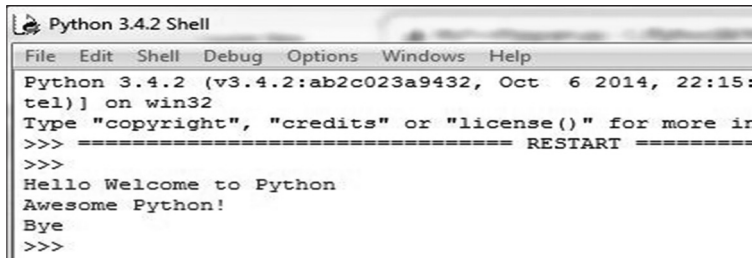


Figure 1.27 Output of a Python program in Python IDLE's interactive shell prompt



Note: Hereafter all the Python programs given as examples in the forthcoming chapters of this book are executed in Python 3.4 IDLE's script mode.

1.7 COMMENTING IN PYTHON

Comments in Python are preceded by a hash symbol (#) on a line and called a **line comment**. Three consecutive single quotation marks ''' are used to give multiple comments or comments on several lines at once and called **paragraph comment**.

When the Python interpreter sees #, it ignores all the text after # on the same line. Similarly, when it sees the triple quotation marks ''' it scans for the next ''' and ignores any text in between the triple quotation marks.

The following program demonstrates the use of comment statements in Python.

#Learn How to Comment in Python

```
print('I Learnt How to Comment in Python')
''' Amazing tool
in Python called Comment'''
print('Bye')
```

Output

```
I Learnt How to Comment in Python
Bye
```

Explanation As explained above, Python ignores all the text in a statement if it is preceded by the # symbol. When the above program is executed, it ignores all the text followed by the # symbol and triple quotation marks.

1.8 INTERNAL WORKING OF PYTHON

When a programmer tries to run a Python code as a script or instructions in an interactive manner in a Python shell, then Python performs various operations internally. All such internal operations can be broken down into a series of steps as shown in **Figure 1.28**.

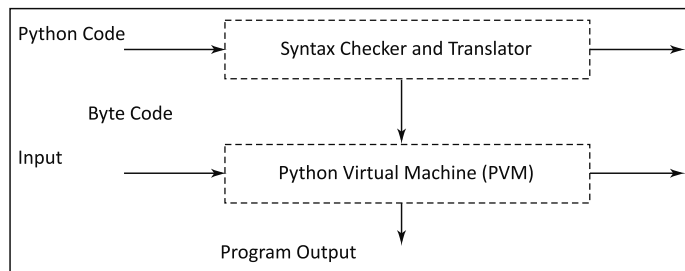


Figure 1.28 Internal working of Python

The Python interpreter performs the following steps to execute a Python program or run a set of instructions in interactive mode.

- ⊙ **STEP 1:** The interpreter reads a Python code or instruction. Then it verifies that the instruction is well formatted, i.e. it checks the syntax of each line. If it encounters any error, it immediately halts the translation and shows an error message.
- ⊙ **STEP 2:** If there is no error, i.e. if the Python instruction or code is well formatted then the interpreter translates it into its equivalent form in low level language called “**Byte Code**”. Thus, after successful execution of Python script or code, it is completely translated into byte code.
- ⊙ **STEP 3:** Byte code is sent to the **Python Virtual Machine (PVM)**. Here again the byte code is executed on **PVM**. If an error occurs during this execution then the execution is halted with an error message.

1.9 PYTHON IMPLEMENTATIONS

The standard implementation of Python is usually called “**CPython**”. It is the default and widely used implementation of the Python programming language. It is written in C. Besides C, there are different implementation alternatives of Python, such as **Jython**, **IronPython**, **Stackless** and **Pypy**. All these Python implementations have specific purposes and roles. All of them make use of simple

Python language but execute programs in different ways. Different Python implementations are briefly explained ahead.

1.9.1 Jython

Originally, Jython was known as “**JPython**”. JPython takes Python programming language syntax and enables it to run on the Java platform. In short, it is used to run Python programs on Java platforms. More details about JPython can be found at <http://jPython.org>.

1.9.2 IronPython

IronPython is an open source implementation of Python for the .NET framework. It uses dynamic language runtime (**DLR**), which is a framework for writing dynamic languages for **.net**. A major use of **IronPython** is to embed **.net** applications. More details about IronPython can be found at <http://ironpythonPython.net>.

1.9.3 Stackless Python

It is a Python programming language interpreter. If you run a program on Stackless Python environment then the running program is split into multithreads. The best thing about multithreads in Stackless Python is the handling of multithreads, which are managed by the language interpreter itself and not by the operating system. More details about Stackless Python can be found at <http://www.stackless.com>.

1.9.4 PyPy

The **PyPy** is a reimplement of Python in Python. In short, the Python interpreter is itself written in Python. It focuses on speed, efficiency and compatibility. It makes use of **Just-in-Time compiler (JIT)** to run the code more quickly as compared to running the same code in regular Python language. More details about **PyPy** Python can be found at <http://pypy.org>.

SUMMARY

- ◆ A computer is an electronic device which accepts data from a user, processes it for calculations specified by the user and generates an output.
- ◆ The hardware of a computer system consists of three main components, viz. **input/output (I/O) unit**, **central processing unit (CPU)** and **memory unit**.
- ◆ A program written in **1s** and **0s** is called machine language.
- ◆ In assembly languages, machine operations are represented by **mnemonic codes** such as **ADD**, **MUL**, etc. and symbolic names that specify the memory address.
- ◆ Programs written in high-level languages are similar to instructions written in English language.
- ◆ An **assembler** is used to translate an **assembly language program** into an equivalent **machine language program**.
- ◆ An **interpreter** or **compiler** is used to translate a program written in a high-level language into an equivalent machine code for execution.
- ◆ An **interpreter** reads the source code line by line and converts it into object code.

- ♦ A **compiler** is a software which translates an entire program written in a high-level language into machine language at one go.
- ♦ A **loader** is a software used to load and relocate the executable program in the main memory during execution.
- ♦ Python is a general purpose, interpreted and objects oriented programming language.
- ♦ You can enter Python statements interactively from the Python prompt `>>>`.
- ♦ Python source programs are case sensitive.
- ♦ The `#` symbol is used to comment a single line in Python.
- ♦ Triple single quotation `'''` marks are used to comment multiple lines in Python.
- ♦ Python programs can be executed on any operating system like Windows, Linux or Ubuntu.

KEY TERMS

- ⇒ **Assembly Language:** Machine operations are represented by mnemonic code
- ⇒ **Byte Code:** The Python interpreter translates Python codes/instructions into their equivalent low level language
- ⇒ **Central Processing Unit (CPU):** It consists of Arithmetic Logical Unit and Control Unit
- ⇒ **High-level Language:** Programs are written in a manner similar to writing instructions in English language
- ⇒ **Jython, IronPython, Stackless, PyPy:** Different implementation alternatives of Python
- ⇒ **Machine Language:** Instructions are written in binary form, i.e. 0s and 1s
- ⇒ **Python Virtual Machine (PVM):** Used to check if the object code contains errors

REVIEW QUESTIONS

A. Multiple Choice Questions

1. Which of the following memory is used to store temporary results in registers when the computation is in progress?
 - a. Primary Memory
 - b. Secondary Memory
 - c. Internal Memory
 - d. None of the above
2. Secondary memory is also called _____.
 - a. Storage Memory
 - b. External Memory
 - c. Only b
 - d. Both a and b
3. _____ is used to translate a program written in a high-level language into its equivalent machine code.
 - a. Compiler
 - b. Linker
 - c. Loader
 - d. Both a and b
4. _____ is used to relocate executable programs to the main memory during execution.
 - a. Linker
 - b. Compiler
 - c. Interpreter
 - d. Loader
5. What is the correct syntax for the print statement in Python 3.0?
 - a. `print()`
 - b. `print`
 - c. `print()`
 - d. None of the above

6. Which of the following symbols is used to make single line comments in Python?
- #
 - '
 - ""
 - &

B. True or False

- Python is not case sensitive.
- We can execute Python on Windows.
- We cannot comment on multiple lines in a Python program.
- An assembler is used to translate an assembly language program into its equivalent machine language program.
- Python is an interpreted language.

C. Exercise Questions

- Explain the classification of programming languages in brief.
- What is a compiler?
- What is an interpreter?
- Differentiate between a compiler and an interpreter.
- What is a linker?
- What is a loader?
- Explain the internal working of Python in brief.
- Describe the memory unit of a computer system in brief.

PROGRAMMING ASSIGNMENTS

- Write a program to display the statement given below in two different lines.
"I am using Python" and "It's my First Assignment"

- Write a program to display the statements given below.

ohhh!!!

What a Python language is!!!

It's Easy!Get Started

- Write a program to display the pattern given below.

```

      A
    A  A
  A    A
A      A

```

- Write a program to display the pattern given below.

```

00000
0  0
0  0
0  0
00000

```

Basics of Python Programming

2

CHAPTER OUTLINE

2.1 Introduction	2.7 Multiple Assignments
2.2 Python Character Set	2.8 Writing Simple Programs in Python
2.3 Token	2.9 The <code>input()</code> Function
2.4 Python Core Data Type	2.10 The <code>eval()</code> Function
2.5 The <code>print()</code> Function	2.11 Formatting Number and Strings
2.6 Assigning Value to a Variable	2.12 Python Inbuilt Functions

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Describe keywords, delimiters, literals, operators and identifiers supported by Python
- Read data from the console using input function
- Assign value or data to a variable and multiple values to multiple variables at a time
- Use `ord` function to obtain a numeric code for a given character value, `chr` function to convert numeric value to a character and `str` function to convert numbers to a string
- Format strings and numbers using the `format` function
- Identify and use various in-built math functions supported by Python

2.1 INTRODUCTION

Computer programming languages are designed to process different kinds of data, viz. numbers, characters and strings in the form of digits, alphabets, symbols etc. to get a meaningful output known as result or information. Thus, program written in any programming language consists of a set of instructions or statements which executes a specific task in a sequential form. Whereas all these instructions are written using specific words and symbols according to syntax rules or the grammar of a language. Hence, every programmer should know the rules, i.e. syntax supported by language for the proper execution of a program.

This chapter describes basics of python programming, i.e. syntax, data types, identifiers, tokens, and how to read values from the user using input function, etc.

2.2 PYTHON CHARACTER SET

Any program written in Python contains words or statements which follow a sequence of characters. When these characters are submitted to the Python interpreter, they are interpreted or uniquely identified in various contexts, such as characters, identifiers, names or constants. Python uses the following character set:

- **Letters:** Upper case and lower case letters
- **Digits:** 0,1,2,3,4,5,6,7,8,9
- **Special Symbols:** Underscore (_), (,), [,], {, }, +, -, *, &, ^, %, \$, #, !, Single quote('), Double quotes(""), Back slash(\), Colon(:), and Semi Colon (;)
- **White Spaces:** ('\\t\\n\\x0b\\x0c\\r'), Space, Tab.

2.3 TOKEN

A program in Python contains a sequence of instructions. Python breaks each statement into a sequence of lexical components known as **tokens**. Each token corresponds to a substring of a statement. Python contains various types of tokens. Figure 2.1 shows the list of tokens supported by Python.

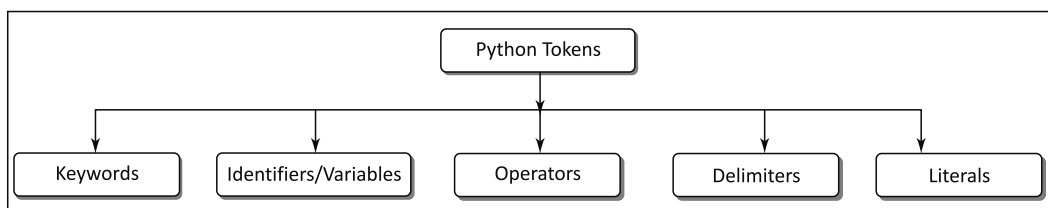


Figure 2.1 Tokens in Python

Details for all the tokens are given next.

2.3.1 Literal

Literals are numbers or strings or characters that appear directly in a program. A list of some literals in Python is as follows:

Example

```
78                #Integer Literal
21.98             #Floating Point Literal
'Q'               #Character Literal
"Hello"           #String Literal
```

Python also contains other literals, such as lists, tuple and dictionary. Details of all such literals are given in the forthcoming chapters.

Display Literals in Interactive Mode

Let us consider a simple example. Print the message “Hello World” as a string literal in Python interactive mode.

Example

```
>>> 'Hello World'
'Hello World'
```

As shown above, type **Hello World** in interactive mode and press enter. Immediately after pressing enter you will see the required message.

2.3.2 Value and Type on Literals

Programming languages contain data in terms of input and output and any kind of data can be presented in terms of **value**. Here **value** can be of any form like literals containing numbers, characters and strings.

You may have noticed that in the previous example we wrote ‘Hello World’ in single quotes. However, we don’t know the type of value in it. To know the exact type of any value, Python offers an in-built method called **type**.

The syntax to know the type of any value is **type (value)**

Example

```
>>> type('Hello World')
<class 'str'>
>>> type(123)
<class 'int'>
```

Thus, when the above examples are executed in Python interactive mode, return type of value is passed to the in-built function `type()`.

2.3.3 Keywords

Keywords are reserved words with fixed meanings assigned to them. Keywords cannot be used as identifiers or variables. Table 2.1. shows the complete list of keywords supported by Python.

Table 2.1 List of Python keywords for Python version 3.0

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

2.3.4 Operator

Python contains various operators, viz. arithmetic, relational, logical and bitwise operators, as shown in Table 2.2.

Table 2.2 Operators in Python

<i>Operator Type</i>	<i>Operators</i>
+ - * / // % **	Arithmetic Operator
== != < > <= >=	Relational Operator
and not or	Logical Operator
& ~ ^ << >>	Bitwise Operator

Details about Python operators like operators and expressions are given in Chapter 3.

2.3.5 Delimiter

Delimiters are symbols that perform a special role in Python like grouping, punctuation and assignment. Python uses the following symbols and symbol combinations as delimiters.

```
( ) [ ] { }
, : . \ = ;
+= -= *= /= //= %=
&= |= ^= >>= <<= **=
```

2.3.6 Identifier/Variable

Identifier is the name used to find a variable, function, class or other objects. All identifiers must obey the following rules.

An identifier:

- Is a sequence of characters that consists of letters, digits and underscore

- Can be of any length
- Starts with a letter which can be either lower or upper case
- Can start with an underscore '_'
- Cannot start with a digit
- Cannot be a keyword.

Some examples of valid identifiers are Name, Roll_NO, A1, _Address etc.

Python gives a **syntax error** if a programmer writes an invalid identifier. Some examples of invalid identifiers are First Name, 12Name, for, Salary@

If we type the invalid identifiers given above in Python interactive shell, it will show an error as these are invalid.

Example

```
>>> First Name
SyntaxError: invalid syntax
>>> 12Name
SyntaxError: invalid syntax
>>> for
SyntaxError: invalid syntax
```

2.4 PYTHON CORE DATA TYPE

All features in Python are associated with an **object**. It is one of the primitive elements of Python. Further, all kinds of objects are classified into **types**. One of the easiest types to work with is numbers, and the native data types supported by Python are string, integer, floating point numbers and complex numbers.

The following section details the basic data types supported by Python.

2.4.1 Integer

From simple Mathematics, we know that an integer is a combination of positive and negative numbers including (zero) 0. In a program, integer literals are written without commas and a leading minus sign to indicate a negative value. Following is an example of simple integer literals displayed in Python interactive mode.

Example

```
>>> 10
10
>>> 1220303
1220303
>>> -87
-87
```

Integer literals can be octal or hexadecimal in format. All the above examples are of decimal type integers. Decimal integers or literals are represented by a sequence of digits in which the first digit is non-zero. To represent an **octal**, **0o**, i.e. a zero and a lower or upper case letter **O** followed by a sequence of digits from **0 to 7** is used. An example of octal literals is given as follows.

Example

```
>>> 0o12
10
>>> 0o100
64
```



Note: In Python version 2.6 or earlier, octal literals were represented by the leading letter **O**, followed by a sequence of digits. In Python 3.0, octal literals have to be accompanied by a leading **0o**, i.e. a zero and a lower or upper case letter **O**.

In the previous section, we have learnt about representation of numbers as default **decimal (base 10)** notation and octal (**base 8**) notation. Similarly, numbers can also be represented as hexadecimal (**base 16**) notation using **0x** (zero and the letter X) followed by a sequence of digits. Simple examples of hexadecimal literals displayed in Python interactive mode are given as follows:

Example

```
>>> 0x20
32
>>> 0x33
51
```



Note: Integer in Python 2.6 (int and long)—In Python 2.6 there are two types of integers. One of 32 bits and another having unlimited precession. Python 2.6 automatically converts integers to long integers if the value of the integer overflows 32 bits.

Integers in Python 3.0 (Only int type)—In Python 3.0 the normal **int** and **long** integer have been merged. Hence, there is only one type called integer.

The int Function

The **int** function converts a string or a number into a whole number to integer. The **int** function removes everything after the decimal point. Consider the following example.

Example

```
>>> int(12.456)
12
```

The following example converts a string to an integer.

Example

```
>>> int('123')
123
```

2.4.2 Floating Point Number

The value of π (3.14) is an example of a real number in mathematics. It consists of a whole number, decimal point and fractional part. The length of real numbers has infinite precession, i.e. the digits in the fractional part can continue forever. Thus, Python uses floating point numbers to represent real numbers. A floating-point number can be written using a **decimal notation** or **scientific notation**. Some examples of floating point numbers displayed in Python interactive mode are given as follows:

Example

```
>>> 3.7e1
37.0
>>> 3.7
3.7
>>> 3.7*10
37.0
```

The above example shows the representation of floating point number **37.0** in both decimal and scientific manner. Scientific notations are very helpful because they help programmers to represent very large numbers. Table 2.3 shows decimal notations in scientific notation format.

Table 2.3 Example of floating point numbers

<i>Decimal Notation</i>	<i>Scientific Notation</i>	<i>Meaning</i>
2.34	2.34e0	$2.34 * 10^0$
23.4	2.34e1	$2.34 * 10^1$
234.0	2.34e2	$2.34 * 10^2$

The float Function

The **float** function converts a string into a floating-point number. A programmer can make use of float to convert string into float. Consider the following example.

Example

```
>>>float('10.23')
10.23
```

2.4.3 Complex Number

A complex number is a number that can be expressed in the form **a+bj**, where **a** and **b** are real numbers and **j** is an imaginary unit. Simple example of complex numbers displayed in Python interactive mode is given as follows:

Example

```
>>> 2+4j
(2+4j)
```

```
..
>>> type(2+4j)
<class 'complex'>
>>> 9j
9j
>>> type(9j)
<class 'complex'>
```

Here we have written a simple kind of complex number and using **type** we have checked the type of the number.

2.4.4 Boolean Type

The Boolean data type is represented in Python as type **bool**. It is a primitive data type having one of the two values, viz. **True** or **False**. Internally, the **True** value is represented as 1 and False as 0. In the following example check the type of **True** and **False** value in Python interactive mode.

```
>>> type(True)
<class 'bool'>
>>> False
False
>>> type(False)
<class 'bool'>
```

The Boolean type is used to compare the two values. For example, when relational operators, such as **==**, **!=**, **<=**, **>=** are used in between two operands then it returns the value as **True** or **False**.

Example

```
>>> 5 == 4
False
>>> 5 == 5
True
>>> 4 < 6
True
>>> 6 > 3
True
```

2.4.5 String Type

A string literal or string in Python can be created using single, double and triple quotes. A simple example of type as string is given as follows:

Example

```
>>> D = 'Hello World'
>>> D
'Hello World'
>>> D="Good Bye"
```

```

>>> D
'Good Bye'
>>> Sentence
'Hello, How are you? Welcome to the world of Python Programming. It is just the
beginning. Let us move on to the next topic.'
>>> Sentence
'Hello, How are you? Welcome to the world of Python Programming. It is just the
beginning. Let us move on to the next topic.'

```

In the previous examples, we presented string literals in three different formats, viz. single quote, double quote and triple single quotes. The triple single quotes are used to write a multiline string.

The str Function

The `str` function is used to convert a number into a string. The following example illustrates the same.

```

>>> 12.5          #Floating Point Number
12.5
>>> type(12.5)
<class 'float'>
>>> str(12.5)     #Convert floating point number to string
'12.5'

```

The String Concatenation (+) Operator In both mathematics and programming, we make use of '+' operator to add two numbers. Similarly, '+' operator is used to concatenate two different strings. The following example illustrates the use of + operator on strings.

```

>>> "Wooooow" + "Python Programming"
'WooooowPython Programming'      #Concatenates two different strings

```

2.5 THE print() FUNCTION

In Python, a function is a group of statements that are put together to perform a specific task. The task of `print` function is to display the contents on the screen. The syntax of `print` function is:

Syntax of `print()` function:

```
print(argument)
```

The argument of the `print` function can be a value of any type `int`, `str`, `float` etc. It can also be a value stored in a variable. Simple examples of `print()` function executed in interactive mode of Python are given as follows:

Example

Display messages using `print()`

```

>>> print('Hello Welcome to Python Programming')
Hello Welcome to Python Programming

```

```
..
>>> print(10000)
10000
>>> print("Display String Demo")
Display String Demo
```

Suppose you want to print a message with quotation marks in the output as

```
print("The flight attendant asked, "May I see your boarding pass?")
```

If you try to run the above statement as is, Python will show an error. For Python, the second quotation mark is the end of the string and hence it does not know what to do with the rest of the characters. To overcome this problem Python has a special notation to represent a special character. This special notation consists of a **backslash (\)** followed by a letter or a combination of digits and is called an **escape sequence**. Using **backslash**, the special characters within **print** can be written as shown below.

Example

```
>>> print("The flight attendant asked,\"May I see your boarding pass?\" ")
The flight attendant asked, "May I see your boarding pass?"
```

In the above example, we have used backslash before the quotation marks to display the quotation marks in the output.

Table 2.4 illustrates a list of escape sequences used in Python.

Table 2.4 Python escape sequences

Character Escape Sequence	Name
\'	Single Quote
\"	Double Quote
\n	Linefeed
\f	Formfeed
\r	Carriage return
\t	Tab
\\	Backslash
\b	Backspace



Note: The syntax of **print** function is different in Python 2.X. It is **print arguments**

Python 2.X does not use an additional parenthesis. If you try to execute the print statement without parenthesis, unlike Python 3, it will raise a syntax error.

Example:

```
>>> print 'Hello World'
```

Syntax Error: Missing parentheses in call to 'print'

(Contd.)

Python programs are case sensitive. Python raises an error if a programmer tries to replace `print` by `Print`.

Example:

```
>>> Print('hi')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    Print('hi')
NameError: name 'Print' is not defined
```

2.5.1 The `print()` Function with end Argument

Consider a simple program of a print statement.

PROGRAM 2.1 | Write a program to display the messages “Hello”, “World” and “Good Bye”. Each of the three messages should get displayed on a different line.

```
print('Hello')
print('World')
print('Good Bye')
```

Output

```
Hello
World
Good Bye
```

In the above program, we have displayed each message in a different line. In short, the `print` function automatically prints a **linefeed** (`\n`) to cause the output to advance to the next line. However, if you want to display the messages “Hello” “World” and “Good Bye” in one line without using a single print statement, then you can invoke the **print** function by passing a special argument named `end=' '`. The following program illustrates the use of the end argument within the print function.

PROGRAM 2.2 | Write a basic program to make use of the end key and display the messages “Hello” “World” and “Good Bye” in one line.

```
print(' Hello',end=' ')
print(' World',end=' ')
print(' Good Bye')
```

Output

```
Hello World Good Bye
```


2.6 ASSIGNING VALUE TO A VARIABLE

In Python, the equal sign (=) is used as the assignment operator. The statement for assigning a value to a variable is called an **assignment statement**. The syntax used to assign value to a variable or identifier is:

Variable = expression

In the above syntax, expression may contain information in terms of values, even some time expression may contain operands with operators which evaluates to a value.

Let us consider the following example of assigning and displaying the value of a variable in Python interactive mode.

Example

```
>>> Z = 1                                # Assign value 1 to variable Z
>>> Z                                    # Display value of Z
1
>>> radius = 5                          #Assign value 5 to the variable radius
>>> radius                               #Display value of variable radius
5
>>> R = radius + Z                       #Assign the addition of radius and Z to R
>>> R                                    #Display value of Variable R
6
>>> E = (5 + 10 * (10 + 5))             #Assign the value of the expression to E
>>> E
155
```

This example explains how a variable can be used to assign a value and how a variable can be used on both the sides of = operator. As given in the above example:

$R = \text{radius} + Z$

In the above assignment statement, the result of **radius+Z** is assigned to R. Initially the value assigned to Z is 1. Once Python executes the above statement, it adds the most recent value of Z and assigns the final value to a variable R.



Note: To assign a value to a variable, you must place the variable name to the left of the assignment operator. If you write in the following manner, Python will display an error.

```
>>> 10 = X
```

Syntax Error: can't assign to literal

In Mathematics, $E = (5 + 10 * (10 + 5))$, denotes an equation, but in Python $E = (5 + 10 * (10 + 5))$ is an assignment statement that evaluates the expression and assigns the result to E.

2.6.1 More on Assigning Values to Variables

Consider the following example where a value has been assigned to multiple variables.

Example

```

>>> P = Q = R = 100           #Assign 100 to P, Q and R
>>> P                         #Display value of Variable P
100
>>> Q                         #Display value of Variable Q
100
>>> R                         #Display Value of Variable R
100

```

In the above example, we have assigned value 100 to **P**, **Q** and **R**. The statement **P = Q = R = 100** is equivalent to

```

P = 100
Q = 100
R = 100

```

2.6.2 Scope of a Variable

Each variable has a scope. The scope of a variable is a part of the program where a variable can be referenced. More details on scope of variables are given in Chapter 6. Consider the following simple example and run it on Python interpreter.

```

>>> C = Count + 1
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    C = Count + 1
NameError: name 'Count' is not defined

```

In the above example, we have written a statement as **C = Count + 1**, but when Python tries to execute the above statement it raises an error, viz. **"Count is not defined"**. To fix the above error in Python the variable must be assigned some value before it is used in an expression. Thus, the correct version of the above code written in Python interactive mode is as given as follows:

```

>>> Count = 1
>>> C = Count + 1
>>> C
2

```



Note: A variable must be assigned a value before it can be used in an expression.

2.7 MULTIPLE ASSIGNMENTS

Python supports simultaneous assignment to multiple variables. The syntax of multiple assignments is

```
Var1, Var2, Var3, ..... = Exp1, Exp2, Exp3, ..... ExpN
```

In the above syntax, Python simultaneously evaluates all the expressions on the right and assigns them to a corresponding variable on the left.

Consider the following statements to swap the values of the two variables **P** and **Q**. The common approach to swap the contents of the two variables is shown as follows:

Example

```
>>> P = 20
>>> Q = 30
>>> Temp = P           #Save value of variable P into a variable Temp
>>> P = Q               #Assign value of Q to P
>>> Q = Temp           #Assign the value of Temp to Q
#After Swapping the value of P, and Q are as follows.
>>> P
30
>>>
>>> Q
20
```

In the above code, we have used the following statements to swap the values of the two variables **P** and **Q**.

```
Temp = P
P = Q
Q = Temp
```

However, by using the concept of multiple assignment, you can simplify the task of swapping two numbers.

```
>>> P, Q = Q, P      #Swap P with Q & Q with P
```

Thus, the entire code to swap two numbers using multiple assignment is as follows:

```
>>> P = 20           #Initial Values of P and Q
>>> Q = 30
>>> P
20
>>> Q
30
>>> P, Q = Q, P      #Swap values of P and Q
>>> P               #Display value of P
30
>>> Q               #Display Value of Q
20
```

2.8 WRITING SIMPLE PROGRAMS IN PYTHON

How can a simple program to calculate the area of a rectangle be written in Python?

We know that a program is written in a step-wise manner. Consider the initial steps given as follows:.

- ☉ **STEP 1:** Design an algorithm for the given problem.

An algorithm describes how a problem is to be solved by listing all the actions that need to be taken. It also describes the order in which the series of actions need to be carried out. An algorithm helps a programmer to plan for the program before actually writing it in a programming language. Algorithms are written in simple English language along with some programming code.

- ☉ **STEP 2:** Translate an algorithm to programming instructions or code.

Let us now write an algorithm to calculate the area of a rectangle.

Algorithm to Calculate the Area of a Rectangle

- a. Get the length and breadth of the rectangle from the user.
- b. Use the relevant formula to calculate the area

$$\text{Area} = \text{Length} * \text{Breadth}$$

- c. Finally display the area of the rectangle.

This algorithm can be written as code as shown in program 2.3.

PROGRAM 2.3 | Write a program to calculate the area of a rectangle.

```
Length = 10
breadth = 20
print(' Length = ',length,' Breadth = ',breadth)
area = length * breadth
print(' Area of Rectangle is = ',area)
```

Output

```
Length = 10 Breadth = 20
Area of Rectangle is = 200
```

Explanation In the above program, two variables, viz. length and breadth are initialized with values 10 and 20, respectively. The statement **area = length x breadth** is used to compute the area of the rectangle.

Here the values of the variables are fixed. However, a user may want to calculate the area of different rectangles with different dimensions in future. In order to get the values according to the user's choice, a programmer must know how to read the input values from the console. This is described in the next section.

2.9 THE `input()` FUNCTION

The `input()` function is used to accept an input from a user. A programmer can ask a user to input a value by making use of `input()`.

`input()` function is used to assign a value to a variable.

Syntax

```
Variable_Name = input()
```

OR

```
Variable_Name = input('String')
```

2.9.1 Reading String from the Console

A simple program of `input()` function to read strings from the keyboard is given in Program 2.4.

PROGRAM 2.4 | Write a program to read strings from the keyboard.

```
Str1 = input('Enter String1:')
Str2 = input('Enter String2: ')
print(' String1 = ',Str1)
print(' String2 = ',Str2)
```

Output

```
Enter String1:Hello
Enter String2: Welcome to Python Programming
String1 = Hello
String2 = Welcome to Python Programming
```

Explanation The `input()` function is used to read the string from the user. The string values entered from the user are stored in two separate variables, viz. **Str1** and **Str2**. Finally all the values are printed by making use of `print()` function.

Let us also check what happens if by mistake the user enters digits instead of characters. Program 2.5 illustrates the same.

PROGRAM 2.5 | Write a program to enter digits instead of characters.

```
print(' Please Enter the Number:')
X = input()
print(' Entered Number is: ',X)
print(' Type of X is:')
print(type(X))
```

(Contd.)

Output

```
Please Enter the number:
60
Entered Number is: 60
Type of X is:
<class 'str'>
```

Explanation We know that Python executes statements sequentially. Hence, in the above program the first print statement is printed, i.e. 'Please Enter the Number.' But when it runs the second statement, i.e. **X = input()** the programming execution stops and waits for the user to type the text using the keyboard. The text that the user types is not committed until he/she presses Enter. Once the user enters some text from the keyboard, the value gets stored in an associated variable. Finally, the entered value is printed on the console. The last statement is used to check the type of value entered.



Note: The input function produces only string. Therefore, in the above program even if the user enters a numeric, i.e. integer value, Python returns the type of input value as string.

In the above program, how does a programmer read integer values using the input function?

Python has provided an alternative mechanism to convert existing string to int. A programmer can use **int** to convert a string of digits into an integer. Program 2.6 illustrates the use of **int** and **input ()**.

PROGRAM 2.6 | Write a program to demonstrate the use of **int** and **input** function.

```
print(' Please Enter Number')
Num1 = input()                #Get input from user
print(' Num1 = ',Num1)        #Print value of Num1
print(type(Num1))             #Check type of Num1
print(' Converting type of Num1 to int ')
Num1 = int(Num1)              #Convert type of Num1 from str to int
print(Num1)                   #print the value of Num1
print(type(Num1))             #Check type of Num1
```

Output

```
Please Enter Number
12
Num1 = 12
<class 'str'>
Converting type of Num1 to int
12
<class 'int'>
```

Explanation The above program asks the user for input. The user has entered the input as 12 but it is of type str. By making use of int, i.e. the statement **Num1 = int(Num1)**, it converts the existing type to int.

We can minimise the number of lines in a program directly by making use of **int** before **input** function. A shorter version of the above program is given in Program 2.7.

PROGRAM 2.7 | Write a program to demonstrate the use of **int** before input.

```
Num1 = int(input(' Please Enter Number:'))
print(' Num1 = ',Num1) #Print the value of Num1
print(type(Num1)) #Check type of Num1
```

Output

```
Please Enter Number:
20
Num1 = 20
<class 'int'>
```

PROGRAM 2.8 | Write a program to read the length and breadth of a rectangle from a user and display the area of the rectangle.

```
print(' Enter Length of Rectangle:', end=' ')
Length = int(input()) #Read Length of Rectangle
print(' Enter Breadth of Rectangle:', end=' ')
Breadth = int(input()) #Read Breadth of Rectangle
Area = Length * Breadth #Compute Area of Rectangle
print('-----Details of Rectangle-----')
print(' Length = ',Length) #Display Length
print(' Breadth = ',Breadth) #Display Breadth
print(' Area of rectangle is :',Area)
```

Output

```
Enter Length of Rectangle: 10
Enter Breadth of Rectangle: 20
-----Details of Rectangle-----
Length = 10
Breadth = 20
Area of rectangle is: 200
```



Note: A programmer can make use of any type to convert the string into a specific type.

Example:

```
X = int(input())      #Convert it to int
X = float(input())    #Convert it to float
```

PROGRAM 2.9 | Write a program to add one integer and floating type number.

```
print('Enter integer number: ',end='')
Num1 = int(input()) # Read Num1
print('Enter Floating type number:',end='')
Num2 = float(input()) #Read Num2
print(' Number1 = ',Num1) #Print Num1
print(' Number2 = ',Num2) #Print Num2
sum = Num1 + Num2 #Calculate Sum
print(' sum = ',sum) #Display Sum
```

Output

```
Enter integer number: 2
Enter Floating type number:2.5
Number1 = 2
Number2 = 2.5
sum = 4.5
```



Note: Python 3 uses **input()** method to read the input from the user.

Python 2 uses **raw _ input()** method to read the input from the user.

In subsequent programs in this chapter we are going to use **input()** method only as all programs are executed in Python 3.

2.10 THE **eval()** FUNCTION

The full form of **eval** function is to evaluate. It takes a string as parameter and returns it as if it is a Python expression. For example, if we try to run the statement **eval('print("Hello")')** in Python interactive mode, it will actually run the statement **print("Hello")**.

Example

```
>>> eval('print("Hello")')
Hello
```

The **eval** function takes a string and returns it in the type it is expected. The following example illustrates this concept.

Example

```
>>> X = eval('123')
>>> X
123
>>> print(type(X))
<class 'int'>
```

2.10.1 Apply eval() to input() Function

In the previous section we learnt about the `input()` function in detail. We know that the `input()` function returns every input by the user as string, including numbers. And this problem was solved by making use of `type` before `input()` function.

Example

```
X = int (input('Enter the Number'))
```

Once the above statement is executed, Python returns it into its respective type.

By making use of `eval()` function, we can avoid specifying a particular type in front of `input()` function. Thus, the above statement,

```
X = int (input('Enter the Number'))
```

can be written as:

```
X = eval(input('Enter the Number'))
```

With respect to the above statement, a programmer does not know what values a user can enter. He/she may enter a value of any type, i.e. int, float, string, complex etc. By making use of `eval`, Python automatically determines the type of value entered by the user. Program 2.10 demonstrates the use of `eval()`.

PROGRAM 2.10 | Write a program to display details entered by a user, i.e. name, age, gender and height.

```
Name = (input('Enter Name :'))
Age = eval(input('Enter Age :')) #eval() determine input type
Gender = (input('Enter gender:'))
Height = eval(input('Enter Height:')) #eval() determine input type
print(' User Details are as follows: ')
print(' Name: ',Name)
print(' Age: ',Age)
print(' Gender: ',Gender)
print(' Height ',Height)
```

Output

```
Enter Name: Donald Trump
```

(Contd.)

```

Enter Age: 60
Enter Gender: M
Enter Height:5.9
User details are as follows:
Name: Donald Trump
Age: 60
Gender: M
Height: 5.9

```

Explanation In the above program we have used `eval()` in front of `input()` function as:

```
Age = eval(input('Enter Age :'))
```

The above statement reads the input as a string and converts a string into a number. After the user enters a number and presses Enter, the number is read and assigned to a variable name.

2.11 FORMATTING NUMBER AND STRINGS

A formatting string helps make string look presentable to the user for printing. A programmer can make use of **format** function to return a formatted string. Consider the following example to calculate the area of a circle before using this function.

PROGRAM 2.11 | Write a program to calculate the area of a circle.

```

radius = int(input('Please Enter the Radius of Circle: '))
print(' Radius = ', radius)           #Print Radius
PI = 3.1428                           #Initialize value of PI
Area = PI * radius * radius           #Calculate Area
print(' Area of Circle is: ',Area)    #Print Area

```

Output

```

Please Enter the Radius of Circle: 4
Radius = 4
Area of Circle is: 50.2848

```

In the above program, the user entered the radius as 4. Thus, for a circle having radius 4, it has displayed the area as **50.2848**. To display only two digits after the decimal point, make use of **format()** function. The syntax of format function is

```
format(item, format-specifier)
```

item is the number or string and **format-specifier** is a string that specifies how the item is formatted. A simple example of **format()** function executed in Python interactive mode is given as follows:

Example

```

>>> x = 12.3897                                #Assign value to variable x
>>> print(x)                                    # print x
12.3897
>>> format(x," .2f")                            #Return formatted string
'12.39'

```

PROGRAM 2.12 | Make use of `format()` function and display the area of a circle.

```

radius = int(input('Please Enter the Radius of Circle: '))
print(' Radius = ', radius)
PI = 3.1428
Area = PI * radius * radius
print(' Area of Circle is: ',format(Area,'.2f'))

```

Output

```

Please Enter the Radius of Circle: 4
Radius = 4
Area of Circle is: 50.28

```

In the above program, the statement, `print('Area of Circle is: ',format(Area,'.2f'))` is used to display the area of the circle. Within format function, `Area` is an item and `'2f'` is the format specifier which tells the Python interpreter to display only two digits after the decimal point.

2.11.1 Formatting Floating Point Numbers

If the item is a float value, we can make use of specifiers to give the **width** and **precision**. We can use `format` function in the form **width.precisionf**. Precision specifies the number of digits after the decimal point and width specifies the width of the resultant string. In **width.precisionf** `'f'` is called conversion code. `'f'` indicates the formatting for floating point numbers. Examples of floating point numbers are

```

print(format(10.345,"10.2f"))
print(format(10,"10.2f"))
print(format(10.32245,"10.2f"))

```

displays the output as follows:

```

10.35
10.00
10.32

```

In above example, print statement uses 10.2f as the format specifier. 10.2f is explained in detail in **Fig. 2.2**.

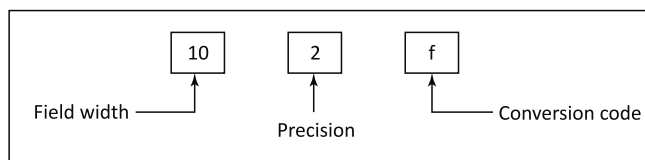


Figure 2.2 Format specifier details

The actual representation of the above output is:

10									
					1	0	.	3	5
					1	0	.	0	0
					1	0	.	3	2

The gray box denotes a blank space. The decimal point is also counted as 1.

2.11.2 Justifying Format

By default, the integer number is right justified. You can insert < in the format specifier to specify an item to be left justified. The following example illustrates the use of right and left justification.

Example

```
>>>print(format(10.234566,"10.2f")) #Right Justification Example
10.23
>>> print(format(10.234566,"<10.2f")) #Left Justification Example
10.23
```

The actual representation of the above output for left justification is:

1	0	.	2	3					
10									

2.11.3 Integer Formatting

In case of integer formatting, you can make use of conversion code **d** and **x**. **d** indicates that the integer is to be formatted, whereas **x** specifies that the integer is formatted into a hexadecimal integer. The following example illustrates integer formatting.

Example

```
>>>print(format(20,"10x")) #Integer formatted to Hexadecimal Integer
14
>>> print(format(20,"<10x"))
14
```

```
..
>>> print(format(1234,"10d"))    #Right Justification
1234
```

In the above example the statement **print(format(20,"10x"))** converts the number 20 into a hexadecimal, i.e. 14.

2.11.4 Formatting String

A programmer can make use of conversion code **s** to format a string with a specified width. However, by default, string is left justified. Following are some examples of string formatting.

Example

```
>>> print(format("Hello World!","25s")) #Left Justification Example
Hello World!

>>>print(format("HELLO WORLD!",">20s"))    #String Right Justification
        HELLO WORLD!
```

In the above example, the statement **(format("HELLO WORLD!",">20s"))** displays the output as:

								H	E	L	L	O		W	O	R	L	D	!
←								20 →											

In print function 20 specifies the string to be formatted with a width of 20. In the second print statement, **">"** is used for right justification of the given string.

2.11.5 Formatting as a Percentage

The conversion code **%** is used to format a number as a percentage. The following example illustrates the same.

Example

```
>>> print(format(0.31456,"10.2%"))
31.46%

>>> print(format(3.1,"10.2%"))
310.00%

>>> print(format(1.765,"10.2%"))
176.50%
```

In the above example, the statement **print(format(0.31456,"10.2%"))**, contains the format specifier **10.2%**. It causes the number to be multiplied by 100. The **10.2%** denotes the integer to be formatted with a width of 10. In width, **%** is counted as one space.

2.11.6 Formatting Scientific Notation

While formatting floating point numbers we have used the conversion code **f**. However, if we want to format a given floating point number in scientific notation then the conversion code **e** will be used. An example of formatting floating point numbers is given as follows:

Example

```
>>> print(format(31.2345, "10.2e"))
      3.12e+01
>>> print(format(131.2345, "10.2e"))
      1.31e+02
```

Most frequently used specifiers are shown in Table 2.5.

Table 2.5 Frequently used specifiers

<i>Specifier</i>	<i>Format</i>
10.2f	Format floating point number with precision 2 and width 10.
<10.2f	Left Justify the floating point number.
>10.2f	Right Justify the formatted item.
10X	Format integer in hexadecimal with width 10
20s	Format String with width 20
10.2%	Format the number in decimal

2.12 PYTHON INBUILT FUNCTIONS

In the previous sections of this chapter we have learnt how to use the functions **print**, **eval**, **input** and **int**. We know that a function is a group of statements that performs a specific task. Apart from the above functions, Python supports various inbuilt functions as well. A list of all inbuilt functions supported by Python is given in **Table 2.6**. It provides the name of a function, its description and examples executed in Python interactive mode.

Table 2.6 Inbuilt functions in Python

<i>Function</i>	<i>Description</i>
abs(x)	Returns absolute value of x
Example	
<pre>>>> abs(-2) 2 >>> abs(4) 4</pre>	

(Contd.)

max(x1, X2, X3,.....,XN)	Returns largest value among X1, X2, X3, X4,... , XN
Example:	
<pre>>>> max(10,20,30,40) 40</pre>	
min(x1, X2, X3,.....,XN)	Returns minimum value among X1, X2, X3, X4,... , XN.
pow(X, Y)	Return the X^Y
Example:	
<pre>>>> pow(2,3) 8</pre>	
round(x)	Returns an integer nearest to the value of x.
Example:	
<pre>>>> round(10.34) 10 >>> round(10.89) 11</pre>	

Functions given in Table 2.6 are not enough to solve mathematical calculations. Thus, Python has an additional list of functions defined under Python's **math** module to solve problems related to mathematical calculations. List of functions under the **math** module is given in Table 2.7.

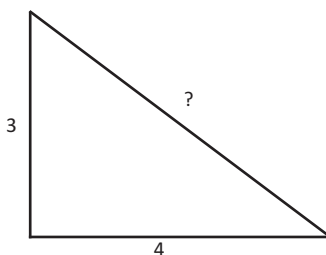
Table 2.7 Inbuilt mathematical functions in Python

Function	Example	Description
ceil(X)	<pre>>>> math.ceil(10.23) 11</pre>	Round X to nearest integer and returns that integer.
floor(X)	<pre>>>> math.floor(18.9) 18</pre>	Returns the largest value not greater than X
exp(X)	<pre>>>> math.exp(1) 2.718281828459045</pre>	Returns the exponential value for e^x
log(X)	<pre>>>> math.log(2.71828) 0.999999327347282</pre>	Returns the natural logarithmic of x (to base e)
log(x,base)	<pre>>>> math.log(8,2) 3.0</pre>	Returns the logarithmic of x to the given base
sqrt(X)	<pre>>>>math.sqrt(9) 3.0</pre>	Return the square root of x
Sin(X)	<pre>>>> math.sin(3.14159/2) 0.999999999991198</pre>	Return the sin of X, where X is the value in radians
asin(X)	<pre>>>> math.asin(1) 1.5707963267948966</pre>	Return the angle in radians for the inverse of sine
cos(X)	<pre>>>> math.cos(0) 1.0</pre>	Return the sin of X, where X is the value in radians
aCos(X)	<pre>>>> math.acos(1) 0.0</pre>	Return the angle in radians for the inverse of cosine

(Contd.)

tan(X)	>>> math.tan(3.14/4) 0.9992039901050427	Return the tangent of X, where X is the value in radians
degrees(X)	>>> math.degrees(1.57) 89.95437383553924	Convert angle X from radians to degrees
Radians(X)	>>> math. radians(89.99999) 1.5707961522619713	Convert angle x from degrees to radians

PROGRAM 2.13 | Write a program to calculate the hypotenuse of the right-angled triangle given as follows:



$$\begin{aligned}
 \text{Hypotenuse} &= \text{Square_Root} \{(\text{Base})^2 + (\text{Height})^2\} \\
 &= \text{Square_Root} \{(3)^2 + (4)^2\} \\
 &= 5
 \end{aligned}$$

```

import math #Import Math Module
Base = int(input('Enter the base of a right-angled triangle:'))
Height = int(input('Enter the height of a right-angled triangle:'))
print(' Triangle details are as follows: ')
print(' Base = ',Base)
print(' Height = ',Height)
Hypotenuse = math.sqrt(Base * Base + Height * Height )
print(' Hypotenuse =',Hypotenuse)

```

Output

```

Enter the base of a right-angled triangle:3
Enter the height of a right-angled triangle:4
Triangle details are as follows:
Base = 3
Height = 4
Hypotenuse = 5.0

```

Explanation In the above program, the first line '**import math**' is used to include all in-built functions supported by Python under the math module. The **input** function is used to read the

base and height of the right-angled triangle. The statement **math.sqrt** is executed to find the square root of the number. Finally, print for the hypotenuse of the right-angled triangle is given.

2.12.1 The ord and chr Functions

As we know, a string is a sequence of characters. It can include both text and numbers. All these characters are stored in a computer as a sequence of 0s and 1s. Therefore, a process of mapping a character to its binary representation is called **character encoding**.

There are different ways to encode a character. The **encoding scheme** decides the manner in which characters are encoded. The American Standard Code for Information Interchangeable (**ASCII**) is one of the most popular encoding schemes. It is a 7-bit encoding scheme for representation of all lower and upper case letters, digits and punctuation marks. The ASCII uses numbers from 0 to 127 to represent all characters. Python uses the in-built function **ord(ch)** to return the ASCII value for a given character. The following example demonstrates the use of the in-built function **ord()**.

Example

```
>>> ord('A') #Returns ASCII value of Character 'A'
65
>>> ord('Z') #Returns ASCII Value of Character of 'Z'
90
>>> ord('a') #Returns ASCII Value of Character of 'a'
97
>>> ord('z') #Returns ASCII Value of Character of 'z'
122
```

The **chr(Code)** returns the character corresponding to its code, i.e. the ASCII value. The following example demonstrates the use of in-built function **chr()**.

Example

```
>>> chr(90)
'Z'
>>> chr(65)
'A'
>>> chr(97)
'a'
>>> chr(122)
'z'
```

PROGRAM 2.14 | Write a program to find the difference between the ASCII code of any lower case letter and its corresponding upper case letter.

```
Char1 = 'b'
Char2 = 'B'
print('Letter\tASCII Value')
```

(Contd.)

```

print(Char1, '\t', ord(Char1))
print(Char2, '\t', ord(Char2))
print(' Difference between ASCII value of two Letters:')
print(ord(Char1), '-', ord(Char2), '=', end=' ')
print(ord(Char1) - ord(Char2))

```

Output

```

Letter ASCII Value
b      98
B      66
Difference between ASCII value of two Letters:
98 - 66 = 32

```

Explanation In the above program, the letter 'b' is stored in variable **Char1** and the letter 'B' is stored in variable **Char2**. The **ord()** function is used to find the ASCII value of the letters. Finally, the statement **ord(Char1) - ord(Char2)** is used to find the difference between the ASCII values of the two letters **Char1** and **Char2**.

SUMMARY

- ◆ Python breaks each statement into a sequence of lexical components called tokens.
- ◆ Literals are numbers, strings or characters that appear directly in a program.
- ◆ Python offers an inbuilt method called type to know the exact type of any value.
- ◆ Keywords are reserved words.
- ◆ Keywords cannot be used as identifiers or variables.
- ◆ An identifier is a name used to identify a variable, function, class or other objects.
- ◆ Everything in Python is an object.
- ◆ The **int** function converts a string or a number into a whole number or integer.
- ◆ The **float** function converts a string into a floating-point number.
- ◆ The Boolean data type is represented in Python as of type **bool**.
- ◆ **print()** function is used to display contents on the screen.
- ◆ **input()** function is used to accept input from the user.
- ◆ **format()** function can be used to return a formatted string.

KEY TERMS

- ⇒ **chr()**: Returns a character for a given ASCII value
- ⇒ **end()**: Used as argument with **print()** function
- ⇒ **format()**: Formats string and integer
- ⇒ **Identifier**: Name to identify a variable

- ⇒ **Inbuilt Math Functions:** `abs()`, `max()`, `round()`, `ceil()`, `log()`, `exp()`, `sqrt()`, `sin()`, `asin()`, `acos()`, `atan()`, `cos()`, `degrees()`, `radians()` and `floor()`.
- ⇒ **`input()`:** Used to accept data from the user
- ⇒ **`int()`:** Used to convert string or float into integer
- ⇒ **`ord()`:** Returns ASCII value of a character
- ⇒ **`print()`:** Prints contents on the screen
- ⇒ **`str()`:** Used to convert a number into string
- ⇒ **`type()`:** Used to know the exact type of any value
- ⇒ **Tokens:** Breaks each statement into a sequence of lexical components

REVIEW QUESTIONS

A. Multiple Choice Questions

- Which of the following is not a valid identifier?
 - `A_`
 - `_A`
 - `1a`
 - `_1`
- Which of the following is an invalid statement?
 - `w,X,Y,Z = 1,00,000,0000`
 - `WXYZ = 1,0,00,000`
 - `W X Y Z =10 10 11 10`
 - `W_X_Y = 1,100,1000`
- Which of the following is not a complex number?
 - `A = 1+2j`
 - `B = complex(1,2)`
 - `C = 2+2i`
 - None of the above
- What is the output of the following statement?


```
round(1.5)-round(-1.5)
```

 - 1
 - 2
 - 3
 - 4
- What is the output of the following statement?


```
print('{:,}'.format('100000'))
```

 - 1,00,000
 - 1,0,0,0,0,0
 - 10,00,00,0
 - Error
- Which type of error will occur on executing the following statement?


```
Name = MyName
```

 - Syntax Error
 - Name Error
 - Type Error
 - Value Error
- What is the output of following statement?


```
Sum = 10 + '10'
```

 - 1010
 - 20
 - TypeError
 - None of the above
- What will be printed if we write `print()` statement as


```
Print("Hello Python!")
```

 - Hello Python
 - Syntax Error
 - Name Error
 - Both a and b

9. Which of the following is a valid input() statement?
- `x = input(Enter number:)`
 - `X = Input(Enter number:)`
 - `X = input('Enter number:')`
 - `X = Input('Enter Number:')`
10. What will be the output of the following statement if the user has entered 20 as the value of x.
- ```
x = input('Enter Number:')
print(10+x)
```
- 1010
  - 20
  - 30
  - Error

## B. True or False

- Python breaks each statement into a sequence of lexical components known as tokens.
- Keywords are tokens of Python.
- Operators are not a part of tokens.
- Python keywords do not have fixed meaning.
- Keywords can be used as identifiers or variables.
- Strings are part of literals.
- An identifier is a name used to identify a variable, function etc.
- Python classifies different kinds of objects into types.
- The float function converts a string into a whole integer number.
- The str function is used to convert a number into a string.

## C. Exercise Questions

- Which of the following identifiers are valid?  
Name, Roll\_No , Sr.No, Roll-No, break, elif, DoB
- What will be the output of the following statements if all of them are executed in Python interactive mode?
  - `abs(-2)`
  - `min(102,220,130)`
  - `max(-1,-4,-10)`
  - `max('A','B','Z')`
  - `max('a','B','Z')`
  - `round(1.6)`
  - `math.ceil(1.2)`
  - `math.floor(1.8)`
  - `math.log(16,2)`
  - `math.exp(1)`
  - `math.`
  - `cos(math.pi)`
  - `math.cos(math.pi)`
- What will be the output of the following statements if all of them are executed in Python interactive mode?
  - `ord('a')`
  - `ord('F')`
  - `ord('f')`
  - `chr(97)`
  - `chr(100)`
- Identify the error in the following piece of code. Explain how you will fix it.
 

```
num1 = '10'
num2 = 20.65
sum = num1 + num2
```

```
print(sum)
```

5. State the output of following statements.

- |                                          |                                                |
|------------------------------------------|------------------------------------------------|
| a. <code>print(format(16,'x'))</code>    | b. <code>print(format(10,'x'))</code>          |
| c. <code>print(format(10+10,'x'))</code> | d. <code>print(format(10+ord('a'),'x'))</code> |
| e. <code>print(format(20,'o'))</code>    | f. <code>print(format(100,'b'))</code>         |
| g. <code>print(format(10,'b'))</code>    |                                                |

6. State the output of the following statements.

- |                                                |                                                 |
|------------------------------------------------|-------------------------------------------------|
| a. <code>print(format('Hello','&gt;2'))</code> | b. <code>print(format('Hello','&lt;2'))</code>  |
| c. <code>print(format('Hello','&gt;4'))</code> | d. <code>print(format('Hello','&gt;20'))</code> |

7. State the output of the following statements.

- |                                                     |                                                   |
|-----------------------------------------------------|---------------------------------------------------|
| a. <code>print(format(10,'&gt;20'))</code>          | b. <code>print(format('10','&lt;20'))</code>      |
| c. <code>print(format(10.76121421431,'.2f'))</code> | d. <code>print(format(10.76121421431,'f'))</code> |

8. Explain the use of end keyword with a suitable example.

9. Explain character set supported by Python in detail.

10. How are complex numbers displayed in interactive mode? Give an example.

11. State the output of the following code.

```
num1 = '10'
num2 = '20'
sum = num1 + num2
print(sum)
```

## PROGRAMMING ASSIGNMENTS

- Write a program to print 'F' to 'A' in five different lines.
- Write a program to read and store the name of three different cities in three different variables and print all the contents of variables on the console.
- Write a program to prompt the user to enter and display their personal details, such as name, address and mobile number.
- By making use of five different print statements, write a program to print 'A' to 'F' in one single line.
- Write a program to read an integer as string. Convert the string into integer and display the type of value before and after converting to int.
- Write a program initialize the string "hello world" to a variable Str1 and convert the string into upper case.
- Translate the following algorithm into Python code.  
Step 1: Initialize variable named Pounds with value 10.  
Step 2: Multiply Pounds by 0.45 and assign it to a variable Kilogram.  
Step 3: Display the value of variable Pounds and Variable.
- Write a program to read the radius of a circle and print the area of the circle.



# UNIT 2

---

- Operators and Expressions
- Decision Statements
- Loop Control Statements

# Operators and Expressions

## 3

### CHAPTER OUTLINE

|                                                                   |                                                                          |
|-------------------------------------------------------------------|--------------------------------------------------------------------------|
| 3.1 Introduction                                                  | 3.6 Translating Mathematical Formulae into Equivalent Python Expressions |
| 3.2 Operators and Expressions                                     | 3.7 Bitwise Operator                                                     |
| 3.3 Arithmetic Operators                                          | 3.8 The Compound Assignment Operator                                     |
| 3.4 Operator Precedence and Associativity                         |                                                                          |
| 3.5 Changing Precedence and Associativity of Arithmetic Operators |                                                                          |

### LEARNING OUTCOMES

---

*After completing this chapter, students will be able to:*

- Perform simple arithmetic operations
  - Explain the difference between division and floor division operators
  - Use unary, binary and bitwise operators, and perform multiplication and division operations using bitwise left and right shift operators
  - Evaluate numeric expressions and translate mathematical formulae into expressions
  - Recognise the importance of associativity and operator precedence in programming languages
- 

### 3.1 INTRODUCTION

An operator indicates an operation to be performed on data to yield a result. In our day to day life, we use various kinds of operators to perform diverse data operations. Python supports different

operators which can be used to link variables and constants. These include arithmetic operators, Boolean operators, bitwise operators, relational operators and simple assignment and compound assignment operators.

Table 3.1 lists basic operators in Python with their symbolic representation

**Table 3.1** Types of operators

| Type of Operator                                    | Symbolic Representation |
|-----------------------------------------------------|-------------------------|
| Arithmetic Operators                                | +, -, /, //, *, %, %%   |
| Boolean Operators                                   | and, or, not            |
| Relational Operators                                | >, <, <=, >=, !=        |
| Bitwise Operators                                   | &,  , ^, >>, <<, ~      |
| Simple Assignment and Compound Assignment Operators | =, +=, *=, /=, %=, **=  |

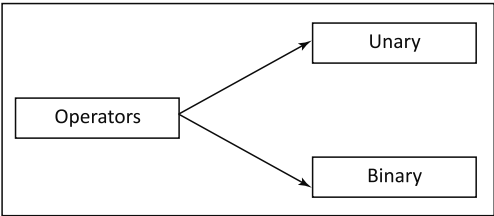
3.2 OPERATORS AND EXPRESSIONS

Most statements contain expressions. An expression in Python is a block of code that produces a result or value upon evaluation. A simple example of an expression is `6 + 3`. An expression can be broken down into operators and operands. Operators are symbols which help the user or command computer to perform mathematical or logical operations. In the expression `6 + 3`, the `+` acts as the operator. An operator requires data to operate and this data is called **operand**. In this example, `6` and `3` are the operands.

The following sections describe the various kinds of operators and their usage. The expressions given in the examples are executed in Python interactive mode.

3.3 ARITHMETIC OPERATORS

There are two types of arithmetic operators in Python, viz. binary and unary (as shown in Fig. 3.1).



**Figure 3.1** Types of arithmetic operators

3.3.1 Unary Operators

Unary arithmetic operators perform mathematical operations on one operand only. The `+` and `-` are two unary operators. The unary operator minus (`-`) produces the negation of its numeric



operand. The unary operator plus (+) returns the numeric operand without change. Table 3.2 gives the details of unary operators.

**Table 3.2** Unary operators

| Unary Operator | Example                                   | Description                                                                           |
|----------------|-------------------------------------------|---------------------------------------------------------------------------------------|
| +              | +X<br>(+X returns the same value, i.e. X) | Returns the same value that is provided as input                                      |
| -              | -X<br>(-x returns the negation of x)      | Negates the original value so that the positive value becomes negative and vice versa |

### Examples of Unary Operators

```
>>> x=-5 #Negates the value of X
>>> x
-5
>>> x+=6 #Returns the numeric operand, i.e. 6, without any change
>>> x
6
```

### Some More Complex Examples of Unary Operators

```
>>> +-5
-5
```

In the above expression **+-5**, the first '+' operator represents the unary plus operation and the second '-' operator represents the unary minus operation. The expression **+-5** is equivalent to **+(-(5))**, which is equal to -5.

```
>>> 1--3 #Equivalent to 1-(-3)
4
>>> 2---3 #Equivalent to 2-(-(-3))
-1
>>> 3+--2 #Equivalent to 3+(-(-2))
5
```

## 3.3.2 Binary Operators

Binary operators are operators which require two operands. They are written in **infix** form, i.e. the operator is written in between two operands.

### The Addition (+) Operator

The '+' operator in Python can be used with binary and unary form. If the addition operator is applied in between two operands, it returns the result as the arithmetic sum of the operands. Some examples of addition operators executed in Python interactive mode are given as follows:

**Example**

```
>>> 4+7 #Addition
11
```

```
>>>5+5 Addition
10
```

Table 3.3. explains the syntax and semantics of the addition operator in Python, using its three numeric types, viz. **int**, **float** and **complex**.

**Table 3.3** Addition operator

| <i>Syntax</i>               | <i>Example</i>      |
|-----------------------------|---------------------|
| (int, int)-> int            | 2+4 returns 6       |
| (float, float)->float       | 1.0+4.0 returns 5.0 |
| (int, float)->float         | 1+2.0 returns 3.0   |
| (float, int)->float         | 2.0+1 returns 3.0   |
| (complex, complex)->complex | 3j+2j returns 5j    |

**The Subtraction (-) Operator**

The '-' operator in Python can be used with binary and unary form. If the subtraction operator is applied in between two operands, the result is returned as the arithmetic difference of the operands. Some examples of subtraction operators executed in Python interactive mode are given as follows:

**Example**

```
>>> 7-4 #Subtraction
3
```

```
>>>5-2 #Subtraction
3
```

Table 3.4 explains the syntax and semantics of the subtraction operator in Python, using its three numeric types, viz. **int**, **float** and **complex**.

**Table 3.4** Subtraction operator

| <i>Syntax</i>               | <i>Example</i>      |
|-----------------------------|---------------------|
| (int, int)-> int            | 4-2 returns 2       |
| (float, float)->float       | 3.5-1.5 returns 2.0 |
| (int, float)->float         | 4-1.5 returns 2.5   |
| (float, int)->float         | 4.0-2 returns 2.0   |
| (complex, complex)->complex | 3j-2j returns 1j    |

**PROGRAM 3.1** | Read the cost and selling price of an object and write a program to find the profit earned by a seller (in rupees). The selling price is greater than the cost price.

```
SP=eval(input('Enter the Selling Price of an Object:'))
CP=eval(input('Enter the Cost Price of an Object:'))
print('-----')
print(' Selling Price = ',SP)
print(' Cost Price = ',CP)
print('-----')
Profit=SP - CP #Formula to Calculate Profit
print(' Profit Earned by Selling = ',Profit)
```

### Output

```
Enter the Selling Price of an Object: 45
Enter the Cost Price of an Object: 20
```

```

Selling Price = 45
Cost Price = 20
```

```

Profit Earned by Selling = 25
```

**Explanation** At the start of the program, the selling price and cost price of the object is read using eval. The statement, **Profit = SP - CP** is executed to calculate the profit earned by the seller.

### The Multiplication (\*) Operator

The '\*' operator in Python can be used only with binary form. If the multiplication operator is applied in between two operands, it returns the result as the arithmetic product of the operands. Some examples of multiplication operators executed in Python interactive mode are given as follows:

### Example

```
>>> 7*4 #Multiplication
28
>>>5*2 #Multiplication
10
```

Table 3.5 explains the syntax and semantics of the multiplication operator in Python, using its three numeric types, viz. **int**, **float** and **complex**.

**Table 3.5** Multiplication operator

| <i>Syntax</i>               | <i>Example</i>      |
|-----------------------------|---------------------|
| (int, int)-> int            | 4*2 returns 8       |
| (float, float)->float       | 1.5*3.0 returns 4.5 |
| (int, float)->float         | 2*1.5 returns 3.0   |
| (float, int)->float         | 1.5* 5 returns 7.5  |
| (complex, complex)->complex | 2j*2j returns -4+0j |

**PROGRAM 3.2** | Write a program to calculate the square and cube of a number using \* operator.

```

num=eval(input('Enter the number:')) # Read Number
print('Number = ',num)
Square=num* num #Calculate Square
Cube = num * num * num #Calculate Cube
print('Square of a Number = ',num,' is ',Square)
print('Cube of a Number = ',num,' is ',Cube)

```

### Output

```

Enter the number: 5
Number = 5
Square of a Number = 5 is 25
Cube of a Number = 5 is 125

```

## The Division (/) Operator

The '/' operator in Python can be used only with binary form. If the division operator is applied in between two operands, it returns the result as the arithmetic quotient of the operands. Some examples of division operators executed in Python interactive mode are given as follows:

### Example

```

>>> 4/2 #Division
2.0

>>> 10/3
3.3333333333333335 #Division

```

Table 3.6 explains the syntax and semantics of the division operator in Python, using its three numeric types, viz. **int**, **float** and **complex**.

**Table 3.6** Division (/) operator

| Syntax                      | Example             |
|-----------------------------|---------------------|
| (int, int)-> float          | 25/5 returns 5.0    |
| (float, float)->float       | 0.6/2.0 returns 0.3 |
| (int, float)->float         | 4/0.2 returns 20.0  |
| (float, int)->float         | 1.5/2 returns 0.75  |
| (complex, complex)->complex | 6j/2j returns 3+0j  |



**Note:** When the division (/) operator is applied on two **int** operands, Python returns a **float** result.

**PROGRAM 3.3** | Write a program to calculate simple interest (SI). Read the principle, rate of interest and number of years from the user.

```
P=eval(input('Enter principle Amount in Rs = ')) #Read P
ROI=eval(input('Enter Rate of Interest = ')) #Read ROI
years=eval(input('Enter the Number of years ='))#Read years
print(' Principle = ',P)
print(' Rate of Interest = ',ROI)
print(' Number of Years = ',years)
SI = P*ROI*Years/100 #Calculate SI
print('Simple Interest = ',SI)
```

### Output

```
Enter Principle Amount in Rs = 1000
Enter Rate of Interest = 8.5
Enter the Number of Years = 3
Principle = 1000
Rate of Interest = 8.5
Number of Years = 3
Simple Interest = 255.0
```

**PROGRAM 3.4** | Write a program to read a temperature in Celsius from the user and convert it into Fahrenheit.

```
Celsius =eval(input('Enter Degree is Celsius:'))#Read Celsius from User
print('Celsius = ', Celsius) #Print Celsius
Fahrenheit = (9 / 5) * Celsius + 32 # Convert Celsius to Fahrenheit
print(' Fahrenheit = ', Fahrenheit) # Print Fahrenheit
```

(Contd.)

**Output**

```
Enter Degree is Celsius: 23
Celsius = 23
Fahrenheit = 73.4
```



**Note:** Formula to convert Celsius into Fahrenheit is:  

$$\text{Fahrenheit} = (9/5) * \text{Celsius} + 32$$

**The Floor Division (//) Operator**

The `'//'` operator in Python can be used only with binary form. If the floor division operator is applied in between two operands, it returns the result as the arithmetic quotient of the operands. Some examples of floor division operators executed in Python interactive mode are given as follows:

**Example**

```
>>> 4//2 # Floor Division
2
>>> 10//3
3 #Floor Division
```

Table 3.7 explains the syntax and semantics of the floor division operator in Python, using its numeric types, viz. **int** and **float**.

**Table 3.7** Floor division (`//`) operator

| Syntax                | Example               |
|-----------------------|-----------------------|
| (int, int)-> int      | 25//5 returns 5       |
| (float, float)->float | 10.5//5.0 returns 2.0 |
| (int, float)->float   | 11//2.5 returns 4.0   |
| (float, int)->float   | 4.0//3 returns 1.0    |



**Note:** a. From the above example, it is clear that when the floor division (`//`) operator is applied on two **int** operands, Python returns an **int** result.  
 b. In the second example `10.5//5.0`, the result returned is `2.0`. However, if `10.5//5.0` returns `2.1`, it means the floor division operator has been applied on two float operands. Hence, it returns the result in float but ignores the decimal number after the decimal point.

**The Modulo (%) Operator**

When the second number divides the first number, the modulo operator returns the remainder. The `%` modulo operator is also known as the remainder operator. If the remainder of `x` divided by `y` is zero then we can say that `x` is divisible by `y` or `x` is a multiple of `y`.

Consider the following example.

$$\begin{array}{rcl}
 & & 3 \leftarrow \text{Quotient} \\
 \text{Divisor} \longrightarrow 4 & \left| & 14 \leftarrow \text{Dividend} \\
 & & - 12 \\
 & & \hline
 & & 2 \leftarrow \text{Remainder}
 \end{array}$$

In the above example,  $14 \% 4$  returns 3 as the remainder. Thus, the left-side operand, i.e. 14 is the dividend and the right-side operand, i.e. 4 is the divisor. Some more examples of the modulo operator executed in Python interactive mode are given below.

### Example

```
>>> 10 % 4 # 10 is divided by 4 returns remainder as 2
2
>>> 13%5
3
```

Table 3.8 explains the syntax and semantics of the modulo (%) operator in Python, using its numeric types, viz. **int** and **float**.

**Table 3.8** Modulo (%) operator

| Syntax                | Example                |
|-----------------------|------------------------|
| (int, int)-> int      | 25%4 returns 1         |
| (float, float)->float | 2.5 % 1.2 returns 0.10 |
| (int, float)->float   | 13%2.0 returns 1.0     |
| (float, int)->float   | 1.5 % 2 returns 1.5    |



**Note:** Mathematically,  $X \% Y$  is equivalent to  $X - Y * (x // Y)$

**Example:**  $14 \% 5$  returns 4  
 Therefore,  
 $14 \% 5 = 14 - 5 * (14 // 5)$   
 $= 14 - 5 * (2)$   
 $= 14 - 10$   
 $= 4$

**Use of % Modulo Operator in Programming** The modulo operator, i.e. the remainder operator is very useful in programming. It is used to check if a number is even or odd, i.e. if  $\text{number} \% 2$  returns zero then it is an even number and if  $\text{number} \% 2 == 1$  then it is an odd number.

**PROGRAM 3.5** | Write a program to read the weight of an object in grams and display its weight in kilograms and grams, respectively.

**Example**

**Input:** Enter the weight of the object in grams: 2,500

**Output:** Weight of the object (kilograms and grams): 2 kg and 500 g

**Note:** 1 kilogram = 1,000 grams

```
W1 = eval(input('Enter the Weight of Object in grams:')) #Input Weight
print(' Weight of Object = ',W1,' grams') # Print Weight
W2 = W1 // 1000 #Calculate No of kg
W3 = W1 % 1000 #Calculate No of g
print(' Weight of Object = ',W2,' kg and ',W3,' g')
```

**Output**

```
Enter the Weight of Object in g : 1250
Weight of Object = 1250 g
Weight of Object = 1 kg and 250 g
```

**PROGRAM 3.6** | Write a program to reverse a four-digit number using % and // operators.

```
Num=eval(input('Enter four-digit number: '))
print('Entered number is:',num)
r1=num%10
q1=num//10
r2=q1%10
q2=q1//10
r3=q2%10
q3=q2//10
r4=q3%10
print('Reverse of ',num,'is:',r1,r2,r3,r4)
```

**Output**

```
Enter four-digit number: 8763
Entered number is: 8763
Reverse of 8763 is: 3 6 7 8
```

**Explanation** In the above program, initially the number is read from the user. For instance, the number read through the user is **8763**. To reverse the contents of the number, initially the operation  $(8763 \% 10)$  gives a remainder 3. To display the second digit 6, the number has to be divided by 10. Hence,  $(8763 // 10)$  gives **876**. After obtaining the quotient as **876**, the modulus operation  $(876 \% 10)$  is performed again to obtain the digit 6. This process is continued three times to obtain the reverse of the four-digit number entered by the user.



### The Exponent \*\* Operator

The '\*\*' exponent operator is used to calculate the power or exponent of a number. To compute  $x^y$  ( $x$  raised to  $y$ ), the expression is written as  $x**y$ . The exponent operator is also called **power operator**.

#### Example

```
>>> 4**2 #Calculate Square of a Number 4
16
>>> 2**3 #Calculate Cube of a Number 2
8
```

Table 3.9 explains the syntax and semantics of the exponent (\*\*) operator in Python, using its numeric types, viz. **int** and **float**.

**Table 3.9** Exponent(\*\*) operator

| Syntax                | Example              |
|-----------------------|----------------------|
| (int, int)-> int      | 2**4 returns 16      |
| (float, float)->float | 2.0**3.0 returns 8.0 |
| (int, float)->float   | 5**2.0 returns 25.0  |
| (float, int)->float   | 4.0 **3 returns 64.0 |

**PROGRAM 3.7** Write a program to calculate the distance between two points. The formula for computing distance is

$$\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

We can use  $Z^{0.5}$  to compute the square root of the expression  $\sqrt{Z}$ . The program below prompts the user to read the coordinates of the two points and compute the distance between them.

```
print('Point1')
X1 = eval(input('Enter X1 coordinate:')) #Read X1
Y1 = eval(input('Enter Y1 coordinate:')) #Read Y1
print('point2')
X2 = eval(input('Enter X2 coordinate: ')) #Read X2
Y2 = eval(input('Enter Y2 coordinate: ')) #Read Y2
L1=(X2-X1)**2 + (Y2-Y1)**2 #Computer inner expression
Distance = L1**0.5 #Compute Square root.
print('Distance between two point is as follows')
print('(X1,Y1)',(X2,Y2),'= ', Distance)
```

#### Output

```
Point1
Enter X1 Coordinate :4
Enter Y1 Coordinate :6
```

(Contd.)

```

point2
Enter X2 Coordinate: 8
Enter Y2 Coordinate: 10
Distance between the two points is as follows
(4 6) (8 10) = 5.656854249492381

```

**PROGRAM 3.8** | Write a program to display the following table.

| X  | Y | X**Y   |
|----|---|--------|
| 10 | 2 | 100    |
| 10 | 3 | 1000   |
| 10 | 4 | 10000  |
| 10 | 5 | 100000 |

```

print('X \t Y \t X**Y')
print('10 \t 2 \t ',10**2)
print('10 \t 3 \t ',10**3)
print('10 \t 4 \t ',10**4)
print('10 \t 5 \t ',10**5)

```

#### Output

| X  | Y | X**Y   |
|----|---|--------|
| 10 | 2 | 100    |
| 10 | 3 | 1000   |
| 10 | 4 | 10000  |
| 10 | 5 | 100000 |

## 3.4 OPERATOR PRECEDENCE AND ASSOCIATIVITY

Operator precedence determines the order in which the Python interpreter evaluates the operators in an expression.


Consider the expression  $4+5*3$ .

Now, you may ask so how does Python know which operation to perform first? In the above example  $4+5*3$ , it is important to know whether  $4+5*3$  evaluates to 19 (where the multiplication is done first) or 27 (where the addition is done first).

The default order of precedence determines that multiplication is computed first so the result is 19. As an expression may contain a lot of operators, operations on the operands are carried out according to the priority, also called the precedence of the operator. The operator having higher priority is evaluated first.

Table 3.10 gives the list of operator precedence in the descending order. The operators on the top rows have higher precedence and the operators on the bottom rows have lower precedence. If a row contains multiple operators, it means all the operators are of equal priority or precedence.

**Table 3.10** Operator precedence

| Precedence                                                                        | Operator                     | Name                                                      |
|-----------------------------------------------------------------------------------|------------------------------|-----------------------------------------------------------|
|  | **                           | Exponential                                               |
|                                                                                   | +, -, ~                      | Plus, Minus, Bitwise not                                  |
|                                                                                   | *, /, //, %                  | Multiplication, division, integer division, and remainder |
|                                                                                   | +, -                         | Binary Addition, Subtraction                              |
|                                                                                   | <<, >>                       | Left and Right Shift                                      |
|                                                                                   | &                            | Bitwise AND                                               |
|                                                                                   | ^                            | Bitwise XOR                                               |
|                                                                                   |                              | Bitwise OR                                                |
|                                                                                   | <, <=, >, >=                 | Comparison                                                |
|                                                                                   | ==, !=                       | Equality                                                  |
|                                                                                   | =%, /=, //=, -=, +=, *=, **= | Assignment Operators                                      |
|                                                                                   | is, is not                   | Identity Operators                                        |
|                                                                                   | in, not in                   | Membership Operator                                       |
|                                                                                   | Not                          | Boolean Not                                               |
|                                                                                   | And                          | Boolean and                                               |
|                                                                                   | Or                           | Boolean or                                                |

### 3.4.1 Example of Operator Precedence

Consider arithmetic operators `*`, `/`, `//` and `%`, which have higher precedence as compared to operators `+` and `-`.

#### Example

$$4+5*3-10$$

As compared to `+` and `*` operators, the `*` operator has higher priority. Hence, the multiplication operation is performed first. Therefore, above expression becomes,

$$4+15-10$$

Now in above expression, `+` and `-` have the same priority. In such a situation, the leftmost operation is evaluated first. Hence, the above expression becomes

$$19 - 10$$

Consequently, subtraction is performed last and the final answer of the expression will be 9.

### 3.4.2 Associativity

When an expression contains operators with equal precedence then the associativity property decides which operation is to be performed first. Associativity implies the direction of execution and is of two types, viz. left to right and right to left.

- (i) **Left to Right:** In this type of expression, the evaluation is executed from the left to right.

$$4 + 6 - 3 + 2$$

In the above example, all operators have the same precedence. Therefore, associativity rule is followed (i.e. the direction of execution is from the left to right).

The evaluation of the expression  $4+6-3+2$  is equivalent to

$$\begin{aligned} &= ((4+6)-3)+2 \\ &= ((10)-3)+2 \\ &= (7)+2 \\ &= 9 \end{aligned}$$

- (ii) **Right to Left:** In this type of expression, the evaluation is executed from the right to left.

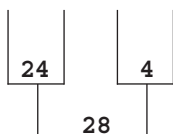
$$X = Y = Z = \text{Value}$$

In the above example, assignment operators are used. The value of  $Z$  is assigned to  $Y$  and then to  $X$ . Thus, the evaluation starts from the right.

### Example of Associativity

- (i) When operators of the same priority are found in an expression, precedence is given to the leftmost operator.

$$Z = 4 * 6 + 8 // 2$$



In the above expression  $*$  is evaluated first, even though  $*$  and  $//$  have the same priorities. The operator  $*$  occurs before  $//$  and hence the evaluation starts from the left. Therefore, the final answer for the above expression is 28.

The examples so far illustrated how Python uses associativity rules for evaluating expressions. Table 3.11 shows the precedence and associativity for arithmetic operators.

**Table 3.11** Associativity table for arithmetic operators

| Precedence | Operators | Associativity          |
|------------|-----------|------------------------|
| Highest    | ()        | Innermost to Outermost |
|            | **        | Highest                |
|            | *,//,/,%  | Left to Right          |
| Lowest     | + -       | Left to Right          |

## 3.5 CHANGING PRECEDENCE AND ASSOCIATIVITY OF ARITHMETIC OPERATORS

One can change the precedence and associativity of arithmetic operators by using  $()$ , i.e. the parentheses operator. The  $()$  operator has the highest precedence among all other arithmetic

operators. It can be used to force an expression to evaluate in any order. Parentheses operator () also makes an expression more readable.

Some examples of parentheses operator executed in Python interactive mode are given as follows:

### Example

```
>>> z=(5+6)*10
>>> z
110
```

### Explanation

In the above example, z is initialized with one expression (5+6)\*10. The sub expression (5 + 6) is evaluated first, followed by the multiplication operation.

### Some More Complex Examples

```
>>> A= 100 / (2*5)
>>> A
10.0

>>> B= 4 + (5 * (4/2) + (4 + 3))
>>> B
21.0
```

**PROGRAM 3.9** | Write a program to find the area and perimeter of a rectangle using (), i.e. the parenthesis operator.

```
Length = eval(input('Enter the Length of Rectangle:'))
Breadth = eval(input('Enter the Breadth of Rectangle:'))
print('- - - - -')
print(' Length = ',Length)
print(' Breadth = ',Breadth)
print('- - - - -')
print(' Area = ', Length * Breadth)
print(' Perimeter = ',2 * (Length + Breadth))
```

### Output

```
Enter the Length of Rectangle: 10
Enter the Breadth of Rectangle: 20
- - - - -
Length = 10
Breadth = 20
- - - - -
Area = 200
Perimeter = 60
```

**Explanation** In the above program, the values of variables length and breadth of the rectangle are initially read from the user. Then using the multiplication \* operator, the area of the rectangle is computed. Finally, in order to compute the perimeter, the addition of length and breadth is performed and the result is multiplied by 2.



**Note:** Area of Rectangle = Length \* Breadth  
Perimeter of Rectangle = 2 \* (Length + Breadth)

### 3.6 TRANSLATING MATHEMATICAL FORMULAE INTO EQUIVALENT PYTHON EXPRESSIONS

Consider the following quadratic equation written in normal arithmetic manner.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The steps required to convert this quadratic equation into its equivalent Python expression are given as follows:

- ⊙ **STEP 1:** The numerator and denominator are computed first to find the roots of the quadratic equation. Division between the numerator and denominator is performed as the last step. Hence, we can write the above expression as:

$$\text{Numerator/Denominator}$$

- ⊙ **STEP 2:** The denominator is just 2a, so we can rewrite the formula as:

$$\text{Numerator}/((2 * a))$$

- ⊙ **STEP 3:** Now we can split the numerator into two parts, i.e. left and right as follows:

$$(\text{Left} + \text{Right}) / ((2 * a))$$

- ⊙ **STEP 4:** Substitute -b for left. There is no need to put parenthesis for -b because unary operator has higher precedence than binary addition. Hence, the above equation becomes:

$$(-b + \text{Right}) / ((2 * a))$$

- ⊙ **STEP 5:** The right contains the expression inside the square root. Therefore, the above equation can be rewritten as:

$$(-b + \text{sqrt}(\text{expression})) / ((2 * a))$$

- ⊙ **STEP 6:** But the expression inside the square root contains two parts left and right. Hence, the above equation is further rewritten as

$$(-b + \text{sqrt}(\text{left} - \text{right})) / ((2 * a))$$

- ⊙ **STEP 7:** Now the left part contains the expression b\*\*2 and the right part contains the expression 4\*a\*c. There is no need to put parenthesis for b\*\*2 because the exponent operator has

higher precedence than the \* operator since the expression  $4*a*c$  is present on the right side. The above equation can be rewritten as

$$(-b + \sqrt{b^2 - 4*a*c}) / ((2*a))$$

Thus, we have converted the mathematical expression into a Python expression. While converting an equation into a Python expression, one needs to only remember the rules of operator precedence and associativity.

**PROGRAM 3.10** | Write the following numeric expression in Python and evaluate it.

$$\frac{2+8P}{2} - \frac{(P-Q)(P+Q)}{2} + 4 * \frac{(P+Q)}{2}$$

Consider the value of variables P and Q as 4 and 2, respectively.

```
P = 4
Q = 2
Z = (2 + 8 * P) / 2 - ((P-Q)*(P+Q))/2 + 4 * ((P+Q)/2)
print('(2 + 8 * P) / 2 - ((P-Q)*(P+Q))/2 + 4 * ((P+Q)/2)')
print(' Where P = ',P,' and Q = ',Q)
print(' Answer of above expression = ',Z)
```

#### Output

```
(2 + 8 * P) / 2 - ((P-Q)*(P+Q))/2 + 4 * ((P+Q)/2)
Where P = 4 and Q = 2
Answer of above expression = 23.0
```

**Explanation** In the above program, initially the equation

$$\frac{2+8P}{2} - \frac{(P-Q)(P+Q)}{2} + 4 * \frac{(P+Q)}{2}$$

is translated into a Python expression as

$$(2 + 8 * P) / 2 - ((P - Q) * (P + Q)) / 2 + 4 * ((P + Q) / 2).$$

Once the expression is converted into a Python expression, the values of P and Q are substituted by the Python interpreter and finally the expression is evaluated considering Python precedence and associativity rules.

## 3.7 BITWISE OPERATOR

Python has six bitwise operators for bitwise manipulation. The bitwise operator permits a programmer to access and manipulate individual bits within a piece of data. Table 3.12. shows various bitwise operators supported by Python.

**Table 3.12** Bitwise operators

| <i>Operator</i> | <i>Meaning</i> |
|-----------------|----------------|
| <b>&amp;</b>    | Bitwise AND    |
| <b> </b>        | Bitwise OR     |
| <b>^</b>        | Bitwise XOR    |
| <b>&gt;&gt;</b> | Right Shift    |
| <b>&lt;&lt;</b> | Left Shift     |
| <b>~</b>        | Bitwise NOT    |

### 3.7.1 The Bitwise AND (&) Operator

This operator performs **AND** operation on input bits of numbers. The Bitwise **AND** operator is represented as '&'. The '&' operator operates on two operands bit-by-bit. Table 3.13. explains the AND operator.

**Table 3.13** AND operator

| <i>Input</i> |   | <i>Output</i> |
|--------------|---|---------------|
| X            | Y | X & Y         |
| 0            | 0 | 0             |
| 0            | 1 | 0             |
| 1            | 0 | 0             |
| 1            | 1 | 1             |

We can conclude from this table that the output is obtained by multiplying the input bits.

#### Example of AND Operator

```
>>> 1 & 3
1 # The bitwise & operator on 1 and 3 returns 1
>>> 5 & 4
4 # The bitwise & operator on 5 and 4 returns 4
```

Working of the bitwise operator is given as follows:

|                                                           |   |   |   |   |         |
|-----------------------------------------------------------|---|---|---|---|---------|
| 1 and 3 are converted into their equivalent binary format |   |   |   |   |         |
|                                                           | 0 | 0 | 0 | 1 | (one)   |
| <b>&amp;</b>                                              |   |   |   |   |         |
|                                                           | 0 | 0 | 1 | 1 | (Three) |
| -----                                                     |   |   |   |   |         |
| Bitwise operation (0 & 0) (0 & 0) (0 & 1) (1 & 1)         |   |   |   |   |         |
| -----                                                     |   |   |   |   |         |
| <b>Result</b>                                             | 0 | 0 | 0 | 1 | (One)   |
| Decimal equivalent of (0 0 0 1) = 1                       |   |   |   |   |         |
| Therefore, 1 & 3 = 1                                      |   |   |   |   |         |



**PROGRAM 3.11**

Write a program to read two numbers from the user. Display the result using bitwise & operator on the numbers.

```
num1 = int(input('Enter First Number: '))
num2 = int(input('Enter Second Number: '))
print(num1, ' & ', num2, ' = ', num1 & num2)
```

**Output**

```
#Test Case 1
Enter First Number: 1
Enter Second Number: 3
 1 & 3 = 1

#Test Case 2
Enter First Number: 5
Enter Second Number: 6
 5 & 6 = 4
```

**3.7.2 The Bitwise OR (|) Operator**

This operator performs bitwise OR operation on the numbers. The bitwise OR operator is represented as '|'. It also operates on two operands and the two operands are compared bit-by-bit. Table 3.14 explains the '|' (OR) operator.

**Table 3.14** Bitwise OR operator

| Input |   | Output |
|-------|---|--------|
| X     | Y | X   Y  |
| 0     | 0 | 0      |
| 0     | 1 | 1      |
| 1     | 0 | 1      |
| 1     | 1 | 1      |

We can conclude from this table that the output is obtained by adding the input bits.

**Examples of Bitwise '|' (OR) Operator**

```
>>> 3 | 5
7 # The bitwise | operator on 3 and 5 returns 7
```

```
>>> 1 | 5
5 # The bitwise | operator on 1 and 5 returns 5
```

Working of the bitwise OR ('|') operator is given as follows:

```

Working of expression 3 | 5 is as below.
Initially 3 and 5 are converted into their equivalent binary format
 0 0 1 1 (Three)
 |
 0 1 0 1 (Five)

Bitwise operation (0 | 0) (0 | 1) (1 | 0) (1 | 1)

Result 0 1 1 1 (seven)
Decimal Equivalent of (0 1 1 1) = 7
Therefore 3 | 5 = 7

```

**PROGRAM 3.12** | Write a program to read two numbers from the user. Display the result using bitwise | operator on the numbers.

```

num1 = int(input('Enter First Number: '))
num2 = int(input('Enter Second Number: '))
print(num1, ' | ', num2, ' = ', num1 | num2)

```

### Output

```

#Test Case 1
Enter First Number: 3
Enter Second Number: 5
3 | 5 = 7

#Test Case 2
Enter First Number: 6
Enter Second Number: 1
6 | 1 = 7

```

## 3.7.3 The Bitwise XOR (^) Operator

This operator performs bitwise exclusive or XOR operation on the numbers. It is represented as '^'. The '^' operator also operates on two operands and these two operands are compared bit-by-bit. Table 3.15. explains the '^' (XOR) operator.

**Table 3.15** The Table for Bitwise XOR Operator

| Input |   | Output       |
|-------|---|--------------|
| X     | Y | $X \wedge Y$ |
| 0     | 0 | 0            |
| 0     | 1 | 1            |
| 1     | 0 | 1            |
| 1     | 1 | 0            |

We can conclude from this table that the output is logic one when one of the input bits is logic one.

### Examples of Bitwise XOR (^) Operator

```
>>> 3 ^ 5
6 # The bitwise ^ operator on 3 and 5 returns 6
>>> 1 ^ 5
4 # The bitwise ^ operator on 1 and 5 returns 4
```

Working of the bitwise XOR (^) operator is given as follows:

Working of expression  $3 \wedge 5$  is as below.

Initially 3 and 5 are converted into their equivalent binary format

|                                     |                |                |                |                |         |
|-------------------------------------|----------------|----------------|----------------|----------------|---------|
|                                     | 0              | 0              | 1              | 1              | (Three) |
| $\wedge$                            |                |                |                |                |         |
|                                     | 0              | 1              | 0              | 1              | (Five)  |
| -----                               |                |                |                |                |         |
| <b>Bitwise operation</b>            | $(0 \wedge 0)$ | $(0 \wedge 1)$ | $(1 \wedge 0)$ | $(1 \wedge 1)$ |         |
| -----                               |                |                |                |                |         |
| <b>Result</b>                       | 0              | 1              | 1              | 0              | (Six)   |
| Decimal Equivalent of (0 1 1 0) = 6 |                |                |                |                |         |
| Therefore $3 \wedge 5 = 6$          |                |                |                |                |         |

**PROGRAM 3.13** Write a program to read two numbers from the user. Operate bitwise ^ operator on them and display the result.

```
num1 = int(input('Enter First Number: '))
num2 = int(input('Enter Second Number: '))
print(num1, ' ^ ', num2, ' = ', num1 ^ num2)
```

### Output

#### #Test Case 1

```
Enter First Number: 3
```

(Contd.)

```

Enter Second Number: 5
3 ^ 5 = 6
#Test Case 2
Enter First Number: 1
Enter Second Number: 2
1 ^ 2 = 3

```

### 3.7.4 The Right Shift (>>) Operator

The right shift operator is represented as >>. It also needs two operands. It is used to shift bits to the right by  $n$  position. Working of the right shift operator (>>) is explained as follows:

#### Example

```

>>>4 >> 2 # The input data 4 is to be shifted by 2 bits towards the right side
1
>>>8>>2
2

```

#### Explanation

Consider the expression  $4 \gg 2$ .

Initially, the number 4 is converted into its corresponding binary format, i.e. 0 1 0 0

0 0 0 0 0 0 1 0 0      Binary 4 ←

8 7 6 5 4 3 2 1 0      Bit Index ←

The input data 4 is to be shifted by 2 bits towards the right side.

The answer in binary bits would be

0 0 0 0 0 0 0 0 1      Binary 1 ←

8 7 6 5 4 3 2 1 0      Bit Index ←



**Note:** Shifting the input number by  $N$  bits towards the right means the number is divided by  $2^S$ . In short, it means  $Y = N/2^S$ .

Where,

$N$  = The Number

$S$  = The Number of Bit Positions to Shift

Consider the above example  $4 \gg 2$ . Let us solve this using the above formula, i.e.  $y = N / 2^S$

$$= 4 / 2^2$$

$$= 4 / 4$$

$$= 1$$

Therefore,  $4 \gg 2$  returns 1 in Python interactive mode.

**PROGRAM 3.14** | Write a program to shift input data by 2 bits towards the right.

```
N = int(input('Enter Number: '))
S = int(input('Enter Number of Bits to be shift Right: '))
print(N,' >> ',S,' = ', N >> S)
```

### Output

```
Enter Number: 8
Enter Number of Bits to be shift Right: 2
8 >> 2 = 2
```

## 3.7.5 The Left Shift (<<) Operator

The left shift operator is represented as <<. It also needs two operands. It is used to shift bits to the left by N position. The working of the left shift operator is given as follows:

### Example

```
>>> 4 << 2 # The input data 4 is to be shifted by 2 bits towards the left side
16
```

```
>>> 8 << 2 # The input data 8 is to be shifted by 2 bits towards the left side
32
```

### Explanation

Consider the expression  $4 \ll 2$ .

Initially, the number 4 is converted into its corresponding binary format, i.e. 0 1 0 0

0 0 0 0 0 0 1 0 0      Binary 4 ←

8 7 6 5 4 3 2 1 0      Bit Index ←

The input data 4 is to be shifted by 2 bits towards the left side.

The answer in binary bits would be

0 0 0 0 1 0 0 0 0      Binary 16 ←

8 7 6 5 4 3 2 1 0      Bit Index ←



**Note:** Shifting the input number by N bits towards the left side means the number is multiplied by  $2^S$ . In short, it means  $Y = N * 2^S$ .

Where,

N = The Number

S = The Number of Bit Positions to Shift

Consider the above example  $4 \ll 2$ . Let us solve this using the above formula, i.e.

$$\begin{aligned} y &= N * 2^S \\ &= 4 * 2^2 \\ &= 4 * 4 \\ &= 16 \end{aligned}$$

Therefore,  $4 \ll 2$  returns 16 in Python interactive mode.

**PROGRAM 3.15** | Write a program to shift input data by four bits towards the left.

```
N = int(input('Enter Number: '))
S = int(input('Enter Number of Bits to be shift Left: '))
print(N, ' << ', S, ' = ', N << S)
```

### Output

```
Enter Number: 4
Enter Number of Bits to be shift Left: 2
4 << 2 = 16
```

## 3.8 THE COMPOUND ASSIGNMENT OPERATOR

The operators +, \*, //, /, % and \*\* are used with the assignment operator (=) to form the compound or augmented assignment operator.

### Example

Consider the following example, where the value of a variable X is increased by 1.

$$X = X + 1$$

Python allows a programmer to combine the assignment and addition operator. Thus, the above statement  $X = X + 1$  can also be written as

$$X += 1$$

The += operator is called the addition operator. A list of all other compound assignment operators is given in Table 3.16.

**Table 3.16** Compound assignment operators

| Operator | Example | Equivalent | Explanation                                                                   |
|----------|---------|------------|-------------------------------------------------------------------------------|
| +=       | Z+=X    | Z=Z+X      | Add the value of Z to X                                                       |
| -=       | Z-=X    | Z=Z-X      | Subtract X from Z                                                             |
| *=       | Z*=X    | Z=Z*X      | Multiplies the value of X, Y and stores the result in Z                       |
| /=       | Z/=X    | Z=Z/X      | Performs floating point division operation and stores the result in Z         |
| //=      | Z//=X   | Z=Z//X     | Performs normal integer floor division and stores the result in Z             |
| **=      | Z**=X   | Z=Z**X     | The value of variable X is raised to Z and the result is stored in variable Z |
| %=       | Z%=X    | Z=Z%X      | The Z modulo X operation is performed.                                        |

**PROGRAM 3.16** | Write a program using compound assignment operators to calculate the area of a circle.

```
radius = eval(input('Enter the Radius of Circle: ')) #Read Radius
print(' Radius = ',radius) #Display Radius
area = 3.14
radius **=2 #Radius = Radius ** 2
area*=radius #Area=Area*Radius
print(' Radius of Circle is = ',area) #Print area
```

### Output

```
Enter the Radius of Circle: 2
Radius = 2
Radius of Circle is = 12.56
```

Thus, to perform various operations in the above program we have to make use of compound assignment operators such as **\*\*=**, and **\*=**.

## MINI PROJECT

### Goods Service Tax (GST) Calculator

#### What is GST?

Goods and services tax is a comprehensive tax **levied on the manufacture, sale and consumption of goods and services** at a national level. This tax has **substituted all indirect taxes** levied on goods and services earlier by the central and state governments in India.

#### Problem Statement

We all buy various goods from a store. Along with the price of the goods we wish to buy, we also have to pay an additional tax, which is calculated as a specific percentage on the total price of the goods. This is called GST on the products.

## Model of GST Using an Example

The GST has two components, viz. one which is levied by the central government (referred to as central GST or **CGST**), and one levied by the state government (referred to as state GST or **SGST**). The **rates** for central GST and state GST are given as follows:

| Type of Tax | Tax Rate |
|-------------|----------|
| CGST        | @9%      |
| SGST        | @9%      |

### Example

Invoice of a product

| Particulars                   | GST on Particulars |
|-------------------------------|--------------------|
| Cost of Production            | 5,000              |
| Add: CGST @ 9%                | 450                |
| Add: SGST @ 9%                | 450                |
| <b>Total Cost of Product:</b> | <b>₹5,900</b>      |

### Formula to Calculate Total Cost

(CGST Tax Rate on product) + (SGST Tax Rate on product)



**Note:** Make use of proper operators to solve the above problem.

### Algorithm

- ⊙ **STEP 1:** Read Cost of Production
- ⊙ **STEP 2:** Input the CGST tax rate
- ⊙ **STEP 3:** Input the SGST tax rate
- ⊙ **STEP 4:** Calculate and print the total cost of the product.

### Program

```
CP = float(input('Enter the Cost of Product:'))
CGST = float(input('Enter tax % imposed by Centre, i.e. CGST:'))
SGST = float(input('Enter tax % imposed by State, i.e. SGST:'))
total = 0
Amount_CGST = ((CGST/100) * CP)
Amount_SGST = ((SGST/100) * CP)
total = CP + Amount_CGST + Amount_SGST
print('Total Cost of Product: Rs ',total)
```

(Contd.)



**Output**

```
Enter the Cost of Product: 5000
Enter tax % imposed by Centre, i.e. CGST: 9
Enter tax % imposed by State, i.e. SGST: 9
Total Cost of Product: Rs. 5900.0
```

In the above example, we have calculated final cost of the product based on the tax rate.

**SUMMARY**

- ◆ Python supports various operators such as Arithmetic, Boolean, Relational, Bitwise and compound Assignment Operator.
- ◆ Unary Operator perform operation on one operand only whereas Binary operator requires two operands.
- ◆ The Division(/) operator applied on two operands returns a float value.
- ◆ Modulo (%) operator returns remainder when first number is divided by the second.
- ◆ Exponent (\*\*) operator calculates power of number.
- ◆ Operator precedence determines the order in which python evaluates the operators in an expression.
- ◆ Associativity gives direction of execution, i.e. **left to right** or **right to left**.

**KEY TERMS**

- ⇒ **Arithmetic Operators:** Binary and Unary Operators
- ⇒ **Bitwise Operators:** and (&), or (|), xor (^), left shift (<<) and right shift (>>)
- ⇒ **Augmented Assignment Operator:** Operators used with the assignment operator
- ⇒ **Operator Precedence:** Determines the order in which the Python interpreter evaluates an expression
- ⇒ **Associativity:** Determines which operation is to be performed first.

**REVIEW QUESTIONS****A. Multiple Choice Questions**

1. What will be the output of the following expression if it is executed in Python interactive mode?

**16 % 3**

- |      |       |
|------|-------|
| a. 5 | b. 1  |
| c. 0 | d. -1 |

2. What will be the output of the following program?

```
X=5
Y=5
print (X/Y)
```

- a. 1  
c. 0.1
- b. 1.0  
d. None of the above
3. What will be the output of the following statement?  
`print (15 + 20 / 5 + 3 * 2 - 1)`  
 a. 19.0  
b. 19  
c. 12.0  
d. 24.0
4. What will be the output of the following program?  
`A=7`  
`B=4`  
`C=2`  
`print (a//b/c)`  
 a. 0.85  
b. 0  
c. 0.5  
d. 0.0
5. Which one of the following operators belongs to floor division?  
 a. %  
b. /  
c. //  
d. None of the above
6. What will be the output of the following expression?  
`4*1**2`  
 a. 16  
b. 4  
c. 8  
d. 1
7. What will be the output of the following program?  
`X=4.6`  
`Y=15`  
`Z=X//Y`  
`print (Z)`  
 a. 0  
b. 0.0  
c. 0.30  
d. None of the above
8. Operators with the same precedence are evaluated in which of the following orders?  
 a. Left to Right  
b. Right to Left  
c. Unpredictable  
d. None of the above
9. What will be the output, if the input data 5 is shifted towards the left by 2 bits?  
 a. 20  
b. 10  
c. 1  
d. 25
10. Which of the following have the highest precedence in an expression?  
 a. Addition  
b. Multiplication  
c. Exponent  
d. parenthesis

## B. True or False

- Operators operate on operands.
- Binary operators operate on at least two operators.
- The '-' operator in Python can be used with binary and unary form.
- 4.5-1.5 returns 3.0.

5. The unary arithmetic operator performs mathematical operations on more than one operand.
6. The operator precedence determines the order in which the Python interpreter operates the operators in an expression.
7. Associativity implies the direction of execution of an expression.
8. Shifting the input number by N bits towards the left means the number is divided by  $2^s$ .
9. Shifting the input number by N bits towards the right means the number is divided by  $2^s$ .
10. The right shift operator is represented as `>>`.
11. The `()` operator has the highest precedence among all other arithmetic operators.

### C. Exercise Questions

1. State the results of the following expressions.

| Expression | Results |
|------------|---------|
| 40/8       |         |
| 40//8      |         |
| 50%5       |         |
| 3%2        |         |
| 3**3       |         |

2. State the output of each of the following expression, if each expression is independent. Assume the value of X as 4.

| Expression | Output |
|------------|--------|
| X +=10     |        |
| X -=4      |        |
| X *=6      |        |
| X **=2     |        |
| X %=2      |        |
| X /=2      |        |

3. Values assigned to different variables are

A = 10

B = 20

C = 40

D = 4

E = 5

Evaluate each of the following Python expression.

- (i)  $(A + B) * C$
- (ii)  $A + (B - E)$
- (iii)  $A*B/E$
- (iv)  $C/B//5$
- (v)  $C+(A*E)/(B-A)$

4. Convert the following expressions into their shortest form.

| Expression        | Equivalent Expression |
|-------------------|-----------------------|
| $Z = Z * 10 + 4$  |                       |
| $A = A \% 20$     |                       |
| $B = B ** 10 + 2$ |                       |
| $C = C / 3$       |                       |

5. Find the output of each expression given below if Python executes each expression separately.

Initially the value of  $X = 4$

| Expression   | Output |
|--------------|--------|
| $X = X << 2$ |        |
| $X = X >> 2$ |        |
| $X = x >> 3$ |        |
| $X = X << 3$ |        |

6. Determine the hierarchy of operations and evaluate the following expressions.

$$X = 4/2*2+16/8+5$$

$$Y = 3*4/2+2/2+6-4+4/2$$

7. Convert the following equations into their corresponding Python expressions.

$$(a) \quad \frac{2XY}{C+10} - \frac{X}{4(Z+D)}$$

$$(b) \quad Z = \frac{\frac{10Y(ab+C)}{d} - 0.8 + 2b}{(x+a)\left(\frac{1}{z}\right)}$$

8. The programmer has to find out the area of a rectangle but he/she has one constraint, viz. he/she has to take the value of the length and breadth of the rectangle from the user. The programmer has written the following program but he/she is unable to detect the bug in the program. Go through the following program to find the bug and then rewrite the whole program.

```
area=0
length = 0
breadth = 0
area= length * breadth
length=eval(input('Enter the Length of Rectangle:'))
breadth=eval(input('Enter the Breadth of Rectangle:'))
print('Area of Rectangle = ',area)
```

9. Evaluate the expression,  $(X + Y - \text{abs}(X - Y))/2$ , when

$$X = 4 \text{ and } Y = 6$$

$$X = 5 \text{ and } Y = 4$$

## PROGRAMMING ASSIGNMENTS

1. Write a program to read the marks of 5 subjects through the keyboard. Find out the aggregate and percentage of marks obtained by the student. Assume maximum marks that can be obtained by a student in each subject as 100.
2. Write a program to read a four-digit number through the keyboard and calculate the sum of its digits.
3. Write a program to read the distance between any two cities in kilometer (km) and print the distances in meters (m), centimeters (cm) and miles.

**Note:** 1 km = 1000 meter

1 km = 100000 centimeter

1 km = 0.6213 miles

4. Write a program to read the weight of an object in kilogram and print its weight in pound and tonne.

**Note:** 1 kg = 2.20 pound

1 kg = 0.001 tonne

5. Read a distance in meters and a time in seconds through the keyboard. Write a program to calculate the speed of a car in meter/second.

**Note:**  $\text{Speed} = \frac{\text{Distance}}{\text{Time}}$

6. Write a program to read the radius of a sphere from the user and calculate the volume of the sphere.

**Note:**  $\text{Volume of sphere} = \frac{4}{3} \times 3.14 \times r^3$

7. An ATM contains Indian currency notes of 100, 500 and 1000. To withdraw cash from this ATM, the user has to enter the number of notes he/she wants of each currency, i.e. of 100, 500 and 1000. Write a program to calculate the total amount withdrawn by the person from the ATM in rupees.

# 4

# Decision Statements

## CHAPTER OUTLINE

|                                          |                                                  |
|------------------------------------------|--------------------------------------------------|
| 4.1 Introduction                         | 4.6 Boolean Expressions and Relational Operators |
| 4.2 Boolean Type                         | 4.7 Decision Making Statements                   |
| 4.3 Boolean Operators                    | 4.8 Conditional Expressions                      |
| 4.4 Using Numbers with Boolean Operators |                                                  |
| 4.5 Using String with Boolean Operators  |                                                  |

## LEARNING OUTCOMES

*After completing this chapter, students will be able to:*

- Describe Boolean expressions and **bool** data type
- Perform operations on numbers and strings using **Boolean** and **Relational** operators (>, <, >=, <= and !=)
- Write a simple decision making statement and its implementation with **if** statement, two-way decision making statements and their implementation with **if else** statement, nested statements and their implementation with **if** statements and multi-way decision making statements and their implementation with **if-elif-else** statements
- Explain and use **conditional expressions** to write programs
- Write non-sequential programs using Boolean expressions

## 4.1 INTRODUCTION

So far, we have seen programs that contain a sequence of instructions. These programs are executed by the compiler line by line, in the way the program line appears. The control flow in

such programs is sequential. Control flow refers to the order in which program statements are executed, i.e. when the execution of one statement is complete, the computer control passes to the next statement in the code. This process is similar to reading the text, figures and tables on a page of a book.

In monolithic programs, instructions are executed sequentially one by one in the order in which they come into sight in the program. Of course, this is a fundamental programming concept for beginners to develop simple programs. It is not advisable to have a sequential program writing style for solving every problem. Quite often, it is advantageous in a program to alter the sequence of the flow of statements depending upon the circumstances. In real-time applications, there are a number of situations where a programmer has to change the order of execution of statements based on certain conditions. Therefore, when a programmer desires the control flow to be non-sequential then he/she may use control structures or decision statements. Thus, decision making statements help a programmer in transferring the control from one statement to another in the program. In short, a **programmer decides which statement is to be executed based on a condition**. Decision making statements use conditions which are similar to **Boolean expressions**.

After reading this chapter, a programmer is expected to take up real life problems/applications and implement with Python programming containing conditional statements. Programmer may think the programming pattern for preparation of mark sheet, grade sheet, preparation of electricity bill for residential and commercial consumers, Railway tariff based on distances, simple calculations of interest on deposits for banking problems, etc. Of course, unlimited problems are existing in the nature for which a programmer is expected to give programming solution.

## 4.2 BOOLEAN TYPE

Python has a type called '**bool**'. The bool has only two values, viz. **true** and **false**. The term, 'Boolean' comes from the name of the British mathematician, George Boole. In the 1840s, Boole showed that the classical rules of logic could be expressed in purely mathematical form using only two values, viz. true and false. The simplest Boolean expression in Python is True and False. In Python interactive shell, a programmer can check if the type of two values, viz. true and false belong to the type '**bool**' in the following manner:

```
>>> True
True
>>> False
False
>>> type(True)
<class 'bool'> #The Value True belongs to the class type bool
>>> type(False)
<class 'bool'> #The Value False belongs to the class type bool
```



**Note:** There are only two Boolean values, **True** and **False**. Capitalisation of the first letter is important for these values and so **true** and **false** are not considered Boolean values in Python. As illustrated, the Python interpreter will show an error if a programmer checks the type of 'true' or 'false'.

(Contd.)

```
>>> type(true)
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
type(true)
NameError: name 'true' is not defined
```

## 4.3 BOOLEAN OPERATORS

The **and**, **or** and **not** are the only three basic Boolean operators. Boolean operators are also called **logical operators**. The **not** operator has the highest precedence, followed by **and** and then **or**.

### 4.3.1 The not Operator

The **not** operator is a unary operator. It is applied to just one value. The **not** operator takes a single operand and negates or inverts its Boolean value. If we apply the not operator on an expression having false value then it returns it as true. Similarly, if we apply the not operator on an expression having true value then it returns it as false.

#### Example

Use of the not operator on a simple Boolean expression in Python, i.e. true and false.

```
>>> True
True
>>> not True
False
>>> False
False
>>> not False
True
```

### 4.3.2 The and Operator

The **and** is a binary operator. The and operator takes two operands and performs left to right evaluation to determine whether both the operands are true. Thus, and of Boolean operand is **true** if and only if both operands are **true**. Table 4.1 explains the add operator.

**Table 4.1** The **and** operator

| X     | Y     | X and Y |
|-------|-------|---------|
| True  | True  | True    |
| True  | False | False   |
| False | True  | False   |
| False | False | False   |



**Example**

Evaluation of the **and** operator in Python interactive mode.

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

**4.3.3 The or Operator**

The **or** of two Boolean operands is true if at least one of the operands is true. Table 4.2 explains the **or** operator.

**Table 4.2** The **or** operator

| X     | Y     | X or Y |
|-------|-------|--------|
| True  | True  | True   |
| True  | False | True   |
| False | True  | True   |
| False | False | False  |

**Example**

Evaluation of the **or** operator in Python interactive mode.

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

**4.4 USING NUMBERS WITH BOOLEAN OPERATORS**

A programmer can use numbers with Boolean operators in Python. One such example is given as follows:

**Example**

```
>>> not 1
False
```

```

>>> 5
5
>>> not 5
False
>>> not 0
True
>>> not 0.0
True

```

**Explanation** Here, Python uses the Boolean operator **not** on the numbers and treats all numbers as **True**. Therefore, by writing **not 1**, Python substitutes 1 as **True** and evaluates **not True**, which returns **False**. Similarly, **not** is used before 5 and Python substitute **True** in place of 5 and it again evaluates the expression **not True**, which returns **False**. But in case of the numbers 0 and 0.0, Python treats them as **False**. Therefore, while evaluating **not 0**, it substitutes **False** in place of 0 and again evaluates the expression **not False**, which returns **True**.

## 4.5 USING STRING WITH BOOLEAN OPERATORS

Like numbers, a programmer can use strings with Boolean operators in Python. One such example is given as follows:

### Example

```

>>> not 'hello'
False
>>> not ''
True

```

**Explanation** Here, Python uses the Boolean operator **not** on string. The expression **not hello** returns **True** since Python treats all strings as **True**. Therefore, it substitutes **True** in place of 'hello' and again reevaluates the expression **not True**, which returns **False**. However, if it is an empty string, Python will treat it as **False**. Therefore, it substitutes **False** in place of an empty string "and reevaluates the expression **not False**, which in turn returns **True**.

## 4.6 BOOLEAN EXPRESSIONS AND RELATIONAL OPERATORS

A Boolean expression is an expression that is either true or false. The following example compares the value of two operands using the **==** operator and produces the result true if the values of both the operands are equal.

### Example

The **==** operator compares two values and produces a Boolean value.

```

>>> 2==2
True

```

```
..
>>> a=2
>>> b=2
>>> a==b
True
```



**Note:** The comparison operator `==` contains two equal signs. Whereas the assignment operator `=` contains only one equal sign.

From the above example, it is clear how we can compare two values or two operands. Thus, `==` is one of the Python **relational operators**. Other relational operators supported in Python are given in Table 4.3.

**Table 4.3** Relational operators

| Operator           | Meaning                  | Example              | Python Return Value |
|--------------------|--------------------------|----------------------|---------------------|
| <code>&gt;</code>  | Greater than             | <code>4&gt;1</code>  | True                |
| <code>&lt;</code>  | Less than                | <code>4&lt;9</code>  | True                |
| <code>&gt;=</code> | Greater than or equal to | <code>4&gt;=4</code> | True                |
| <code>&lt;=</code> | Less than or equal to    | <code>4&lt;=3</code> | True                |
| <code>!=</code>    | Not equal to             | <code>5!=4</code>    | True                |

**PROGRAM 4.1** | Write a program to prompt a user to enter the values of the three different variables and display the output of the following expressions.

- `p>q>r`
- `p<q<r`
- `p<q` and `q<z`
- `p<q` or `q<z`

```
p,q,r=eval(input('Enter Three Numbers:'))
print(' p =',p,' q = ',q,' r = ',r)
print('(p > q > r) is ', p > q > r)
print('(p < q < r) is ', p < q < r)
print('(p < q) and (q < r) is ', (p < q) and (q < r))
print('(p < q) or (q < r) is ', (p < q) or (q < r))
```

### Output

```
Enter Three Numbers:1,2,3
p = 1 q = 2 r = 3
(p > q > r) is False
(p < q < r) is True
(p < q) and (q < r) is True
(p < q) or (q < r) is True
```



**Note:** An expression always returns a value and a statement does not return any value. A statement may include one or more than one expression.

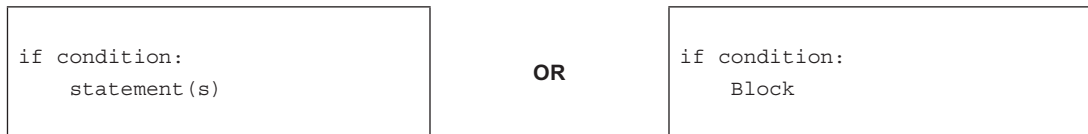
## 4.7 DECISION MAKING STATEMENTS

Python supports various decision-making statements. These are:

1. if statements
2. if-else statements
3. Nested if statements
4. Multi-way if-elif-else statements

### 4.7.1 The if Statements

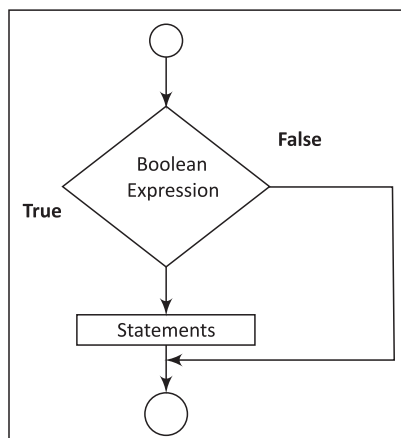
The **if** statement executes a statement if a condition is true. The syntax for if statement is shown in Figure 4.1.



**Figure 4.1** Syntax for **if** statement

#### *Details of the if Statement*

The keyword **if** begins the **if** statement. The condition is a Boolean expression which determines whether or not the body of **if** block will be executed. A colon (:) must always be followed by the condition. The block may contain one or more statements. The statement or statements are executed if and only if the condition within the **if** statement is true. The flow chart for **if** statement is given in Figure 4.2.



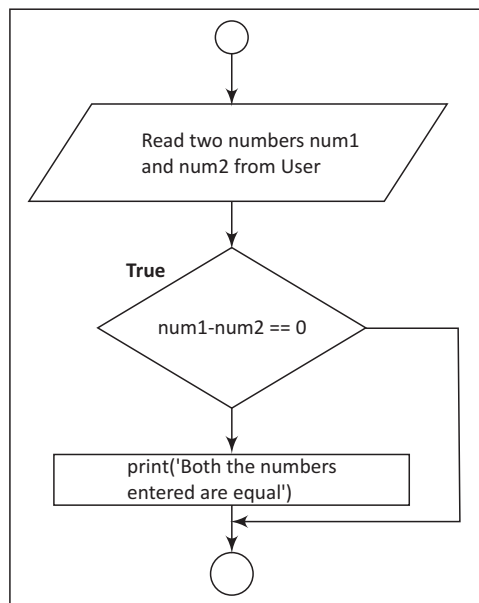
**Figure 4.2** Flowchart for **if** statement

**Points to Remember**

- (a) The statement(s) must be indented at least one space right of the **if** statement.
- (b) In case there is more than one statement after the **if** condition, then each statement must be indented using the same number of spaces to avoid indentation errors.

The statement(s) within the if block are executed if the Boolean expression evaluates to true.

**PROGRAM 4.2** | Write a program that prompts a user to enter two integer values. Print the message '**Equals**' if both the entered values are equal.

**Flow Chart**

```

num1=eval(input("Enter First Number: "))
num2=eval(input("Enter Second Number: "))
if num1-num2==0:
 print("Both the numbers entered are Equal")

```

**Output**

```

Enter First Number: 12
Enter Second Number: 12
Both the numbers entered are Equal

```

**Explanation** In the above program, the two numbers are provided by a user. The statement within the if block is executed if and only if the Boolean expression `num1 - num2` evaluates to True.

**Precautions** Sometimes a program may contain only one statement within the if block. In this case a programmer can write the block of code in two different ways.

(a) Consider the code given as:

```
Number=eval(input("Enter the Number: "))
if Number>0:
 Number = Number * Number
```

This code can also be written as:

```
Number=eval(input("Enter the Number: "))
if Number>0: Number = Number * Number
```

(b) The above code **cannot** be written as:

```
Number=eval(input("Enter the Number: "))
if Number>0:
Number = Number * Number
```

The above code does not run and displays an error called **indentation error**. Thus, Python determines which statement makes a block using indentation.

**PROGRAM 4.3** | Write a program which prompts a user to enter the radius of a circle. If the radius is greater than zero then calculate and print the area and circumference of the circle.

```
from math import pi
Radius=eval(input("Enter Radius of Circle: "))
if Radius>0:
 Area=Radius*Radius*pi
 print(" Area of Circle is = ",format(Area,".2f"))
 Circumference=2*pi*Radius
 print("Circumference of Circle is = ",format(Circumference,".2f"))
```

### Output

```
Enter Radius of Circle: 5
Area of Circle is = 78.54
Circumference of Circle is = 31.42
```

## 4.7.2 The if-else Statement

The execution of the **if** statement has been explained in the previous programs. We know, the **if** statement executes when the condition following **if** is true and it does nothing when the condition is false. The **if-else** statement takes care of a true as well a false condition. The syntax for **if-else** statement is given in Figure 4.3.

```

if condition:
 statement(s)
else:
 statement(s)

```

OR

```

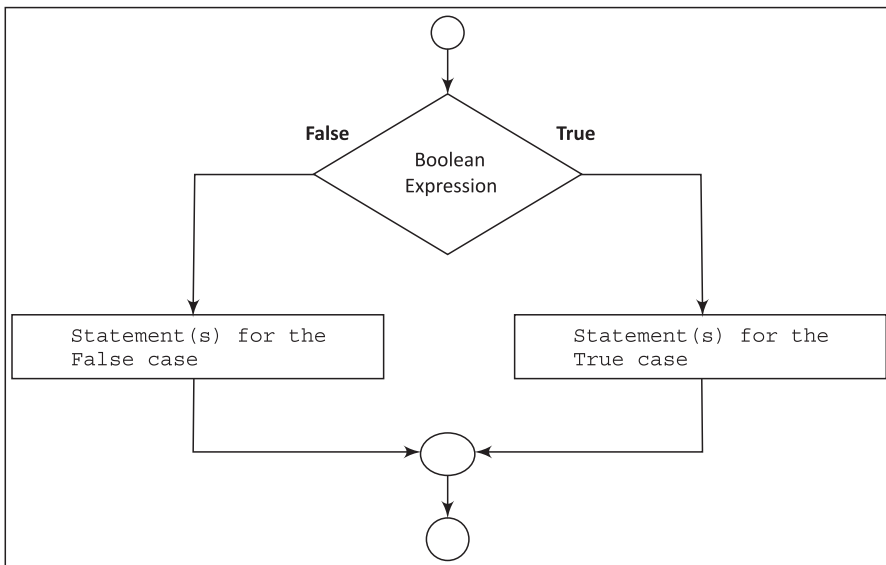
if condition:
 if_Block
else:
 else_Block

```

Figure 4.3 Syntax for **if-else** statement

### Details of if-else Statement

The **if-else** statement takes care of both true and false conditions. It has two blocks. One block is for **if** and it may contain one or more than one statements. The block is executed when the condition is true. The other block is for **else**. The **else** block may also have one or more than one statements. It is executed when the condition is false. A colon (:) must always be followed by the condition. The keyword **else** should also be followed by a colon (:). The flow chart for **if-else** statement is given in Figure 4.4.

Figure 4.4 Flow chart for **if-else** statement

**PROGRAM 4.4** | Write a program to prompt a user to enter two numbers. Find the greater number.

```

num1=int(input("Enter the First Number:"))
num2=int(input("Enter the Second Number:"))
if num1>num2:
 print(num1,"is greater than ",num2)
else:
 print(num2,"is greater than ",num1)

```

(Contd.)

**Output**

```
Enter the First Number:100
Enter the Second Number:43
100 is greater than 43
```

**Explanation** The above program prompts a user to read any two numbers. The two numbers entered are stored in variables num1 and num2, respectively. If the value of num1 is greater than num2 is checked using the **if** condition. If the value of num1 is greater then the message 'num1 is greater than num2' is displayed. Otherwise, the message 'num2 is greater than num1' is displayed.

**PROGRAM 4.5** Write a program to calculate the salary of a medical representative considering the sales bonus and incentives offered to him are based on the total sales. If the sales exceed or equal to ₹1,00,000 follow the particulars of Column 1, else follow Column 2.

| Column 1                 | Column 2                |
|--------------------------|-------------------------|
| Basic = ₹4000            | Basic = ₹4000           |
| HRA = 20% of Basic       | HRA = 10% of Basic      |
| DA = 110 % of Basic      | DA = 110 % of Basic     |
| Conveyance = ₹500        | Conveyance = ₹500       |
| Incentive = 10% of Sales | Incentive = 4% of Sales |
| Bonus = ₹1000            | Bonus = ₹500            |

```
Sales=float(input('Enter Total Sales of the Month:'))
if Sales >= 100000:
 basic = 4000
 hra = 20 * basic/100
 da = 110 * basic/100
 incentive = Sales * 10/100
 bonus = 1000
 conveyance = 500
else:
 basic = 4000
 hra = 10 * basic/100
 da = 110 * basic/100
 incentive = Sales * 4/100
 bonus = 500
 conveyance = 500

salary= basic+hra+da+incentive+bonus+conveyance
print('Salary Receipt of Employee ')
```

(Contd.)



```

print(' Total Sales = ',Sales)
print(' Basic = ',basic)
print(' HRA = ',hra)
print(' DA = ',da)
print(' Incentive = ',incentive)
print(' Bonus = ',bonus)
print(' Conveyance = ',conveyance)
print(' Gross Salary = ',salary)

```

### Output

```

Enter Total Sales of the Month:100000
Salary Receipt of Employee
Total Sales = 100000.0
Basic = 4000
HRA = 800.0
DA = 4400.0
Incentive = 10000.0
Bonus = 1000
Conveyance = 500
Gross Salary = 20700.0

```

**Explanation** The program calculates the salary of a medical representative according to the total sale of products. The basic salary is the same but other allowances and incentives change according to the total sales. **If** the total sale is more than ₹1,00,000 the rate of allowances and incentive is calculated as per Column 1, **else** as per Column 2. The **if** condition checks the given figure of total sale. If the total sale is more than ₹1,00,000 the first block following the **if** statement is executed, otherwise the **else** block is executed.

### Points to Remember

- Indentation is very important in Python. The **else** keyword must properly line up with the **if** statement.
- If a programmer does not line up **if** and **else** in exactly the same columns then Python will not know that **if** and **else** will go together. Consequentially, it will show an indentation error.
- Both statements within the **if** block and **else** block must be indented and must be indented the same amount.

**PROGRAM 4.6** | Write a program to test whether a number is divisible by 5 and 10 or by 5 or 10.

```

num=int(input('Enter the number:'))
print('Entered Number is: ',num)

```

(Contd.)

```

if(num % 5 == 0 and num % 10==0):
 print(num,' is divisible by both 5 and 10')
if(num % 5 == 0 or num % 10 == 0):
 print(num,'is divisible by 5 or 10')
else:
 print(num,' is not divisible either by 5 or 10')

```

### Output

#### #Test Case 1:

```

Enter the number:45
Entered Number is: 45
45 is divisible by 5 or 10

```

#### #Test Case 2:

```

Enter the number:100
Entered Number is: 100
100 is divisible by both 5 and 10
100 is divisible by 5 or 10

```

**Explanation** In the above program, the number is read from the user. The Boolean expression **num % 5 == 0 and num % 10==0** checks whether the number is divisible by both 5 and 10. Again the Boolean expression **num % 5 == 0 or num % 10 == 0** is used to check if the number entered is divisible either by 5 or by 10.



**Note: Conditional or Short Circuit AND Operator:** If one of the operands of an AND operator is false, the expression is false. Consider two operands OP1 and OP2. When evaluating OP1 and OP2, Python first evaluates OP1 and if OP1 is **True** then Python evaluates the second operand OP2. Python improves the performance of the AND operator, i.e. if the operand OP1 is **False**, it does not evaluate the value of the second operand OP2. The AND operator is also referred to as **conditional or short circuit AND operator**.

**Conditional or Short Circuit OR Operator:** We have seen in Table 4.2 that even if one of the operands of an OR operator is **True**, the expression is **True**. Python improves the performance of the OR operator. Consider two operands OP1 and OP2 and the expression OP1 or OP2. While evaluating the expression OP1 or OP2, Python first evaluates OP1. If OP1 is **False**, it evaluates OP2. If OP1 is **True**, it does not evaluate OP2. The OR operator is also referred to as **conditional or short circuit OR operator**.

## 4.7.3 Nested if Statements

When a programmer writes one **if** statement inside another **if** statement then it is called a **nested if** statement. A general syntax for **nested if** statements is given as follows:

```

if Boolean-expression1:
 if Boolean-expression2:
 statement1

```

```

..
 else:
 statement2
else:
 statement3

```

In the above syntax, if the Boolean-expression1 and Boolean-expression2 are correct then statement1 will execute. If the Boolean-expression1 is correct and Boolean-expression2 is incorrect then statement2 will execute. And if both Boolean-expression1 and Boolean-expression2 are incorrect then statement3 will execute.

A program to demonstrate the use of **nested if** statements is given as follows:

**PROGRAM 4.7** | Write a program to read three numbers from a user and check if the first number is greater or less than the other two numbers.

```

num1=int(input("Enter the number:"))
num2=int(input("Enter the number:"))
num3=int(input("Enter the number:"))
if num1>num2:
 if num2>num3:
 print(num1,"is greater than ",num2,"and ",num3)
else:
 print(num1," is less than ",num2,"and",num3)
print("End of Nested if")

```

### Output

```

Enter the number:12
Enter the number:34
Enter the number:56
12 is less than 34 and 56
End of Nested if

```

**Explanation** In the above program, three numbers—num1, num2 and num3—are provided from the user through a keyboard. Initially, the **if** condition with Boolean expression **num1>num2** is checked if it is true and the then other nested **if** condition with Boolean expression **num2>num3** is checked. If both the **if** conditions are true then the statements following the second if statement are executed.

## 4.7.4 Multi-way if-elif-else Statements

The syntax for **if-elif-else** statements is given as follows:

```

If Boolean-expression1:
 statement1
elif Boolean-expression2 :

```

```

 statement2
 elif Boolean-expression3 :
 statement3
- - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - -
elif Boolean-expression n :
 statement N
else :
 Statement(s)

```

In this kind of statements, the number of conditions, i.e. Boolean expressions are checked from top to bottom. When a true condition is found, the statement associated with it is executed and the rest of the conditional statements are skipped. If none of the conditions are found true then the last else statement is executed. If all other conditions are false and if the final else statement is not present then no action takes place.

**PROGRAM 4.8** | Write a program to prompt a user to read the marks of five different subjects. Calculate the total marks and percentage of the marks and display the message according to the range of percentage given in table.

| Percentage            | Message      |
|-----------------------|--------------|
| per >= 90             | Distinction  |
| per >= 80 && per < 90 | First Class  |
| per >= 70 && per < 80 | Second Class |
| per >= 60 && per < 70 | First Class  |
| per <60               | Fail         |

```

Subject1=float(input("Enter the Marks of Data-Structure:"))
Subject2=float(input("Enter the Marks of Python:"))
Subject3=float(input("Enter the Marks of Java:"))
Subject4=float(input("Enter the Marks of C Programming:"))
Subject5=float(input("Enter the Marks of HTML:"))
sum=Subject1+Subject1+Subject3+Subject4+Subject5
per=sum/5
print("Total Marks Obtained", sum, "Out of 500")
print("Percentage = ",per)

if per>=90:
 print("Distinction")
else:
 if per>=80:
 print(" First Class")

```

(Contd.)

```

else:
 if per >= 70:
 print("Second Class")
else:
 if per >= 60:
 print("Pass")
 else:
 print("Fail")

```

### Output

```

Enter the Marks of Data-Structure: 60
Enter the Marks of Python: 70
Enter the Marks of Java: 80
Enter the Marks of C Programming: 90
Enter the Marks of HTML: 95
Total Marks Obtained 385.0 out of 500
Percentage = 77.0
Second Class

```

**Explanation** In the above program, the marks of five subjects are entered through a keyboard. Their sum and average is calculated. The percentage obtained is stored in the variable 'per'. The obtained percentages are checked with different conditions using if-else blocks and the statements are executed according to the conditions.



**Note:** The above program consists of **if-else-if** statements. It can also be written in **if-elif-else** form as shown in Figure 4.5(b).

```

if per >= 90:
 print("Distinction")
else:
 if per >= 80:
 print(" First Class")
 else:
 if per >= 70:
 print("Second Class")
 else:
 if per >= 60:
 print("Pass")
 else:
 print("Fail")

```

(a)

Equivalent

```

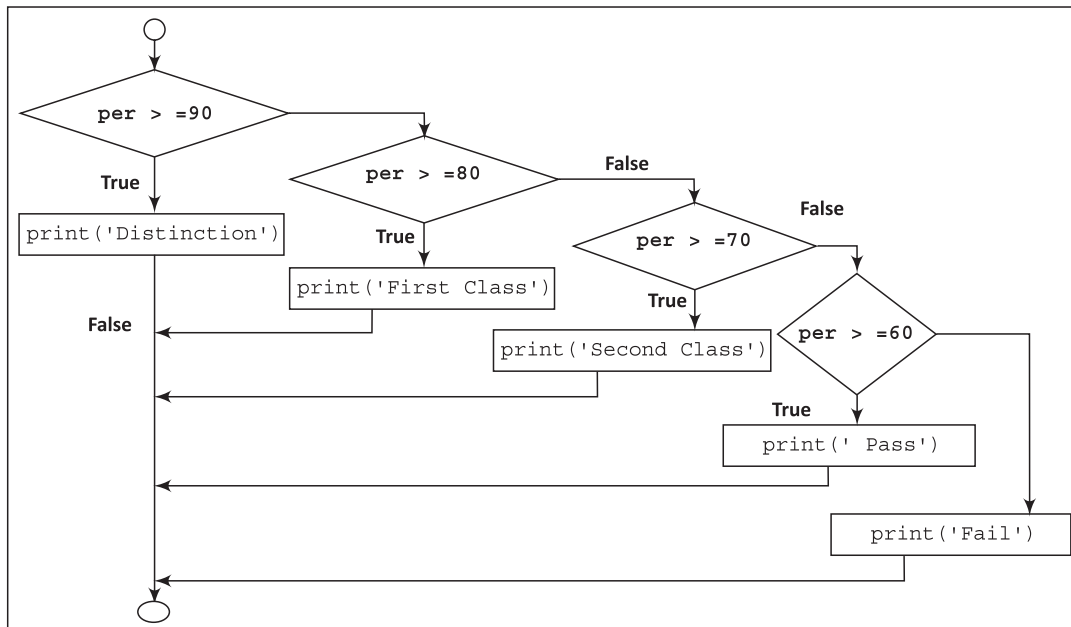
if per >= 90:
 print("Distinction")
elif per >= 80:
 print(" First Class")
elif per >= 70:
 print("Second Class")
elif per >= 60:
 print("Pass")
else:
 print("Fail")

```

(b)

**Figure 4.5** (a) if-else-if-else (b) if-elif-else

The flowchart for multi-way **if-else-if** statements for the above program is given in Figure 4.6.



**Figure 4.6** Flowchart for multi-way **if-else-if** statements

**PROGRAM 4.9** Write a program to prompt a user to enter a day of the week. If the entered day of the week is between 1 and 7 then display the respective name of the day.

```

Day=int(input("Enter the day of week:"))
if day==1:
 print(" Its Monday")
elif day==2:
 print("Its Tuesday")
elif day==3:
 print("Its Wednesday")
elif day==4:
 print("Its Thursday")
elif day==5:
 print("Its Friday")
elif day==6:
 print("Its Saturday")
elif day==7:
 print(" Its Sunday")
else:
 print("Sorry!!! Week contains only 7 days")

```

(Contd.)

**Output**

```
Enter the day of week: 7
Its Sunday
```

**PROGRAM 4.10** | Write a program that prompts a user to enter two different numbers. Perform basic arithmetic operations based on the choices.

```
num1=float(input("Enter the first number:"))
num2=float(input("Enter the Second number:"))

print("1) Addition ")
print("2) Subtraction ")
print("3) Multiplication ")
print("4) Division ")

choice = int(input("Please Enter the Choice:"))
if choice==1:
 print(" Addition of ",num1,"and",num2,"is:",num1+num2)
elif choice==2:
 print(" Subtraction of ",num1,"and",num2,"is:",num1-num2)
elif choice==3:
 print(" Multiplication of ",num1,"and",num2,"is:",num1*num2)
elif choice==4:
 print(" Division of ",num1,"and",num2,"is:",num1/num2)
else:
 print("Sorry!!! Invalid Choice")
```

**Output**

```
Enter the first number:15
Enter the Second number:10
1) Addition
2) Subtraction
3) Multiplication
4) Division
Please Enter the Choice:3
Multiplication of 15.0 and 10.0 is: 150.0
```

## 4.8 CONDITIONAL EXPRESSIONS

Consider the following piece of code.

```
if x%2==0:
 x = x*x
```

```

else:
 x = x*x*x

```

In the above code, initially, x is divided by 2. If x is divisible by 2 then the square of the number is assigned to variable x, else the cube of the number is assigned. To improve the performance of simple if-else statements, Python provides a conditional expression. Using this conditional expression, the code above can be rewritten as:

```
x=x*x if x % 2 == 0 else x*x*x
```

Therefore, the general form of conditional expression is:

```
Expression1 if condition else Expression2
```

Expression<sub>1</sub> is the value of the conditional expression if the condition is true.

Condition is a normal Boolean expression that generally appears in front of an if statement.

Expression<sub>2</sub> is the value of the conditional expression if the condition is false.

Consider the program without conditional expression given as follows:

**PROGRAM 4.11** | Write a program to find the smaller number among the two numbers.

```

num1=int(input('Enter two Numbers:'))
num2=int(input('Enter two Numbers:'))
if num1 < num2:
 min=num1
 print('min = ',min)
else:
 min=num2
 print('min = ',min)

```

### Output

```

Enter two Numbers: 20
Enter two Numbers: 30
min = 20

```

The same program can be written using conditional expression as follows:

```

num1=int(input('Enter two Numbers:'))
num2=int(input('Enter two Numbers:'))
min = print('min = ',num1) if num1 < num2 else print('min = ',num2)

```

### Output

```

Enter two Numbers: 45
Enter two Numbers: 60
min = 45

```





**Note:** Many programming languages, such as Java, C++ have a '?:', i.e. **ternary operator**. This is a conditional operator. The syntax for the '?:' ternary operator is:

```
Boolean expression? if_true_return_value1: if_false_return_value2
```

The ternary operator works like **if-else**. If the Boolean expression is true, it returns value1 and if the Boolean expression is false, it returns the second value.

Python does not have a ternary operator. It uses a **conditional expression**.

## MINI PROJECT Finding the Number of Days in a Month

This mini project will make use of programming features such as **if statement** and **elif** statements. It will help a programmer to know the number of days in a month.

**Hint:** If entered the month is 2 then read the corresponding year. To know the number of days in month 2 check if the entered year is a leap year. If leap then **num\_days = 29** or not leap then **num\_days = 28** for month 2, respectively.

**Leap year:** A leap year is divisible by 4 but not by 100 or divisible by 400.

### Algorithm

- ⊙ **STEP 1:** Prompt the month from the user.
- ⊙ **STEP 2:** Check if the entered month is 2, i.e. February. If so then go to **Step 3**, else go to **Step 4**.
- ⊙ **STEP 3:** If the entered month is 2 then check if the year is a **leap** year. If it is a leap year then store **num\_days = 29**, else **num\_days = 28**.
- ⊙ **STEP 4:** If the entered month is one of the following from the list (1, 3, 5, 7, 8, 12) then store **num\_days = 31**. Or if the entered month is from the list (4, 6, 9, 11) then store **num\_days = 29**. If the entered month is different from the range (1 to 12) then display message "Invalid Month".
- ⊙ **STEP 5:** If the input is valid then display the message as "there are **N** number of days in the month **M**".

### Program

**#Number of Days in a Month**

```
print('Program will print number of days in a given month')
```

```
#init
```

```
flag = 1 # Assumes user enters valid input
```

```
#Get month from the user
```

```
month = (int(input('Enter the month(1-12):')))
```

(Contd.)

```

Check if entered month = 2 i.e. February
if month == 2:
 year = int(input('Enter year:'))

 if (year % 4 == 0) and (not(year % 100 == 0)) or (year % 400 == 0):
 num_days = 29
 else:
 num_days = 28
if entered month is one from (jan, march, may, july, august, october, or december)
elif month in (1,3,5,7,8,10,12):
 num_days = 31
if entered month is one from (April, June, September November,)
elif month in (4, 6, 9, 11):
 num_days = 30
else:
 print('Please Enter Valid Month')
 flag = 0

#Finally print num_days
if flag == 1:
 print('There are ',num_days, 'days in', month,' month')

```

### Output (Case 1)

Program will print number of days in a given month  
Enter the month(1-12):2  
Enter year: 2020  
There are 29 days in 2 month

### Output (Case 2)

Program will print number of days in a given month  
Enter the month(1-12):4  
There are 30 days in 4 month

Thus, the above case study helps the user to know the number of days for the entered year.

## SUMMARY

- ◆ A Boolean expression contains two values, viz. True and False.
- ◆ True and False are of type 'bool'.
- ◆ The and, or and not are the three basic Boolean operators.

- ♦ The not operator has highest precedence, followed by and and then or.
- ♦ A programmer can use strings with Boolean operators.
- ♦ The == operator compares two values and produces a Boolean value.
- ♦ Python supports various relational Operators such as >, <, >=, <= and !=.
- ♦ Applying relational operators on numbers and characters yields a Boolean value.
- ♦ Python Supports various decision statements, such as if, if-else and multi-way if-elif-else statements.
- ♦ Python does not have a ternary operator. It uses a conditional expression instead.

## KEY TERMS

- ⇒ **Boolean Expressions:** An expression whose value is either **True** or **False**.
- ⇒ **Logical Operators:** Comprise the **and**, **or** and **not** operators.
- ⇒ **Relational Operators:** Comparison of two values with relational operators, such as <, <=, >, >=, != and == operators. One of the operators among them is used while comparing two operands.
- ⇒ **Conditional Expression:** Evaluates expression based on condition.
- ⇒ **Conditional or Short Circuit AND Operator:** Improves performance. Python avoids executing the second operand in case the first operand is false.
- ⇒ **Conditional or Short circuit OR Operator:** Improves performance. Python avoids executing the second operand in case the first operand is true.

## REVIEW QUESTIONS

### A. Multiple Choice Questions

1. What will be the output of following program after the execution of the following code?

```
x = 0
y = 0
if x > 0:
 y = y + 1
else:
 if x < 0 :
 y = y + 2
 else:
 y = y + 5
print(' Y =',y)
```

- a. 1
  - b. 0
  - c. 2
  - d. 5
2. What will be stored in num after the execution of the following code?

```
i=10
j=20
k=30
```

```

if j>k:
 if i > j:
 num = i
 else:
 num = j
else:
 if i > k:
 num = i
 else:
 num = k
print('Num = ',num)

```

- a. 10
  - b. 20
  - c. 30
  - d. None of the above
3. Which of the following Python logical expressions can determine whether x and y are greater than z?
- a.  $x \& y > z$
  - b.  $(x > z) \& (y > z)$
  - c.  $(y > z) \& (x > y)$
  - d. Both b and c
  - e. All of the above
4. Evaluate the following Python expression and tell the value in terms of true and false.
- a. `i=5`  
`j=10`  
`k=15`  
`print(i == k / j)`
  - b. `i=5`  
`k=15`  
`print( k % i < k / i)`
5. What will be the output of the following code fragment assuming num is 10?
- ```

num=10
if num == 20:
    print('Apple')
    print('Grapes')
print('No Output')

```
- a. Apple
 - b. Grapes
 - c. Apple Grapes
 - d. No Output
6. What will be the output of the following program?
- ```

P=int(True)
q=int(False)
print('P = ',p)
print('q = ',q)

```
- a. Error
  - b. p=0 q=1
  - c. p=True q=False
  - d. p = 1 and q = 0
7. Evaluate the following Boolean expressions. Consider the value of P, Q and R as 4, 5 and 6, respectively.
- a.  $P > 7$
  - b.  $P < 7$  and  $Q > 2$
  - c.  $P == 1$
  - d.  $P > 2 || Q > 6$

8. What will be the output of the following program if the value stored in variable num is 19?

```
if num % 2 == 1:
 print(num, ' is odd number')
print(num, ' is even number ')
```

9. Consider the two different blocks of codes a) and b) given as follows. State which of the following codes is better and why.

a.

```
weight = 10
if weight >= 55:
 print(' The person is eligible for Blood Donation ')
if weight < 55:
 print(' The person is not eligible for Blood Donation')
```

b.

```
weight = 10
if weight >= 55:
 print(' The person is eligible for Blood Donation ')
else:
 print(' The person is not eligible for Blood Donation')
```

10. What will be the output of the following program?

```
if (20 < 1) and (1 < -1):
 print("Hello")
elif (20 > 10) or False:
 print('Hii')
else:
 print('Bye')
```

- a. Hello  
c. Bye

- b. Hii  
d. Error

## B. True or False

1. In monolithic programs, the instructions are executed sequentially one by one.
2. There are only three Boolean values.
3. The and, or and not are only three basic Boolean operators.
4. The not operator is a binary operator.
5. In Python, a programmer cannot use numbers along with Boolean operators.
6. A Python programmer can use strings with Boolean operators.
7. The if statement executes a statement if the condition is true.
8. The == operator compares two values and produces a Boolean value.
9. With if-elif-else statements, the number of Boolean expressions is checked from top to bottom. When a true condition is found, the statement associated with it is executed.
10. Integer equivalent of True is 0.

### C. Exercise Questions

- Write the following statement in terms of if-else statement in Python.
  - If temperature is greater than 50 then temperature is hot, otherwise temperature is cold.
  - If age is greater than 18 then fare is \$400, otherwise fare is \$200.
- Write the Boolean expressions for the following statements.
  - If age is greater than 5 and less than 10.
  - If age is less than 3 and greater than 70, display the message “No Air Fare”.
- What are Boolean operators? Explain each operator.
- Is it necessary to change the flow control in a program?
- What are the different ways in which the flow control can be changed in Python?
- List few Boolean expressions with relational operators.
- Give the syntax for if\_else statement.
- Illustrate the nested if statements with a suitable example.
- What is a conditional expression?
- Draw and explain multi-way if-elif-else statements.

## PROGRAMMING ASSIGNMENTS

- Write a program to prompt (input) year and check if it is a leap year.
- Write a program to calculate an Internet browsing bill. Use the conditions specified as follows:
  - 1 Hour – ₹20
  - ½ Hour – ₹10
  - Unlimited hours in a day – ₹100

The owner should enter the number of hours spent on browsing.

- Write nested if statements to print the appropriate message depending on the value of the variables temperature and humidity as given as follows. Assume that the temperature can only be warm and cold and the humidity can only be dry and humid.

| <i>if temperature is</i> | <i>if humidity is</i> | <i>Print this activity</i> |
|--------------------------|-----------------------|----------------------------|
| Warm                     | Dry                   | Play Basketball            |
| Warm                     | Humid                 | Play Tennis                |
| Cold                     | Dry                   | Play Cricket               |
| Cold                     | Humid                 | Swim                       |

- Write a program to calculate the square of only those numbers whose least significant digit is 5.

**Example:** Enter the number: 25

Square: 25\*25 = 625

- Consider a college cricket club in which a student can enroll only if he/she is less than 18 and greater than 15 years old. Write a program using the not operator.

# 5

# Loop Control Statements

## CHAPTER OUTLINE

|                                       |                                         |
|---------------------------------------|-----------------------------------------|
| 5.1 Introduction                      | 5.5 Nested Loops                        |
| 5.2 The <code>while</code> Loop       | 5.6 The <code>break</code> Statement    |
| 5.3 The <code>range()</code> Function | 5.7 The <code>continue</code> Statement |
| 5.4 The <code>for</code> Loop         |                                         |

## LEARNING OUTCOMES

*After completing this chapter, students will be able to:*

- Write programs using **for** and **while** loop to repeat a sequence of instructions
- Write a program and perform a task until a condition is satisfied
- Use **loops** to traverse the sequence of characters in string or traverse the sequence of integers
- Apply the syntax and working of **range()** function
- Control the execution of programs using **break** or **continue** statement

## 5.1 INTRODUCTION

In our day-to-day life, we perform certain tasks repeatedly. It can be tedious to perform such tasks using pen and paper. For instance, teaching multiplication tables to multiple classes can become easier if the teacher uses a simple computer program with loop instructions instead of pen and paper.

Let us try to understand the concept of control statements in this context. Suppose a programmer wants to display the message, "I Love Python" 50 times. It would be tedious for him/her to write the statement 50 times on a computer screen or even on paper. This task can become very easy, quick and accurate if the programmer completes it using loop instructions in a computer programming language. Almost all computer programming languages facilitate the use of control loop statements to repeatedly execute a block of code until a condition is satisfied.

Consider the example to print the statement, "I Love Python" 50 times. Assume that the programmer doesn't know the concept of control statements and writes the code in the following manner.

### Example

```
print("I Love Python")
print("I Love Python")
print("I Love Python")
.
.
.
.
.
} print "I Love Python" for 50 times
print('I Love Python')
```

In the above example, the print statement is written for displaying the message 50 times. This can be done more easily using loop in Python. Loops are used to repeat the same code multiple times. Python provides two types of loop statements, viz. **while** and **for** loops. The **while** loop is a **condition controlled** loop. It is controlled by true or false conditions. The **for** loop is a **count controlled** loop which repeats for a specific number of times.

After understanding the concept of loop, a programmer can take up any challenging application in which statements/actions are to be repeated several times.

## 5.2 THE while LOOP

The **while** loop is a loop control statement in Python and frequently used in programming for repeated execution of statement(s) in a loop. It executes a sequence of statements repeatedly as long as a condition remains true. The syntax for while loop is given as follows:

```
while test-condition:
 #Loop Body
 statement(s)
```

### 5.2.1 Details of while Loop

The reserved keyword **while** begins with the **while** statement. The test condition is a Boolean expression. The colon (:) must follow the test condition, i.e. the **while** statement be terminated with a colon (:). The statement(s) within the while loop will be executed till the condition is true, i.e. the condition is evaluated and if the condition is true then the body of the loop is executed. When the



condition is false, the execution will be completed out of the loop or in other words, the control goes out of the loop. The flowchart in Fig. 5.2 shows the execution of the while loop.

## 5.2.2 Flowchart for while Loop

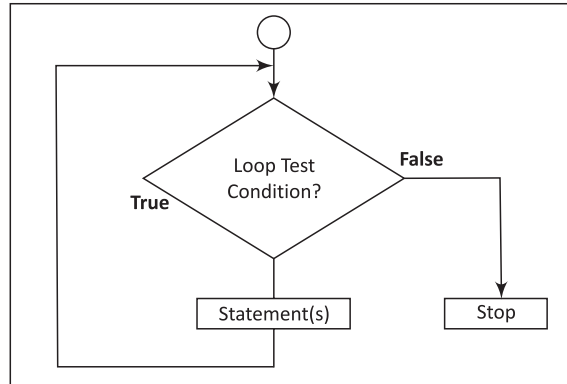


Figure 5.1 Flowchart of while loop

**PROGRAM 5.1** | Write a program to print the numbers from one to five using the while loop.

```

count=0 #initialize the counter
while count<=5: # Test condition
 print("Count = ",count) # print the value of count
 count=count+1 # Increment the value of count by 1

```

### Output

```

Count = 0
Count = 1
Count = 2
Count = 3
Count = 4
Count = 5

```

**Explanation** In the above program, initially the value of a variable **count** is initialised to 0. The loop checks whether the value of the count is less than 5 (**count<=5**). If the condition is **true**, it executes the part of the loop that contains the statements to be repeated in order to display the value of **count** and it increments the value of **count** by 1. It repeatedly executes the statements within the loop until **count<=5**. The loop terminates when the value of **count** reaches 6.



**Note:** Precaution is to be taken while writing statements within the while loop.

Consider the program as shown in Figure 5.2.

|                                                                                 |                                                                                     |
|---------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <pre>count=0 while count&lt;=5:     print("Count = ",count) count=count+1</pre> | <pre>count=0 while count&lt;=5:     print("Count = ",count)     count=count+1</pre> |
| (a) Good Code                                                                   | (b) Bad Code                                                                        |

**Figure 5.2** Precautions regarding the while loop

In Figure 5.2 (a) the value of count is initially set to 0. Then it increments to 2, 3, 4 and 5. When the value of count becomes 6, the condition **count<=5** is false and the loop exits.

Consider the Figure 5.2 (b) where the loop is mistakenly written as:

```
count=0
while count<=5:
 print("Count = ",count)
count=count+1
```

The above code is called **bad code** because the entire loop body must be indented inside the loop. Since the statement **count=count+1** is not in the loop body, the loop executes for infinite number of times. And because the value of count is always 0, the condition **count <=5** is always true.



**Note:** All statements within the while block must be indented with the same number of spaces.

**PROGRAM 5.2** | Write a program to add 10 consecutive numbers starting from 1 using the while loop.

```
count=0 #initialize the counter
sum=0 #initialize sum to zero
while count<=10:
 sum= sum +count
 count=count+1
1
print("Sum of First 10 Numbers = ",sum)
#test condition if true
#add sum + count
#increase the value of count by
#print sum
```

### Output

```
Sum of First 10 Numbers = 55
```

**PROGRAM 5.3** | Write a program to find the sum of the digits of a given number.

For example, if a user enters 123. The program should return (3+2+1), i.e. 6 as the sum of all the digits in a number.

```
num=int(input("Please Enter the number:"))#Read Number from User
x=num #Assign value of num to x
```

(Contd.)

```

sum=0
rem=0
while num>0:
 rem=num % 10
 num=num // 10
 sum=sum + rem
print("Sum of the digits of an entered number ",x," is = ",sum)

```

**Output**

```

Please Enter the number: 12345
Sum of the digits of an entered number 12345 is = 15

```

**Explanation** The integer number is read from the user through the keyboard and it is stored in variable num. Initially, the value of sum and rem are initialised to 0. Unless and until the value of num>0 the statements within the loop continue to be executed. The modulus operator, i.e. **num%10** and the division operator, i.e. **num//10** are used frequently to obtain the sum of the numbers entered.

**5.2.3 Some More Programs on while Loop**

**PROGRAM 5.4** | Write a program to display the reverse of the number entered.

For example, if a user enters 12345. The program should return (54321), i.e. the reverse of the number entered.

```

num =int(input("Please Enter the number: "))
x=num
rev=0
while num>0:
 rem=num % 10
 num=num // 10
 rev=rev*10+rem
print("Reverse of a entered number ",x," is = ",rev)

```

**Output**

```

Please Enter the number: 8759
Reverse of a entered number 8759 is = 9578

```

**PROGRAM 5.5** | Write a program to print the sum of the numbers from 1 to 20 (1 and 20 are included) that are divisible by 5 using the while loop.

```

count=1
sum=0

```

(Contd.)

```

while count<=20:
 if count%5 == 0:
 sum=sum+count
 count=count+1
print("The Sum of Numbers from 1 to 20 divisible by 5 is: ",sum)

```

**Output**

The Sum of Numbers from 1 to 20 divisible by 5 is: 50

**PROGRAM 5.6** | Write a program using the while loop to print the factorial of a number.

Factorial of 6 =  $6*5*4*3*2*1 = 720$

```

Num=int(input("Enter the number:"))
fact=1
ans=1
while fact<=num:
 ans=ans*fact
 fact=fact+1
print("Factorial of",num," is: ",ans)

```

**Output**

Enter the number:6  
Factorial of 6 is: 720



**Note:** The factorial of a number is defined as the product of all the numbers from 1 to n.

**PROGRAM 5.7** | Write a program to check whether the number entered is an Armstrong number or not.

$153 = 1^3 + 5^3 + 3^3 = 153$

```

num=int(input("Please enter the number: "))
sum=0
x=num
while num>0:
 d=num%10
 num=num // 10
 sum=sum+(d*d*d)

```

(Contd.)

```

if (x==sum) :
 print("The number ", x ,"is Armstrong Number")
else:
 print(" The number ", x ,"is not Armstrong Number")

```

**Output**

```

Please enter the number: 153
The number 153 is Armstrong Number

```



**Note:** An Armstrong number is a number which is equal to the sum of the cube of its digits.

## 5.3 THE `range()` FUNCTION

There is a inbuilt function in Python called **`range()`**, which is used to generate a list of integers. The range function has one, two or three parameters. The last two parameters in **`range()`** are optional.

The general form of the range function is:

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

The 'begin' is the first beginning number in the sequence at which the list starts.

The 'end' is the limit, i.e. the last number in the sequence.

The 'step' is the difference between each number in the sequence.

### 5.3.1 Examples of `range()` Function

#### **Example 1**

Create a list of integers from 1 to 5.

```

>>> list(range(1,6))
[1,2,3,4,5]

```

`range(1,6)` function is used in the above example. It generates a list of integers starting from 1 to 5. Note that the second number, i.e. 6 is not included in the elements of this list. By default, the difference between the two successive numbers is one.



**Note:** The above range (1,6) is equivalent to range(6). The output of both the range functions will be the same.

#### **Example 2**

Create a list of integers from 1 to 20 with a difference of 2 between two successive integers.

```

>>> list(range(1,20,2))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

```

`range(1,20,2)` function is used in the above example. It generates a list of integers starting from 1 with a difference of two between two successive integers up to 20.

Table 5.1 shows different examples of the **range()** function with relevant outputs.

**Table 5.1** Examples of `range()` function

| <i>Example of Range Function</i> | <i>Output</i>                |
|----------------------------------|------------------------------|
| <code>range(5)</code>            | [0, 1, 2, 3, 4]              |
| <code>range(1,5)</code>          | [1, 2, 3, 4]                 |
| <code>range(1,10,2)</code>       | [1, 3, 5, 7, 9]              |
| <code>range(5,0,-1)</code>       | [5, 4, 3, 2, 1]              |
| <code>range(5,0,-2)</code>       | [5, 3, 1]                    |
| <code>range (-4,4)</code>        | [-4, -3, -2, -1, 0, 1, 2, 3] |
| <code>range (-4,4,2)</code>      | [-4, -2, 0, 2]               |
| <code>range(0,1)</code>          | [0]                          |
| <code>range(1,1)</code>          | Empty                        |
| <code>range(0)</code>            | Empty                        |

## 5.4 THE for LOOP

The **for** loops in Python are slightly different from the **for** loops in other programming languages. The Python for loop iterates through a sequence of objects, i.e. it iterates through each value in a sequence, where the sequence of object holds multiple items of data stored one after another.

In the forthcoming chapters, we will study various sequence type objects of Python, such as string, list and tuples. The syntax of for loop is given as follows:

```
for var in sequence:
 statement(s)
.....
.....
.....
```

### 5.4.1 Details of for Loop

The **for** loop is a Python statement which repeats a group of statements for a specified number of times. As described in the syntax, the keywords **for** and **in** are essential keywords to iterate the sequence of values. The variable **var** takes on each consecutive value in the sequence and the statements in the body of the loop are executed once for each value. A simple example of for loop is:

```
for var in range(m,n):
 print var
```

As discussed in Section 5.3, the function **range(m, n)** returns the sequence of integers starting from m, m+1, m+2, m+3..... n-1.

**PROGRAM 5.8** | Use for loop to print numbers from 1 to 5.

```
for i in range(1,6):
 print(i)
print("End of The Program")
```

**Output**

```
1
2
3
4
5
End of The Program
```

**Explanation** In the above program, the sequence of numbers from 1 to 5 is printed. These numbers are generated using the inbuilt **range()** function. The expression **range(1, 6)** creates an object known as an iterable. This allows the for loop to assign the values 1, 2, 3, 4 and 5 to the iteration variable *i*. During the first iteration of the loop, the value of *i* is 1 within the block. During the second iteration, the value of *i* is 2 and so on.

**PROGRAM 5.9** | Display capital letters from A to Z.

```
print(" The Capital Letters A to Z are as follows:")
for i in range(65,91,1):
 print(chr(i),end=" ")
```

**Output**

```
The Capital Letters A to Z are as follows:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

**Explanation** The **range()** function contains three different parameters, viz. (**begin, end, step\_size**). As in the above program, the range function contains the values 65, 90 and 1. It indicates to print the characters whose ASCII value starts from 65 and ends at 90. Therefore, the statement **print(chr(i),end=" ")** is used to print equivalent character value of ASCII value.

**5.4.2 Some More Programs on for Loop****PROGRAM 5.10** | Use for loop to print numbers from 1 to 10 in the reverse order.

```
print("Numbers from 1 to 10 in Reverse Order: ")
for i in range(10,0,-1):
 print(i,end=" ")
print("\n End of Program")
```

(Contd.)

**Output**

```
Numbers from 1 to 10 in Reverse Order:
10 9 8 7 6 5 4 3 2 1
End of Program
```

**PROGRAM 5.11** | Write a program to print squares of the first five numbers.

```
for i in range(1,6):
 square=i*i
 print("Square of ",i," is: ",square)
print("End of Program")
```

**Output**

```
Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
Square of 5 is: 25
End of Program
```

**PROGRAM 5.12** | Write a program to print even numbers from 0 to 10 and find their sum.

```
sum=0
print("Even numbers from 0 to 10 are as follows")
for i in range(0,11,1):
 if i%2==0:
 print(i)
 sum=sum+i
print("Sum of Even numbers from 0 to 10 is = ",sum)
```

**Output**

```
Even numbers from 0 to 10 are as follows
0
2
4
6
8
10
Sum of Even numbers from 0 to 10 is = 30
```



**PROGRAM 5.13** | Write a program to calculate the sum of numbers from 1 to 20 which are not divisible 2, 3 or 5.

```
Sum=0
print("Numbers from 1 to 20 which are not divisible by 2,3,or 5")
for i in range(1,20):
 if i%2==0 or i%3==0 or i%5==0:
 print("")
 else:
 print(i)
 sum=sum+i
print("Sum of Even numbers from 1 to 10 is = ",sum)
```

### Output

```
Numbers from 1 to 20 which are not divisible by 2, 3, and 5
1
7
11
13
17
19
Sum of Even numbers from 1 to 10 is = 68
```

**PROGRAM 5.14** | Write a program that prompts a user to enter four numbers and find the greatest number among the four numbers entered.

```
Num1=int(input("Enter the first Number:"))
num2=int(input("Enter the first Number:"))
num3=int(input("Enter the first Number:"))
num4=int(input("Enter the first Number:"))
sum=num1+num2+num3+num4
print("The sum of Entered 5 Numbers is = ",sum)
for i in range(sum):
 if i==num1 or i==num2 or i==num3 or i==num4:
 Large=i
print(" Largest Number = ",Large)
print("End of Program")
```

### Output

```
Enter the first Number: 4
Enter the first Number: 3
Enter the first Number: 12
```

(Contd.)

```

Enter the first Number: 2
The sum of Entered 5 Numbers is = 21
Largest Number = 12
End of Program

```

**PROGRAM 5.15** | Write a program to generate a triangular number.

If the number entered is 5, its triangular number would be  $(1+2+3+4+5) = 15$ .

```

Num=int(input("Please enter the Number: "))
Triangular_Num=0
for i in range(Num,0,-1):
 Triangular_Num=Triangular_Num+i
print(" Triangular Number of ",Num," is = ",Triangular_Num)

```

#### Output

```

Please enter the Number: 10
Triangular Number of 10 is = 55

```



**Note:** A triangular number is nothing but the summation of 1 to the given number.

**PROGRAM 5.16** | Write a program to print Fibonacci series up to 8.

First\_Number = 0

Second\_Number = 1

Fibonacci Series = 0 1 1 2 3 5 8 13 21 34 55

```

First_Number=int(input("Please enter First Number:"))
Second_Number=int(input("Please enter First Number:"))
Limit=int(input(" Number of Fibonacci Numbers to be Print: "))
print(First_Number,end=" ")
print(Second_Number,end=" ")
for i in range(Limit+1):
 sum=First_Number+Second_Number
 First_Number=Second_Number
 Second_Number=sum
 print(sum,end=" ")

```

(Contd.)

**Output**

```
Please enter First Number:0
Please enter First Number:1
Number of Fibonacci Numbers to be Print: 8
0 1 1 2 3 5 8 13 21 34 55
```

## 5.5 NESTED LOOPS

The **for** and **while** loop statements can be nested in the same manner in which the if statements are nested. Loops within the loops or when one loop is inserted completely within another loop, then it is called **nested loop**.

**PROGRAM 5.17** | Write a program to demonstrate the use of the nested for loop.

```
for i in range(1,4,1): #Outer Loop
 for j in range(1,4,1): #Inner Loop
 print("i = ",i," j = ",j," i + j =",i + j)
print("End of Program")
```

**Output**

```
i = 1 j = 1 i + j = 2
i = 1 j = 2 i + j = 3
i = 1 j = 3 i + j = 4
i = 2 j = 1 i + j = 3
i = 2 j = 2 i + j = 4
i = 2 j = 3 i + j = 5
i = 3 j = 1 i + j = 4
i = 3 j = 2 i + j = 5
i = 3 j = 3 i + j = 6
End of Program
```

**Explanation** In the above program, we have used two loops. One is the outer loop and the other is the inner loop. The inner loop 'j' terminates when the value of j exceeds 3. Whereas, outer loop 'i' terminates when the value of i exceeds 3.

**PROGRAM 5.18** | Write a program to display multiplication tables from 1 to 5.

```
Print("Multiplication Table from 1 to 5 ")
for i in range(1,11,1): #Outer Loop
 for j in range(1,6,1): #Inner Loop
```

(Contd.)

```

 print(format(i * j,"4d"),end=" ")
 print()
print("End of Program")

```

### Output

Multiplication Table from 1 to 5

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  |
| 2  | 4  | 6  | 8  | 10 |
| 3  | 6  | 9  | 12 | 15 |
| 4  | 8  | 12 | 16 | 20 |
| 5  | 10 | 15 | 20 | 25 |
| 6  | 12 | 18 | 24 | 30 |
| 7  | 14 | 21 | 28 | 35 |
| 8  | 16 | 24 | 32 | 40 |
| 9  | 18 | 27 | 36 | 45 |
| 10 | 20 | 30 | 40 | 50 |

End of Program

**Explanation** The program contains two for loops. The 'j' for loop is the innermost for loop and the 'i' for loop is the outermost for loop. The outermost loop 'i' executes for 10 times. For each value of 'i', the innermost loop 'j' executes 5 times. At the same time for each value of 'i', the product i\*j is carried out. To align the numbers properly, the program formats the product of i\*j using **format(i\*j,"4d")**. The digit **4d** within **format()** specifies a decimal integer format with width 4.

## 5.5.1 Some More Programs on Nested Loops

**PROGRAM 5.19** | Write a program to display the pattern of stars given as follows:

```

* * * * *
* * * *
* * *
* *
*

```

```

print(" Star Pattern Display")
num=7
x=num
for i in range(1,6,1):
 num=num-1;
 for j in range(1,num,1):
 print(" * ",end=" ")
 x=num-1

```

(Contd.)

```
print()
print("End of Program")
```

**Output**

Star Pattern Display

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

End of Program

**PROGRAM 5.20** | Write a program to display the pattern of stars given as follows:

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

```
print(" Star Pattern Display")
num=1
x=num
for i in range(1,6,1):
 num=num+1;
 for j in range(1,num,1):
 print(" * ",end=" ")
 x=num+1
 print()
print("End of Program")
```

**Output**

Star Pattern Display

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

End of Program

**PROGRAM 5.21** | Write a program to display the pattern of numbers given as follows:

```
1
1 2
```

```

1 2 3
1 2 3 4
1 2 3 4 5

```

```

print(" Number Pattern Display")
num=1
x=num
for i in range(1,6,1):
 num=num+1;
 for j in range(1,num,1):
 print(j, end=" ")
 x=num+1
 print()
print("End of Program")

```

### Output

```

Number Pattern Display
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
End of Program

```

**PROGRAM 5.22** | Write a program to display the pattern of numbers given as follows:

```

1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1

```

```

print(" Number Pattern Display")
num=1
x=num
for i in range(1,5,1):
 num=num+1;
 for j in range(1,num,1):
 print(j, end=" ")
 x=num+1
 print()

```

(Contd.)

```

num=5
x=num
for i in range(1,5,1):
 num=num-1;
 for j in range(1,num,1):
 print(j, end=" ")
 x=num-1
 print()

```

### Output

Number Pattern Display

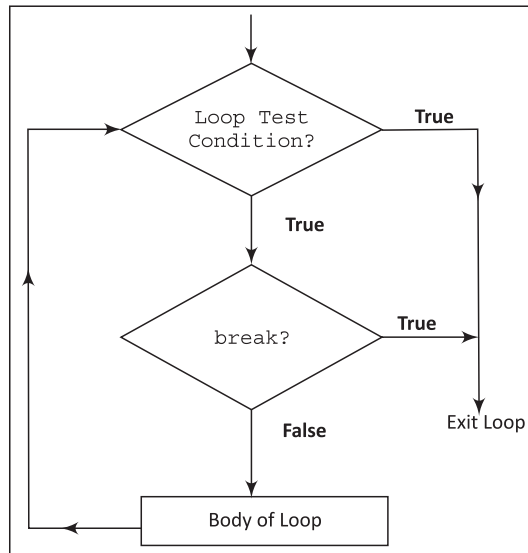
```

1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1

```

## 5.6 THE break STATEMENT

The keyword **break** allows a programmer to terminate a loop. When the break statement is encountered inside a loop, the loop is immediately terminated and the program control automatically goes to the first statement following the loop. The flowchart for break is shown in Figure 5.3.



**Figure 5.3** Flowchart for **break** statement

The working of break in while and for loop is shown as follows:

### Working of break in while loop:

```

while test-Boolean-expression:
 body of while
 if condition:
 break
 body of while
 statement(s)

```

### Working of break in for loop:

```

for var in sequence:
 body of for
 if condition:
 break
 body of for
 statement(s)

```

**PROGRAM 5.23** | Write a program to demonstrate the use of the break statement.

```

print("The Numbers from 1 to 10 are as follows:")
for i in range(1,100,1):
 if(i==11):
 break
 else:
 print(i, end=" ")

```

### Output

```

The Numbers from 1 to 10 are as follows:
1 2 3 4 5 6 7 8 9 10

```

**Explanation** The above program prints the numbers from 0 to 10 on the screen. The loop terminates because 'break' causes immediate exit from the loop.

**PROGRAM 5.24** | Check if the number entered is prime or not.

```

num=int(input("Enter the Number:"))
x=num
for i in range(2,num):
 if num%i==0: #Check if entered number is divisible by i
 flag=0
 break
 else:

```

(Contd.)



```

 flag=1
if(flag==1):
 print(num," is Prime ")
else:
 print(num," is not prime ")

```

### Output

```

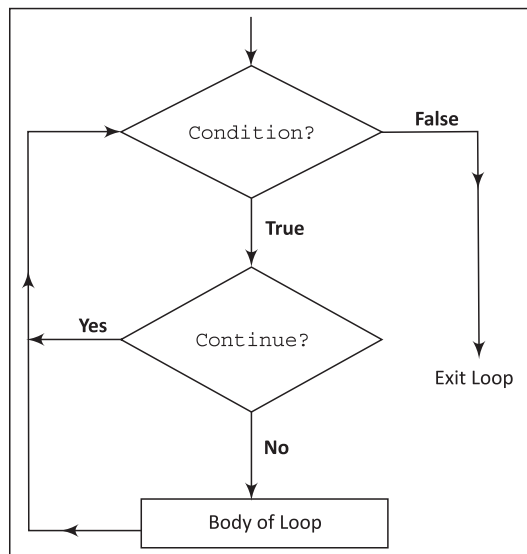
#Test case 1:
Enter the Number:23
23 is Prime
#Test case 2:
Enter the Number:12
12 is not prime

```

**Explanation** The number is read from the user through the keyboard. A prime number should be divisible by 1 and itself. Therefore, the variable 'i' is iterated from 2 to one less than the number entered. Each value of 'i' is used to check if 'i' can divide the number entered.

## 5.7 THE continue STATEMENT

The **continue** statement is exactly opposite of the **break** statement. When **continue** is encountered within a loop, the remaining statements within the body are skipped but the loop condition is checked to see if the loop should continue or exit. Flowchart for continue statements is shown in Figure 5.4.



**Figure 5.4** Flowchart for **continue** statement

The working of **continue** in while loop is shown as follows:

```

➔ while test-boolean-expression:
 body of while
 if condition:
 continue
 body of while
 statement(s)

```

Alternatively, the working of **continue** in for loop is shown as follows:

```

➔ for var in sequence:
 body of for
 if condition:
 continue
 body of for
 statement(s)

```

The difference between **break** and **continue** is given in Table 5.2.

**Table 5.2** Difference between **break** and **continue** functions

| <i>Break</i>                          | <i>Continue</i>                                                                      |
|---------------------------------------|--------------------------------------------------------------------------------------|
| Exits from current block or loop.     | Skips the current iteration and also skips the remaining statements within the body. |
| Control passes to the next statement. | Control passes at the beginning of the loop.                                         |
| Terminates the loop.                  | Never terminates the loop.                                                           |

**PROGRAM 5.25** | Demonstrate the use of **continue** keyword.

```

for i in range(1,11,1):
 if i == 5:
 continue
 print(i, end=" ")

```

**Output**

1 2 3 4 6 7 8 9 10

**Explanation** In each iteration in the above program, the value of the variable ‘i’ is checked. If the value of ‘i’ is 5 then continue statement is executed and the statements following the continue statement are skipped.

**PROGRAM 5.26** | Read the string “Hello World” from the user. Make use of **continue** keyword and remove space.

```

str1=str(input("Please Enter the String: "))
print(" Entered String is : ", str1)
print(" After Removing Spaces, the String becomes:")
for i in str1:
 if i==" ":
 continue
 print(i, end=" ")

```

### Output

```

Please Enter the String: Hello World
Entered String is : Hello World
After Removing Spaces, the String becomes:
HelloWorld

```

**Explanation** The string str1 is read from the user. Each character of entered string is iterated through the variable 'i'. The statement **if i == " "** is used to check if the entered string contains any space. If it contains space, the **continue** statement is executed and the rest of the statements following the **continue** statement are skipped. Finally, we obtain the string without spaces.

## MINI PROJECT

### Generate Prime Numbers using Charles Babbage Function

Charles Babbage discovered the first calculating machine to print prime numbers for a given equation. This mini project will make use of **if**, **if-else**, **if-elif** and for loop concepts of programming.

Let us consider the formula used by Charles Babbage:

$$T = X^2 + X + 41$$

The above formula generates a sequence of values for  $T$ , which happen to be prime numbers. Thus, calculate the sequence of prime numbers  $T$  for the values of  $x$  starting from 0 to 5. The following table contains prime numbers generated by the Charles Babbage function.

**Table 5.3** Evaluation of Charles Babbage function

| D2 | D1 | $T = X^2 + X + 41$ | (Value of X) |
|----|----|--------------------|--------------|
|    |    | 41                 | 0            |
| 2  | 2  | 43                 | 1            |
| 2  | 4  | 47                 | 2            |
| 2  | 6  | 53                 | 3            |
| 2  | 8  | 61                 | 4            |
| 2  | 10 | 71                 | 5            |



---

## SUMMARY

- ♦ Loop is the process of executing a set of statements for fixed number of times.
- ♦ Iteration refers to one time execution of a statements within a loop.
- ♦ Python supports two types of loop control statements, i.e. for loop and while loop.
- ♦ While loop is condition controlled loop.
- ♦ for loop is count controlled loop and it execute statements within the body of loop for fixed number of times.
- ♦ The break and continue keywords can be in the loops.
- ♦ The break statement exits from the current block or loop and control passes to the next statement.
- ♦ The continue statement skips the current iteration and also skips the remaining statements within the body of a loop.

---

## KEY TERMS

- ⇒ **while Loop:** Condition controlled loop
- ⇒ **for Loop:** Count controlled loop
- ⇒ **range():** Generates a list of integers
- ⇒ **nested Loop:** Loop within a loop
- ⇒ **break Statement:** The break statement within a loop helps a programmer to terminate the loop immediately
- ⇒ **continue Statement:** Skips the current iteration and also skips the remaining statement within the body.

---

## REVIEW QUESTIONS

### A. Multiple Choice Questions

1. How many times will a loop with header for count in range(5): execute statements in its body?
  - a. 5 times
  - b. 4 times
  - c. 6 times
  - d. 3 times
2. What will be the output of the following program?

```
count = 35
for x in range(0,10):
 count = count - 1
 if x == 2:
 break
print(count)
```

  - a. 35
  - b. 32
  - c. 35, 34, 33
  - d. 34, 33, 32

3. What will be the output of the following program?

```
Z = 1
while Z<5:
 if Z % 7 == 0:
 break
 Z = Z + 2
print(Z)
```

- |      |      |
|------|------|
| a. 5 | b. 3 |
| c. 4 | d. 2 |

4. What will be the output of the following program?

```
My_str = "I LOVE PHYTHON"
count = 0
for char in my_str:
 if char == "O":
 continue
 else:
 count = count + 1
print(count)
```

- |       |       |
|-------|-------|
| a. 10 | b. 9  |
| c. 11 | d. 12 |

5. What will be the output of the following program?

```
my_str = "I LOVE PYTHON"
count = 0
for char in my_str:
 count = count + 1
 if char == "E":
 break
print(count)
```

- |       |       |
|-------|-------|
| a. 11 | b. 13 |
| c. 10 | d. 12 |

6. What will be the output of the following program?

```
i = 1
for x in range(1,4):
 for y in range(1,3):
 i = i +(i * 1)
print(i)
```

- |       |       |
|-------|-------|
| a. 32 | b. 62 |
| c. 63 | d. 64 |

7. What will be the output of the following program?

```
count = 0
for x in range (1,3):
 for y in range (4,6):
 count = count + (x * y)
print (count)
```

- |       |       |
|-------|-------|
| a. 32 | b. 27 |
| c. 57 | d. 64 |

8. What will be the output of the following program?

```
i = 0
for x in range (1,3):
 j = 0
 for y in range (-2,0):
 j = j + y
 i = i + j
print (i)
```

- |       |                      |
|-------|----------------------|
| a. 10 | b. -10               |
| c. 0  | d. None of the above |

9. By default, while is:

- |                                |                           |
|--------------------------------|---------------------------|
| a. Condition control statement | b. Loop control statement |
| c. Both a and b                | d. None of the above      |

10. What will be the output of the following program?

```
Count = 0
num = 10
while num > 8:
 for y in range(1,5):
 count = count + 1
 num = num - 1
print(count)
```

- |       |       |
|-------|-------|
| a. 10 | b. 8  |
| c. 12 | d. 11 |

## B. True or False

- Python facilitates the use of control statements to change the flow of execution of programs.
- The while loop is not a keyword supported by Python.
- A loop cannot repeatedly execute a block of statements for a specified number of times.
- A loop cannot be nested.
- The continue statement is a keyword.
- The break statement is used to terminate from the loop.
- The break statement is not a keyword.
- The while statement is terminated by a semicolon (;).

9. The meaning of while(1) implies it is true.
10. Indentation does not play a major role for the statements within the body of a loop.

### C. Exercise Questions

1. Give the syntax for control statements supported by Python.
2. Explain the working of the while loop with a flowchart.
3. What happens if we create a loop that never ends?
4. What is meant by nested loops?
5. Find the bugs in the following programs.

```
a. count = 0
 s=0
 while count<10:
 s += count
 count=count+1
 print (s)
```

```
b. count=0
 for i in range(10,0,-1)
 print(i)
```

6. Is it possible to nest the while loop within for loops?
7. When is the break statement used?
8. When is the continue statement used?
9. Convert the following for loop into while loop.

```
for i in range(50,0,-2):
 print(i,end=' ')
```

10. Answer the following questions.

- a. How many times will the following loop execute and what will be its output for both the programs, a and b?

```
sum=0
for i in range(20,0,-2):
 sum=sum+i
 print(i)
 if i==14:
 continue
print (sum)
```

(a)

```
sum=0
for i in range(20,0,-2):
 sum=sum+i
 print(i)
 if i==14:
 break
print (sum)
```

(b)

11. Convert the following while loop into for loop

```
i=0
s=0
while i<=50:
 if i%7==0:
 s = s+i
 i = i+7
print (s)
```



## PROGRAMMING ASSIGNMENTS

1. Write a program that asks for input  $n$  and prints a sequence of powers of 5 from  $5^0$  to  $5^n$  in separate lines.

**Note:** The input number  $n$  should be positive.

**Example:**

**Input:**  $N=4$

**Output:** 1

5

25

125

625

2. Write a program to display the following table.

| <i>Kilogram</i> | <i>Gram</i> |
|-----------------|-------------|
| 1               | 1000        |
| 2               | 2000        |
| 3               | 3000        |

**Note:** 1 kilogram = 1000 grams

3. Write a program to display the numbers of a series 1, 4, 9, 16, 25, ...,  $n$  by using **for** loop.
4. Write a program using the while loop, which prints the sum of every fifth number from 0 to 500 (including both 0 and 500).
5. Write a program using the while loop to read the positive integer and count the number of decimal digits in a positive integer.
6. Write a program to read the password from a user. If the user types the correct password, i.e. "Python" then display the message, "Welcome to Python Programming".
7. Write programs for the following series using the while loop.

a.  $x + x^2/2! + x^3/3! + \dots + x^n$

b.  $1 + x + x^2 + x^3 + \dots + x^n$

8. Consider a scenario where a son eats five chocolates every day. The price of each chocolate is different. His father pays the bill to the chocolate vendor at the end of every week.

Develop a program that can generate the bills for the chocolates and send to the father. Also state which loop will be used to solve this problem.

# UNIT 3

---

- Lists

- Tuples, Sets and Dictionaries

## CHAPTER OUTLINE

|                                        |                                     |
|----------------------------------------|-------------------------------------|
| 6.1 Introduction                       | 6.8 The List Operator               |
| 6.2 Creating Lists                     | 6.9 List Comprehensions             |
| 6.3 Accessing the Elements of a List   | 6.10 List Methods                   |
| 6.4 Negative List Indices              | 6.11 List and Strings               |
| 6.5 List Slicing [Start: end]          | 6.12 Splitting a String in List     |
| 6.6 List Slicing with Step Size        | 6.13 Passing List to a Function     |
| 6.7 Python Inbuilt Functions for Lists | 6.14 Returning List from a Function |

## LEARNING OUTCOMES

*After completing this chapter, students will be able to:*

- Explain the necessity and importance of 'list' in programming languages
- Create a list of different and mixed types
- Write programs to access the elements of a list using the positive index operator and the negative index operator
- Explain list slicing with different features and programs
- Use various operators, such as +, \* and in operators on lists
- Create a new list from an existing list, learn to pass a list to a function and write programs to return lists from a function

## 6.1 INTRODUCTION

We may need to store variables of the same data type on many occasions. For example, currency notes used in daily life in India are of denominations ₹5, 10, 20, 100, 500 and 2000. If a programmer wishes to display all the six currency notes then by regular programming methods he/she may print them by reading all the currency notes with six different variables. Through a list, however, a programmer can use a single variable to store all the elements of the same or different data type and even print them. Similarly, miscellaneous lists to display top 100 countries in the world, students qualifying GRE exams, buying groceries etc. can be created.

In Python, a list is a sequence of values called **items** or **elements**. The elements can be of any type. The structure of a list is similar to the structure of a string.

## 6.2 CREATING LISTS

The List class defines lists. A programmer can use a list's constructor to create a list. Consider the following example.

**Example: Create a list using the constructor of the list class**

- a. Create an empty list.

```
L1 = list();
```

- b. Create a list with any three integer elements, such as 10, 20 and 30.

```
L2 = list([10,20,30])
```

- c. Create a list with three string elements, such as "Apple", "Banana" and "Grapes".

```
L3 = list(["Apple","Banana","Grapes"])
```

- d. Create a list using inbuilt **range()** function.

```
L4 = list(range(0,6)) # create a list with elements from 0 to 5
```

- e. Create a list with inbuilt characters X, Y and Z.

```
L5=list("xyz")
```

**Example: Creating a list without using the constructor of the list class**

- a. Create a list with any three integer elements, such as 10, 20 and 30.

```
L1=[10,20,30]
```

- b. Create a list with three string elements, such as "Apple", "Banana" and "Grapes".

```
L2 = ["Apple", "Banana", "Grapes"]
```



**Note:** A list can contain the elements of mixed type.

**Example:**

```
L3=list(["Jhon","Male",25,5.8])
```

The above example creates a list L3, which is of mixed type, i.e. it contains elements of different types, such as string, float and integer.

## 6.3 ACCESSING THE ELEMENTS OF A LIST

The elements of a list are unidentifiable by their positions. Hence, the **index** `[]` operator is used to access them. The syntax is:

```
Name_of_Variable_of_a_List[index]
```

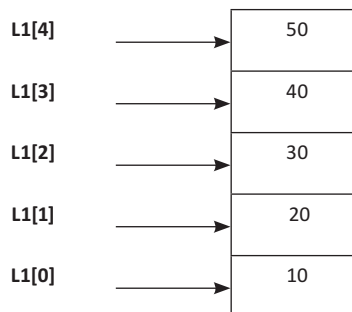
### Example

```
>>> L1=([10,20,30,40,50]) #Create a List with 5 Different Elements
>>> L1 #print the complete List
[10, 20, 30, 40, 50]
>>> L1[0] # Print the first element of the List
10
```

**Explanation** The above example L1 creates a list of five elements

```
L1 = [10,20,30,40,50]
```

where L1 is the reference variable of the list.



**Figure 6.1** The list has five elements with index from 0 to 4



**Note:** A list retains its original order. Therefore, a list is an ordered set of elements enclosed in square brackets separated by commas. The index of a non-empty list always starts from zero.

## 6.4 NEGATIVE LIST INDICES

The negative index accesses the elements from the end of a list counting in backward direction. The **index** of the last element of any non-empty list is always **-1**, as shown in Figure 6.2.

|         |    |    |    |    |    |    |
|---------|----|----|----|----|----|----|
| List1 = | 10 | 20 | 30 | 40 | 50 | 60 |
|         | -6 | -5 | -4 | -3 | -2 | -1 |

**Figure 6.2** List with negative index

Accessing the elements of a list using a negative index.

**Example**

```
>>> List1=[10,20,30,40,50,60]#Create a List
>>> List1[-1] #Access Last element of a List
60
>>>List1[-2] #Access the second last element of List
50
>>> List1[-3] #Access the Third last element of List
40
>>>List1[-6] #Access the first Element of the List
10
```

**Note:**

```
List[-n] == List[Length_of(List)-n]
```

**Example:**

```
>>>List1=[10,20,30,40,50,60]
>>>List1[-3]
40
```

**Explanation:**

```
List1[-3]==List1[Len(List1)-3]=List1[6-3]=List1[3] .
```

Thus, List1[-3]==List1[3] prints the element stored at index 3 counting in a forward direction from the list or we can say it prints the element stored at index -3 counting in a backward direction from the list.

**6.5 LIST SLICING [START: END]**

The slicing operator returns a subset of a list called **slice** by specifying two indices, i.e. **start** and **end**. The syntax is:

```
Name_of_Variable_of_a_List[Start_Index: End_Index]
```

**Example**

```
>>> L1=([10,20,30,40,50]) #Create a List with 5 Different Elements
>>> L1[1:4]
20,30,40
```

The L1[1:4] returns the subset of the list starting from index the start index 1 to one index less than that of the end index, i.e.  $4-1 = 3$ .

**Example**

```
>>> L1=([10,20,30,40,50]) #Create a List with 5 Different Elements
>>> L1[2:5]
[30, 40, 50]
```

The above example L1 creates a list of five elements. The index operator L1[2:5] returns all the elements stored between the index 2 and one less than the end index, i.e.  $5-1 = 4$ .

## 6.6 LIST SLICING WITH STEP SIZE

So far, we learnt how to select a portion of a list. In this section, we will explore how to select every second or third element of a list using **step size**. In slicing, the first two parameters are **start index** and **end index**. Thus, we need to add a third parameter as **step size** to select a list with step size. To be able to do this we use the syntax:

```
List_Name[Start_Index:End_Index:Step_Size]
```

### Example

```
>>> MyList1 = ["Hello", 1, "Monkey", 2, "Dog", 3, "Donkey"]
>>> New_List1 = MyList1[0:6:2]
print(New_List1)
['Hello', 'Monkey', 'Dog']
```

**Explanation** Initially we created a list named **MyList1** with five elements. The statement **MyList1[0:6:2]** indicates the programmer to select the portion of a list which starts at index 0 and ends at index 6 with the step size as 2. It means we first extract a section or slice of the list which starts at the index 0 and ends at the index 6 and then selects every other second element.

### Example

```
>>> List1 = ["Python", 450, "C", 300, "C++", 670]
>>> List1[0:6:3] #Start from Zero and Select every Third Element
['Python', 300] #Output
```

### 6.6.1 Some More Complex Examples of List Slicing

```
>>> List1 = [1, 2, 3, 4] #List With Four Elements

>>> MyList1[:2] #Access first two elements of the List.
[1, 2]

>>> MyList1[::-1] #Display List in Reverse Order
[4, 3, 2, 1]

#Start index with -1 and End Index with 0 and step Size with -1
>>> MyList1[-1:0:-1]
[4, 3, 2]
```

## 6.7 PYTHON INBUILT FUNCTIONS FOR LISTS

Python has various inbuilt functions that can be used with lists. Some of these are listed in Table 6.1.

**Table 6.1** Inbuilt functions that can be used with lists

| <i>Inbuilt Functions</i> | <i>Meaning</i>                               |
|--------------------------|----------------------------------------------|
| <b>Len()</b>             | Returns the number of elements in a list.    |
| <b>Max()</b>             | Returns the element with the greatest value. |
| <b>Min()</b>             | Returns the element with the lowest value.   |
| <b>Sum()</b>             | Returns the sum of all the elements.         |
| <b>random.shuffle()</b>  | Shuffles the elements randomly.              |

### Example

**#Creates a List to store the names of Colors and return size of list.**

```
>>> List1=["Red","Orange","Pink","Green"]
>>> List1
['Red', 'Orange', 'Pink', 'Green']
>>> len(List1) #Returns the Size of List
4
```

**#Create a List, find the Greatest and Minimum value from the list.**

```
>>> List2=[10,20,30,50,60]
>>> List2
[10, 20, 30, 50, 60]
>>> max(List2) #Returns the greatest element from the list.
60
>>> min(List2) #Returns the minimum element from the list.
10
```

**#Create a List, and Shuffle the elements in random manner.**

**#Test Case 1**

```
>>>import random
>>> random.shuffle(List2)
>>> List2
[30, 10, 20, 50, 60]
>>> List2
[30, 10, 20, 50, 60]
```

**#Test Case2**

```
>>> random.shuffle(List2)
>>> List2
[20, 10, 30, 50, 60]
```



```
..
#Create a List, and find the sum of all the elements of a List.
>>> List2=[10,20,30,50,60]
>>> List2
[10, 20, 30, 50, 60]
>>> sum(List2) # Returns the sum of all the elements
170
```

## 6.8 THE LIST OPERATOR

1. **The + Operator:** The concatenation operator is used to join two lists.

### Example

```
>>> a=[1,2,3] #Create a list with three elements 1,2, and 3
>>> a #Prints the list
[1, 2, 3]
>>> b=[4,5,6] #Create a list with three elements 4,5, and 6
>>> b #print the list
[4, 5, 6]
>>> a+b #Concatenate the list a and b
[1, 2, 3, 4, 5, 6]
```

2. **The \* Operator:** The multiplication operator is used to replicate the elements of a list.

### Example

```
>>> List1=[10,20]
>>> List1
[10, 20]
>>> List2=[20,30]
>>> List2
[20, 30]
>>> List3=2*List1 #Print each element of a List1 twice.
>>> List3
[10, 20, 10, 20]
```

3. **The in Operator:** The in operator used to determine whether an element is in a list. It returns True if the element is present and False if the element is absent in the list.

### Example

```
>>> List1= [10,20]
>>> List1
[10, 20]
>>> 40 in List1 #To Check if 40 is present in List1
False
>>> 10 in List1 #To Check if 10 is present in List1
True
```

4. **The isOperator:** Let us execute the following two statements:  
A='Python'  
B='Python'

We know that both A and B refer to a string but we don't know whether they refer to the same string or not. Two possible situations are:



In the first case, A and B refer to two different objects that have the same values. In second case, they refer to the same object. To understand whether two variables refer to the same object, a programmer can use the 'is' operator.

**Example**

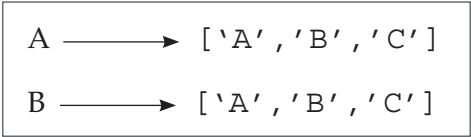
```
>>> A='Microsoft'
>>> B='Microsoft'
>>> A is B #Check if two variable refer to the same Object
True
```

From the above example, it is clear that Python created only one string object and both A and B refer to the same object. However, when we create two lists with the same elements, Python creates two different objects as well.

**Example**

```
>>> A=['A','B','C']
>>> B=['A','B','C']
>>> A is B #Check if two lists refer to the same Object
False
```

**Explanation** In the above example, the two lists A and B contain exactly the same number of elements. The is operator is used to check if both the variables A and B refer to the same object, but it returns False. It means that even if the two lists are the same, Python creates two different objects. State diagram for the above example is given in Figure 6.3.



**Figure 6.3** Effect of is operator on a list

It is important to note that in the above example, we can say that the two lists are **equivalent** because they have the same elements. We cannot say that both the lists are **identical** because they don't refer to the same object.

**Note:**

1. In case of strings, if both the variables contain the same values then both the variables refer to the same object.
2. In case of a list, if two variables contain the list with the same number of elements then both the variables refer to two different objects.
3. If two objects are **identical** then they are also **equivalent**.
4. If two objects are **equivalent** then it is not necessary that they will also be **identical**.

5. **The del Operator:** The del operator stands for Delete. The del operator is used to remove the elements from a list. To delete the element of a list, the elements of the list are accessed using their index position and the del operator is placed before them.

**Example**

```
Lst=[10,20,30,40,50,60,70]
>>> del Lst[2] #Removes 3rd element from the List
>>> Lst
[10, 20, 40, 50, 60, 70]

Lst=[10,20,30,40,50,60,70]
>>> del Lst[-1]
>>> Lst #Removes last element from the List
[10, 20, 30, 40, 50, 60]

>>> Lst=[10,20,30,40,50,60,70]
>>> del Lst[2:5] #Removes element from index position 2 to 4
>>> Lst
[10, 20, 60, 70]

>>> Lst=[10,20,30,40,50,60,70]
>>> del Lst[:] #Removes all the element from the List
>>> Lst
[]
```



**Note:** The del operator uses index to access the elements of a list. It gives a run time error if the index is out of range.

Example:

```
>>> del Lst[4]
```

Traceback (most recent call last):

```
File "<pyshell#37>", line 1, in <module>
 del Lst[4]
```

IndexError: list assignment index out of range

## 6.9 LIST COMPREHENSIONS

List comprehension is used to create a new list from existing sequences. It is a tool for transforming a given list into another list.

### **Example: Without list comprehension**

Create a list to store five different numbers such as 10, 20, 30, 40 and 50. Using the for loop, add number 5 to the existing elements of the list.

```
>>> List1= [10, 20, 30, 40, 50]
>>> List1
[10, 20, 30, 40, 50]
>>> for i in range(0,len(List1)):
 List1[i]=List1[i]+5 #Add 5 to each element of List1
>>> List1 #print the List1 After Performing
[15, 25, 35, 45, 55]
```

The above code is workable but **not** the **optimal code** or the best way to write a code in Python. Using list comprehension, we can replace the loop with a single expression that produces the same result.

The syntax of list comprehension is based on set builder notation in mathematics. Set builder notation is a mathematical notation for describing a set by stating the property that its members should satisfy. The syntax is

```
[<expression> for <element> in <sequence> if <conditional>]
```

The syntax is designed to read as “Compute the expression for each element in the sequence, if the conditional is true”.

### **Example: Using list comprehension**

```
>>> List1= [10, 20, 30, 40, 50]
>>> List1
[10, 20, 30, 40, 50]

>>>for i in range(0,len(List1)):
 List1[i]=List1[i]+10

>>>List1
[20, 30, 40, 50, 60]
```

**Without List Comprehension**

```
>>> List1= [10,20,30,40,50]
>>> List1= [x+10 for x in List1]
>>> List1
[20, 30, 40, 50, 60]
```

**Using List Comprehension**

In the above example, the output for both without list comprehension and using list comprehension is the same. The use of list comprehension requires **lesser code** and also runs faster. With reference to the above example we can say that list comprehension contains:

- a. An input sequence

- b. A variable referencing the input sequence
- c. An optional expression
- d. An output expression or output variable

### Example

```
List1= [20, 30, 40, 50, 60]
List1= [x+10 for x in List1]
```

(An output variable)

(A variable referencing an input sequence)

(An input sequence)

**Output** [20, 30, 40, 50, 60]

**PROGRAM 6.1** | Write a program to create a list with elements 1,2,3,4 and 5. Display even elements of the list using list comprehension.

```
List1=[1,2,3,4,5]
print("Content of List1")
print(List1)
List1=[x for x in List1 if x%2==0]
print("Even elements from the List1")
print(List1)
```

### Output

```
Content of List1
[1, 2, 3, 4, 5]
Even elements from the List1
[2, 4]
```

**Explanation** The List1 contains element 1,2,3,4 and 5. The statement **List1=[x for x in List1 if x%2==0]** consists of an output variable x and an input sequence List1 and an expression x%2==0.

## 6.9.1 Some More Examples of List Comprehension

**PROGRAM 6.2** | Consider the following mathematical expressions

```
A = {x : x in {0.....9}}
B = {x : x in {0.....9}}
C = {X : x in A and even}
```

Write a program to create a list **'A'** to generate squares of a number (from 1 to 10), list **'B'** to generate cubes of a number (from 1 to 10) and list **'C'** with those elements that are even and present in list **'A'**.

```
print("List A = ",end=" ")
A=[x**2 for x in range(11)] #Computes Square of a number X
print(A)
B=[x**3 for x in range(11)] # Computes Cube of a number X
print("List B = ",end=" ")
print(B)
print("Only Even Numbers from List A = ",end=" ")
C=[x for x in A if x%2==0]
print(C)
```

### Output

```
List A = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
List B = [0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
Only Even Numbers from List A = [0, 4, 16, 36, 64, 100]
```

**PROGRAM 6.3** | Consider a list with five different Celsius values. Convert all the Celsius values into Fahrenheit.

```
print("All the elements with Celsius Value:")
print("Celsius= ",end="")
Celsius=[10,20,31.3,40,39.2] #List with Celsius Value
print(Celsius)
print(" Celsius to Fahrenheit Conversion ")
print("Fahrenheit = ",end="")
Fahrenheit=[((float(9)/5)*x + 32) for x in Celsius]
print(Fahrenheit)
```

### Output

```
All the elements with Celsius Value:
Celsius= [10, 20, 31.3, 40, 39.2]
 Celsius to Fahrenheit Conversion
Fahrenheit = [50.0, 68.0, 88.34, 104.0, 102.56]
```



**Note:** Formula to convert Celsius values into Fahrenheit.  
Fahrenheit = (9/5) \* Celsius + 32

**PROGRAM 6.4** | Consider the list with mixed type of elements, such as L1 = [1,'x',4,5.6,'z', 9, 'a', 0, 4]. Create another list using list comprehension which consists of only the integer element present within the list L1.

```
print("List With Mixed Elements")
L1 = [1,'x' ,4,5.6,'z' , 9, 'a', 0, 4]
print(L1)
print("List With only Integer Elements:")
L2 = [e for e in L1 if type(e) == int]
print(L2)
```

### Output

```
List With Mixed Elements
[1, 'x', 4, 5.6, 'z', 9, 'a', 0, 4]
List With only Integer Elements:
[1, 4, 9, 0, 4]
```

## 6.10 LIST METHODS

Once a list is created, we can use the methods of the list class to manipulate the list. Table 6.2 contains the methods of the list class along with examples.

**Table 6.2** Methods of list along with example

| Methods of List                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Meaning                                                                                      |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <p><b>None</b> <b>append(object x)</b></p> <p><b>Example:</b></p> <pre>&gt;&gt;&gt; List1=['X','Y','Z'] &gt;&gt;&gt; List1 ['X', 'Y', 'Z'] &gt;&gt;&gt; List1.append('A') <b>#Append element 'A' to the end of the List1</b> &gt;&gt;&gt; List1 ['X', 'Y', 'Z', 'A']</pre> <p><b>Note:</b> Append method is equivalent to doing:</p> <pre>List1[len(List1):]=[Element_Name]</pre> <p><b>Example:</b></p> <pre>&gt;&gt;&gt;List1=["Red","Blue","Pink"] &gt;&gt;&gt; List1 ['Red', 'Blue', 'Pink'] &gt;&gt;&gt; List1[len(List1):]=['Yellow'] &gt;&gt;&gt; List1 ['Red', 'Blue', 'Pink', 'Yellow']</pre> | <p>Adds an element x to the end of the list. None is the return type of method appended.</p> |

(Contd.)

|                                                                                                                                                                                                                                                                                                                                                                                    |                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|
| <b>None clear()</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; List1=["Red", "Blue", "Pink"] &gt;&gt;&gt; List1 ['Red', 'Blue', 'Pink'] &gt;&gt;&gt; List1.clear() # Removes all the element of List &gt;&gt;&gt; List1          # Returns Empty List after removing all elements []</pre>                                                                                            | Removes all the items from the list.                           |
| <b>int count(object x)</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; List1=['A','B','C','A','B','Z'] &gt;&gt;&gt; List1 ['A', 'B', 'C', 'A', 'B', 'Z'] #Count the number of times the element 'A' has appeared in the list &gt;&gt;&gt; List1.count('A') 2 # Thus, 'A' has appeared 2 times in List1</pre>                                                                           | Returns the number of times the element x appears in the list. |
| <b>List copy()</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; List1=["Red","Blue","Pink"] &gt;&gt;&gt; List1 ['Red', 'Blue', 'Pink'] &gt;&gt;&gt; List2=List1.copy() # Copy the contents of List1 to List2 &gt;&gt;&gt; List2 ['Red', 'Blue', 'Pink']</pre> <p><b>Note:</b> Copy() Method is equivalent to doing</p> <pre>List2=List1[:] # Copies the content of List1 to List2</pre> | This method returns a shallow copy of the list.                |
| <b>Example:</b><br><pre>&gt;&gt;&gt; List1=["Red","Blue","Pink"] &gt;&gt;&gt; List2=List1[:] &gt;&gt;&gt; List2 ['Red', 'Blue', 'Pink']</pre>                                                                                                                                                                                                                                      |                                                                |
| <b>None extend(list L2)</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; List1= [1,2,3] &gt;&gt;&gt; List2= [4,5,6] &gt;&gt;&gt; List1 [1, 2, 3]</pre>                                                                                                                                                                                                                                  | Appends all the elements of list L2 to the list.               |

(Contd.)



```
>>> List2
[4, 5, 6]
>>> List1.extend(List2) #Appends all the elements
of List2 to List1
>>> List1
[1, 2, 3, 4, 5, 6]
```

**int index(object x)**

**Example:**

```
>>> List1=['A','B','C','B','D','A']
```

```
>>> List1
```

```
['A', 'B', 'C', 'B', 'D', 'A']
```

**#Returns the index of first occurrence of element 'B' from the list1**

```
>>> List1.index('B')
```

```
1 #Returns the index of element B
```

Returns the index of the first occurrence of the element *x* from the list.

**None insert(int index, Object X)**

**Example:**

```
>>> Lis1=[10,20,30,40,60]
```

```
>>> Lis1
```

```
[10, 20, 30, 40, 60]
```

```
>>> Lis1.insert(4,50) #Insert Element 50 at index 4
```

```
>>> Lis1
```

```
[10, 20, 30, 40, 50, 60]
```

Insert the element at a given index.

**Note:** The index of the first element of a list is always zero.

**Object pop(i)**

**Example:**

```
>>> Lis1=[10,20,30,40,60]
```

```
>>> Lis1
```

```
[10, 20, 30, 40, 60]
```

```
>>> Lis1.pop(1) # Remove the element which is at
index 1.
```

```
20
```

```
>>> Lis1 # Display List after removing the
element from index 1.
```

```
[10, 30, 40, 60]
```

```
>>> Lis1.pop() # Remove the last element from the
list
```

```
60
```

```
>>> Lis1
```

```
[10, 30, 40] #Display the list after removing last
element
```

Removes the element from the given position. Also, it returns the removed element.

**Note:** The parameter *i* is optional. If it is not specified then it removes the last element from the list.

(Contd.)

|                                                                                                                                                                                                                                                                                                   |                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| <p><b>None remove(object x)</b></p> <p><b>Example:</b></p> <pre>&gt;&gt;&gt; List1=['A','B','C','B','D','E'] &gt;&gt;&gt; List1 ['A', 'B', 'C', 'B', 'D', 'E'] &gt;&gt;&gt; List1.remove('B')<b>#Removes the first occurrence of element B</b> &gt;&gt;&gt; List1 ['A', 'C', 'B', 'D', 'E']</pre> | Removes the first occurrence of element x from the list. |
| <p><b>None reverse()</b></p> <p><b>Example:</b></p> <pre>&gt;&gt;&gt; List1=['A','B','C','B','D','E'] &gt;&gt;&gt; List1 ['A', 'B', 'C', 'B', 'D', 'E'] &gt;&gt;&gt; List1.reverse() <b># Reverse all the elements of the list.</b> &gt;&gt;&gt; List1 ['E', 'D', 'B', 'C', 'B', 'A']</pre>       | Reverses the element of the list.                        |
| <p><b>None sort()</b></p> <p><b>Example:</b></p> <pre>&gt;&gt;&gt; List1=['G','F','A','C','B'] &gt;&gt;&gt; List1 ['G', 'F', 'A', 'C', 'B']      <b>#Unsorted List</b> &gt;&gt;&gt; List1.sort() &gt;&gt;&gt; List1                      <b>#Sorted List</b> ['A', 'B', 'C', 'F', 'G']</pre>      | Sort the elements of list.                               |

**Q. What will be the output of the following program?**

```
my_list = ['two', 5, ['one', 2]]
print(len(my_list))
```

**Output**

3

**Explanation** ['one',2] is one element so the overall length is 3.

**Q. What will be the output of the following program?**

```
Mixed_List=['pet' , 'dog' ,5, 'cat', 'good','dog']
Mixed_List.count('dog')
```

**Output**

2

**Explanation** It returns the number of occurrence of “dog” in the list.

**Q. What will be the output of the following program?**

```
Mylist=['Red',3]
Mylist.extend('Green')
print(Mylist)
```

**Output**

```
['Red', 3, 'G', 'r', 'e', 'e', 'n']
```

**Explanation** Extend the list by adding each character to it.

**Q. What will be the output of the following program?**

```
Mylist=[3,'Roses',2,'Chocolate ']
Mylist.remove(3)
Mylist
```

**Output**

```
['Roses', 2, 'Chocolate']
```

**Explanation** Remove the item from the list whose value is 3.

## 6.11 LIST AND STRINGS

A String is a sequence of characters and list is sequence of values, but a list of characters is not the same as string. To convert from string to a list of character, you can use list.

**Example: Convert String to list of Characters**

```
>>> p='Python'
>>> p
'Python'
>>> L=list(p)
>>> L
['p', 'y', 't', 'h', 'o', 'n']
```

## 6.12 SPLITTING A STRING IN LIST

In the above example, we have used the inbuilt function list. The list() function breaks a string into individual letters. In this section, we will explore how to split a string into words.

The **str** class contains the **split** method and is used to split a string into words.

**Example**

```
>>> A="Wow!!! I Love Python Programming" #A Complete String
>>> B=A.split() # Split a String into Words
>>> B #Print the contents of B
['Wow!!!', 'I', 'Love', 'Python', 'Programming']
```

**Explanation** In the above example, we have initialised string to A as “Wow!!! I Love Python Programming”. In the next line, the statement, B = A.split() is used to split “Wow!!! I Love Python Programming” into the list ['Wow!!!', 'I', 'Love', 'Python', 'Programming'].



**Note:** In the above program, we have used the following two lines to split string into words:

```
>>> A="Wow!!! I Love Python Programming"
>>> B=A.split()
```

We can also write the split method as

```
>>> A="Wow!!! I Love Python Programming".split()
```

It is fine to split a string without a delimiter. But what if the string contains the delimiter? A string containing a delimiter can be split into words by removing the delimiter. It is also possible to remove the delimiter from the string and convert the entire string into a list of words. In order to remove the delimiter, the **split()** method has a parameter called **split(delimiter)**. The parameter **delimiter** specifies the character to be removed from the string. The following example illustrates the use of a delimiter inside the split() method.

**Example**

```
>>> P="My-Data-of-Birth-03-June-1991" # String with Delimiter '-'
>>> P # Print the Entire String
'My-Data-of-Birth-03-June-1991'
>>> P.split('-') #Remove the delimiter '-' using split method.
['My', 'Data', 'of', 'Birth', '03', 'June', '1991']
```

**6.13 PASSING LIST TO A FUNCTION**

As list is a mutable object. A programmer can pass a list to a function and can perform various operations on it. We can change the contents of a list after passing it to a function. Since a list is an object, passing a list to a function is just like passing an object to a function.

Consider the following example to print the contents of a list after the list is passed to a function.

**PROGRAM 6.5** | Create a list of 5 elements. Pass the list to a function and print the contents of the list inside the function.

```
def Print_List(Lst):
 for num in Lst:
```

(Contd.)

```

 print(num,end=" ")
Lst=[10,20,30,40,100]
Print_List(Lst) # Invoke Function by Passing List as Parameter

```

**Output**

```
10 20 30 40 100
```

**PROGRAM 6.6** | Create a list of five elements. Pass the list to a function and compute the average of five numbers.

```

def Calculate_Avg(Lst):
 print('\tLst= ',Lst)
 print('\tSum = ',sum(Lst))
 avg=sum(Lst)/len(Lst)
 print('\tAverage = ',avg)
Lst=[10,20,30,40,3]
Calculate_Avg(Lst)

```

**Output**

```

Lst= [10, 20, 30, 40, 3]
Sum = 103
Average = 20.6

```

**PROGRAM 6.7** | Write a function Split\_List(Lst,n), where list Lst is split into two parts and the length of the first part is given as  $n$ .

```

Lst = [1,2,3,4,5,6]
Split_List(Lst,2)
Lst1=[1,2]
Lst2=[3,4,5,6]

```

```

def Split_List(Lst,n) :
 list1 = Lst[:n]
 list2 = Lst[n:]
 print('\tFirst List with ',n,' elements')
 print(list1)
 print('\tSecond List with ',len(Lst)-n,' elements ')
 print(list2)
#Sample test:
Lst = [100,22,32,43,51,64]
print('\tList Lst Before Splitting')

```

(Contd.)

```
print(Lst)
Split_List(Lst,4)
```

### Output

```
List Lst Before Splitting
[100, 22, 32, 43, 51, 64]
First List with 4 elements
[100, 22, 32, 43]
Second List with 2 elements
[51, 64]
```

## 6.14 RETURNING LIST FROM A FUNCTION

We can pass a list while invoking a function. Similarly, a function can return a list. When a function returns a list, the list's reference value is returned.

Consider the following example to pass a list to a function. After passing, reverse the elements of the list and return the list.

**PROGRAM 6.8** | Write a program to pass a list to function

```
def Reverse_List(Lst):
 print('List Before Reversing = ',Lst)
 Lst.reverse() # The reverse() to reverse the contents of list
 return Lst # Return List
Lst=[10,20,30,40,3]
print('List after Reversing = ',Reverse_List(Lst))
```

### Output

```
List Before Reversing = [10, 20, 30, 40, 3]
List after Reversing = [3, 40, 30, 20, 10]
```

**PROGRAM 6.9** | Write a function that accepts a positive integer k and returns a list that contains the first five multiples of k.

The first five multiples of 3 are 3, 6, 9, 12, and 15.

```
def list_of_multiples(k):
 my_list=[]
 for i in range(1,6):
 res=k*i
 my_list.append(res)
```

(Contd.)

```

 return my_list
print(list_of_multiples(3))

```

**Output**

```
[3, 6, 9, 12, 15]
```

**Some More Programs on List**

**PROGRAM 6.10** | Write a function that accepts two positive integers, viz. a and b and returns a list of all the even numbers between a and b (including a and not including b).

Even numbers between 10 and 20.

```
[10,12,14,16,18]
```

```

def list_of_even_numbers(start, end):
 output_list = []
 for number in range(start, end):
 # check if the number is even
 if number % 2 == 0:
 # if true put the numbers in the output list
 output_list.append(number)
 return output_list
print(list_of_even_numbers(10, 20))

```

**Output**

```
[10, 12, 14, 16, 18]
```

**PROGRAM 6.11** | Write a function is\_Lst\_Palindrome(Lst) to check whether a list is palindrome. It should return True if Lst is palindrome and False if Lst is not palindrome.



**Note:** List is palindrome if it contains the same elements in forward direction & reverse direction.

**Lst = [1,2,3,2,1]** #Should return true

**Lst = [1,2,3]** #Should return false.

```

def is_Lst_Palindrome(Lst):
 r = Lst[::-1]
 for i in range (0, (len(Lst) + 1)//2):
 if r[i] != Lst[i]:
 return False
 return True

```

(Contd.)

```
#Sample test
Lst = [1,2,3,2,1]
x = is_Lst_Palindrome(Lst)
print(Lst,"(is palindrome): ",x)
Lst1 = [1,2,3,4]
x = is_Lst_Palindrome(Lst1)
print(Lst1,"(is palindrome): ",x)
```

**Output**

```
[1, 2, 3, 2, 1] (is palindrome): True
[1, 2, 3, 4] (is palindrome): False
```

**PROGRAM 6.12** | Write a function `check_duplicate(Lst)` which returns True if a list `Lst` contains duplicate elements. It should return False, if all the elements in the list `Lst` are unique.

```
Lst = [4,6,2,1,6,7,4]
Should return true as 4 and 6 appears more than once.
Lst = [1,2,3,12,4]
Should return false as all the elements appears only once.
```

```
def check_duplicate(Lst) :
 dup_Lst = []
 for i in Lst:
 if i not in dup_Lst:
 dup_Lst.append(i)
 else:
 return True
 return False

#Sample test:
Lst = [4,6,2,1,6,7,4]
print(Lst)
x = check_duplicate(Lst)
print(x)
Lst1 = [1,2,3,12,4]
print(Lst1)
x = check_duplicate(Lst1)
print(x)
```

**Output**

```
[4, 6, 2, 1, 6, 7, 4]
True # Returns true since 4 and 6 is repeated twice
```

(Contd.)



```
[1, 2, 3, 12, 4]
False
#Returns false since no element from above list is repeated twice
```

**PROGRAM 6.13** | Write a program that prompts a user to enter the element of a list and add the element to a list. Write a function **maximum(Lst)** and **minimum(Lst)** to find the maximum and minimum number from the list.

```
Lst = [12,34,45,77]
```

#Should return 12 as Minimum and 77 as Maximum.

```
lst = []
for i in range(0,4):
 x = input('Enter element to add to the list:')
 x = int(x)
 lst.append(x)
print('Elements of List are as follows:')
print(lst)
def maximum(lst):
 myMax = lst[0]
 for num in lst:
 if myMax < num:
 myMax = num
 return myMax

def minimum(lst):
 myMin = lst[0]
 for num in lst:
 if myMin > num:
 myMin = num
 return myMin

#Sample test
y = maximum(lst)
print('Maximum Element from List = ',y)
y = minimum(lst)
print('Minimum Element from the List = ',y)
```

### Output

```
Enter element to add to the list:665
Enter element to add to the list:234
```

(Contd.)

```

Enter element to add to the list:213
Enter element to add to the list:908
Elements of List are as follows:
[665, 234, 213, 908]
Maximum Element from List = 908
Minimum Element from the List = 213

```

**PROGRAM 6.14** | Write a function **Assign\_grade(Lst)** which reads the marks of a student from a list and assigns a grade based on the following conditions:

```

if Marks >=90 then grade A
if Marks >=80 && <90 then grade B
if Marks >65 && < 80 then grade C
if Marks > =40 && <=65 then grade D
if Marks <40 then grade F

```

Consider the List of Marks of a 5 Student in English Subject.

```
Lst=[78,90,34,56,89]
```

#### #Should return

```

Student 1 Marks 78 grade C
Student 2 Marks 90 grade A
Student 3 Marks 34 grade F
Student 4 Marks 56 grade D
Student 5 Marks 89 grade B

```

```

def Assign_grade(Lst):
 for Marks in Lst :
 if Marks >= 90:
 print('Student',Lst.index(Marks) + 1,'Marks =',Marks,' grade A')
 elif Marks >=80 and Marks<90:
 print('Student',Lst.index(Marks)+ 1,'Marks =',Marks,' grade B')
 elif Marks >65 and Marks< 80 :
 print('Student',Lst.index(Marks)+ 1,'Marks =',Marks,' grade C')
 elif Marks >=40 and Marks<=65:
 print('Student',Lst.index(Marks)+ 1,'Marks =',Marks,' grade D')
 else:
 print('Student',Lst.index(Marks)+ 1,'Marks =',Marks,' grade F')
#Sample test
Lst=[78,90,34,56,89]

```

(Contd.)

```
print('Marks of 5 Student = ',Lst)
Assign_grade(Lst)
```

### Output

```
Marks of 5 Student = [78, 90, 34, 56, 89]
Student 1 Marks = 78 grade C
Student 2 Marks = 90 grade A
Student 3 Marks = 34 grade F
Student 4 Marks = 56 grade D
Student 5 Marks = 89 grade B
```

**PROGRAM 6.15** | Write a function `check_duplicate(Lst)` which returns True if a list `Lst` contains duplicate elements. It should return False if all the elements in the list `Lst` are unique.

```
Lst = [4,6,2,1,6,7,4]
Should return true as 4 and 6 appears more than once.
Lst = [1,2,3,12,4]
Should return false as all the elements appears only once.
```

```
def check_duplicate(Lst) :
 dup_Lst = []
 for i in Lst:
 if i not in dup_Lst:
 dup_Lst.append(i)
 else:
 return True
 return False
```

#### #Sample test:

```
Lst = [4,6,2,1,6,7,4]
print(Lst)
x = check_duplicate(Lst)
print(x)
Lst1 = [1,2,3,12,4]
print(Lst1)
x = check_duplicate(Lst1)
print(x)
```

(Contd.)

**Output**

```
[4, 6, 2, 1, 6, 7, 4]
True # Returns true since 4 and 6 is repeated twice
[1, 2, 3, 12, 4]
False #Returns false since no element from above list is repeated twice
```

**PROGRAM 6.16** | Write a function `print_reverse(Lst)` to reverse the elements of a list.



**Note:** Reverse the contents of a list without using the `reverse()` method of a list and without using slicing.

```
Lst=[12,23,4,5]
Should reverse the contents of list as follows
Lst=[5,4,23,12]
```

```
def print_reverse(Lst):
 print('List Before Reversing')
 print(Lst)
 lst = []
 count = 1
 for i in range(0,len(Lst)):
 lst.append(Lst[len(Lst)-count])
 count += 1
 lst = str(lst)
 lst = ''.join(lst)
 return lst

#Sample test:
Lst=[12,23,4,5,1,9]
x = print_reverse(Lst)
print('List After Reversing')
print(x)
```

**Output**

```
List Before Reversing
[12, 23, 4, 5, 1, 9]
List After Reversing
[9, 1, 5, 4, 23, 12]
```

**PROGRAM 6.17** | Write a function that accepts two positive integers `a` and `b` (`a` is smaller than `b`) and returns a list that contains all the odd numbers between `a` and `b` (including `a` and including `b` if applicable) in descending order.

Odd numbers between 10 and 20 should create the list and print the list in descending order as follows  
[19, 17, 15, 13, 11]

```
def list_of_odd_numbers(start, end):
 output_list = []
 for number in range(start, end+1):
 # check if the number is odd
 if number % 2 == 1:
 # if true put the numbers in the output list
 output_list.append(number)
 # Sort the List
 output_list.sort()
 # Reverse the list to display elements in descending order
 output_list.reverse()
 return output_list
print(list_of_odd_numbers(10, 20))
```

### Output

```
[19, 17, 15, 13, 11]
```

**PROGRAM 6.18** | Write a program to return prime numbers from a list.

```
List1=[3,17,9,2,4,8,97,43,39]
print('List1= ',List1)
lst = []
print('Prime Numbers from the List are as Follows:')
for a in List1 :
 prime = True
 for i in range(2, a):
 if (a%i == 0):
 prime = False
 break
 if prime:
 lst.append(a)
print(lst)
```

### Output

```
List1= [3, 17, 9, 2, 4, 8, 97, 43, 39]
Prime Numbers from the List are as Follows:
[3, 17, 2, 97, 43]
```

## SUMMARY

- ◆ A list is a sequence of zero or more elements.
- ◆ The element within a list can be of any data type.
- ◆ List is a mutable kind of data structure.
- ◆ A list can be initialised in different ways, viz. with and without using constructor lists.
- ◆ The index operator is used to access the elements of a list.
- ◆ The negative index accesses elements from the end of a list by counting in backward direction.
- ◆ The slicing operator and the list slicing with step size operator return a subset of a list.
- ◆ Various inbuilt functions can be used with lists.
- ◆ The for loop can be used to traverse the elements of a list.
- ◆ List comprehension can be used to create a new list from existing sequences. It is a tool for transforming a given list into another list.
- ◆ Methods such as copy, reverse and sort can be used to copy, reverse and sort the elements of a list.
- ◆ Methods such as append(), extend(), insert() are used to insert the elements within the list whereas methods such as pop(), and remove() are used to remove the contents from the list
- ◆ Proficiency in a list is impossible unless the unanswered problems are taken up for solving.

## KEY TERMS

- ⇒ **The index[] Operator:** Accesses elements of a list
- ⇒ **List Slicing:** Returns a subset of a list
- ⇒ **List Comprehensions:** Creates a new list from an existing list
- ⇒ **The split() Method:** Splits a string into words
- ⇒ **The List Inbuilt Method:** min(), max(), shuffle(), len() and sum()

## REVIEW QUESTIONS

## A. Multiple Choice Questions

1. Given: List1 = ['a','b','c','d'].  
What will be the output of the following statement?  
List1 = [x for x in List1 if ord(x) > 97]  
print(List1)
  - a. ['a','b','c']
  - b. ['b','c','d']
  - c. ['a','b','c','d']
  - d. None of the above
2. Consider the list, L = ['a','b','c','d','e','f','g','h','i','j']. Which one of the following outputs is correct?
  - a. >>> L[0:3]  
['a', 'c', 'f', 'i']
  - b. >>> L[0:-1]  
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
  - c. >>> L[0:2]
  - d. None of the above

d. 30, 40

10. List1 = ['A','B','C'] and List2 = ['B','A','C']

Is List1 == List2?

- a. Yes
- b. No
- c. Cannot predict
- d. None of the above

## B. True or False

- 1. The `list()` is used to create an empty list.
- 2. The `range()` is used for creating a list with elements from 0 to 5.
- 3. A list can be created without using a constructor.
- 4. The elements of a list are not identified by their positions.
- 5. The negative index accesses elements from the start of a list.
- 6. `List1[-1]` accesses the first element of a list.
- 7. `List1[2:5]` returns all the elements stored between the index 2 and the one less than the end index, i.e. 5-1 = 4.
- 8. It is possible to access the elements of a list only in sequence.
- 9. The `len()` returns a number of elements in a list.
- 10. The `sum()` returns the sum of all the elements in a list.
- 11. It is impossible to shuffle elements randomly in a list.
- 12. The concatenation operator '+' is used to join two lists.
- 13. The multiplication operator '\*' is used to replicate the elements in a list.
- 14. The `del` operator is used to remove a specific element from a list.
- 15. Odd elements of a list can be displayed using list comprehension.
- 16. One can insert an element at a given index.
- 17. The `pop(1)` removes an element from a list which is at index 1.
- 18. The `pop()` removes the last element from a list.
- 19. A string is a sequence of characters.
- 20. A programmer can pass a list to a function and perform various operations.

## C. Exercise Questions

- 1. How is a list created?
- 2. Explain the different ways to create a list with suitable examples.
- 3. What is meant by slicing operation?
- 4. What is the benefit of step size in a list?
- 5. Explain the supporting inbuilt functions used to create lists.
- 6. List and clarify the operators supporting lists.
- 7. What is the use of the 'is' operator in Python?
- 8. Which operator is used to delete elements from a list?
- 9. What application is used for list comprehension?
- 10. What facilitates counting of similar elements in a list?
- 11. How are elements of a list reversed?
- 12. How is a string converted into characters?
- 13. How is an empty list and list with five integers, i.e. 10, 15, 30, 50 and 40 created?



14. Given: List1 = ['a','b','c','d','e'] and List2= [1, 2, 3]. What is the return value of each of the following statements?
- ice
  - List1+List2
  - List2\*2
  - 2\*List2
15. Given: List1 = [100,200,400,500]. What is the return value of the following statements?
- min(List1)
  - max(List1)
  - sum(List1)
  - List.count(400)
  - List1.count(100)+List1.count(200)
16. Given: List1 = [12, 23, 45, 23]. What is the return value of the following statements?
- List1\*List1.count(23)
  - List1+List1[:]
  - List1+List1[-1:]
  - List1+List1[::]
  - List1+List1[::-1]
17. Given: List Lst = [10,23,5,56,78,90]. Evaluate the following expressions.
- Lst[:]
  - Lst[0:4]
  - Lst[:-1]
  - Lst[-1:]
  - Lst[-1]
  - Lst[::-1]
  - Lst[:1:]
  - Lst[:-2:]
18. Given: List1= [12, 45, 7, 89, 90]. What is the return value of the following statements?
- List1.reverse()
  - List1.sort()
  - List1.append(10)
  - List.pop(2)
  - List1.clear()
19. What is the error in the following program?
- ```
List1=['a','b','c','d']
List2=[]+List1
List1[1]='f'
print(List1*List2)
print(List2)
```
20. Write a program to pass a list to a function and return it in the reverse order.

PROGRAMMING ASSIGNMENTS

1. Write the function Replicate_n_times(Lst,n) to replicate the elements of a list n times, i.e. to replicate the elements of a list for a given number of times.

Example:

```
Lst = [1, 2, 3, 4]
Replicate_n_time(Lst,2)
Lst = [1,1,2,2,3,3,4,4]
```

2. Write a program to count the occurrences of each element within a list.

Example:

```
Lst = [1, 23, 0, 9, 0, 23]
1 occurs 1 time
```

(Contd.)

PROGRAMMING ASSIGNMENTS (Contd.)

```
23 occurs 2 times
0 occurs 2 times
9 occurs 1 times
```

3. Write the function `remove_negative(Lst)` to remove the negative elements and return the positive elements from a list.

Example:

```
Lst = [-1, 0, 2, -4, 12]
#Should return list with positive elements
Lst = [ 0, 2, 12]
```

4. Write a program to duplicate all the elements of a list.

Example:

```
Lst=[1,2,3]
#Should return
Lst=[1,1,2,2,3,3]
```

5. Write a program to check if an element of a list is a prime number. If it is prime, return True or else return False.

Example:

```
List1=[3,17,9,2,4,8]
#Should display
Lsit1=[True, True, False, False, False, False]
```

6. Write the function `remove_first_last(list)` to remove the first and last element from a list.

Example:

```
List1=[10,20,30,40,50]
removeFirstAndLast (Lis1)
#should return
[20, 30, 40]
```

7. Write a function `Extract_Even(List)` to return all even numbers from a list.

Example:

```
List1=[1,2,3,4,5,6]
Extract_Even(List1)
#should return
[2,4,6]
```

Tuples, Sets and Dictionaries

7

CHAPTER OUTLINE

7.1 Introduction to Tuples

7.3 Dictionaries

7.2 Sets

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Create tuples, sets and dictionaries and explain their necessity and importance in programming
 - Pass variable length arguments to tuples and use inbuilt functions such as `len`, `min`, `max` and `sum`, and other functions such as `zip()` and `sort()` on tuples
 - Perform different operations on sets such as union, intersection, difference and symmetric difference
 - Create dictionaries and add, retrieve, modify and delete the values of dictionaries
 - Traverse the contents of sets, tuples and dictionaries using the `for` loop function
-

7.1 INTRODUCTION

Tuples work exactly like lists. A tuple contains a sequence of items of many types. The elements of tuples are fixed. Once a tuple has been created, we cannot add or delete elements, or even shuffle their order. Hence, tuples are immutable. This means that once created, they cannot be changed. Since tuples are immutable, their length is also fixed. A new tuple must be created to grow or shrink an earlier one.

7.1.1 Creating Tuples

A tuple is an inbuilt data type in Python. In order to create a tuple, the elements of tuples are enclosed in parenthesis instead of square brackets. All the elements of a tuple are separated by commas.

Example: Defining a tuple

```
T1 = ()      #Create an empty tuple
T2 = (12,34,56,90)  #Create a tuple with 4 elements
T3 = ('a','b','c','d','e') #Create a tuple of 5 characters
T4 = 'a','b','c','d','e'   #Create a tuple without parenthesis
```



Note: #To create a tuple of a single element, it should be followed by a comma.

```
>>> T1=(4,)
```

```
>>> type(T1)
```

```
<class 'tuple'>
```

Is it possible to create a tuple of a single element without a comma?

```
>>> T1=(4)
```

```
>>> type(T1)
```

```
<class 'int'>
```

Point to Remember

A single value in parenthesis is not a tuple.

7.1.2 The tuple() Function

In the above section, we learnt how to create a tuple. For example, an empty tuple is created by using empty parenthesis.

```
>>>t1=()      # Create Empty tuple
>>> t1        # Print Empty tuple
()
>>> type(t1)   #Check the type of t1
<class 'tuple'>
```

An alternate way of creating a tuple is by using the **tuple()** function.

Example

```
>>> t1=tuple() # Create Empty tuple using tuple() function
>>> t1         # Print tuple t1
()
```

If the argument to a **tuple()** function is a sequence, i.e. string, list or a tuple then the result is a tuple with the elements of the sequence.

Example

```
>>> t1=tuple("TENNIS") #Tuple function with string as argument
>>> t1
('T', 'E', 'N', 'N', 'I', 'S')
```

7.1.3 Inbuilt Functions for Tuples

Python provides various inbuilt functions that can be used with tuples. Some of these are shown in Table 7.1.

Table 7.1 Inbuilt functions that can be used with tuples

<i>Inbuilt Functions</i>	<i>Meaning</i>
len()	Returns the number of elements in a tuple
max()	Returns the element with the greatest value
min()	Returns the element with the smallest value
sum()	Returns the sum of all the elements of a tuple
index(x)	Returns the index of element x
count(x)	Returns the number of occurrences of element x

Example

```
>>> t1= ("APPLE")
>>> len(t1) #Return the length of tuple t1
5
>>> max(t1) #Return Element from tuple with Maximum Value
'P'
>>> min(t1) #Return Element from tuple with Minimum Value
'A'
>>> t1.index('A')
0
>>> t1.count('P')
2
```

7.1.4 Indexing and Slicing

Since tuples are like lists, the indexing and slicing of tuples is also similar to that of lists. The index [] operator is used to access the elements of a tuple.

Example

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	← Positive Index
P	Y	T	H	O	N	
t[-6]	t[-5]	t[-4]	t[-3]	t[-2]	t[-1]	← Negative Index

```

>>> t=('P','Y','T','H','O','N')    #Create Tuple
>>>t                                #Print Tuple
>>>('P', 'Y', 'T', 'H', 'O', 'N')
>>> t[0]
'P'
>>> t[5]
'N'
>>> t[-1]
'N'
>>> t[-6]
'P'

```

Tuple Slicing Example

```

>>> t=('P','Y','T','H','O','N')    #Create Tuple
>>>t                                #Print Tuple
>>>('P', 'Y', 'T', 'H', 'O', 'N')
>>>t[0:]    #Print the contents of tuple t starting from index 0
('P', 'Y', 'T', 'H', 'O', 'N')
>>> t[0:3]  #Print the contents of tuple t starting from 0 to 2
('P', 'Y', 'T')

```



Note: More details of slicing can be found in Chapter 8: Lists.

7.1.5 Operations on Tuples

Tuples don't support all methods supported by lists. A tuple supports the usual sequence operations supported by a list.

1. **The + Operator:** The concatenation + operator is used to join two tuples.

```

>>> (1,2)+(3,4)    #The concatenation operator
(1, 2, 3, 4)

```

2. **The * Operator:** The multiplication operator is used to replicate the elements of a tuple.

```

>>> (1,2)*3        #The Repetition Operator
(1, 2, 1, 2, 1, 2)

```

7.1.6 Passing Variable Length Arguments to Tuples

We can pass variable number of parameters to a function. An argument which begins with the * in function definition gathers all arguments into a tuple.

PROGRAM 7.1 | Write a program to create a function `create_tup()` which accepts a variable number of arguments and prints all of them.

```

def create_tup(*args):
    print(args)

```

(Contd.)

Output

Run the above program in interactive mode of Python.

```
>>> create_tup(1,2,3,4)
(1, 2, 3, 4)
>>> create_tup('a','b')
('a', 'b')
```

The inbuilt **sum()** function takes two parameters to add the elements within it.

How can we create a function that takes variable arguments and adds all the elements present in it?

The following program creates the function **sum_all()** which accepts a variable number of arguments and displays the sum of all the arguments.

PROGRAM 7.2 | Create function **sum_all()** to accept a variable number of arguments and display the sum of all the elements present in it.

```
def sum_all(*args):
    t=()
    s=0
    for i in args:
        s=s+i
    print(s)
```

Output

#Running the above program in Python interactive mode.

```
>>>sum_all(10,20,30,40) #Function sum_all with variable arguments
100

>>> sum_all(1,2,3)
6
```

7.1.7 Lists and Tuples

A tuple can also be created from a list. This is illustrated in the following example.

Example

```
>>> List1=[1,2,3,4]      #Create List
>>> print(List1)         #Print List1
[1, 2, 3, 4]
>>> type(List1)          #Print the type of variable List1
<class 'list'>
>>> t1=tuple(List1)      #Convert List to tuple
>>> t1                   #print t1
```

```

(1, 2, 3, 4)
>>> type(t1)    #Check type of t1 after converting List to Tuple
<class 'tuple'>

```

7.1.8 Sort Tuples

If a programmer wants to sort a tuple, he/she can use the inbuilt `sort()` method. A tuple does not contain any method named `sort`. Therefore, to sort a tuple, a programmer will have to first convert a tuple into a list. After conversion, he/she can use `sort()` method for lists and then again convert the sorted list into a tuple.

```

>>> t1=(7,2,1,8)    #Create Tuple t1
>>> t1              #Print t1
(7, 2, 1, 8)
>>> L1=list(t1)     #Convert Tuple t1 to List
>>> L1              #Print L1
[7, 2, 1, 8]
>>> L1.sort()       #Sort List
>>> t2=tuple(L1)    #Convert Sorted List to Tuple
>>> t2              #Print sorted tuple
(1, 2, 7, 8)

```

7.1.9 Traverse Tuples from a List

A tuple assignment can be used in the for loop to traverse a list of tuples.

PROGRAM 7.3 | Write a program to traverse tuples from a list.

```

T=[(1, "Amit"),(2,"Divya"),(3,"Sameer")]
for no, name in t:
    print(no, name)

```

Output

```

1 Amit
2 Divya
3 Sameer

```

7.1.10 The `zip()` Function

The `zip()` is an inbuilt function in Python. It takes items in sequence from a number of collections to make a list of tuples, where each tuple contains one item from each collection. The function is often used to group items from a list which has the same index.

Example

```

>>> A1=[1,2,3]

```



```

..
>>> A2="XYZ"
>>> list(zip(A1,A2))          #Zip List A1 and A2
[(1, 'X'), (2, 'Y'), (3, 'Z')]

```

Explanation

The result of `list(zip(A1,A2))` is a list of tuples where each tuple contains an index wise element from each list as a pair.

Example

```

>>> L1=['Laptop', 'Desktop', 'Mobile']          #Create List1
>>> L2=[40000,30000,15000]                      #Create List2
>>> L3=tuple((list(zip(L1,L2))))                #Group item from List 1 and 2
>>> L3      #print L3
(('Laptop', 40000), ('Desktop', 30000), ('Mobile', 15000))

```



Note: If the sequences are not of the same length then the result of `zip()` has the length of the shorter sequence.

Example:

```

>>> a="abcd"          #Sequence of length 4
>>> b=[1,2,3]         #Sequence of length 3
>>> list(zip(a,b))    #Zip() on a and b returns list of tuples
[('a', 1), ('b', 2), ('c', 3)]

```

PROGRAM 7.4 | Consider two lists, viz. List L1 and L2. Here, L1 contains a list of colors and L2 contains their color code as:

```

L1=['Black','White','Gray']
L2=[255,0,100]

```

Display the contents as:

```

('Black',255)
('white',0)
('Gray',100)

```

```

L1=['Black','White','Gray']    #Create List L1
L2=[255,0,100]                 #Create List L2
for Color,Code in zip(L1,L2):  # Use of zip in for loop
    print((Color,Code))

```

Output

```

('Black', 255)
('White', 0)
('Gray', 100)

```

7.1.11 The Inverse `zip(*)` Function

The `*` operator is used within the `zip()` function. The `*` operator unpacks a sequence into positional arguments. A simple example of the `*` operator on positional arguments is given as follows:

PROGRAM 7.5 | Demonstrate the use of the `*` operator on positional arguments.

```
def print_all(Country, Capital):
    print(Country)
    print(Capital)
```

Output

```
>>> args=("INDIA","DELHI")
>>> print_all(*args)
INDIA
DELHI
```

Explanation In the above program, the function `print_all()` is created. When `*args` are passed to the function `print_all`, its values are *unpacked* into the function's positional arguments `arg1` to `Country` and `Capital` to `arg2`.

The function `zip(*)` also performs the same operation, i.e. unpacks a sequence into positional arguments.

PROGRAM 7.6 | Demonstrate the use of the `zip(*)` function.

```
X=[("APPLE",50000),("DELL",30000)] #List of tuples
Laptop,Prize=zip(*X) # Unpacking Values
print(Laptop)
print(Prize)
```

Output

```
('APPLE', 'DELL')
(50000, 30000)
```

Explanation In the above program, initially the list is created. List `x` contains a sequence of tuples. The function `zip(*)` is used to unpack the values of `x`.

7.1.12 More Examples on `zip(*)` Function

#Transpose of a matrix

```
>>> Matrix=[(1,2),(3,4),(5,6)]
```

```
..
>>> Matrix
[(1, 2), (3, 4), (5, 6)]
>>> x=zip(*Matrix)
>>> tuple(x)
((1, 3, 5), (2, 4, 6))
```

7.1.13 More Programs on Tuples

PROGRAM 7.7 | Consider an example of a tuple as T = (1, 3, 2, 4, 6, 5). Write a program to store numbers present at odd index into a new tuple.

```
def oddTuples(aTup): #Function with tuple as an argument
    rTup = ()        #Initially the output tuple rTup is empty
    index = 0
    while index < len(aTup):
        rTup += (aTup[index],)
        index += 2    #index increased by 2
    return rTup
t=(1, 3, 2, 4, 6, 5)
print(oddTuples(t))
```

Output

```
(1, 2, 6)
```

Explanation In the above program, initially a tuple 't' is created. This tuple 't' is passed as a parameter to a function. The while loop iterates till the length of the tuple. In each iteration, the number stored at an odd index is accessed and stored into the output tuple 'rTup'.

7.2 SETS

A set is an unordered collection of unique elements without duplicates. A set is mutable. Hence, we can easily add or remove elements from a set. The set data structure in Python is used to support mathematical set operations.

7.2.1 Creating Sets

A programmer can create a set by enclosing the elements inside a pair of curly brackets {}. The elements within a set can be separated using commas. We can also create a set using the inbuilt `set()` function, or from an existing list or tuple.

Examples

```
>>>S1 =set() # Creates an empty Set
>>>S1        # Print Set S1
```

```

set()
>>> type(S1)  # Check type of S1
<class 'set'>

>>> S1={10,20,30,40}    # Create set of 4 elements
>>> S1                  # Print Set S1
{40, 10, 20, 30}

>>> S2=[1,2,3,2,5]      # Create List
>>> S2                  # Print List
[1, 2, 3, 2, 5]
>>> S3=set(S2)          # Convert List S2 to Set
>>> S3                  #Print S3 (Removes duplicate from the List)
{1, 2, 3, 5}

>>> S4=(1,2,3,4)        # Create Tuple
>>> S5=set(S4)          # Convert Tuple to Set
>>> S5                  # Print S5
{1, 2, 3, 4}

```

7.2.2 The Set in and not in Operator

The **in** operator is used to check if an element is in a set. The **in** operator returns True if the element is present in the list. The **not in** operator returns True if the said element is not present in the set.

Example

```

>>> S1={1,2,3}
>>> 3 in S1              # Check if 3 is in S1
True
>>> 4 not in S1          # Check if 4 is not in the S1
True

```

7.2.3 The Python Set Class

Python contains a **set** class. The most commonly used methods within the **set** class (with examples) are listed in Table 7.2.

Table 7.2 Methods of set class

Function	Meaning
s.add(x) Example: <pre>>>> s1={1,2,19,90} # Create set of 4 elements >>> s1.add(100) # Add 100 to the existing list s1 >>> s1 # Print s1 {1, 90, 19, 2, 100}</pre>	Adds the element x to an existing set s .
s.clear() Example: <pre>>>> s1={1,2,3,4} #Create set of 4 elements >>> s1.clear() #Remove all the elements from the set s1 >>> s1 #Print s1 set()</pre>	Removes the entire element from an existing set.
S.remove(x) Example: <pre>>>> s1={1,2,3,4} >>> s1.remove(2) #Remove element 2 from Set s1 >>> S1 {1, 3, 4}</pre> <p>Note: The <code>discard()</code> function is similar to <code>remove</code> function.</p>	Removes the item x from a set.
s1.issubset(s2) Example: <pre>>>> s1={1,2,3,4} >>> s2={1,2,3,4,5} >>> s1.issubset(s2) # Check if all the elements of s1 are in s2. True</pre>	If every element in $s1$ is also in $s2$ then set $s1$ is a subset of $s2$. The <code>issubset()</code> is used to check whether $s1$ is a subset of $s2$.
s2.issuperset(s1) Example: <pre>>>> s1={1,2,3} >>> s2={1,2,3,4} >>> s2.issuperset(s1) True</pre>	Let $s1$ and $s2$ be two sets. If $s1$ is a subset of $s2$ and the set $s1$ is not equal to $s2$ then the set $s2$ is called a superset of A .

7.2.4 Set Operations

In mathematics or everyday applications we often use various set operations, such as `union()`, `intersection()`, `difference()` and `symmetric_difference()`. All these methods are part of the set class.

The `union()` Method

The union of two sets A and B is a set of elements which are in A, in B or in both A and B. We can use the **union** method or the `|` operator to perform this operation.

Example

```
>>> S1={1,2,3,4}
>>> S2={2,4,5,6}
>>> S1.union(S2)
{1, 2, 3, 4, 5, 6}

>>>S1 | S2
{1, 2, 3, 4, 5, 6}
```



Note: Sets cannot have duplicate elements. So, the union of sets {1,2,3,4} and {2,4,5,6} is {1,2,3,4,5,6}.

The `intersection()` Method

The intersection of two sets A and B is a set which contains all the elements of A that also belong to B. In short, intersection is a set which contains elements that appear in both sets. We can use intersection methods or the `&` operator to perform this operation.

Example

```
>>> S1={1,2,3,4}
>>> S2={3,4,5,6}
>>> S1.intersection(S2)
{3, 4}
>>> S1 & S2
{3, 4}
```

The `difference()` Method

The difference between two sets A and B is a set which contains the elements in set A but not in set B. We can use the **difference** method or the `-` operator to perform the difference operation.

Example

```
>>> A={1,2,3,4}
>>> B={3,4,5,6}
>>> A.difference(B)
{1, 2}
```

```
..
>>>>> A-B
{1, 2}
```

The symmetric_difference()

The symmetric difference is a set which contains elements from the either set but not in both sets. We can use symmetric_difference method or the ^ (exclusive) operator to perform this operation.

Example

```
>>> S1={1,2,3,4}
>>> S2={3,4,5,6}
>>> S1.symmetric_difference(S2)
{1, 2, 5, 6}
>>> S1^S2
{1, 2, 5, 6}
```

7.3 DICTIONARIES

7.3.1 Need of Dictionaries

In the previous chapter, we learnt about a Python data structure called **list**. Lists organise their elements by position and this kind of structuring is useful when we want to locate elements in a specific order, i.e. locate either first, last element or visit each element in a sequence.

There may be situation where a programmer is not so much interested in the position of the item or element in the structure but in association of that element with some other element in the structure.

For example, to look up Amit's phone number we are just interested in his number from the phonebook and don't care much where the number is located in the phonebook. It means the name of the person is associated with his phone number.

7.3.2 Basics of Dictionaries

In Python, a dictionary is a collection that stores values along with keys. The sequence of such key and value pairs is separated by commas. These pairs are sometimes called **entries or items**. All entries are enclosed in curly brackets { and }. A colon separates a key and its value. Sometimes, items within dictionaries are also called associative arrays because they associate a key with a value.

Simple examples of dictionaries are given as follows:

```
Phonebook - {"Amit": "918624986968", "Amol": "919766962920"}
```

```
Country Code Information - {"India": "+91", "USA": "+1", "Singapore": "+65"}
```

The structure of a dictionary is shown in Figure 7.1a. The above phonebook example is illustrated in Figure 7.1b.

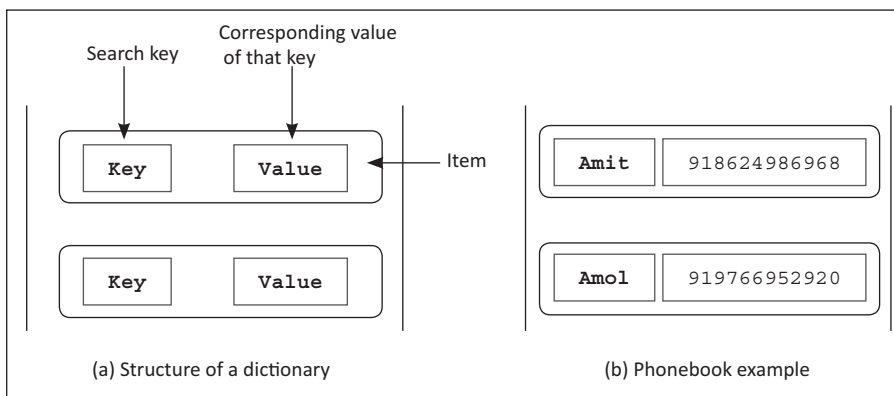


Figure 7.1 a and b Dictionary—structure and example

Keys are like an index operator in a dictionary. A key can be of any type. Therefore, a dictionary maps a set of objects, i.e. keys to another set of objects, i.e. values. It is a mapping of unique keys to values, i.e. each key is mapped to one value. Also, dictionaries do not contain any duplicate keys.

7.3.3 Creating a Dictionary

We can create a dictionary by enclosing the items inside a pair of curly brackets {}. One way to start a dictionary is to create an empty dictionary first and then add items to it.

Creating an Empty Dictionary

Example

```
>>>D1 = {}          # Create Empty Dictionary
>>>D1               # Print Empty Dictionary
{}
>>> type(D1)        # Check the type of D1
<class 'dict'>
```



Note: Python uses curly brackets for sets and dictionaries. Therefore, to create an empty dictionary, we use {} and to create an empty set, we use the function `set()`.

Creating a Dictionary with Two Items

To create a dictionary of two items, the items should be in the form of **key:value** and separated by commas.

Example: Creating a dictionary of two items

```
>>> P={"Amit":"918624986968", "Amol":"919766962920"}
>>> P      #Display P
{'Amit': '918624986968', 'Amol': '919766962920'}
```


Creating Dictionaries in Four Different Ways

Example

#Way 1:

```
>>> D1 = {'Name': 'Sachin', 'Age': 40}
>>> D1
{'Name': 'Sachin', 'Age': 40}
```

#Way 2:

```
>>> D2 = {}
>>> D2['Name'] = 'Sachin'
>>> D2['Age'] = 40
>>> D2
{'Name': 'Sachin', 'Age': 40}
```

#Way 3:

```
>>> D3 = dict(Name='Sachin', Age=40)
>>> D3
{'Name': 'Sachin', 'Age': 40}
```

#Way 4:

```
>>> dict([('name', 'Sachin'), ('age', 40)])
{'age': 40, 'name': 'Sachin'}
```

Explanation

In the above example, we have created dictionaries in four different ways. We can select the first way if we know all the contents of a dictionary in advance. The second way, if we want to add one field at a time. The third way requires all keys to string. The fourth way is good if we want to build keys and values at runtime.

7.3.4 Adding and Replacing Values

To add a new item to a dictionary, we can use the subscript[] operator. The syntax to add and an item to a dictionary is:

```
Dictionary_Name[key] = value
```

Example

```
P["Jhon"] = "913456789087"
```

In the above example, the name of the dictionary is P. We are adding the phone number of "Jhon" into our phonebook. The "Jhon" will act as the key and the phone number of Jhon will be its value.

#Running the above example in Python interactive mode**#Create Dictionary of Phonebook**

```
P={"Amit":"918624986968", "Amol":"919766962920"}
```

```
>>> P      #Display P
```

```
{'Amit': '918624986968', 'Amol': '919766962920'}
```

#Add another element to the existing Dictionary of Phone Book P

```
>>> P["Jhon"]="913456789087"    #Add New element
```

```
>>> P
```

```
{'Jhon': '913456789087', 'Amit': '918624986968', 'Amol': '919766962920'}
```



Note: If a key is already present in a list then it replaces the old value for the said key with the new value.

Example

```
P={"Amit":"918624986968", "Amol":"919766962920"}
```

```
>>> P      #Display P
```

```
{'Amit': '918624986968', 'Amol': '919766962920'}
```

```
>>> P["Amit"]="921029087865"    #Replace the Old value by New
```

```
>>> P      #Print After Replacing
```

```
{'Amit': '921029087865', 'Amol': '919766962920'}
```

7.3.5 Retrieving Values

The subscript[] can also be used to obtain the value associated with a key. The syntax is:

```
Dictinoary_Name[Key] #Retrieve the value associated with the key.
```

Example

```
P={"Amit":"918624986968", "Amol":"919766962920"}
```

```
>>> P      #Display P
```

```
{'Amit': '918624986968', 'Amol': '919766962920'}
```

```
>>> P["Amol"] #Display the value associated with the key "Amol"
```

```
'919766962920'
```



Note: If a key is not in a dictionary, Python raises an error.

Example

```
>>>P={"Amit":"918624986968", "Amol":"919766962920"}
```

```
>>> P["Sachin"]
```

```
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    P["Sachin"]
KeyError: 'Sachin'
```

7.3.6 Formatting Dictionaries

The % operator is used to substitute values from a dictionary, into a string by a name.

Example

```
>>> D={}
>>> D["Laptop"]="MAC"
>>> D["Count"]=10
>>> D          #Print Dictionary D
{'Laptop': 'MAC', 'Count': 10}
>>> P="I want %(Count)d %(Laptop)s Laptops"%D
>>> P
'I want 10 MAC Laptops'
```

Explanation

In the above program, initially a dictionary is created containing two keys, viz. 'Laptop' and 'Count'. In the statement, "I want %(Count)d %(Laptop)s Laptops"%D." The characters 'd' and 's' for integer and string.

7.3.7 Deleting Items

We can delete any entry from a dictionary. The **del** operator is used to remove a key and its associated value. If a key is in a dictionary then it is removed otherwise Python raises an error. The syntax used to remove an element from a dictionary is

```
del dictionary_name[key]
```

Example

```
>>>P={"Amit":"918624986968", "Aml":"919766962920"}
>>> del P["Amit"]  #delete key "Amit"
>>> P              #Print after deleting
{'Aml': '919766962920'}
```

7.3.8 Comparing Two Dictionaries

The == operator is used to test if two dictionaries contain the same items. Also, the != operator returns True if the items within dictionaries are not the same.

Example

```
>>> A={"I":"India","A":"America"}
>>> A
{'I': 'India', 'A': 'America'}
>>> B={"I":"Italy","A":"America"}
>>> B
{'I': 'Italy', 'A': 'America'}
>>> A==B
False
>>> A!=B
True
```

7.3.9 The Methods of Dictionary Class

Python contains dict class for dictionaries. To see the complete documentation for dictionaries we can run `help(dict)` in Python interactive mode. Table 7.3 contains the methods of the dictionary class along with suitable examples.

Table 7.3 Some commonly used dictionary operations

<i>Methods of dict Class</i>	<i>What it does?</i>
keys() Example: <pre>>>> ASCII_CODE={"A":65,"B":66,"C":67,"D":68} >>> ASCII_CODE #Print Dictionary named ASCII_CODE {'D': 68, 'B': 66, 'A': 65, 'C': 67} >>> ASCII_CODE.keys() #Return all keys dict_keys(['D', 'B', 'A', 'C'])</pre>	Returns a sequence of keys.
Values() Example: <pre>>>> ASCII_CODE={"A":65,"B":66,"C":67,"D":68} >>> ASCII_CODE.values() #Return Values dict_values([68, 66, 65, 67])</pre>	Returns a sequence of values.
items() Examples: <pre>>>>ASCII_CODE={"A":65,"B":66,"C":67,"D":68} >>>ASCII_CODE.items() dict_items([('D', 68), ('B', 66), ('A', 65), ('C', 67)])</pre>	Returns a sequence of tuples.
clear() Example: <pre>>>>ASCII_CODE={"A":65,"B":66,"C":67,"D":68} >>> ASCII_CODE.clear() # Delete all entries >>> ASCII_CODE # Print after {}</pre>	Deletes all entries.

(Contd.)

<pre>get(key)</pre> <p>Example:</p> <pre>>>> Temperature={"Mumbai":35,"Delhi":40,"Chennai":54} >>> Temperature.get("Mumbai") 35</pre>	Returns the value for a key.
<pre>pop(key)</pre> <p>Example:</p> <pre>>>> Temperature.pop("Mumbai") 35 >>> Temperature #Print after removing key "Mumbai". {'Delhi': 40, 'Chennai': 54}</pre>	Removes a key and returns the value if the key exists.
<pre>clear()</pre> <p>Example:</p> <pre>>>> Temperature={"Mumabai":35,"Delhi":40,"Chennai":54} >>> Temperature.clear() >>> Temperature</pre>	Removes all the keys.

7.3.10 Traversing Dictionaries

The for loop is used to traverse all the keys and values of a dictionary. A variable of the for loop is bound to each key in an unspecified order. It means it retrieves the key and its value in any order. The following program shows the traversing elements of a dictionary.

PROGRAM 7.8 | Write a program to traverse the elements of a dictionary.

```
Grades={"Tammana":"A","Pranav":"B","Sumit":"C"}
for key in Grades:
    print(key,":",str(Grades[key]))
```

Output

```
Tammana: A
Sumit: C
Pranav: B
```



Note: Write the above program in Python shell and then execute it in Python interpreter. The latter will display all items in a different order.

PROGRAM 7.9 | Write a program to assign grades to students and display all the grades using keys() and get() method of a dictionary.

```
Grades={"Tamana":"A","Pranav":"B","Summit":"C"}
for key in Grades.keys():
```

(Contd.)

```
print(key, ' ', Grades.get(key, 0))
```

Output

```
Summit    C
Pranav    B
Tamana    A
```

Explanation Grades of students are assigned to the dictionary Grades. As discussed in Table 7.3, the `keys()` method is used in for loop to return the sequence of keys. All returned keys are stored in a variable `key`. Finally, `get()` method is used to return values associated with the particular key.

7.3.11 Nested Dictionaries

A dictionary within a dictionary is called a nested dictionary. To understand this, let us make a dictionary of Indian cricket players with some information about them. The key for this dictionary will consist of the cricketers' names. The value will include information such as the runs scored in test and ODI matches.

```
>>> Players={"Virat Kohli" : { "ODI": 7212 ,"Test":3245},
              "Sachin Tendulkar" : { "ODI": 18426 ,"Test":15921}}

>>> Players['Virat Kohli']['ODI'] # Display run scored by Kohli in ODI
7212

>>> Players['Virat Kohli']['Test']#Display run scored by Kohli in Test
3245

>>> Players['Sachin Tendulkar']['Test']
15921

>>> Players['Sachin Tendulkar']['ODI']
18426
```

7.3.12 Traversing Nested Dictionaries

We used the for loop to traverse simple dictionaries. It can also be used to traverse nested dictionaries. Let us write the above example and use the for loop to go through the keys of the dictionaries.

```
Players={"Virat Kohli" : { "ODI": 7212 ,"Test":3245},
          "Sachin Tendulkar" : { "ODI": 18426 ,"Test":15921}}

#Way 1
for Player_Name, Player_Details in Players.items():
```

(Contd.)

```

print("", Player_Name)
print("", Player_Details)
#Way 2
for Player_Name, Player_Details in Players.items():
    print(" Player: ", Player_Name)
    print(" Run Scored in ODI:\t", Player_Details["ODI"])
    print(" Run Scored in Test:\t", Player_Details["Test"])

```

Output

```

Sachin Tendulkar
{'Test': 15921, 'ODI': 18426}
Virat Kohli
{'Test': 3245, 'ODI': 7212}

Player: Sachin Tendulkar
Run Scored in ODI:      18426
Run Scored in Test: 15921

Player: Virat Kohli
Run Scored in ODI:      7212
Run Scored in Test: 3245

```

Explanation The above program shows the two different ways to print the details of the dictionaries. The first way contains the code:

```

for Player_Name, Player_Details in Players.items():
    print("", Player_Name)
    print("", Player_Details)

```

In the above code, `Player_Name` stores the keys, i.e. the name of the player from the outer dictionary and the variable `Player_Details` stores the value associated with the key, i.e. `Player_Name`.

However, the second way is used to access specific information about a player. The code for the second way is:

```

for Player_Name, Player_Details in Players.items():
    print(" Player: ", Player_Name)
    print(" Run Scored in ODI:\t", Player_Details["ODI"])
    print(" Run Scored in Test:\t", Player_Details["Test"])

```

In the for loop we have used `Player_Name` which displays the name of the player as key of the dictionary. To access specific details of that player, key the index `[]` operator is used.

The above program code is much shorter and easier to maintain, but even this code will not keep up with our dictionary. If we add more information to our dictionary, we will have to update our print statements later.

Let us minimise the above piece of code and put a second for loop inside the first for loop in order to run through all the information about each player.

```
Players={"Virat Kohli" : { "ODI": 7212 ,"Test":3245},
        "Sachin Tendulkar" : {"ODI": 18426 ,"Test":15921}}
for Player_Name, Player_Details in Players.items():
    print(" ",Player_Name)
    for key in Player_Details:
        print(key,':',str(Player_Details[key]))
```

Output

```
Sachin Tendulkar
Test : 15921
ODI : 18426
    Virat Kohli
Test : 3245
ODI : 7212
```

Explanation The first loop gives us all the keys in the main dictionary which consist of the name of each player. Each of these names can be used to unlock the dictionary for each player. The inner loop goes through the dictionary for that individual player and pulls out all the keys in that player's dictionary. The inner for loop prints the key, which tells us the kind of information we are about to see and the value for that key.

7.3.13 Simple Programs on Dictionary

PROGRAM 7.10 Write a function histogram that takes string as parameter and generates a frequency of characters contained in it.

```
S = "AAPPLE"
```

The program should create a dictionary

```
D = {'A': 2, 'E': 1, 'P': 2, 'L': 1}
```

```
def Histogram(S):
    D =dict() #Initially Create Empty Dictionary
    for C in S:
        if C not in D:
            D[C] = 1
        else:
            D[C] =D[C]+1
```

(Contd.)


```

    return D
H=Histogram("AAPPLE")
print(H)

```

Output

```
{'A': 2, 'E': 1, 'P': 2, 'L': 1}
```

Explanation In the above program, we created a function Histogram(S). A string S is passed as a parameter to the function. Initially, an empty dictionary is created. The for loop is used to traverse the string. While traversing, each character is stored in C. If the character C is not in the dictionary then we inserted a new item into the dictionary with key C and initial value as 1. If C is already in the dictionary then we incremented D[C].

PROGRAM 7.11 | Write a program to count the frequency of characters using the get() method.

```

def Histogram(S):
    D =dict()
    for C in S:
        if C not in D:
            D[C] = 1
        else:
            D[C]=D.get(C,0)+1

    return D
H=Histogram("AAPPLE")
print(H)

```

Output

```
{'P': 2, 'L': 1, 'A': 2, 'E': 1}
```

PROGRAM 7.12 | Write a program to print and store squares of numbers into a dictionary.

```

def Sq_of_numbers(n):
    d=dict() #Creates A Empty Dictionary
    for i in range(1,n+1): # Iterates from 1 to N
        if i not in d:
            d[i]=i*i #Store the Square of a Number i into dictionary
    return d
print('Squares of Number:')

```

(Contd.)

```
Z=Sq_of_numbers(5)
print(Z)
```

Output

```
Squares of Number:
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

PROGRAM 7.13 | Write a program to pass a list to a function. Calculate the total number of positive and negative numbers from the list and then display the count in terms of dictionary.

Input: L=[1,-2,-3,4]

Output: {'Neg': 2, 'Pos': 2}

```
def abc(L):
    D={}                                #Empty Dictionary
    D["Pos"]=0
    D["Neg"]=0
    for x in L:
        if x>0:
            D["Pos"]+=1
        else:
            D["Neg"]+=1
    print(D)
L=[1,-2,-3,4]
abc(L)
```

Output

```
{'Pos': 2, 'Neg': 2}
```

Explanation In the above program, an empty dictionary D is created. Initially two keys are added to the dictionary, viz. Pos and Neg with count 0 as their respective value. The list L is passed to the function abc(). If the number is positive or negative, the count is increased accordingly.

PROGRAM 7.14 | Write a program to convert an octal number into binary.

Input: (543)₈

Output: (101100011)

```
def Convert_Oct_Bin(Number,Table):
    binary=''
    for digit in Number:
```

(Contd.)

```

        binary = binary +Table[digit]
    return binary

```

```

octToBinaryTable={ '0': '000', '1': '001', '2': '010',
                   '3': '011', '4': '100', '5': '101',
                   '6': '110', '7': '111' }

```

Output

#Sample Input 1:

```

>>> Convert_Oct_Bin("553",octToBinaryTable)
'101101011'

```

#Sample Input 2:

```

>>> Convert_Oct_Bin("127",octToBinaryTable)
'001010111'

```

Explanation In the above program we created the function `Convert_oct_Bin()`. It accepts two parameters. The first parameter is the octal number as a string which we want to convert into binary and the second parameter is the dictionary which contains the decimal number and its equivalent binary number.

The above algorithm visits each digit of the octal number, selects the corresponding three bits which represent that digit in binary and add these bits to the result string `binary`.

7.3.14 Polynomials as Dictionaries

As we learnt in previous chapters, Python has two data types, viz. mutable and immutable. Python objects that cannot change their contents are known as immutable data types. The immutable data types consist of `str` and `tuple`. List and dictionaries can change their contents so they are called mutable objects. The keys in a dictionary are not restricted to be strings. Any immutable Python object can be used as a key. However, a common type of key used in a dictionary is of type integers.

Let us consider the following example of how dictionaries with integers as keys represent a polynomial.

Example of Polynomial

$$P(Y) = -2 + Y^2 + 3Y^6$$

The above example is a polynomial of a single variable, i.e. y . The above polynomial consists of three terms, viz. (-2) , (Y^2) and $(3Y^6)$. All the terms can be viewed as set of power and coefficient terms. The first term, i.e. (-2) contains power of y as 0 and coefficient as -2. Similarly, the term two (Y^2) contains the power of y as 2 and coefficient as 1. And the last term $(3Y^6)$ contains the power of y as 6 and coefficient as 3. A dictionary can be used to map a power to a coefficient.

Representing above Polynomial using Dictionaries

$$P = \{0:-2, 2:1, 6:3\}$$

The above polynomial can also be represented as a list. But we have to fill in all zero coefficients too since the index must match power. Therefore, representing the above polynomial as list is

$$P(Y) = -2 + Y^2 + 3Y^6$$

$$P = [-2, 0, 1, 0, 0, 0, 3]$$

After representing the said polynomial in terms of list, we can compare the representation of the polynomial in terms of dictionary and list. The advantage of a dictionary is that the user has to store only the non-zero coefficient. For the polynomial $1+x^{50}$, the dictionary holds only two elements, while the list holds 51 elements.

PROGRAM 7.15 | Write a program to evaluate a polynomial of one variable, i.e. x if the value of x is 2.

$$P(X) = -2+X^2+3X^3$$

$$P(2) = 26$$

```
def Eval_Poly(P,X):
    sum = 0
    for Power in P:
        sum = sum + P[Power]*X**Power
    print('The Value of Polynomial after Evaluation:',sum)
P = {0:-2, 2:1, 3:3}
Eval_Poly(P,2)
```

Output

The Value of Polynomial after Evaluation: 26

Explanation The function Eval_Poly() is created. The polynomial P is represented in the form of a dictionary. The argument to the function is a dictionary of polynomial P. Where P[Power] holds the coefficient associated with the term X**Power.

MINI PROJECT Orange Cap Calculator

The orange cap is an annual cricket award presented to the leading run scorer in a cricket series.

Example

Consider an ongoing test cricket series. Following are the names of the players and their scores in the test1 and 2. Calculate the highest number of runs scored by an individual cricketer in both the tests.

```
orangepcap({ 'test1': { 'Dhoni': 74, 'Kohli': 150 }, 'test2': { 'Dhoni': 29,
'Pujara': 42 } })
```

From the above example we can analyse the runs scored by each player in both tests as

$$\text{Dhoni} = 74 + 29 = 103$$

$$\text{Kohli} = 150 + 0 = 150$$

$$\text{Pujara} = 0 + 42 = 42$$

Thus, Kohli scored the most runs in both the test matches and he will be awarded the orange cap for this tournament.

Program Statement

Define a Python function 'orangepcap(d)' which reads a dictionary 'd' of the following form and identifies the player with the highest total score. The function should return a pair (playername, topscore), where playername is the name of the player with the highest score and topscore is the total runs scored by the player.

Input

```
orangepcap({ 'test1': { 'Dhoni': 74, 'Kohli': 150 }, 'test2': { 'Dhoni': 29,
'Pujara': 42 } })
```

Output

```
('Kohli', 150)
```

Algorithm

- ⊙ **STEP 1:** Create a dictionary 'd' consisting of overall score details of test1 and test2.
- ⊙ **STEP 2:** Pass the dictionary 'd' to the Orangepcap() function.
- ⊙ **STEP 3:** Use the for loop to traverse the contents of dictionaries and nested dictionaries.
- ⊙ **STEP 4:** In each iteration for each player store the runs scored by each player.
- ⊙ **STEP 5:** Display information about a player with name and maximum runs in all the matches.

Program

```
def orangecap(d):
    total = {}
    for k in d.keys():
        for n in d[k].keys():
            if n in total.keys():
                total[n] = total[n] + d[k][n]
            else:
```

(Contd.)

```

        total[n] = d[k][n]
    print('Total Run Scored by Each Player in 2 Tests: ')
    print(total)

    print('Player With Highest Score')
    maxtotal = -1
    for n in total.keys():
        if total[n] > maxtotal:
            maxname = n
            maxtotal = total[n]

    return(maxname,maxtotal)
d=orangecap({'test1':{'Dhoni':74, 'Kohli':150}, 'test2':{'Dhoni':29,
'Pujara':42}})
print(d)

```

Output

```

Total Run Scored by Each Player
{'Dhoni': 103, 'Pujara': 42, 'Kohli': 150}
Player With Highest Score
('Kohli', 150)

```

SUMMARY

- ◆ A tuple contains a sequence of items of any type.
- ◆ The elements of tuples are fixed.
- ◆ Tuples are immutable.
- ◆ A tuple can also be created from a list.
- ◆ The elements of tuples are enclosed in parenthesis instead of square brackets.
- ◆ Tuples don't not contain any method named sort.
- ◆ A set is an unordered collection of elements without duplicates.
- ◆ Sets are mutable.
- ◆ Different mathematical operations such as union, intersection, difference and symmetric difference can be performed on sets.
- ◆ A dictionary is a collection which stores values along with keys.
- ◆ A for loop is used to traverse all keys and values of a dictionary.
- ◆ The in and not in can be used to check if a key is present in a dictionary.

KEY TERMS

- ⇒ **Tuple:** Sequence of elements of any type
- ⇒ **Set:** Unordered collection of elements without duplicates

- ⇒ **Dictionary:** Collection of key and value pair
- ⇒ **Immutable:** Python objects which can't be changed
- ⇒ **Nested Dictionary:** A dictionary within a dictionary
- ⇒ **zip() Function:** Inbuilt Python function used to make a list of tuples
- ⇒ **zip(*) Function:** Zip inverse
- ⇒ **set Functions:** union(), intersection(), difference() and symmetric_difference()

REVIEW QUESTIONS

A. Multiple Choice Questions

1. What will be the output of the following code?

```
def main():
    Average_Rainfall={}
    Average_Rainfall['Mumbai']=765
    Average_Rainfall['Chennai']=850
    print(Average_Rainfall)
main()
```

- | | |
|------------------------------------|------------------------------------|
| a. ['Mumbai': 765, 'Chennai': 850] | b. {'Mumbai': 765, 'Chennai': 850} |
| c. ('Mumbai': 765, 'Chennai': 850) | d. None of the above |

2. What will be the output of the following code?

```
init_tuple=()
print(init_tuple.__len__())
```

- | | |
|---------|----------|
| a. 1 | b. 0 |
| c. NULL | d. Empty |

3. What will be the output of the following code?

```
t = (1, 2, 3, 4)
t[2] = 10
print(t)
```

- | | |
|-------------|--------------|
| a. 1,2,10,4 | b. 1,10,2,4 |
| c. Error | d. 1,10,10,4 |

4. What will be the output of the following code?

```
a = ((1,2),)*7
print(len(a[3:6]))
```

- | | |
|------|----------|
| a. 2 | b. 3 |
| c. 4 | d. Error |

5. What will be the output of the following program?

```
my_dict={}
my_dict[(1, 2, 3)] = 12
my_dict[(4,5)] = 2
```


6. The elements of tuples are not separated by commas.
7. Indexing and slicing of tuples is similar to that of lists.
8. The index [] operator is used to access the elements of a tuple.
9. The zip() function takes items in a sequence from a number of collections to make a list of tuples.
10. The * operator unpacks a sequence into positional arguments.
11. A dictionary within a dictionary is called a nested dictionary.
12. A for loop can be used to traverse nested dictionaries.
13. Python objects which cannot change their contents are known as mutable data types.
14. Immutable data types consist of int, float, complex, str and tuple.
15. List and dictionaries can change their contents, so they are called mutable.
16. A dictionary holds only two elements for the polynomial $1+x^{50}$.

C. Exercise Questions

1. What is meant by a tuple and how is it created?
2. What are the functions of tuples?
3. Compare tuples and lists.
4. How is a single element using a tuple created?
5. List the inbuilt functions supported by tuples.
6. How is indexing and slicing of tuples done?
7. Which operator is used to access the elements in a tuple?
8. Can a programmer pass a variable to a function? If yes, how?
9. Consider the example of a tuple as follows:

`x = (11, 12, (13, 'Sachin', 14), 'Hii')`

- | | |
|--------------------------|---------------------------|
| a. <code>x[0]</code> | b. <code>x[2]</code> |
| c. <code>x[-1]</code> | d. <code>x[2][2]</code> |
| e. <code>x[2][-1]</code> | f. <code>x[-1][-1]</code> |
| g. <code>x[-1][2]</code> | h. <code>x[0:1]</code> |
| i. <code>x[0:-1]</code> | j. <code>len(x)</code> |
| k. <code>2 in x</code> | l. <code>3 in x</code> |
| m. <code>x[0] = 8</code> | |

Write the output for the expression

10. What is the function of zip()?
11. What is the role of the * operator within the zip() function?
12. Describe the basics of dictionaries.
13. A dictionary named 'Grades' is created as

`Grades = {"Sahil":90,"Abhijeet":65}`

What do the following statements do?

- | | |
|---|--|
| a. <code>print(Grades.keys())</code> | b. <code>print(Grades.values())</code> |
| c. <code>print(len(Grades))</code> | d. <code>Grades["Kuruss"]=99</code> |
| e. <code>Grades["Abhijeet"] += 5</code> | f. <code>del Grades["Abhijeet"]</code> |
| g. <code>print(Grades.items())</code> | |

14. What will be the output of the following code?

```
Set1 = {10, 20, 30, 40}
```

- | | |
|--|---|
| a. <code>S1.issubset({10,20,30,40,50,60})</code> | b. <code>S1.issuperset({20,30,40})</code> |
| c. <code>print(10 in S1)</code> | d. <code>print(101 in S1)</code> |
| e. <code>print(len(S1))</code> | f. <code>print(max(S1))</code> |
| g. <code>print(sum(s1))</code> | |

15. Show the output of the following code.

```
S1 = { 'A' , 'B' , 'C' }
```

```
S2 = { 'C' , 'D' , 'E' }
```

- | | |
|---|--|
| a. <code>print(S1.union(S2))</code> | b. <code>print(S1.intersection(S2))</code> |
| c. <code>print(S1.difference (S2))</code> | d. <code>print(S1.symmetric_difference(S2))</code> |
| e. <code>print(S1 ^ S2)</code> | f. <code>print(S1 S2)</code> |
| g. <code>print(S1 & S2)</code> | |

16. What will be the output of the following code?

```
T = (10, 34, 22, 87, 90)
```

- | | |
|-------------------------------|-------------------------------|
| a. <code>print(t)</code> | b. <code>t[0]</code> |
| c. <code>print(t[0:4])</code> | d. <code>print(t[:-1])</code> |

17. How can all keys and values of a dictionary be traversed?

18. How are nested dictionaries created?

19. How can a polynomial be represented using dictionaries?

PROGRAMMING ASSIGNMENTS

1. Write a function which takes a tuple as a parameter and returns a new tuple as the output, where every other element of the input tuple is copied, starting from the first one.

```
T = ('Hello', 'Are', 'You', 'Loving', 'Python?')
```

```
Output_Tuple = ('Hello', 'You', 'Python?')
```

2. Write a function called `how_many`, which returns the sum of the number of values associated with a dictionary.

```
T= animals = { 'L': ['Lion'], 'D': ['Donkey'], 'E': ['Elephant'] }
```

```
>>>print(how_many(animals))
```

```
3
```

3. Write a function 'biggest' which takes a dictionary as a parameter and returns the key corresponding to the entry with the largest number of values associated with it.

```
>>>animals = { 'L': ['Lion'], 'D': ['Donkey', 'Deer'], 'E': ['Elephant'] }
```

```
>>>biggest(animals)
```

```
>>>d #Since d contains two values
```

4. Write a function `Count_Each_vowel` which accepts string from a user. The function should return a dictionary which contains the count of each vowel.

```
>>> Count_Each_vowel("HELLO")
```

```
>>>{'H':1, 'E':1, 'L':2 , 'O':2}
```



UNIT 4

- Functions

- Modules, Packages and Introduction to PIP

8

Functions

CHAPTER OUTLINE

8.1 Introduction	8.5 The Local and Global Scope of a Variable
8.2 Syntax and Basics of a Function	8.6 The <code>return</code> Statement
8.3 Use of a Function	8.7 Recursive Functions
8.4 Parameters and Arguments in a Function	8.8 The Lambda Function

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Describe the necessity and importance of functions in programming languages
- Invoke functions with actual parameters and write a program by invoking a function using keyword or positional arguments
- Use local and global scope of a variable appropriately
- Define recursive function and its implementation with programs
- Write functions that return multiple values with programs

8.1 INTRODUCTION

It is difficult to prepare and maintain a large-scale program and the identification of the flow of data subsequently gets harder to understand. The best way to create a programming application is to divide a big program into small modules and repeatedly call these modules.

With the help of functions, an entire program can be divided into small independent modules (each small module is called a function). This improves the code's readability as well as the flow of execution as small modules can be managed easily.

8.2 SYNTAX AND BASICS OF A FUNCTION

A function is a self-contained block of one or more statements that performs a special task when called. The syntax for function is given as follows:

```
def name_of_function(Parameters): ← Function Header
    statement1
    statement2
    statement3
    .....
    .....
    statementN } ← Function Body
```

The syntax for the Python function contains a **header** and **body**. The function header begins with the '**def**' keyword. The **def** keyword signifies the beginning of the function's definition. The name of the function is followed by the **def** keyword. The function header may contain zero or more number of parameters. These parameters are called **formal parameters**. If a function contains more than one parameter then all the parameters are separated by commas. A function's body is a block of statements. The statements within the function's body define the actions that the function needs to perform.

A simple example for creating a function is explained in the following program.

PROGRAM 8.1 | Write a program to create a function having a name, **display**. Print the message, "Welcome to Python Programming" inside the function.

```
def Display():
    print("Welcome to Python Programming ")
Display()           #call function
```

Output

```
Welcome to Python Programming
```

Explanation In the above program, a function having the name **display()** is created. This function takes no parameters. The body of the function contains only one statement. Finally, function **display()** is called to print the message "Welcome to Python Programming" within the block of the function.

PROGRAM 8.2 | Write a program to prompt the name of a user and print the welcome message, "Dear Name_ of_user Welcome to Python Programming!!!"

```
def print_msg():
    str1=input("Please Enter Your Name:")
    print("Dear ",str1," Welcome to Python Programming ")
print_msg() #call function
```

Output

```
Please Enter Your Name: Virat
Dear Virat Welcome to Python Programming
```

Explanation The function named `print _msg()` is created. Initially, the function `print _msg()` is called and the control of the program passes to the called function `print _msg()`. The function reads the name of the user by making use of the `input` reserved keyword and finally the welcome message is printed.

8.3 USE OF A FUNCTION

A programmer wants to find the sum of numbers starting from 1 to 25, 50 to 75 and 90 to 100. Without functions, he/she will write the code in the following manner.

PROGRAM 8.3 | Write a program to add the sum of digits from 1 to 25, 50 to 76 and 90 to 101 using three different for loops.

```
sum=0
for i in range(1,26):
    sum=sum+i
print('Sum of integers from 1 to 25 is:',sum)

sum=0
for i in range(50,76):
    sum=sum+i
print('Sum of integer from 50 to 76 is:',sum)

sum=0
for i in range(90,101):
    sum=sum+i
print('Sum of integer from 90 to 100 is:',sum)
```

Output

```
Sum of integers from 1 to 25 is: 325
Sum of integer from 50 to 76 is: 1625
Sum of integer from 90 to 100 is: 1045
```

The programmer has created the above code. Observe that the code to compute the sum of numbers is conventional. However, there is a slight difference in the range of numbers, i.e. starting integers and ending integers. Here, all the three for loops contain a different range, i.e. from 1 to 26, 50 to 76 and 90 to 101. Thus, by observing the above code, we can say that it would be better if we could simply write the common code once and then use it repeatedly. A programmer can accomplish this by defining function and using it repeatedly. The code above can be simplified and written using functions as shown in Program 8.4.

PROGRAM 8.4 | Write a program to illustrate the use of functions.

```
def sum(x,y):
    s=0;
    for i in range(x,y+1):
        s=s+i
    print('Sum of integers from ',x,' to ',y,' is ',s)
sum(1,25)
sum(50,75)
sum(90,100)
```

Explanation The function named sum is created with two parameters 'x' and 'y'. Initially, the function invokes the first function call, i.e. **sum(1, 25)** to compute the sum of numbers from 1 to 25. After computing the sum of numbers from 1 to 25 the control passes to the next function call, i.e. **sum(50, 75)**. After computing the sum of integers from 50 to 75, the third function is finally called, i.e. **sum(90,100)**.

Thus, a programmer can effectively make use of functions to write this program.

- If a programmer wants to perform a task repetitively, then it is not necessary to re-write the particular block of the program repeatedly. A particular block of statements can be shifted in a user-defined function. The function defined can be then called any number of times to perform a task.
- Large programs can be reduced to smaller ones using functions. It is easy to debug, i.e. find out the errors in it and hence, it also increases readability.

8.4 PARAMETERS AND ARGUMENTS IN A FUNCTION

Parameters are used to give inputs to a function. They are specified with a pair of parenthesis in the function's definition. When a programmer calls a function, the values are also passed to the function.

While parameters are defined by names that appear in the function's definition, arguments are values actually passed to a function when calling it. Thus, parameters define what types of arguments a function can accept.

Let us consider the example of passing parameters to a function given as follows and use it to differentiate between argument and parameter.

Example

```
def printMax(num1,num2):
    Statemen1
    Statemen2
    .....
    .....
    StatementN
    } #Define a Function

printMax(10,20) ← Call a function (Invoke)
```

In the above example, `printMax(num1, num2)` has two parameters, viz. **num1** and **num2**. The parameters **num1** and **num2** are also called formal parameters. A function is invoked by calling the name of the function, i.e. `printMax(10,20)`, where 10, 20 are the actual parameters. Actual parameters are also called arguments. **num1** and **num2** are the parameters of a function. Program 8.5 demonstrates the use of parameters and arguments in a function.

PROGRAM 8.5 | Write a program to find the maximum of two numbers.

```
def printMax(num1,num2): #Function Definition
    print(" num1 = ",num1)
    print(" num2 = ",num2)

    if num1>num2:
        print("The Number ",num1," is Greater than ",num2)
    elif num2>num1:
        print("The Number ",num2," is Greater than ",num1)
    else:
        print(" Both Numbers ",num1," and ",num2," are equal")
printMax(20,10) #call to function printMax
```

Output

```
num1 = 20
num2 = 10
The Number 20 is Greater than 10
```

Explanation In the above program we have defined a function `printMax()`. The function contains two parameters, viz. **num1** and **num2**. The function `printMax()` is called by passing the values as arguments to the function. The statement `printMax(10, 20)` causes the value of 10 and 20 to be assigned to parameters **num1** and **num2**, respectively. Finally, based on the values of **num1** and **num2** within the function, greatest of the two numbers is calculated and displayed.

PROGRAM 8.6 | Write a program to find the maximum of two numbers.

```
def calc_factorial(num):
    fact=1
    print(" Entered Number is: ",num)
    for i in range(1,num+1):
        fact=fact*i
    print("Factorial of Number ",num," is  = ",fact)
number=int(input("Enter the Number:"))
calc_factorial(number)
```

Output

```
Enter the Number:5
Entered Number is:  5
Factorial of Number 5  is  =  120
```

8.4.1 Positional Arguments

Consider the question—If there are more than one parameters, how does Python know which argument in the call statement has to be assigned to which parameter?

The answer is quite simple. The parameters are assigned by default according to their position, i.e. the first argument in the call statement is assigned to the first parameter listed in the function definition. Similarly, the second argument in the call statement is assigned to the second parameter listed in the function's definition and so on.

Consider a simple example to demonstrate the use of positional arguments.

Example

```
def Display(Name,age):
    print("Name = ",Name,"age = ",age)
Display("John",25)
Display(40,"Sachin")
```

In the above example, the evaluation of statement **Display("John",25)** prints the result as Name = John and age = 25. However, the statement **Display(40,"Sachin")** has a different meaning. It passes 40 to name and Sachin to age. It means the first argument binds to the first parameter and the second argument binds to the second parameter. This style of matching up arguments and parameter is called **positional argument style** or **positional parameter style**.

In the above example, the function definition **Display(Name, age)** contains two parameters. Thus, the call is made to function **Display()** by passing exactly two parameters.

What will be the output of the following program?

```
def Display(Name,age):
    print("Name = ",Name,"age = ",age)
Display("John")
```

Output

Prints the error message

```
Traceback (most recent call last):
File "C:\Python34\keyword_1.py", line 3, in <module>
Display("John")TypeError: Display() missing 1 required positional
argument: 'age'
```

Explanation In the above program, there is no output due to an error. The third line of the program contains the statement `Display("John")`, i.e. the statement has made a call to function `Display(name, age)`. As the function call contains lesser number of arguments as compared to the function definition, Python will report a missing argument error.



Note: Python will show an error when an incorrect number of arguments are passed to the function call. The arguments must match the parameters in order, number and type as defined in the function.

8.4.2 Keyword Arguments

An alternative to positional argument is keyword argument. If a programmer knows the parameter name used within the function then he/she can explicitly use the parameter name while calling the function. A programmer can pass a keyword argument to a function by using its corresponding parameter name rather than its position. This can be done by simply typing **Parameter_name = value** in the function call.

Syntax to call a function using keyword argument is:

Name_of_Function(pos_args, keyword1=value, keyword2=value2.....)

PROGRAM 8.7 | Write a simple program on keyword argument.

```
def Display(Name,age):
    print("Name = ",Name,"age = ",age)
Display(age=25,Name="John") #Call function using keyword
                           arguments
```

Output

Name = John age = 25

Explanation Thus, in the above program, the statement `Display(age=25,Name="John")` passes the value 25 to the parameter 'age' and 'John' to the parameter 'Name'. It means arguments can appear in any order using keyword arguments.

Precautions for Using Keyword Arguments

1. A positional argument cannot follow a keyword argument.

Example: Consider the function definition,

```
def Display(num1,num2):
```

Thus, a programmer can invoke the above `Display()` function as:

```
Display(40,num2=10)
```

But, he/she cannot invoke the function as:

```
Display(num2=10,40)
```

because the positional argument **40** appears after the keyword argument **num2=10**.

2. A programmer cannot duplicate an argument by specifying it as both, a positional argument and a keyword argument.

Example: Consider the function definition,

```
def Display(num1,num2):
```

Thus, a programmer cannot invoke the above `Display()` function as

```
Display(40,num1=40) #Error
```

because he/she has specified multiple values for parameter `num1`.

8.4.3 Parameter with Default Values

Parameters within a function's definition can have default values. We can provide default value to a parameter by using the assignment (=) operator.

PROGRAM 8.8 | Write a program to illustrate the use of default values in a function's definition.

```
def greet(name,msg="Welcome to Python!!"):
    print(" Hello ",name,msg)
greet("Sachin")
```

Output

```
Hello Sachin Welcome to Python!!
```

In the above example, the function `greet()` has the parameter **name**. The parameter **name** does not have any default value and is mandatory during a function call. On the other hand, the parameter **msg** has a default value as **"Welcome to Python!!"**. Hence, it is optional during

a function call. If a value is provided, it will overwrite the default value. Here are some valid function calls to this function.

#Test case 1

```
>>> greet("Amit")
```

Output

```
Hello Amit Welcome to Python!!
```

#Test case 2

```
>>> greet("Bill Gates","How are You?")
```

Output

```
Hello Bill Gates How are You?
```

The above example has two test cases. In the first test case, only one argument is passed to the function `greet()` during the function call. And the second parameter is not passed. In such a case, Python uses the default value of a parameter specified during function definition. But in case of test case 2, both the parameters `greet("Bill Gates","How are You?")` are passed during the function call. In such a situation, the new argument value overwrites the default parameter value.



Note: During a function's definition, any number of parameters in a function can have default values. But once we have a default value to a parameter, all the parameters to its right must also have default values. For example, if we define a function's definition as:

```
def greet(msg="Welcome to Python!!", name):      #Error
```

Python will give the error as:

Syntax Error: Non-default argument follows default argument

PROGRAM 8.9 | Write a program to calculate the area of a circle using the formula:

$$\text{Area of Circle} = \pi \times (r)^2$$

Declare the default parameter value of pi as 3.14 and radius as 1.

```
def area_circle(pi=3.14,radius=1):
    area=pi*radius*radius
    print("radius=",radius)
    print(" The area of Circle  = ",area)
area_circle()
area_circle(radius=5)
```

Output

```
radius= 1
```

(Contd.)

```
The area of Circle = 3.14
radius= 5
The area of Circle = 78.5
```

What will be the output of the following program?

```
def disp_values(a,b=10,c=20):
    print(" a = ",a," b = ",b,"c= ",c)
disp_values(15)
disp_values(50,b=30)
disp_values(c=80,a=25,b=35)
```

Output

```
a = 15  b = 10  c= 20
a = 50  b = 30  c= 20
a = 25  b = 35  c= 80
```

Explanation In the above program, the function named **disp_values** has one parameter without a default argument value, followed by two parameters with default argument values.

During the first function call **disp_values(15)**, parameter a gets the value 15 and parameters b and c get the default values 10 and 20, respectively.

During the function call **disp_values(50,b=30)**, parameter a gets the value 50, parameter b gets the value 30, i.e. the value of b is overwritten and parameter c gets the default value 20.

During the function call **disp_values(c=80,a=25,b=35)**, the default values of parameters b and c are replaced by the new values 35 and 80, respectively.

8.5 THE LOCAL AND GLOBAL SCOPE OF A VARIABLE

Variables and parameters that are initialised within a function including parameters, are said to exist in that function's local scope. Variables that exist in local scope are called **local variables**. Variables that are assigned outside functions are said to exist in global scope. Therefore, variables that exist in global scope are called **global variables**.

PROGRAM 8.10 | Write a program to show local scope vs global scope.

```
p = 20          #global variable p
def Demo():
    q = 10      #Local variable q
    print('The value of Local variable q:',q)
    #Access global variable p within this function
    print('The value of Global Variable p:',p)
Demo()
```

(Contd.)

```
#Access global variable p outside the function Demo()
print('The value of global variable p:',p)
```

Output

```
The value of Local variable q: 10
The value of Global Variable p: 20
The value of global variable p: 20
```

Explanation In the above example, we have created one **local variable 'q'** and one **global variable 'p'**. As global variables are created outside all functions and are accessible to all functions in their scope, in the above example as well the global variable **'p'** is accessed from the function **Demo()** and it is also accessed outside the function.

Local Variables Cannot be Used in Global Scope

PROGRAM 8.11 | Write a program to access a local variable outside a function.

```
def Demo():
    q = 10          #Local variable q
    print('The value of Local variable q:',q)
Demo()
#Access local variable q outside the function Demo()
print('The value of local variable q:',q)    #Error
```

Output

```
The value of Local variable q: 10
Traceback (most recent call last):
  File "C:/Python34/loc1.py", line 6, in <module>
    print('The value of local variable q:',q)    #Error
NameError: name 'q' is not defined
```

Explanation The local variable **'q'** is defined within the function **Demo()**. The variable **'q'** is accessed from the function **Demo()**. The scope of a local variable lies within the block of the function, i.e. it starts from its creation and continues up to the end of the function. Therefore, any attempt to access the variable from outside of the function causes an error.



Note: Accessing a local variable outside the scope will cause an error.

8.5.1 Reading Global Variables from a Local Scope

Consider the following program where global variables are read from a local scope.

PROGRAM 8.12 | Write a program where global variables are read from a local scope.

```
def Demo():
    print(S)
S='I Love Python'
Demo()
```

Output

```
I Love Python
```

Explanation Before calling the function `Demo()`, the variable 's' is defined as a string, "I Love Python". However, the body of the function `Demo()` contains only one statement `print(s)` statement. As there is no local variable 's' defined within the function `Demo()`, the `print(s)` statement uses the value from the global variable. Hence, the output of the above program will be 'I Love Python'.

8.5.2 Local and Global Variables with the Same Name

What will be the output of the above program if we change the value of 's' inside the function `Demo()`? Will it affect the value of the global variable? Program 8.13 demonstrates the change in value 's' within the function `Demo()`.

PROGRAM 8.13 | Write a program to change the value 's' within the function.

```
def Demo():
    S='I Love Programming'
    print(S)
S='I Love Python'
Demo()
print(S)
```

Output

```
I Love Programming
I Love Python
```

Explanation As we know, the scope of a local variable lies within the block of a function. Initially, the value of 's' is assigned as 'I Love Python'. But after calling the function `Demo()`, the value of 's' is changed to 'I Love Programming'. Therefore, the print statement within the function `Demo()` will print the value of the local variable 's', i.e. 'I Love Programming'. Whereas the print statement after the `Demo()` statement, will print the old value of the variable 's', i.e. 'I Love Python'.

8.5.3 The Global Statement

Consider a situation where a programmer needs to modify the value of a global variable within a function. In such a situation, he/she has to make use of the **global** statement. The following program demonstrates the use of the **global** statement.

PROGRAM 8.14 | Write a program without using the global statement.

```
a = 20
def Display():
    a = 30
    print(' The value of a in function:',a)
Display()
print('The value of an outside function:',a)
```

Output

```
The value of a in function: 30
The value of an outside function: 20
```

Explanation In the above program, we have assigned the value of an outside function as 20. By chance, a programmer uses the same name, i.e. 'a' inside the function. But in this case the variable 'a' within the function is local to the function. Therefore, any changes to the value associated with the name inside the function will change the value of the local variable itself and not the value of the global variable 'a'.

PROGRAM 8.15 | Write a program using the global statement.

```
a = 20
def Display():
    global a
    a = 30
    print(' The value of a in function:',a)
Display()
print('The value of an outside function:',a)
```

Output

```
The value of a in function: 30
The value of an outside function: 30
```

Explanation The program demonstrates the use of the **global** keyword. The **global** keyword has been used before the name of the variable to change the value of the local variable. Since the value of the global variable is changed within the function, the value of 'a' outside the function will be the most recent value of 'a'.

8.6 THE return STATEMENT

The return statement is used to **return a value from the function**. It is also used to return from a function, i.e. break out of the function.

PROGRAM 8.16 | Write a program to return the minimum of two numbers.

```
def minimum(a,b):
    if a<b:
        return a
    elif b<a:
        return b
    else:
        return "Both the numbers are equal"
print(minimum(100,85))
```

Output

8 is minimum

Explanation The minimum function returns the minimum of the two numbers supplied as parameters to a function minimum. It uses simple if..elif..else statement to find the minimum value and then returns that value.

PROGRAM 8.17 | Write a function calc_Distance(x1, y1, x2, y2) to calculate the distance between two points represented by Point1(x1, y1) and Point2 (x2, y2). The formula for calculating distance is:

$$\text{Distance} = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

```
import math
def calc_Distance (x1, y1, x2, y2):
    print(" x1 = ",x1)
    print(" x2 = ",x2)
    print(" y1 = ",y1)
    print(" y2 = ",y2)
    dx=x2-x1
    dx=math.pow(dx,2)
    dy=y2-y1
    dy=math.pow(dy,2)
    z = math.pow((dx + dy), 0.5)
    return z
print("Distance = ",(format(calc_Distance(4,4,2,2),".2f")))
```

Output

```
x1 = 4
x2 = 2
y1 = 4
y2 = 2
Distance = 2.83
```

PROGRAM 8.18 For a quadratic equation in the form of ax^2+bx+c , the discriminant D , is $b^2 - 4ac$. Write a function to compute the discriminant D , that returns the following output depending on the discriminant D .

if $D > 0$: The Equation has two Real Roots

if $D = 0$: The Equation has one Real Root

if $D < 0$: The Equation has two Complex Roots

```
def quad_D(a,b,c):
    d=b*b-4*a*c
    print("a = ",a)
    print("a = ",b)
    print("a = ",c)
    print("D = ",d)
    if d>0:
        return "The Equation has two Real Roots"
    elif d<0:
        return "The Equation has two Complex Roots"
    else:
        return "The Equation has one Real Root"
print(quad_D(1,2,5))
```

Output

```
a = 1
a = 2
a = 3
D = -8
The Equation has two Complex Roots
```



Note: The return statement without a value is equivalent to return 'None'. Where, 'None' is a special type in Python that represents nothingness.

PROGRAM 8.19 Write a program to pass the radius of a circle as a parameter to a function **area_of_circle()**. Return the value **none** if the value of the radius is negative or return the area of the circle.

```
def area_of_Circle(radius):
    if radius<0:
        print(" Try Again, Radius of circle cannot be Negative ")
        return
    else:
        print("Radius = ",radius)
```

(Contd.)

```

        return 3.1459*radius**radius
print("Area of Circle =",area_of_Circle(2))

```

Output

```

Radius = 2
Area of Circle = 12.5836

```

Explanation In the above program, the user has to pass the radius of the circle as a parameter to the function `area_of_circle()`. If the radius of the circle is positive then it calculates and returns the area of the circle. Whereas, if the entered radius of the circle is negative, it returns a none value, i.e. it returns nothing.

What will be the output of the above program?

```

def calc_abs(x):
    if x<0:
        return -x
    elif x>0:
        return x
print(calc_abs(0))

```

Output

```

None

```

Explanation The above piece of code is incorrect because when the user has passed the value 0 as a parameter to the function `calc_abs()`, the value of `x` happened to be 0. Then neither condition is true and the function ends without executing any return statement. In such a situation, the function returns a special value called **None**.

8.6.1 Returning Multiple Values

It is possible to return multiple values in Python.

PROGRAM 8.20 | Write a function `calc_arith_op(num1, num2)` to calculate and return at once the result of arithmetic operations such as addition and subtraction.

```

def calc_arith_op(num1, num2):
    return num1+num2, num1-num2    #Return multiple values
print(" ",calc_arith_op(10,20))

```

Output

```

(30, -10)

```

Explanation In the above program, two parameters, viz. num1 and num2 are passed to a function `calc_arith_op()`. Within the body of the function, the **single return** statement computes the addition and subtraction of the two numbers. Finally, the single return statement returns the result of both the arithmetic operations, viz. addition and subtraction.

8.6.2 Assign Returned Multiple Values to Variable(s)

It is also possible for a function to perform certain operations, return multiple values and assign the returned multiple values to a multiple variable.

PROGRAM 8.21 | Write a program to return multiple values from a function.

```
def compute(num1):
    print("Number = ", num1)
    return num1*num1, num1*num1*num1
square, cube=compute(4)
print("Square = ", square, "Cube = ", cube)
```

Output

```
Number = 4
Square = 16 Cube = 64
```

Explanation The number is passed to the function `compute()`. The return statement calculates the square and cube of a passed number. After computation, it returns both the values simultaneously. The returned square of a number is assigned to a variable `square` and the returned cube of a number is assigned to a variable `cube`.

8.7 RECURSIVE FUNCTIONS

So far, we have seen that it is legal for one function to call another function. In programming, there might be a situation where a function needs to invoke itself. Python also supports the recursive feature, which means that a function is repetitively called by itself. Thus, a function is said to be recursive **if a statement within the body of the function calls itself**.

Let us consider a simple example of recursion. Suppose we want to calculate the factorial value of an integer. We know that the factorial of a number is the product of all the integers between 1 and that number, i.e. $n!$ is defined as $n * (n-1)!$.

Consider the following example.

Formula to calculate the factorial of a number $(n)! = n*(n-1)!$

$$\begin{aligned}
 5! &= 5*(4)! \\
 &= 5*4*(3)! \\
 &= 5*4*3*(2)! \\
 &= 5*4*3*2*(1) \\
 &= 120
 \end{aligned}$$

PROGRAM 8.22 | Calculate the factorial of a number using recursion.

```
def factorial(n):
    if n==0:
        return 1
    return n*factorial(n-1)
print(factorial(5))
```

Output

```
120
```

Explanation In the above program, `factorial()` is a recursive function. The number is passed to function `factorial()`. When the function `factorial` is executed, it is repeatedly invoked by itself. Every time a function is invoked, the value of 'n' is reduced by one and multiplication is carried out. The recursion function produces the number 5, 4, 3, 2 and 1. The multiplication of these numbers is carried out and returned. Finally, the print statement prints the factorial of the number.

PROGRAM 8.23 | Write a recursive function which computes the n^{th} Fibonacci number. Fibonacci numbers are defined as:

```
Fib(0)= 1,
Fib(1)= 1
Fib(n)= Fib(n-1)+Fib(n-2).
```

Write this as a Python code and then find the 8th Fibonacci number.

```
def fib(n):
    if n==0:
        return 1
    if n==1:
        return 1
    return fib(n-1)+fib(n-2)
print(" The Value of 8th Fibonacci number = ",fib(8))
```

Output

```
The Value of 8th Fibonacci number = 34
```

8.8 THE LAMBDA FUNCTION

Lambda functions are named after the Greek letter λ (lambda). These are also known as **anonymous functions**. Such kind of functions are not bound to a name. They only have a code to execute that which is associated with them. The basic syntax for a lambda function is:

Name = lambda(variables): Code

Let us consider a simple example which calculates the cube of a number using simple concepts of a function.

```
>>> def func(x):
    return x*x*x
```

```
>>> print(func(3))
27
```

Without the lambda function Now we will calculate the cube of a number using the lambda function.

```
>>> cube = lambda x: x*x*x      #Define lambda function
>>> print(cube(2))             #Call lambda function
8
```

Using the lambda function Thus, in the above example, both the functions `func()` and `cube()` do exactly the same thing. The statement `cube = lambda x: x*x*x` creates a **lambda** function called `cube`, which takes a single argument and returns the cube of a number.



Note: (a) A lambda function does not contain a return statement.
(b) It contains a single expression as a body and not a block of statements as a body.

MINI PROJECT

Calculation of Compound Interest and Yearly Analysis of Interest and Principal Amount

This mini project will use programming features, such as **decision**, **control statements** and **functions** to calculate the interest deposited for a principal amount for some period of time 'n' at some interest 'r'.

Explanation and Calculation of Compound Interest

Compound interest is the addition of interest to the initial principal amount and also to the accumulated interest over preceding periods of a deposit or loan.

Compound interest is different from simple interest. In simple interest, there is no interest on interest. Simply interest is added to the principal amount.

The formula to calculate annual compound interest including principal amount is

$$CI = P * \left(1 + \frac{r}{t}\right)^n - P$$

where,

P = Principal investment amount

r = Annual interest rate

n = Number of years the money is invested

t = Number of times the interest is compounded per year

The formula to calculate interest if it is compounded once per year is

$$I = P * (1 + r)^n \text{----- } \{A\}$$

Thus, 'I' gives future values of an investment or loan which is compound interest plus the principal. So, we are going to use formula 'A'.

Example

Let principal (P) amount = ₹10,000

Rate (R) of interest = 5

Number of Years = 7

Value of compound interest per year (t) = 1

We will use the above formula 'A' to calculate the interest accumulated each year.

Year	Starting Balance	Interest	Ending Balance
1	10000.00	500.00	10500.00
2	10500.00	525.00	11025.00
3	11025.00	551.25	11576.25
4	11576.25	578.81	12155.06
5	12155.06	607.75	12762.82
6	12762.82	638.14	13400.96
7	13400.96	670.05	14071.00

Algorithm to Calculate Compound Interest

- ⊙ **STEP 1:** Read the principal amount, rate of interest and number of years the amount is to be deposited. (Assuming interest is compounded once per year).
- ⊙ **STEP 2:** Pass the principal, rate of interest and the number of years to the function named **Calculate_Compound_Interest()**.
- ⊙ **STEP 3:** Iterate **for loop** for 'n' number of times to calculate interest generated per year by using the formula for compound interest as discussed above.
- ⊙ **STEP 4:** Display the final compound interest.

PROGRAM STATEMENT | Write a program to calculate compound interest for principal amount as ₹10,000, at rate of interest as 5% and number of years the amount is deposited as 7 years.

```
def Calculate_Compound_Interest(p,n,r):
    print('StartBalance\t','\tInterest\t','Ending Balance')
    total = 0
```

(Contd.)

```

x= r/100
tot = 0
for i in range(1,n+1):
    z_new = p*(1 + x) **i - p
    z_old = p*(1 + x)**(i-1) - p
    tot = tot + (z_new - z_old)
    if(i == 1):
        print('{0:.2f}\t'.format(p),end='')
        print('\t{0:.2f}\t'.format(z_new - z_old),end='')
        print('\t\t{0:.2f}\t'.format(z_new+p))
    else:
        print('{0:.2f}\t'.format(p+z_old),end='')
        print('\t{0:.2f}\t'.format(z_new - z_old ),end='')
        print('\t\t{0:.2f}\t'.format(z_new+p))
    print('Total Interest Deposited:Rs{0:.2f}'.format(tot))
p = int(input('Enter the Principal amount:'))
r = int(input('Enter the rate of interest:'))
n = int(input('Enter number of year:'))
Calculate_Compund_Interest(p,n,r)

```

Output

```

Enter the Principal amount:10000
Enter the rate of interest:5
Enter number of year:7

```

Start Balance	Interest	Ending Balance
10000.00	500.00	10500.00
10500.00	525.00	11025.00
11025.00	551.25	11576.25
11576.25	578.81	12155.06
12155.06	607.75	12762.82
12762.82	638.14	13400.96
13400.96	670.05	14071.00

```
Total Interest Deposited: Rs 4071.00
```

In the above program, initially principal amount, rate of interest and number of years are read from the user. The same values are passed as a parameter to the function **Calculate_Compund_Interest()**. The for loop is iterated for **n** number of times to calculate the annual interest generated per year. The difference between **Z_new** and **Z_old** in above program gives the interest generated per year. At last, the compound interest is displayed.

SUMMARY

- ◆ A function is a self-contained block of one or more statements that perform a special task when called.
- ◆ A function's definition in Python begins with the `def` keyword followed by the function's name, parameter and body.
- ◆ The function header may contain zero or more number of parameters.
- ◆ Parameters are the names that appear in a function's definition.
- ◆ Arguments are the values actually passed to a function while calling a function.
- ◆ Arguments to a function can be passed as positional or keyword arguments.
- ◆ The arguments must match the parameters in order, number and type as defined in the function.
- ◆ A variable must be created before it is used.
- ◆ Variables defined within the scope of a function are said to be local variables.
- ◆ Variables that are assigned outside of functions are said to be global variables.
- ◆ The `return` statement is used to return a value from a function.
- ◆ Functions in Python can return multiple values.
- ◆ Python also supports a recursive feature, i.e. a function can be called repetitively by itself.

KEY TERMS

- ⇒ **The `def` Keyword:** Reserved word to define a function
- ⇒ **Positional Arguments:** By default, parameters are assigned according to their position
- ⇒ **Keyword Arguments:** Use syntax `keyword = Value` to call a function with keyword arguments
- ⇒ **Local and Global Scope of a Variable:** Describes two different scopes of a variable
- ⇒ **The `Return` Keyword:** Used to return single or multiple values
- ⇒ **Lambda:** An anonymous function

REVIEW QUESTIONS

A. Multiple Choice Questions

1. A variable defined outside a function is referred to as
 - a. Local variable
 - b. Only variable
 - c. Global variable
 - d. None of the above
2. Which of the following function headers is correct?
 - a. `def Demo(P, Q = 10):`
 - b. `def Demo(P=10,Q = 20):`
 - c. `def Demo(P=10,Q)`
 - d. Both a and c
3. What will be the output of the following program?


```
x = 10
def f():
    x = x + 10
    print(x)
f()
```

- a. 20
 - b. 10
 - c. Error: Local variable X referenced before assignment
 - d. None of the above
4. What will be the output of the following program?
- ```
def Func_A(P = 10, Q = 20):
 P = P + Q
 Q = Q + 1
 print(P, Q)
Func_A(Q = 20, P = 10)
```
- a. Error: P and Q are not defined.
  - b. 20 10
  - c. 10 20
  - d. 30 21
5. What will be the output of the following program?
- ```
Def test():
    x=10
# Main Program #
x = 11
test()
print(x)
```
- a. 10
 - b. 11
 - c. Garbage value
 - d. None of the above
6. If a function does not return any value, then by default which type of value is returned by the function?
- a. int
 - b. double
 - c. str
 - d. None
7. What will be the output of the following program?
- ```
def test():
 global x
 x='A'
Main Program
x = 'Z'
test()
print(x)
```
- a. Z
  - b. A
  - c. Garbage value
  - d. None of the above
8. What will be the output of following program?
- ```
def test(x):
    x = 200
# Main Program #
x = 100
test(x)
print(x)
```

a. 100

b. Garbage value

c. 200

d. None of the above

9. What will be the output of the following program?

```
def test(x):
    p = 90

    # Main Program #
    p = 50
    print(test(p))
```

a. 90

b. 50

c. Error

d. None

10. What will be the output of the following program?

```
def evaluate_expression_1(Z):
    Z = Z + 5
    def evaluate_expression_2(Z):
        print('Hello')
        return Z
    return Z
value = 10
print(evaluate_expression_1(value))
```

a. Hello 10

b. 10

c. 15 Hello

d. 15

11. What will be the output of the following program?

```
def evaluate_expression_1():
    global x
    x = x - 5

    def evaluate_expression_2():
        global x
        return x + 3

    return evaluate_expression_2()
# Main Program #
x = 10
print(evaluate_expression_1())
```

a. 5

b. 8

c. 10

d. 13

12. What will be the output of the following program?

```
def perform_multiplication(Num1, Num2):
    Num2 = Num1 * Num2
    return Num1, Num2
```

```
# Main Program #
Num2, Num1 = perform_multiplication(5,4)
print (Num1, Num2)
```

- | | |
|----------|----------|
| a. 5, 4 | b. 5, 20 |
| c. 20, 5 | d. 4, 5 |

13. What will be the output of the following program?

```
def Display(Designation, Salary):
    print("Designation = ",Designation, "Salary = ",Salary)
Display("Manager",25000)
Display(300000,'Programmer')
```

- Error: Type Mismatch
- Manger 25000
300000 Programmer
- 300000 Programmer
Manger 25000
- None of the above

B. True or False

- A function divides a program in small independent modules.
- The syntax of Python function contains a header and body.
- The function header begins with the definition keyword.
- Parameters are used to give inputs to a function.
- Parameters are specified with a pair of parenthesis in the function's definition.
- In a function, parameters are defined by the names that appear in the function's definition.
- Arguments are values actually passed to a function when calling it.
- The return statement is used to return a value from a function.
- A function invoking itself is called a recursive function.
- A function is said to be recursive if a statement within the body of the function calls itself.

C. Exercise Questions

- What are the advantages of functions?
- What does a function do?
- Write the definition of a function.
- Write the syntax for a function.
- Differentiate between user-defined and library-defined functions.
- How does a function work? Explain how arguments are passed and results are returned?
- What are arguments? How are arguments passed to a function?
- What is the use of a return statement?
- Is it possible to return multiple values from a function?
- What are local and global variables?

PROGRAMMING ASSIGNMENTS

1. Write a function **eval_Quadratic_Equa(a, b, c, x)** which returns the value of any quadratic equation of form

$$ax^2 + bx + c$$
2. Write a function **calc_exp(base, exp)** which computes the exponent of any number, i.e. **base^{exp}**. The function should take two values as base, which can be float or integer. Exp will be an integer greater than 0.
3. Write a function **Calc_GCD_Recurr(a, b)** which calculates the GCD recursively of two numbers. The function should take two positive integers and should return one integer as GCD.

Note: The greatest common divisor (GCD) of two positive integers is the largest integer that divides each of them without a remainder.

Example:

gcd(12, 2) = 2

gcd(6, 12) = 6

gcd(9, 12) = 3

4. Write a function **reverse_number()** to return the reverse of the number entered.

Example:

Reverse_number(1234) displays 4321

5. A four-digit integer is entered through the keyboard. Write a function to calculate the sum of the four-digit number both without recursion and using recursion.
6. A positive integer is entered through the keyboard. Write a function **factors(num)** to obtain the factors of the given numbers.
7. Write a program to define function **dec_bin(num)** to convert the existing decimal number into its equivalent binary number.

Modules, Packages and Introduction to PIP

CHAPTER OUTLINE

9.1 Introduction to Modules	9.5 Installing and Uninstalling Packages via PIP
9.2 Creating and Importing Modules	
9.3 Packages in Python	9.6 Using Python Packages
9.4 Introduction to PIP	

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Learn to create the modules
- Understand the packages
- Learn to install and uninstall packages via PIP
- Learn to use installed packages

9.1 INTRODUCTION TO MODULES

A module is a piece or part of something. In Python, we can say modules are smaller pieces of a bigger program. Each module is a separate file on hard disk. Programmer can take a big program and split it up into more than one module. As Python programs are written in script mode of Python's IDLE, once the code is written, the file is saved by .py extension. In short, the modules are Python's .py files which contain Python codes. In order to use something that is in a module, you first have to tell Python which module you want to use. The Python **import** keyword lets you to include other modules in your program. The reason behind using modules is as follows.

- a. It makes the file smaller which helps programmer to find the things easily into the written code.
- b. Once module is created, it can be used in lot of programs. This feature saves the programmers from starting all over again next time if the programmer would need the same function again.

9.2 CREATING AND IMPORTING MODULES

Writing a module is like writing a simple Python program in a file and saving it in .py extension. Modules contain definitions of functions, classes and variables which can be utilized in other programs. Let us create a simple file, viz. Demo.py.

```
def Display():  
    print('Hello, Welcome all!')
```

If we try to execute the above code, nothing will happen because we have just written the function and it has not been called from elsewhere to perform its action. So, let us create another file named main.py so that we can import the module Demo.py what we have just created and then call the function Display() present in the file Demo.py from a new file, which is main.py. Therefore, we create another file named main.py. In the file main.py, we will make use of import statement to import the module named Demo.py. The contents of the main.py file are as follows.

```
#main.py  
import Demo      #Importing Module named Demo  
demo.Display()   #Call function Display present within Demo.py
```

Output

```
'Hello, Welcome all!'
```

In the above program, we are importing a module, therefore we need to call the function by referencing the module by ".", i.e. the dot notation. Thus, we use the ModuleName.FunctionName() to reference the function present within the module. The statement demo.Display() calls the function Display() from module named demo.py. The above code contains the following two lines.

```
import demo  
demo.Display()
```

We can use the from keyword and replace the above two lines as from Demo import Display.



Note: The module which we have imported and the file, in which we have used the import statement, should be in the same directory. With respect to the above example, Demo.py and main.py should be stored/located at the same location.

Thus, we can get the same output even if we use the from keyword. In the above example, we have seen how a function being present in another file can be called using the import statement. A programmer can use the import statement to import variables and classes present in another file in this manner.

9.3 PACKAGES IN PYTHON

As we know that the modules are smaller pieces of bigger program. Where such module is a separate file on hard disk and each file does contain the Python statements, definitions like functions and class definitions. Thus, the following section is all about handling the multiple modules together aiming to form a package.

A package is basically a directory with Python file and file with the name `_init_.py`. In short, every directory inside the Python path which contains a file named `_init_.py`, will be treated as a package by Python. In those cases, the programmer can put multiple modules into a package. A package can be imported like a “normal” module.

9.3.1 Creating Package

Let us create a simple package with some Python modules. First step is to need to create a folder. The name of the folder will be the name of package which we want to create. Let the name of package be “`My_First_Package`”. The sample directory structure for creating package is shown in Figure 9.1 as follows.

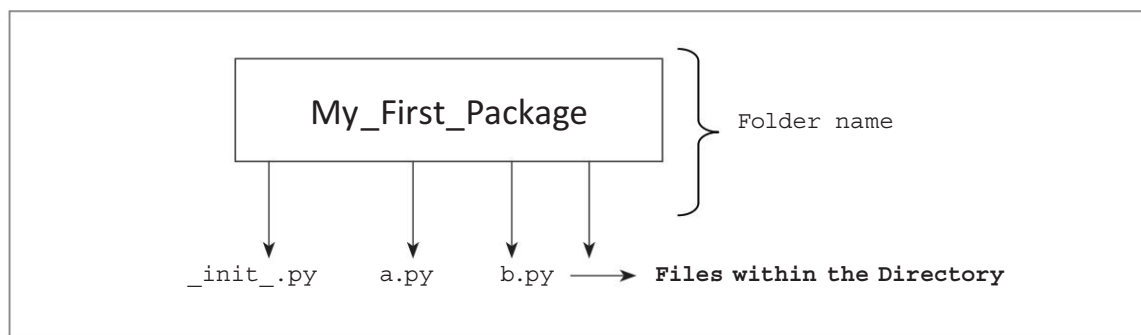


Figure 9.1 Package directory structure



- Note:
- The Package (folder or directory structure) should contain `_init_.py` file.
 - It should also contain one or multiple `.py` files.
 - Apart from `_init_.py` file, all the files will be used as modules and those modules will be imported in `_init_.py` file.

Working Example of Package

- ⊙ **STEP 1:** Let us create a package (folder). The name of the package, say, “`My_First_Package`”.
- ⊙ **STEP 2:** Create `_init_.py` file inside the created package, viz. “`My_First_Package`”. The directory should contain a file named `_init_.py`. This file can be empty or it may contain valid Python code. Let keep it as empty file.
- ⊙ **STEP 3:** Inside the package, let us create two different `.py` files, i.e. `a.py` and `b.py`.

• • • • • • • • • • •

- ```
#a.py
```

```
#b.py
```

Execute both of these files, `a.py` and `b.py`. Make sure that there are no errors present in it.

- In the above example, we have created three files, i.e. `_init_.py`, `a.py` and `b.py`. But the file `_init_.py` is empty. We have used statement from `My_First_Package` and `import a, b` to import the contents of files, `a.py` and `b.py`. Alternatively, we can minimize from `My_First_Package` and `import a, b` statement and automatically load these modules using `_init_.py` file. Thus, the open empty '`_init_.py`' file and insert the following code as follows.

```
>>> My_First_Package.a.call_A()
I am in File a.py
>>> My_First_Package.b.call_B()
I am in File b.py
```

One of the hurdles that new Python developer has to understand about the Python packaging ecosystem. Where PIP is a tool for installing Python packages from “Python Package Index” (PyPI). The PyPI is the repository of software for Python programming language. It helps you to find and install the software being developed and shared by the Python community.

## 9.4.1 PIP Directory Structure

The following is the directory structure where you can find the PIP.

```
Python27 (Base Directory)
|
+-----Scripts (Sub Directory)
|
+-----pip
```

Similarly, following Figure 9.2 shows a snapshot containing PIP into the directory structure.

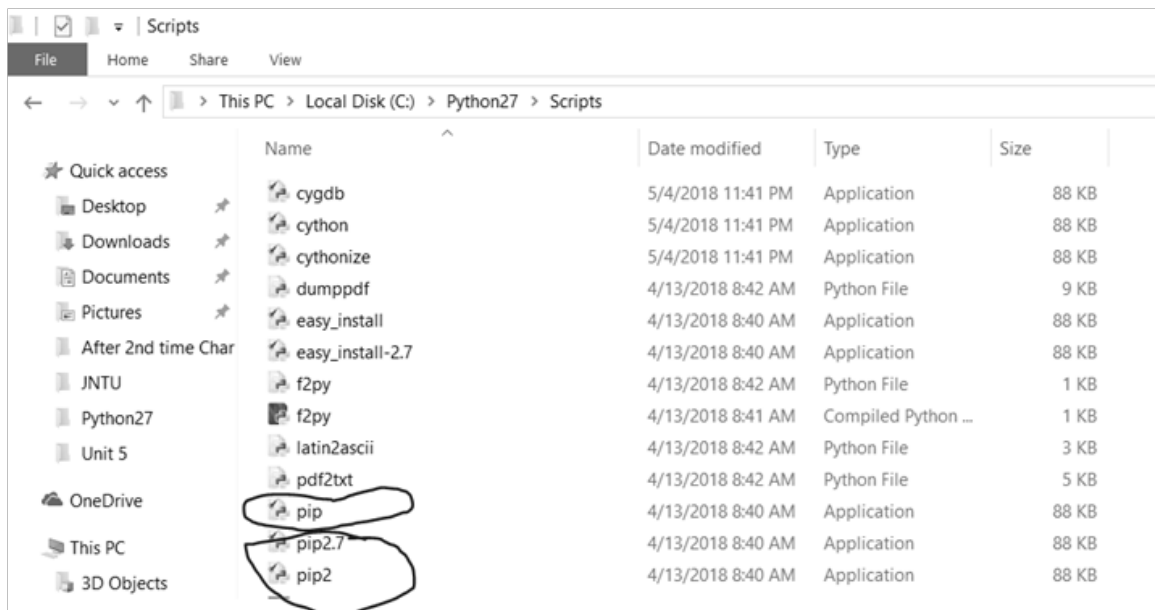
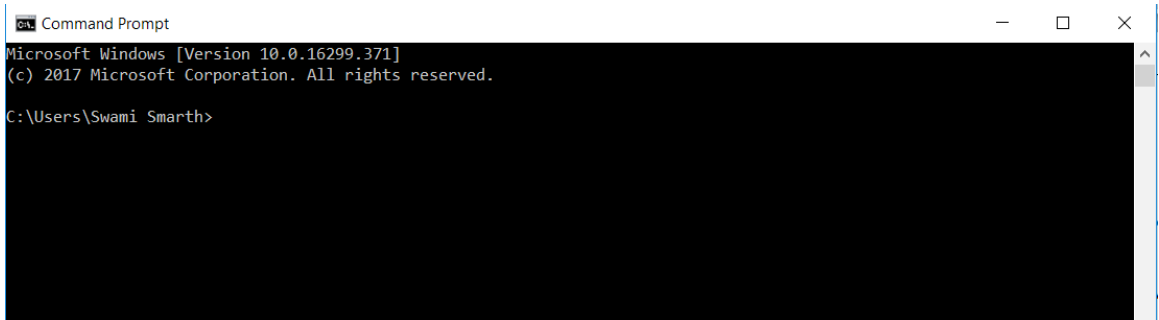


Figure 9.2 PIP Directory Structure

## 9.4.2 Knowing the Version of PIP

If the developer is interested to check the version of PIP, then it can be done in following steps on windows platform.

- ⦿ **STEP 1:** Open CMD (Figure 9.3) (command line interface)—[Press Windows key and in search option, write CMD and press the Tab and Enter.]

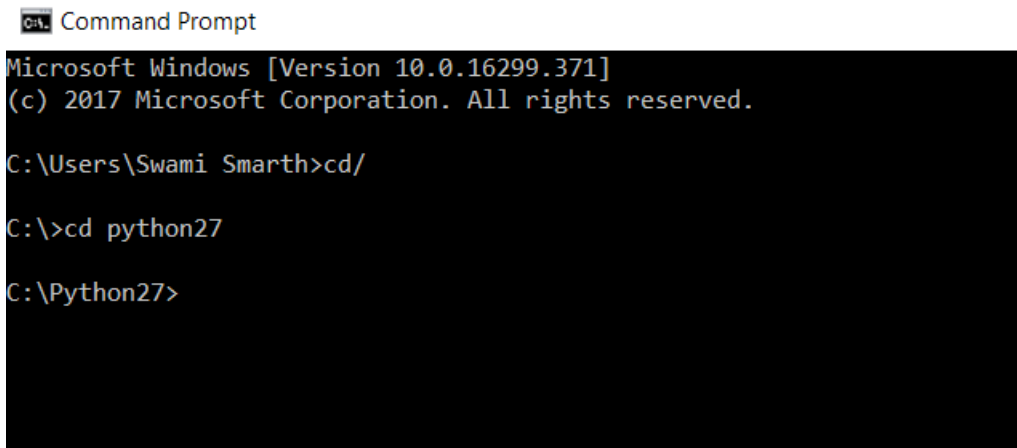


**Figure 9.3** Command Line Interface (CMD)

- ◎ **STEP 2:** Change the directory and go to the base installation directory of Python, i.e. where you have installed Python software. (In my case, I have installed python on C: drive). Type following two commands and press the Tab and Enter (Figure 9.4).

Cd/

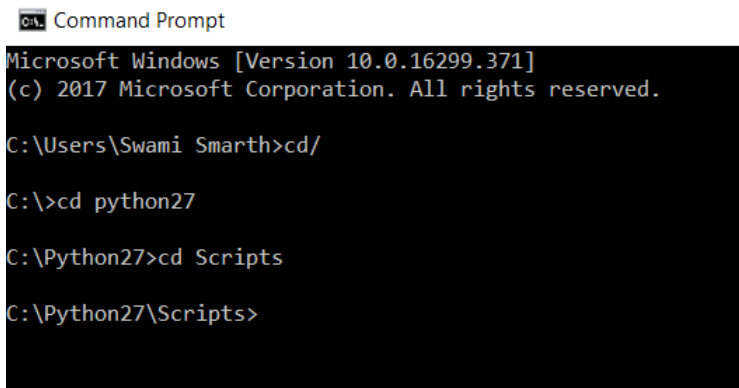
Cd Python27



**Figure 9.4** Root Directory of Python

- ⊙ **STEP 3:** Now, go to the scripts directory inside the root directory. Type following command to go into the script directory (Figure 9.5).

```
cd scripts
```



```

Command Prompt

Microsoft Windows [Version 10.0.16299.371]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Swami Smarth>cd/

C:\>cd python27

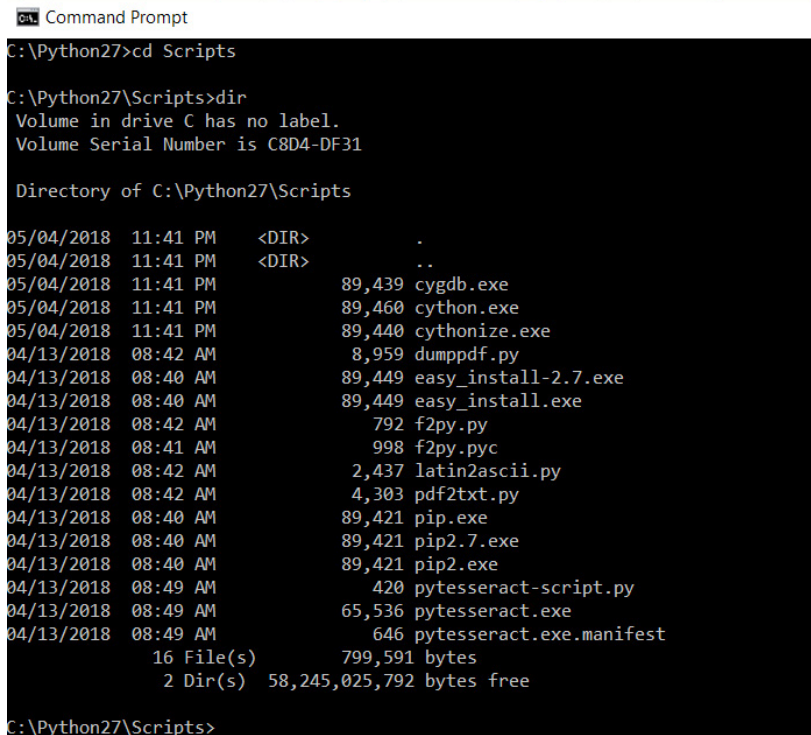
C:\Python27>cd Scripts

C:\Python27\Scripts>

```

**Figure 9.5** Python Script Directory

- ⊙ **STEP 4:** Check if the script directory has PIP. Write command `dir` to list all the files present within the directory structure (see Figure 9.6 below).



```

Command Prompt

C:\Python27>cd Scripts

C:\Python27\Scripts>dir
Volume in drive C has no label.
Volume Serial Number is C8D4-DF31

Directory of C:\Python27\Scripts

05/04/2018 11:41 PM <DIR> .
05/04/2018 11:41 PM <DIR> ..
05/04/2018 11:41 PM 89,439 cygdb.exe
05/04/2018 11:41 PM 89,460 cython.exe
05/04/2018 11:41 PM 89,440 cythonize.exe
04/13/2018 08:42 AM 8,959 dumppdf.py
04/13/2018 08:40 AM 89,449 easy_install-2.7.exe
04/13/2018 08:40 AM 89,449 easy_install.exe
04/13/2018 08:42 AM 792 f2py.py
04/13/2018 08:41 AM 998 f2py.pyc
04/13/2018 08:42 AM 2,437 latin2ascii.py
04/13/2018 08:42 AM 4,303 pdf2txt.py
04/13/2018 08:40 AM 89,421 pip.exe
04/13/2018 08:40 AM 89,421 pip2.7.exe
04/13/2018 08:40 AM 89,421 pip2.exe
04/13/2018 08:49 AM 420 pytesseract-script.py
04/13/2018 08:49 AM 65,536 pytesseract.exe
04/13/2018 08:49 AM 646 pytesseract.exe.manifest
 16 File(s) 799,591 bytes
 2 Dir(s) 58,245,025,792 bytes free

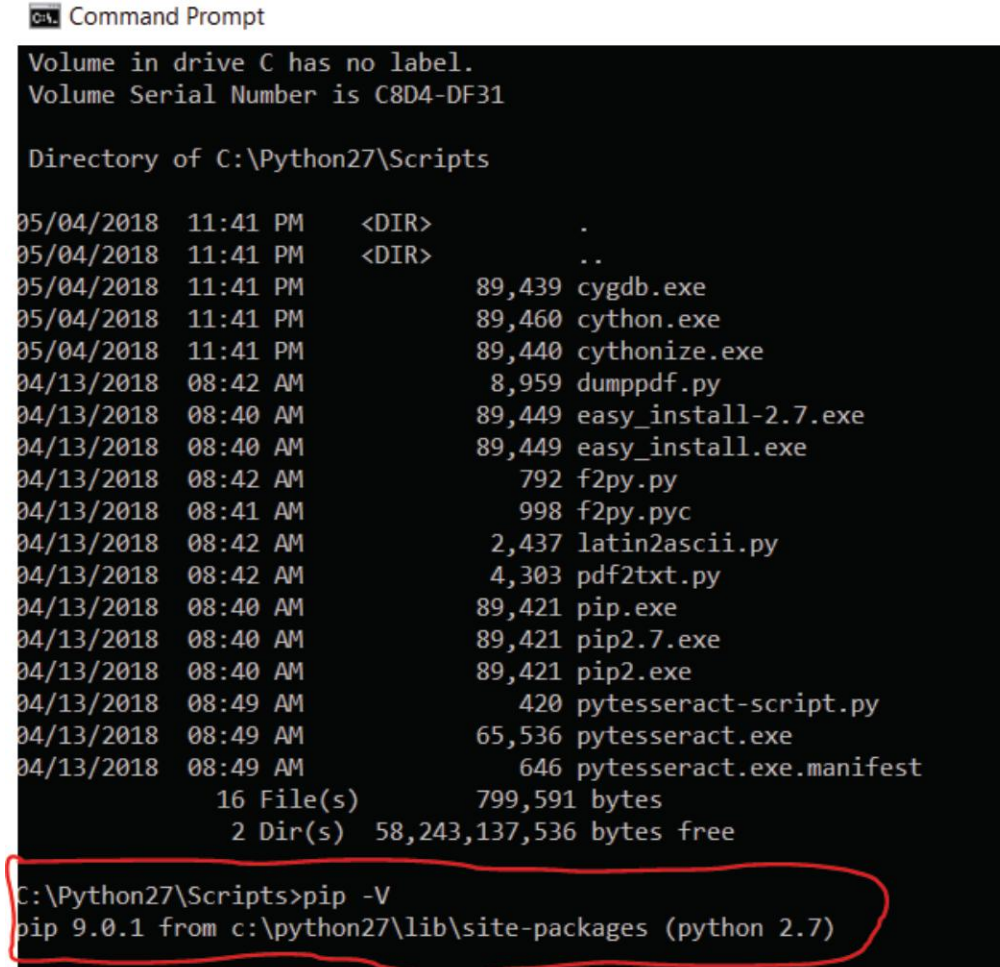
C:\Python27\Scripts>

```

**Figure 9.6** List of files within Script Directory

- ◎ **STEP 5:** If the script directory contains PIP, then write following command on CMD to know the version of PIP (see Figure 9.7).

Pip-V



```

C:\Python27\Scripts>dir
Volume in drive C has no label.
Volume Serial Number is C8D4-DF31

Directory of C:\Python27\Scripts

05/04/2018 11:41 PM <DIR> .
05/04/2018 11:41 PM <DIR> ..
05/04/2018 11:41 PM 89,439 cygdb.exe
05/04/2018 11:41 PM 89,460 cython.exe
05/04/2018 11:41 PM 89,440 cythonize.exe
04/13/2018 08:42 AM 8,959 dumppdf.py
04/13/2018 08:40 AM 89,449 easy_install-2.7.exe
04/13/2018 08:40 AM 89,449 easy_install.exe
04/13/2018 08:42 AM 792 f2py.py
04/13/2018 08:41 AM 998 f2py.pyc
04/13/2018 08:42 AM 2,437 latin2ascii.py
04/13/2018 08:42 AM 4,303 pdf2txt.py
04/13/2018 08:40 AM 89,421 pip.exe
04/13/2018 08:40 AM 89,421 pip2.7.exe
04/13/2018 08:40 AM 89,421 pip2.exe
04/13/2018 08:49 AM 420 pytesseract-script.py
04/13/2018 08:49 AM 65,536 pytesseract.exe
04/13/2018 08:49 AM 646 pytesseract.exe.manifest
 16 File(s) 799,591 bytes
 2 Dir(s) 58,243,137,536 bytes free

C:\Python27\Scripts>pip -V
pip 9.0.1 from c:\python27\lib\site-packages (python 2.7)

```

**Figure 9.7** Knowing the Version of PIP



Note: **pip-h** will list the help options related to PIP.

## 9.5 INSTALLING AND UNINSTALLING PACKAGES VIA PIP

Python software packages can be easily installed by making use of command line interface by just writing one single command. The syntax to install any package is as follows.

```
pip install some-package-name
```

Similarly, the user can uninstall the installed packages using PIP. The syntax is as follows.

```
pip uninstall some-package-name
```

### 9.5.1 The PIP Commands and Options

The PIP provides various commands and options to know various things related to packages. Following Table 9.1 gives the details about commands provided by PIP.

**Table 9.1** Commands and options provided by PIP

| Command                                                                                                                                                                                                                                                                                                                                                                                                                                                                | Meaning                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|
| list<br><b>Example:</b><br>(Run command <b>PIP list</b> on command line interface)<br><b>C:\Python27\Scripts&gt;PIP list</b><br>cyclor (0.10.0)<br>Cython (0.28.2)<br>matplotlib (2.2.2)<br>numpy (1.14.2)<br>pandas (0.22.0)<br>pdfminer (20140328)<br>Pillow (5.1.0)<br>pip (9.0.1)                                                                                                                                                                                  | List installed packages                        |
| list-format columns<br><b>Example:</b><br><b>C:\Python27\Scripts&gt;PIP list--format columns</b><br>Packages                      Version<br>-----<br>backports.functools-lru-cache 1.5<br>cyclor                      0.10.0<br>Cython                      0.28.2<br>matplotlib                      2.2.2<br>numpy                      1.14.2<br>pandas                      0.22.0<br>pdfminer                      20140328<br>Pillow                      5.1.0 | List installed packages with column formatting |
| legacy<br><b>Example:</b><br><b>C:\Python27\Scripts&gt;PIP list--format=legacy</b><br>backports.functools-lru-cache (1.5)<br>cyclor (0.10.0)<br>Cython (0.28.2)<br>matplotlib (2.2.2)<br>numpy (1.14.2)<br>pandas (0.22.0)<br>pdfminer (20140328)<br>Pillow (5.1.0)                                                                                                                                                                                                    | Use legacy formatting                          |

(Contd.)

| json                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Use json formatting to list installed packages.            |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|---------|---------|-------------------------------|-----|--------|--------|--------|--------|------------|-------|-------|--------|--------|--------|----------|----------|--------|-------|-----|-------|
| <b>Example:</b><br><b>C:\Python27\Scripts&gt;PIP list--format=json</b><br><pre>[{"version": "1.5", "name": "backports.functools-lru-cache"}, {"version": "0.10.0", "name": "cyclcr"}, {"version": "0.28.2", "name": "Cython"}, {"version": "2.2.2", "name": "matplotlib"}, {"version": "1.14.2", "name": "numpy"}, {"version": "0.22.0", "name": "pandas"}]</pre>                                                                                                                                                                                                                 |                                                            |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| freeze                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Use freeze formatting to list installed packages.          |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| <b>Example:</b><br><b>C:\Python27\Scripts&gt;PIP list--format=freeze</b><br><pre>backports.functools-lru-cache==1.5 cyclcr==0.10.0 Cython==0.28.2 matplotlib==2.2.2 numpy==1.14.2 pandas==0.22.0</pre>                                                                                                                                                                                                                                                                                                                                                                            |                                                            |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| Outdated                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | List outdated packages                                     |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| <b>Example:</b><br><b>C:\Python27\Scripts&gt;PIP list--outdated--format columns</b><br><table> <thead> <tr> <th>Package</th><th>Version</th></tr> </thead> <tbody> <tr><td>backports.functools-lru-cache</td><td>1.5</td></tr> <tr><td>cyclcr</td><td>0.10.0</td></tr> <tr><td>Cython</td><td>0.28.2</td></tr> <tr><td>matplotlib</td><td>2.2.2</td></tr> <tr><td>numpy</td><td>1.14.2</td></tr> <tr><td>pandas</td><td>0.22.0</td></tr> <tr><td>pdfminer</td><td>20140328</td></tr> <tr><td>Pillow</td><td>5.1.0</td></tr> <tr><td>pip</td><td>9.0.1</td></tr> </tbody> </table> |                                                            | Package | Version | backports.functools-lru-cache | 1.5 | cyclcr | 0.10.0 | Cython | 0.28.2 | matplotlib | 2.2.2 | numpy | 1.14.2 | pandas | 0.22.0 | pdfminer | 20140328 | Pillow | 5.1.0 | pip | 9.0.1 |
| Package                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | Version                                                    |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| backports.functools-lru-cache                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 1.5                                                        |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| cyclcr                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 0.10.0                                                     |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| Cython                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 0.28.2                                                     |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| matplotlib                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | 2.2.2                                                      |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| numpy                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | 1.14.2                                                     |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| pandas                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 0.22.0                                                     |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| pdfminer                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | 20140328                                                   |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| Pillow                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 5.1.0                                                      |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| pip                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | 9.0.1                                                      |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| list-outdated-not-required                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | List packages that have not dependencies of other package. |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |
| <b>Example:</b><br><b>C:\Python27\Scripts&gt;pip list--outdated--not-required</b><br><pre>scipy (1.0.1) - Latest: 1.1.0 [wheel] setuptools (28.8.0) - Latest: 39.1.0 [wheel]</pre>                                                                                                                                                                                                                                                                                                                                                                                                |                                                            |         |         |                               |     |        |        |        |        |            |       |       |        |        |        |          |          |        |       |     |       |

## 9.6 USING PYTHON PACKAGES

Before using any package, the developer should look various installed packages by making use of PIP's list command. Let us assume we have to make use of package "numpy", where numpy is a fundamental package for scientific computing with Python. In short, it is a library which provides support for large multi-dimensional array and matrices. In order to use it, first check if "numpy" package is installed or not.

```
C:\Python27\Scripts>pip list-format columns
```

| Package                       | Version |
|-------------------------------|---------|
| backports.functools-lru-cache | 1.5     |
| cycler                        | 0.10.0  |
| Cython                        | 0.28.2  |
| matplotlib                    | 2.2.2   |

The command `PIP list` shows the installed packages. Among the list of installed packages, there is no package called “numpy”. Thus, the install package, i.e. numpy staying on command line interface as follows.

```
C:\Python27\Scripts>pip install numpy
```

```
Collecting numpy
```

```
 Downloading
```

```
https://files.pythonhosted.org/packages/1c/98/b03970f7e080bccc1118c5ee0f9168ecc735e6c0e618f7de26d4ec163799/numpy-1.14.3-cp27-none-win32.whl (9.8MB)
```

```
100% |#####| 9.8MB 103kB/s
```

```
Installing collected packages: numpy
```

```
Successfully installed numpy-1.14.3
```

Thus, the package numpy is installed and you can start using this package. In order to use it, first statement is “**import numpy**” to import the package numpy.

Using installed “numpy” package:

```
>>> import numpy as np
>>> a = np.array([1,2,3,4]) #Create one dimensional array
>>> a
array([1, 2, 3, 4])

>>> b=np.array([[1,2,3],[4,5,6]]) #Create two dimensional array
>>> b
array([[1, 2, 3],
 [4, 5, 6]])
>>> b.shape
(2, 3)
```

## SUMMARY

- ◆ **Modules** are smaller pieces of bigger program.
- ◆ **import name\_of\_module** is the syntax to loads module.
- ◆ **from name\_of\_module import \*** is the syntax to put all of the contents of module into the current module.
- ◆ Python’s **import** keyword lets you to include other modules in your program.



- ♦ Package is a collection of multiple modules.
- ♦ Directory containing a file `_init_.py` will be treated as a package by Python.
- ♦ The “PIP” is one of the tool for installing Python packages.

## KEY TERMS

- ⇒ **Package:** It is a folder with `_init_.py` and multiple other `.py` files altogether.
- ⇒ **Import:** It is used to include other modules in the main program.
- ⇒ **PIP:** It is a tool for installing packages.

## REVIEW QUESTIONS

### A. Exercise Questions

1. How can we create a module? Explain with examples how to use a module in another program.
2. Explain how we can create a package.
3. What is PIP?
4. List different commands and options provided by PIP.

## PROGRAMMING ASSIGNMENTS

1. Write a program to create package named “Arithmetic\_Operation\_Package”. Inside the package, create two `.py` files containing name as `add.py` and `sub.py`. Now write a code to perform addition and subtraction into `add.py` and `sub.py` and execute the modules of `add.py` and `sub.py` using the package “Arithmetic\_Operation\_Package”.
2. Write a program to get the list of installed packages using PIP.



# UNIT 5

---

- Object-Oriented Programming:  
Class, Objects and Inheritance
- Errors and Exceptions

# Object-Oriented Programming: Class, Objects and Inheritance

## 10

### CHAPTER OUTLINE

|       |                                                  |       |                                                        |
|-------|--------------------------------------------------|-------|--------------------------------------------------------|
| 10.1  | Introduction                                     | 10.12 | Operator Overloading                                   |
| 10.2  | Defining Classes                                 | 10.13 | Inheritance                                            |
| 10.3  | The Self-parameter and Adding Methods to a Class | 10.14 | Types of Inheritance                                   |
| 10.4  | Display Class Attributes and Methods             | 10.15 | The Object Class                                       |
| 10.5  | Special Class Attributes                         | 10.16 | Inheritance in Detail                                  |
| 10.6  | Accessibility                                    | 10.17 | Subclass Accessing Attributes of Parent Class          |
| 10.7  | The <code>__init__</code> Method (Constructor)   | 10.18 | Multilevel Inheritance in Detail                       |
| 10.8  | Passing an Object as Parameter to a Method       | 10.19 | Multiple Inheritance in Detail                         |
| 10.9  | <code>__del__()</code> (Destructor Method)       | 10.20 | Using <code>Super()</code>                             |
| 10.10 | Class Membership Tests                           | 10.21 | Method Overriding                                      |
| 10.11 | Method Overloading in Python                     | 10.22 | Precaution: Overriding Methods in Multiple Inheritance |

### LEARNING OUTCOMES

*After completing this chapter, students will be able to:*

- Explain the necessity and importance of object-oriented features in programming
- Describe attributes and methods for a given object
- Access attributes and member functions, i.e. methods of a class using the dot operator
- Reference an object using the self-parameter
- Overload inbuilt functions using special methods
- Learn to create super and sub classes using the concept of inheritance
- Learn different types of inheritance and make use of them effectively in programming

## 10.1 INTRODUCTION

Python is an object-oriented language. Object-oriented languages help a programmer to reduce the complexity of programs by reusing existing modules or functions. The concept of object-oriented programming language is based on **class**. We know that class is another name for **type** in Python. It means a programmer can create objects of their own class.

So far, we have learnt various inbuilt classes, such as **int**, **str**, **bool**, **float** and **list**. Since all of these are inbuilt classes, Python defines how these classes look and behave. Overall, in any object-oriented language, the class defines how an object of its type looks and behaves. For example, we know how an integer looks. Its 'behavior' will be the operations one can perform on it.

## 10.2 DEFINING CLASSES

As discussed above, class is another name for type in Python. A class may contain data in the form of **fields**. Fields are also called **attributes** and coded in the form of **procedures** known as **methods**. Finally, a programmer has to create an object of its own class, where the object represents an entity which can be easily identified. For example, person, vehicle, fan, book etc. represent real objects. Each object has a unique identity, state and behaviour. The state of the object is also called **property** or **attribute**. For example, a circular object has a data field radius which is a property that characterises a circle. The syntax to define a class in Python is given as follows:

```
Class Class_Name:
 Initializer
 attributes
 methods()
 Statement(s)
```

### PROGRAM 10.1 | Write a simple class program.

```
class Demo:
 pass
D1=Demo() #Instance or Object of the class Demo
print(D1)
```

#### Output

```
<__main__.Demo object at 0x029B3150>
```

**Explanation** In the above example, we have created a new class called **Demo** using the **class** statement. The class is followed by an indented block of statement which forms the body of the class. In the above program, we have an empty block which is indicated using the **pass** statement. The object/instance of this class is created using the name of the class followed by a pair of parentheses. The print statement is used to verify the type of variable D1. Therefore, the print statement tells us

that there is an instance of the Demo class in the `__main__` module. The output of the print statement is `<__main__.Demo object at 0x029B3150>`. It tells us the address of the computer's memory where the object D1 is stored. The value of the address varies from one machine to another. Python stores an object wherever it finds the space. We can say the return value of `print(D1)` is a reference to a Demo class, which we have assigned to D1. Creation of a new object is called **instantiation** and the object is an **instance** of the class.



**Note:** In the above program, we have created an object/instance of the class as:

```
D1=Demo()
```

Creating objects in Python is equivalent to the following code in Java, C++

```
Demo D1 = new Demo();
```

Therefore, there is no **new** keyword in Python as in Java and C++.

**PROGRAM 10.2** | Write a program to create a simple class and print the message, "Welcome to Object-oriented Programming" and print the address of the instance of the class.

```
class MyFirstProgram:
 print(' Welcome to Object-oriented Programming')
C=MyFirstProgram() #Instance of class.
print(C)
```

### Output

```
Welcome to Object-oriented Programming
<__main__.MyFirstProgram object at 0x028B6C90>
```

**Explanation** Name of the class is **MyFirstProgram**. The instance, i.e. 'C' of the class is created. Inside the class, the print statement is used to display the welcome message. Additionally, the last print statement is used to display the address of the computer's memory where the object 'C' is stored.

## 10.2.1 Adding Attributes to a Class

In Program 10.1, we have created a simple class named **Demo**. The class Demo does not contain any data and it does not do anything. What can a programmer do to assign an attribute to a given object? The programs ahead explain how to add an attribute to an existing class.

Let us consider a simple class called **Rectangle** which defines two instance variables **length** and **breadth**. Currently, the class Rectangle does not contain any method.

Class Rectangle:

```
length=0; #Attribute length
breadth=0; #Attribute breadth
```

From above example, it is important to remember that a class declaration only creates a template, i.e. it does not create an actual object. Hence, the above code also does not create any object of type

**Rectangle.** To create a **Rectangle** object we will use the following statement.

```
R1 = Rectangle () # Instance of Class
```

After execution of the above statement R1 will be the instance of the class Rectangle. Each time we create an instance of class, we are creating an object that contains its own copy of each instance variable or attribute defined by that class. Thus, every Rectangle object will contain its copies of instance variables, length and breadth.

### 10.2.2 Accessing Attributes of a Class

The syntax used to access the attributes of a class is:

```
<object>.<attribute>
```

**PROGRAM 10.3** | Write a program to access the attributes of a class.

```
class Rectangle:
 length=0; #Attribute length
 breadth=0; #Attribute breadth
R1 = Rectangle () #Instance of a class
print (R1.length) #Access attribute length
print (R1.breadth) #Access attribute breadth
```

#### Output

```
0
0
```

**Explanation** The class Rectangle is created as shown above. The class contains two attributes, viz. length and breadth. Initially the values of both the attributes are assigned to zero. R1 is the instance of the class. The object R1 and dot operators are used together to print the value of the attributes of a class.

### 10.2.3 Assigning Value to an Attribute

The syntax used to assign a value to an attribute of an object is

```
<object>.<attribute> = <Value>
```

The value can be anything like a Python primitive, an inbuilt data type, another object etc. It can even be a function or another class.

#### Example

```
R1.length=20;
R1.breadth=30;
```

**PROGRAM 10.4** | Write a program to calculate the area of a rectangle by assigning the value to the attributes of a rectangle, i.e. length and breadth.

```
class Rectangle:
 length=0; #Attribute length
 breadth=0; #Attribute breadth
R1 = Rectangle () #Instance of a class
print('Initial values of Attribute')
print('Length = ',R1.length) #Access attribute length
print('Breadth = ',R1.breadth) #Access attribute breadth
print('Area of Rectangle = ',R1.length * R1.breadth)
R1.length = 20 #Assign value to attribute length
R1.breadth = 30 #Assign value to attribute breadth
print('After reassigning the value of attributes')
print('Length = ',R1.length)
print('Breadth = ',R1.breadth)
print('Area of Rectangle is ',R1.length * R1.breadth)
```

### Output

```
Initial values of Attribute
Length = 0
Breadth = 0
Area of Rectangle = 0
After reassigning the value of attributes
Length = 20
Breadth = 30
Area of Rectangle is 600
```

## 10.3 THE SELF-PARAMETER AND ADDING METHODS TO A CLASS

### 10.3.1 Adding Methods to a Class

As discussed at the beginning of this chapter, a class usually consists of **instance variables** and **instance methods**. The syntax to add methods in a class is

```
class Class_Name:
 instance variable; #instance variable with initialisation
def mthod_name(Self,parameter_list):#Paramter List is Optional
 block_of_statements
```

### 10.3.2 The Self-parameter

To add methods to an existing class, the first parameter for each method should be **self**. There is only one difference between class methods and ordinary functions. The self-parameter is used

in the implementation of the method, but it is not used when the method is called. Therefore, the self-parameter references the object itself. Program 10.5 illustrates the self-parameter and addition of methods to an existing class.

**PROGRAM 10.5** Write a program to create a method `Display_Message()` in a class having the name `MethodDemo` and display the message, “Welcome to Python Programming”.

```
class MethodDemo:
 def Display_Message(self):
 print('Welcome to Python Programming')
ob1 = MethodDemo() #Instance of a class
ob1.Display_Message() #Calling Method
```

### Output

```
Welcome to Python Programming
```

**Explanation** In the above program, the `Display_Message()` method takes no parameters but still the method has the self-parameter in the function definition. Therefore, the self-parameter refers to the current object itself. Finally, the method is called and the message is displayed.



- Note:** 1. The first parameter for each method inside a class should be defined by the name 'self'.  
 2. The variable 'self' refers to the object itself. Therefore, for convention it is given the name self.  
 3. The **self in Python** is equivalent to the 'this' pointer in C++ and 'this' reference in Java.

### Important Note

4. Although a programmer can give any name to the parameter self but it is strongly recommended that he/she uses the name 'self'. There are many advantages to using a standard name self, such as any reader of the program will be able to immediately recognise it.

## 10.3.3 Defining Self-parameter and Other Parameters in a Class Method

Program 10.6 explains defining of self and parameters for methods of the existing class.

**PROGRAM 10.6** Write a program to create a class named `Circle`. Pass the parameter `radius` to the method named `Calc_Area()` and calculate the area of the circle.

```
import math
class Circle:
 def Calc_Area(self, radius):
 print('radius = ', radius)
 return math.pi*radius**2
ob1 = Circle()
print('Area of circle is ', ob1.Calc_Area(5))
```

(Contd.)



**Output**

```
radius = 5
Area of circle is 78.53981633974483
```

**Explanation** The class with name Circle is created as shown above. The extra parameter radius is passed to a method defined inside the class `Calc_Area()`. The instance `ob1` of a class is created and used to call the method of the existing class. Even though the method `Calc_Area()` contains two parameters, viz. `self` and `radius`, only one parameter should be passed, viz. the radius of the circle while calling the method.

**PROGRAM 10.7** | Write a program to calculate the area of a rectangle. Pass the length and breadth of the rectangle to the method named `Calc_Rect_Area()`.

```
class Rectangle:
 def Calc_Area_Rect(self,length,breadth):
 print('length = ',length)
 print('breadth = ',breadth)
 return length*breadth
ob1 = Rectangle()
print('Area of Rectangle is ',ob1.Calc_Area_Rect(5,4))
```

**Output**

```
length = 5
breadth = 4
Area of rectangle is 20
```

### 10.3.4 The Self-parameter with Instance Variable

As discussed above, the `self` work as a reference to the current object whose method is invoked. The `self` can also be used to refer any **attribute/member variable** or **instance variable** of the current object from within the instance method. The `self` is used to handle variables.

We cannot create two instance variables/local variables with the same name. However, it is legal to create one instance variable and one local variable or method parameter with the same name. But in this scenario, the local variable tends to hide the value of the instance variable. Program 10.8 illustrates this concept of variable hiding.

**PROGRAM 10.8** | Write a program for variable hiding.

```
class Prac:
 x=5 # attribute x
 def disp(self, x):
```

(Contd.)

```

x=30
print(' The value of local variable x is ',x)
print(' The value of instance variable x is ',x)
ob=Prac()
ob.disp(50)

```

**Output**

```

The value of local variable x is 30
The value of instance variable x is 30

```

**Explanation** The instance variable `x` is initialised with the value 5. Similarly, the method `disp()` has a local variable named `x` and it is initialised with the value 30. Object `ob` is instantiated and the method `disp()` is invoked. Thereafter, it displays both the values of `x` as 30.

Thus, in the above program we have seen the value of the instance variable is hidden by the local variable. If a programmer does not want to hide the value of an instance variable, he/she needs to use `self` with the name of the instance variable. Program 10.9 demonstrates the use of `self` with an instance variable to solve the problem of variable hiding.

**PROGRAM 10.9** | Write a program to demonstrate the use of `self` with an instance variable to solve the problem of variable hiding.

```

class Prac:
 x=5
 def disp(self, x):
 x=30
 print(' The value of local variable x is',x)
 print(' The value of instance variable x is ',self.x)
ob=Prac()
ob.disp(50)

```

**Output**

```

The value of local variable x is 30
The value of instance variable x is 5

```

**Explanation** The `self` is used to differentiate between an instance and local variable. Here `x` displays the value of the local variable and `self.x` displays the value of the instance variable.

### 10.3.5 The Self-parameter with Method

The `self` is also used within methods to call another method from the same class.

**PROGRAM 10.10** | Write a program to create two methods, i.e. **Method\_A()** and **Method\_B()**. Call **Method\_A()** from **Method\_B()** using `self`.

```

class Self_Demo:
 def Method_A(self):
 print('In Method A')
 print('wow got a called from A!!!')
 def Method_B(self):
 print('In Method B calling Method A')
 self.Method_A() #Calling Method_A
Q=Self_Demo()
Q.Method_B() #calling Method_B

```

**Output**

```

In Method B calling Method A
In Method A
wow got a called from A!!!

```

## 10.4 DISPLAY CLASS ATTRIBUTES AND METHODS

There are two ways to determine the attributes in a class. One way is by using the inbuilt function `dir()`. The syntax used to display `dir()` attributes is:

```

dir(name_of_class) or
dir(Instance_of_class)

```

Program 10.11 explains how to display the attributes present in a given class.

**PROGRAM 10.11** | Write a program to display the attributes present in a given class

```

class DisplayDemo:
 Name = ''; #Attribute
 Age = ''; #Attribute
 def read(self):
 Name=input('Enter Name of student: ')
 print('Name = ',Name)
 Age=input('Enter Age of the Student:')
 print('Age = ',Age)
D1 = DisplayDemo()
D1.read()

#Display attributes using dir() on the interactive mode
>>>(dir(DisplayDemo))
['Age', 'Name', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
 '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__
reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__
str__', '__subclasshook__', '__weakref__', 'read']

```

**Explanation** When `dir()` method is executed in interactive mode, the `dir()` function returns a sorted list of attributes and methods belonging to an object. The function returns the existing attributes and methods belonging to the class, including any special methods.

An alternate way to display the attributes of a class is by using a special class attribute `__dict__`. The syntax to display the attributes and methods of an existing class using `__dict__` is

`Class_Name.__dict__`



**Note:** The dict contains two underscores i.e. two underscores before the word dict and two underscores afterwards.

**PROGRAM 10.12** | Write a program executing `__dict__` method on Program 10.11.

```
class DisplayDemo:
 Name = ''; #Attribute
 Age = ' '; #Attribute
 def read(self):
 Name=input('Enter Name of student: ')
 print('Name = ',Name)
 Age=input('Enter Age of the Student:')
 print('Age = ',Age)
D1 = DisplayDemo()
D1.read()

#Display attributes using __dict__
>>> DisplayDemo.__dict__
mappingproxy({'read': <function DisplayDemo.read at 0x02E7C978>, '__weakref__': <attribute '__weakref__' of 'DisplayDemo' objects>, '__doc__': None, '__dict__': <attribute '__dict__' of 'DisplayDemo' objects>, '__module__': '__main__', 'Name': '', 'Age': ''})
```

**Explanation** The special class attribute `__dict__` returns the details of the class containing methods and attributes. The output of the above function returns the address of the method `read`, i.e. `{'read': <function DisplayDemo.read at 0x02E7C978>}`. It also displays the attributes of the class `DisplayDemo`, viz. `Name` and `Age`.

## 10.5 SPECIAL CLASS ATTRIBUTES

Consider a simple program of a class given as follows:

```
class Demo1:
 pass
D1=Demo1()
```

(Contd.)

```
>>>dir(D1)
['_class_', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

In this program, we executed the `dir()` method on the class **Demo1**. The **Demo1** is a simple class. It neither contains any methods nor any attributes. However, by default it contains a list of special methods sorted in an ascending order as output of `dir()`.



**Note:** For any class C, Table 10.1 gives a list of all the special attributes.

**Table 10.1** Special attributes of a class

| Attribute    | Meaning                        |
|--------------|--------------------------------|
| C.__class__  | String name of class           |
| C.__doc__    | Documentation string for class |
| C.__dict__   | Attributes of class            |
| C.__module__ | Module where class is defined  |

## 10.6 ACCESSIBILITY

In Python, there are no keywords like **public**, **protected** or **private**. All attributes and methods are public by default.

There is one way to define private in Python. The syntax to define private attribute and methods is

```
__Attribute
__Methods_Name()
```

To make an attribute and a method private, we need to add two underscores, i.e. “\_\_” in front of the attribute and the method’s name. It helps in hiding these when accessed out of class.

**PROGRAM 10.13** | Write a program to illustrate the use of private.

```
class Person:
 def __init__(self):
 self.Name = 'Bill Gates' #Public attribute
 self.__BankAccNo =10101 #Private attribute

 def Display(self):
```

(Contd.)

```
print(' Name = ',self.Name)
print('Bank Account Number = ',self.__BankAccNo)
```

```
P = Person()
#Access public attribute outside class
print(' Name = ',P.Name)
P.Display()
#Try to access private variable outside class but fails
print(' Salary = ',P.__BankAccNo)
P.Displaay()
```

### Output

```
Name = Bill Gates
Name = Bill Gates
Bank Account Number = 10101
Traceback (most recent call last): #Error
 File "C:/Python34/PrivateDemo.py", line 13, in <module>
 print(' Salary = ',P.__BankAccNo)
AttributeError: 'Person' object has no attribute '__BankAccNo'
```

**Explanation** In the above program, we have defined public and private attributes. The private variable can be accessed within the function. We have created an instance of class **Person**, i.e. **P** to access the attributes defined within the class.

However, it fails and shows an error when instance of class try to access private attributes defined inside the class & are accessed from outside the class.



**Note:** Python hides a private name by changing its name to `_ClassName__AttrName`. This technique is termed as "mangling".

## 10.7 THE `__init__` METHOD (CONSTRUCTOR)

There are many inbuilt methods in Python. Each method has its own significance. The importance of the `__init__` method is explained ahead.

The `__init__` method is known as an initialiser. It is a special method that is used to initialise the instance variable of an object. This method runs as soon as an object of a class is instantiated. The syntax of adding `__init__` method to a class is given as follows:

```
class Class_Name:
 def __init__(self): #__init__ method


```

Note that `init` needs to be preceded and followed by two underscores. Also `__init__` method must have **self** as the first argument. As **self** refers to the object itself, it refers to the object that invokes the method. The **self**-parameter within the `__init__` method automatically sets the reference for the object just created. The `__init__` method can also have positional and/or keyword arguments.

**PROGRAM 10.14** | Write a simple program using the **init** method.

```
class Circle:
 def __init__(self,pi):
 self.pi = pi
 def calc_area(self,radius):
 return self.pi*radius**2
C1=Circle(3.14)
print(' The area of Circle is ',C1.calc_area(5))
```

#### Output

The area of Circle is 78.5

**Explanation** In the above program we have created a class named `Circle`. The class contains two different methods, viz. one is `__init__` method and another `calc_area()` to calculate the area of a circle. Notice that in the above program we do not explicitly call the `__init__` method. We have created an instance of the class `Circle`, i.e. `C1`. While creating the instance of the class we have passed the arguments following the class name to initialise the instance variable of an object.

### 10.7.1 Attributes and `__init__` Method

Programmers can initialise the value of a member variable or attribute by making use of the `__init__` method. Program 10.15 demonstrates the use of the `__init__` method to initialise the attributes with some values.

**PROGRAM 10.15** | Write a program to initialise the value of the attributes by making use of the `init` method.

```
class Circle:
 pi = 0; #Attribute pi
 radius = 0 #Attribute radius
 def __init__(self):
 self.pi = 3.14
 self.radius = 5
 def calc_area(self):
 print('Radius = ',self.radius)
 return self.pi*self.radius**2
C1=Circle()
print(' The area of Circle is ',C1.calc_area())
```

**Explanation** Initially the attributes of the class Circle, i.e. pi and radius are initialised to Zero. With the help of the init method the value of the instance variable pi and radius are reinitialised to 3.14 and 5. These values are initialised upon the creation of the instance of class, i.e. C1. Finally, the area of the circle is calculated by calling the method calc\_area().

## 10.7.2 More Programs on \_\_init\_\_ Method

**PROGRAM 10.16** | Write a program to calculate the volume of a box.

```
class Box:
 width = 0; #Member Variables
 height = 0;
 depth = 0;
 volume = 0;
 def __init__(self):
 self.width = 5
 self.height = 5
 self.depth = 5
 def calc_vol(self):
 print('Width = ',self.width)
 print('Height = ',self.height)
 print('depth = ',self.depth)
 return self.width * self.height * self.depth
B1=Box()
print(' The Volume of Cube is ',B1.calc_vol())
```

### Output

```
Width = 5
Height = 5
Depth = 5
The Volume of Cube is 125
```

**Explanation** The member variable of class Box is initialised to zero. Thereafter, all the member variables, viz. width, height and depth are reinitialised to the value 5 by instantiating the object B1 and using the init method.

## 10.8 PASSING AN OBJECT AS PARAMETER TO A METHOD

So far, we have learnt about passing any kind of parameter of any type to methods.

We can also pass objects as parameter to a method. This is explained in Program 10.17.



**PROGRAM 10.17** | Write a program to pass an object as parameter to a method.

```

class Test:
 a = 0
 b = 0
 def __init__(self, x , y):
 self.a = x
 self.b = y
 def equals(self, obj):
 if(obj.a == self.a and obj.b == self.b):
 return True
 else:
 return False
Obj1 = Test(10,20)
Obj2 = Test(10,20)
Obj3 = Test(12,90)
print(' Obj1 == Obj2 ',Obj1.equals(Obj2))
print(' Obj1 == Obj3 ',Obj1.equals(Obj3))

```

**Output**

```

Obj1 == Obj2 True
Obj1 == Obj3 False

```

**Explanation** In the above program, the **equals()** method inside the class named **Test** compares two objects for equality and returns a result. It compares the invoking object with one that is passed as parameter to the method.

**ob1.equals(ob2)**

As shown above, the invoking object is ob1 and the object being passed to the equals method is ob2. If the values of ob1 and ob2 contain the same value then the method returns **True**, else it returns **False**.

**PROGRAM 10.18** | Write a program to calculate the area of a rectangle by passing an object as parameter to method.

```

class Rectangle:
 length = 0
 breadth = 0
 def __init__(self, l , w):
 self.length = l
 self.breadth = w
 def Calc_Area(self, obj):

```

(Contd.)

```

 print(' Length = ',obj.length)
 print(' Breadth = ',obj.breadth)
 return obj.length * obj.breadth
Obj1 = Rectangle(10,20)
print('The area of Rectangle is ', Obj1.Calc_Area(Obj1))

```

### Output

```

Length = 10
Breadth = 20
The area of Rectangle is 200

```

**Explanation** The object **Obj1** of class **Rectangle** is instantiated. With the help of **init** method, the default values of length and rectangle are initialised to 100 and 200. The **Obj1** itself is passed as parameter to the method **Calc\_area()** and finally the area of the rectangle is computed.

## 10.9 \_\_del\_\_() (DESTRUCTOR METHOD)

Like other object-oriented programming languages, Python also has a destructor. The method **\_\_del\_\_** denotes the destructor and the syntax to define destructor is.

```

def __del__(self)
 block

```

Python invokes the destructor method when the instance is about to be destroyed. It is invoked one per instance. The **self** refers to the instance on which the **\_\_del\_\_()** method is invoked. In other words, Python manages garbage collection of objects by reference counting. This function is executed only if all the references to an instance object have been removed. Program 10.19 illustrates the use of the **\_\_del\_\_** method.

**PROGRAM 10.19** | Write a program to illustrate the use of the **\_\_del\_\_** method.

```

class Destructor_Demo:
 def __init__(self): #Constructor
 print('Welcome')
 def __del__(self): #Destructor
 print('Destructor Executed Successfully')
Ob1=Destructor_Demo() #Instantiation
Ob2 = Ob1
Ob3 = Ob1 #Object Ob2 and Ob3 refers to same object
print(' Id of Ob1 = ',id(Ob1))
print(' Id of Ob2 = ',id(Ob2))
print(' Id of Ob3 = ',id(Ob3))

```

(Contd.)

```

del Ob2 #Remove reference Ob2
del Ob1 #Remove reference Ob1
del Ob3 #Remove reference Ob3

```

### Output

```

Welcome
Id of Ob1 = 47364272
Id of Ob2 = 47364272
Id of Ob3 = 47364272
Destructor Executed Successfully

```

**Explanation** In the above example, we have used constructor `__init__` and destructor `__del__` functions. Initially we have instantiated the object **Ob1** and then assigned aliases **Ob2** and **Ob3** to it. The inbuilt function `id()` is used to confirm that all the three aliases reference to the same object. Finally, all the aliases are removed using the `del` statement.

The destructor `__del__` is not invoked unless all the aliases are deleted.  
The destructor is called exactly once.

```

#Program to demonstrate the above concept
class Destructor_Demo:
 def __init__(self):
 print('Welcome')
 def __del__(self):
 print('Destructor Executed Successfully')
Ob1=Destructor_Demo()
Ob2 = Ob1
Ob3 = Ob1
print(' Id of Ob1 = ',id(Ob1))
print(' Id of Ob2 = ',id(Ob2))
print(' Id of Ob3 = ',id(Ob3))
del Ob1
del Ob2

```

### Output

```

Welcome
Id of Ob1 = 48347312
Id of Ob2 = 48347312
Id of Ob3 = 48347312

```

#From the above program it is clear that the destructor `__del__` is not invoked until all the references to the instance of class are removed. Thus, in order to invoke `__del__`, the reference count has to be decreased to zero.

## 10.10 CLASS MEMBERSHIP TESTS

When we create an instance of a class, the **type** of that instance is the class itself. The inbuilt function **isinstance(obj,Class\_Name)** is used to check for membership in a class. The function returns **True** if an object **obj** belongs to the class **Class\_Name**. Program 10.20 demonstrates the use of the **isinstance()** function.

**PROGRAM 10.20** | Write a program to demonstrate the use of the **isinstance()** method.

```
class A:
 pass
class B:
 pass
class C:
 pass
Ob1=A() #Instance of Class A
Ob2=B() #Instance of Class B
Ob3=C() #Instance of Class C
#Lets make use of isinstance method to check the type.
>>> isinstance(Ob1,A)
True
>>> isinstance(Ob1,B)
False
>>> isinstance(Ob2,B)
True
>>> isinstance(Ob2,C)
False
>>> isinstance(Ob3,B)
False
>>> isinstance(Ob3,C)
True
```

## 10.11 METHOD OVERLOADING IN PYTHON

Most object-oriented programming languages contain the concept of method overloading. It simply refers to having multiple methods with the same name which accept different sets of arguments. Let us consider the code below to understand method overloading.

```
class OverloadDemo:
 def add(self,a,b):
 print(a+b)
 def add(self,a,b,c):
 print(a+b+c)
P = OverloadDemo()
P.add(10,20)
```

If we try to run the code above, it will not execute and show the following error.

```
Traceback (most recent call last):
 File "C:/Python34/Overload_Demo.py", line 7, in <module>
 P.add(10,20)
TypeError: add() missing 1 required positional argument: 'c'
```

This is because Python understands the last definition of the method **add(self, a, b ,c)** which takes only three arguments apart from self. Therefore, while calling the **add ()** method it is forced to pass three arguments. In other words, it forgets the previous definition of method **add ()**.



**Note:** C++ and Java support method overloading. Both the languages allow more than one methods with same name and different signature. The type of method argument defines the signature. In case of overloading, the signature determines which method is actually being invoked. However, Python does not allow method overloading based on **type** as it is not strongly **typed** language.

The above program on method overloading can be solved by using the inbuilt function **isinstanceof**.

#### PROGRAM 10.21 | Write a program on method overloading.

```
class Demo:
 result = 0
 def add(self,instanceOf=None, *args):
 if instanceOf == 'int':
 self.result = 0
 if instanceOf == 'str':
 self.result = ''
 for i in args:
 self.result = self.result + i
 return self.result

D1=Demo()
print(D1.add('int', 10,20,30))
print(D1.add('str', ' I ', ' Love ', ' Python ', ' Programming '))
```

#### Output

```
60
I Love Python Programming
```

**Explanation** The instance of class Demo named D1 is created. The method **add ()** is called twice. The first and second call to **add ()** methods are:

#### #First Call

```
D1.add('int', 10,20,30))
```

**#Second Call**

```
D1.add('str', ' I ', ' Love ', ' Python ', ' Programming ')
```

The **instance** of method checks the type of the first parameter being passed to the `add()` method. It stores the value of **result** process based upon **type**.

**PROGRAM 10.22** | Write a program to display a greeting message. Create a class named **MethodOverloading**. Define the function `greeting()` having one parameter **Name**.

**Input:** `obj.greeting()`

**Output:** Welcome

**Input:** `obj.greeting('Donald Trump')`

**Output:** Welcome Donald Trump

```
class methodOverloading :
 def greeting(self, name=None):
 if name is not None:
 print("Welcome " + name)
 else:
 print("Welcome")

Create an object referencing by variable obj
obj = methodOverloading()

call the method greeting without parameter
obj.greeting()

call the method with parameter
obj.greeting('Donald Trump')
```

In Python, method overloading is a technique to define a method in such way that there are more than one ways to call it. This is different from other programming languages.

## 10.12 OPERATOR OVERLOADING

The idea of operator overloading is not new. It has been used in various object-oriented programming languages, such as C++ and Java. It is one of the best features of a programming language since it makes it possible for a programmer to interact with objects in a natural way. It is the ability to define a data type which provides its own definition of operators. A programmer can overload almost every operator, such as arithmetic, comparison, indexing and slicing, and the number of inbuilt functions, such as length, hashing and type conversion. Overloading operators and inbuilt functions makes user defined types to behave exactly like built-in types.

### 10.12.1 Special Methods

To support operator overloading, Python associates a special method with each inbuilt function and operator. Corresponding to the special method, Python internally converts an expression into a call to perform a certain operation. For example, if a programmer wants to perform a sum of two operands then he/she writes `x + y`. When Python observes the `+` operator, it converts the expression `x + y` to call a special method `__add__`. Thus, to overload the `+` operator, he/she needs to include the implementation of the special method `__add__`.

The details for special methods for arithmetic operations are explained ahead.

### 10.12.2 Special Methods for Arithmetic Operations

Python supports various arithmetic operations, such as addition, subtraction, multiplication and division. It associates a special method with each arithmetic operator. A programmer can overload any arithmetic operation by implementing the corresponding special method. A list of arithmetic operators with their corresponding special method is given in **Table 10.2**.

**Table 10.2** Special methods for operator overloading

| Operation           | Special Method                         | Description                                         |
|---------------------|----------------------------------------|-----------------------------------------------------|
| <code>X + Y</code>  | <code>__add__(self, Other)</code>      | Add X and Y                                         |
| <code>X - Y</code>  | <code>__sub__(Self, Other)</code>      | Subtract Y from X                                   |
| <code>X * Y</code>  | <code>__mul__(self, Other)</code>      | Product of X and Y                                  |
| <code>X / Y</code>  | <code>__truediv__(self, Other)</code>  | Y divides X and it shows the quotient as its output |
| <code>X // Y</code> | <code>__floordiv__(self, Other)</code> | Floored quotient of X and Y                         |
| <code>X % Y</code>  | <code>__mod__(self, Other)</code>      | X mod Y gives a remainder when dividing X by Y      |
| <code>-X</code>     | <code>__neg__(self)</code>             | Arithmetic negation of X                            |

Program 10.23 illustrates operator overloading for adding two objects.

**PROGRAM 10.23** | Write a program to overload the `+` Operator and perform the addition of two objects.

```
class OprOverloadingDemo:
 def __init__(self, X):
 self.X = X

 def __add__(self, other):
 print(' The value of Ob1 =', self.X)
 print(' The value of Ob2 =', other.X)
 print(' The Addition of two objects is:', end='')
 return ((self.X + other.X))

Ob1 = OprOverloadingDemo(20)
Ob2 = OprOverloadingDemo(30)
```

(Contd.)

```
Ob3 = Ob1 + Ob2
print(Ob3)
```

### Output

```
The value of Ob1 = 20
The value of Ob2 = 30
The Addition of two objects is: 50
```

**Explanation** In the above example, we have applied the + operation on two instances, Ob1 and Ob2. When we sum these two objects, the representation is as follows:

**Ob3 = Ob1 + Ob2**

As the above statement contains the + operator, Python automatically invokes the `__add__` method. In the `__add__` method, the first parameter is the object on which the method is invoked and the second parameter is **other**, which is used to distinguish from **self**.



**Note:** It is the responsibility of Python to call a method based on the types of operand of the operator involved while adding two objects.

**Example:** If a programmer writes `Ob1 + Ob2`, it will call the `int` class `__add__` method if `ob1` is an integer. It will call `float` types `__add__` method if `ob1` is float. This is because object on the left side of the + operator corresponds to the object on the right side. Writing `Ob1 + Ob2` is equivalent to `Ob1.__add__(Ob2)`

## 10.12.3 Special Methods for Comparing Types

Comparison is not strictly done on numbers. It can be made on various types, such as list, string and even on dictionaries.

If you are creating your own class, it makes sense to compare your objects to other objects. Similar to the arithmetic operators' above, a programmer can overload any of the following comparison operators. We can use the following special methods to implement comparisons.

**Table 10.3** Special methods for comparison operators

| Operation              | Special Method                   | Description                    |
|------------------------|----------------------------------|--------------------------------|
| <code>X == Y</code>    | <code>__eq__(self, other)</code> | is X equal to Y?               |
| <code>X &lt; Y</code>  | <code>__lt__(self, other)</code> | is X less than Y?              |
| <code>X &lt;= Y</code> | <code>__le__(self, other)</code> | is X less than or equal to Y?  |
| <code>X &gt; Y</code>  | <code>__gt__(self, other)</code> | is X greater than Y?           |
| <code>X &gt;= Y</code> | <code>__ge__(self, other)</code> | is greater than or equal to Y? |

Program 10.24 demonstrates operator overloading for comparing two objects.



**PROGRAM 10.24** | Write a program to use special methods and compare two objects.

```

class CmpOprDemo:
 def __init__(self,X):
 self.X = X

 def __lt__(self,other):
 print(' The value of Ob1 =',self.X)
 print(' The value of Ob2 =',other.X)
 print(' Ob1 < Ob2 :',end='')
 return self.X < other.X

 def __gt__(self,other):
 print(' Ob1 > Ob2 :',end='')
 return self.X > other.X

 def __le__(self,other):
 print(' Ob1 <= Ob2 :',end='')
 return self.X <= other.X

Ob1 = CmpOprDemo(20)
Ob2 = CmpOprDemo(30)
print(Ob1 < Ob2)
print(Ob1 > Ob2)
print(Ob1 <= Ob2)

```

**Output**

```

The value of Ob1 = 20
The value of Ob2 = 30
Ob1 < Ob2 :True
Ob1 > Ob2 :False
Ob1 <= Ob2 :True

```

**Explanation** In the above example, we have applied <, > and <= operator on two instances, Ob1 and Ob2. Therefore, when we need to check if one object is less than other it appears as

**Ob1 < Ob2**

As the above statement contains the < operator, Python automatically invokes the `__lt__` method. Whenever Python observes > and <= operators, it invokes `__gt__` and `__ge__` methods.

**10.12.4 Reference Equality and Object Equality**

Consider the following example which gives more details about equality operators in Python.

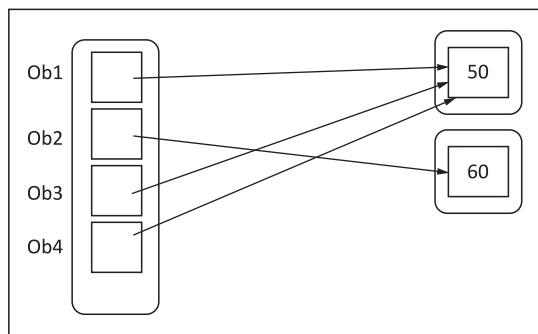
**Example**

```

>>> Ob1 = 50
>>> Ob2 = 60
>>> Ob3 = Ob1
>>> id(Ob1)
1533264672
>>> id(Ob2)
1533264832
>>> Ob1 is Ob2
False
>>> Ob3 is Ob1
True
>>> Ob4 = 50
>>> Ob1 == Ob4
>>> True

```

The above example can be illustrated as shown in Figure 10.1.



**Figure 10.1** Variable referring to objects

In Figure 10.1, objects are referenced by variables, viz. Ob1, Ob2, Ob3 and Ob4. From the Figure, Ob1, Ob3 and Ob4 refer to the same object but Ob2 refer to some other object. If a programmer wants to check the equality of two objects, he/she can use the following two ways:

1. **Reference equality:** If two references are equal and refer to the same object then it is said to be a case of **reference equality**. The inbuilt `id()` function gives the memory address of the object, i.e. identity of the object. The `is` and `is not` operator test whether the two variables refer to the same object. The implementation of statement i.e. `Ob1 is Ob2` checks whether `id(Ob1)` and `id(Ob2)` are the same. If they are same, it returns `True`. In the above example as Ob1 and Ob2 reside at different memory locations, the statement **Ob1 is Ob2** returns `False`.
2. **Object equality:** When two references hold two different/same objects and if the values of the two objects are equal then it is said to be object equality. Thus, in the above example, `Ob1 == Ob4` returns `True` since both of them refer to two different objects with the same value.

### 10.12.5 Special Methods for Overloading Inbuilt Functions

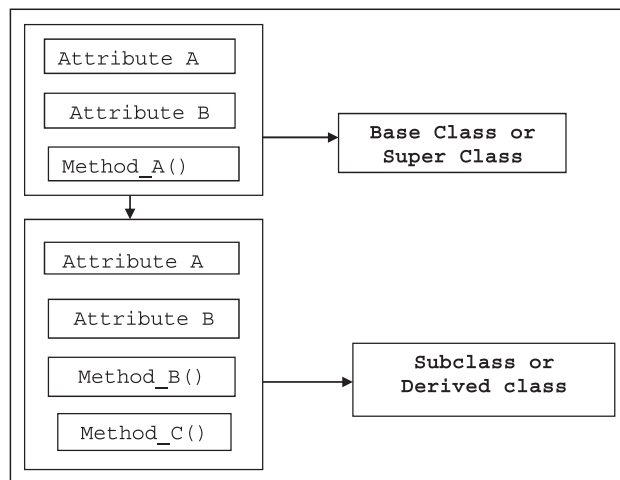
Like operators, we can also overload inbuilt functions. Several inbuilt functions can be overloaded in a manner similar to overloading normal operators in Python. Table 10.4 contains some common inbuilt functions.

**Table 10.4** Special methods for inbuilt functions

| Operation | Special Method               | Description                          |
|-----------|------------------------------|--------------------------------------|
| abs(x)    | <code>__abs__(Self)</code>   | Absolute value of x                  |
| float(x)  | <code>__float__(self)</code> | Float equivalent of x                |
| str(x)    | <code>__str__(self)</code>   | String representation of x           |
| iter(x)   | <code>__itr__(self)</code>   | Iterator of x                        |
| hash(x)   | <code>__hash__(self)</code>  | Generates an integer hash code for x |
| len(x)    | <code>__len__(self)</code>   | Length of x                          |

## 10.13 INHERITANCE

Inheritance is one of the most useful and essential characteristics of object-oriented programming. The existing classes are the main components of inheritance. New classes are created from the existing ones. The properties of the existing classes are simply extended to the new classes. A new class created using an existing one is called a **derived class or subclass** and the existing class is called a **base class or super class**. An example of inheritance is shown in Figure 10.2. The relationship between base and derived class is known as **kind of relationship**. A programmer can define new attributes, i.e. (member variables) and functions in a derived class.



**Figure 10.2** Simple example of inheritance

The procedure of creating a new class from one or more existing classes is called **inheritance**.

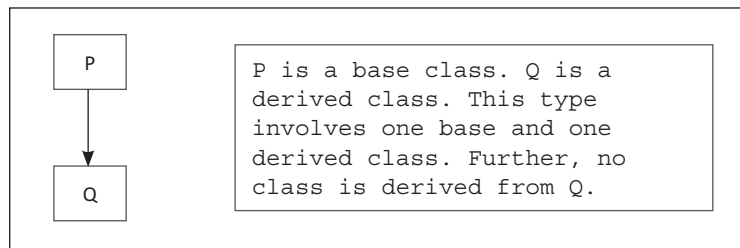
## 10.14 TYPES OF INHERITANCE

We have covered simple examples of inheritance using one base class and one derived class. The process of inheritance can be simple or complex according to the following:

1. **Number of base classes:** A programmer can use one or more base classes to derive a single class.
2. **Nested derivation:** The derived class can be used as the base class and a new class can be derived from it. This is possible at any extent.

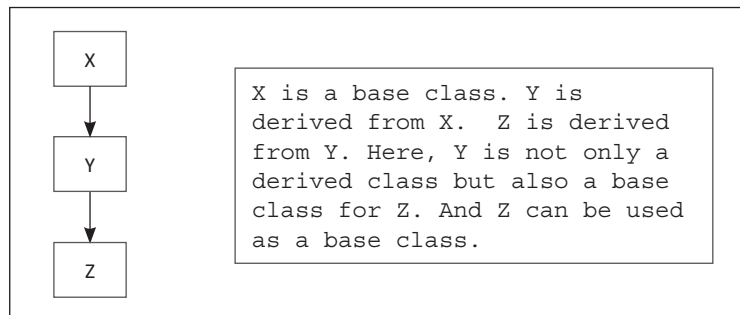
Inheritance can be classified as: (i) single inheritance, (ii) multilevel inheritance and (iii) multiple inheritance. Each of these has been described in detail as follows:

- (i) **Single inheritance:** Only one base class is used for deriving a new class. The derived class is not used as the base class.



**Figure 10.3** Single inheritance

- (ii) **Multilevel inheritance:** When a class is derived from another derived class, the derived class acts as the base class. This is known as multilevel inheritance.



**Figure 10.4** Multilevel inheritance

- (iii) **Multiple inheritance:** When two or more base classes are used for deriving a new class, it is called multiple Inheritance.

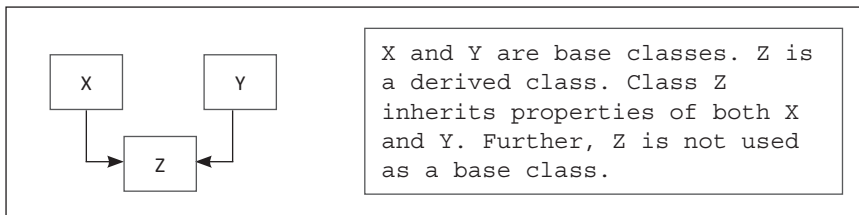


Figure 10.5 Multiple inheritance

## 10.15 THE OBJECT CLASS

Every class in Python is derived from the **object** class. The **object** class is defined in the Python library. Consider the following example of class.

### Example

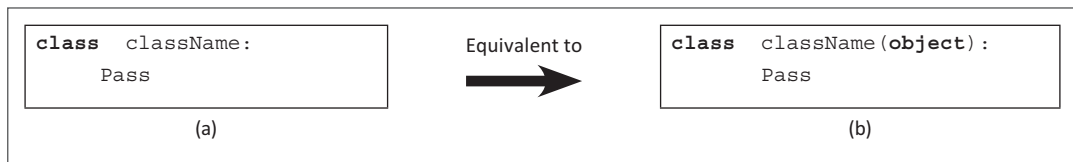


Figure 10.6 The simple class example

Figure 10.6 describes classes in Python. If no inheritance is specified when a class is defined then by default the class is derived from its super class **object**.

## 10.16 INHERITANCE IN DETAIL

Inheritance is a powerful feature of object-oriented programming. It helps creating a new class with little or no modifications on the existing class. The new class called subclass or derived class, inherits the features of its base class. The syntax to define inheritance (i.e. to inherit a single base class) in Python is:

```

Class Derived_Class_Name(Single_Base_Class_Name):
 Body_of_Derived_Class

```

The syntax to inherit multiple base classes is:

```

Class Derived_Class_Name(Comma_Seperated_Base_Class_Names):
 Body_of_Derived_Class

```

Program 10.25 demonstrates the concept of single inheritance.

**PROGRAM 10.25** | Write a simple program on inheritance.

```
class A:
 print('Hello I am in Base Class')
class B(A):
 print('Wow!! Great ! I am Derived class')
ob2 = A() #Instance of class B
```

**Output**

```
Hello I am in Base Class
Wow!! Great! I am Derived class
```

**Explanation** In the above program, we have created the parent class, i.e. (base class) **class A** and child class **class B(A)**: (also called the derived class). The 'A' inside the brackets indicates that class B inherits the properties of its base class A. The instance of the derived class, i.e. the instance ob2 is invoked to execute the functionality of the derived class.

**PROGRAM 10.26** | Write program to create a base class with **Point**. Define the method **Set\_Cordinate(X, Y)**. Define the new class **New\_Point**, which inherits the **Point** class. Also add **draw()** method inside the subclass.

```
Class Point: #Base Class
 def Set_Cordinates(self,X, Y):
 self.X = X
 self.Y = Y

class New_Point(Point): #Derived Class
 def draw(self):
 print(' Locate Point X = ',self.X,' On X axis')
 print(' Locate Point Y = ',self.Y,' On Y axis')

P = New_Point() #Instance of Derived Class
P.Set_Cordinates(10,20)
P.draw()
```

**Output**

```
Locate Point X = 10 On X axis
Locate Point Y = 20 On Y axis
```

**Explanation** The instance of the derived class **P** is created. It is used to initialise the two member variables, **X** and **Y**. **Set\_Cordinates()** method is used to initialise the values of **X** and **Y**. The instance, **P** can access this method since it has been inherited from the parent class. Finally, the **draw()** method is called to draw the point. Thus, the child class **New\_Point** has access to all the attributes and methods defined in its parent class.

## 10.17 SUBCLASS ACCESSING ATTRIBUTES OF PARENT CLASS

Consider Program 10.27 where attributes of the parent class are inherited by its child class.

**PROGRAM 10.27** | Write a program to inherit attributes of the parent class to a child class.

```
class A: # Base Class
 i = 0
 j = 0
 def Showij(self):
 print('i = ',self.i,' j = ',self.j)
class B(A): #Class B inherits attributes and methods of class A
 k = 0
 def Showijk(self):
 print(' i = ',self.i,' j = ',self.j,' k = ',self.k)
 def sum(self):
 print(' i + j + k = ', self.i + self.j + self.k)

Ob1 = A() #Instance of Base class
Ob2 = B() #Instance of Child class
Ob1.i = 100
Ob1.j = 200
print(' Contents of Obj1 ')
Ob1.Showij()
Ob2.i = 100
Ob2.j = 200
Ob2.k = 300
print(' Contents of Obj2 ')
Ob2.Showij() #Sub class Calling method of Base Class
Ob2.Showijk()
print(' Sum of i, j and k in Ob2')
Ob2.sum()
```

### Output

```
Contents of Obj1
i = 100 j = 200
```

(Contd.)

```

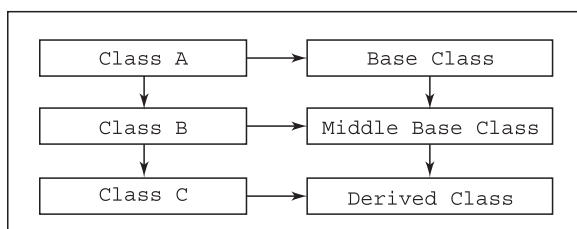
Contents of Obj2
i = 100 j = 200
i = 100 j = 200 k = 300
Sum of i, j and k in Obj2
i + j + k = 600

```

**Explanation** In the above example, the subclass **B** includes all the attributes of its base class **A**. This is why **Obj2** can access **i, j** and call method **showij()**.

## 10.18 MULTILEVEL INHERITANCE IN DETAIL

The procedure of deriving a class from a derived class is called multilevel inheritance.



**Figure 10.7** Multilevel inheritance

**PROGRAM 10.28** | Write a simple program to demonstrate the concept of multilevel inheritance.

```

class A: #Base Class
 name = ' '
 age = 0

class B(A): #Derived Class inheriting Base Class A
 height = ' '

class C(B): #Derived Class inheriting his Base Class B
 weight = ' '

 def Read(self):
 print('Please Enter the Following Values')
 self.name=input('Enter Name:')
 self.age = (int(input('Enter Age:')))
 self.height = (input('Enter Height:'))
 self.weight = (int(input('Enter Weight:')))

 def Display(self):

```

(Contd.)



```

print('Entered Values are as follows')
print(' Name = ',self.name)
print(' Age = ',self.age)
print(' Height = ',self.height)
print(' Weight = ',self.weight)

```

```

B1 = C() #Instance of Class C
B1.Read() #Invoke Method Read
B1.Display() #Invoke Method Display

```

### Output

```

Please Enter the Following Values
Enter Name: Amit
Enter Age:25
Enter Height:5,7'
Enter Weight:60

```

```

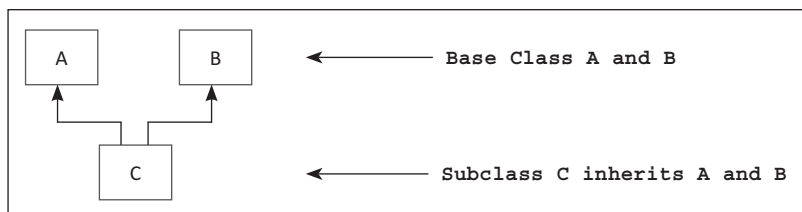
Entered Values are as follows
Name = Amit
Age = 25
Height = 5,7'
Weight = 60

```

**Explanation** In the above program class A, B and C are declared. The member variables of all these classes are initialised with the default value as zero. Class B is derived from class A. Class C is derived from class B. Thus, class B acts as the derived class as well as the base class for class C. The method read() reads data through the keyboard and the method Display() displays data on the screen. Both the functions are invoked using the object B1 of class C.

## 10.19 MULTIPLE INHERITANCE IN DETAIL

When two or more base classes are used for derivation of a new class, it is called multiple inheritance. Let us create a two base classes A, B and one child class C. The child class C inherits the classes A and B.



**Figure 10.8** Example of multiple inheritance

**PROGRAM 10.29** | Write a simple program to demonstrate multiple inheritance.

```

class A: #Base Class A
 a = 0

class B: #Other Base Class B
 b = 0

class C(A,B): #Inherit A and B to create New Class C
 c = 0

 def Read(self):
 self.a = (int(input('Enter the Value of a:')))
 self.b = (int(input('Enter the value of b:')))
 self.c = (int(input('Enter the value of c:')))

 def display(self):
 print(' a = ',self.a)
 print(' b = ',self.b)
 print(' c = ',self.c)

Ob1 = C() #Instance of Child Class
Ob1.Read()
Ob1.display()

```

**Output**

```

Enter the Value of a:10
Enter the value of b:20
Enter the value of c:30
a = 10
b = 20
c = 30

```

**Explanation** In the above program we have created two base classes A and B. Class C is created, which inherits the properties of classes A and B. The statement **class C(A, B)** is used to inherit the properties of classes A and B. Finally, the instance of subclass C is used to call method read and display.

**10.19.1 More Practical Examples on Inheritance**

We have created the parent class **Box**. The constructor **\_\_init\_\_** is used to initialise all the attributes of the class Box. Similarly, the subclass named **ChildBox** is created. The extra attribute named weight is added to the child class, i.e. weight. Thus, all the attributes of the base class along with

the attributes of the child class are initialised in the constructor of the child class by making use of the `__init__` method.

**PROGRAM 10.30** | Write a program to calculate the volume of Box using the **init()** method.

```

Class Box:
 width = 0
 height = 0
 depth = 0
 def __init__(self,W,H,D):
 self.width = W
 self.height = H
 self.depth = D
 def volume(self):
 return self.width * self.height * self.depth

class ChildBox(Box):
 weight = 0
 def __init__(self,W,H,D,WT):
 self.width = W
 self.height = H
 self.depth = D
 self.weight = WT
 def volume(self):
 return self.width * self.height * self.depth

B1 = ChildBox(10,20,30,150)
B2 = ChildBox(5,4,2,100)
vol = B1.volume()
print(' ----- Characteristics of Box1 ---- ')
print(' Width = ',B1.width)
print(' height = ',B1.height)
print(' depth = ',B1.depth)
print(' Weight = ',B1.weight)
print(' Volume of Box1 = ',vol)
print(' ----- Characteristics of Box2---- ')
print(' Width = ',B2.width)
print(' height = ',B2.height)
print(' depth = ',B2.depth)
print(' Weight = ',B2.weight)
vol = B2.volume()
print(' Volume of Box2 =',vol)

```

(Contd.)

**Output**

```
----- Characteristics of Box1 -----
Width = 10
height = 20
depth = 30
Weight = 150
Volume of Box1 = 6000
----- Characteristics of Box2-----
Width = 5
height = 4
depth = 2
Weight = 100
Volume of Box2 = 40
```

## 10.20 USING super()

Consider the following program.

**PROGRAM 10.31**

```
class Demo:
 a = 0
 b = 0
 c = 0
 def __init__(self,A,B,C):
 self.a = A
 self.b = B
 self.c = C
 def display(self):
 print(self.a,self.b,self.c)

class NewDemo(Demo):
 d = 0
 def __init__(self,A,B,C,D):
 self.a = A
 self.b = B
 self.c = C
 self.d = D

 def display(self):
 print(self.a,self.b,self.c,self.d)
```

(Contd.)

```
B1 = Demo(100,200,300)
print(' Contents of Base Class')
B1.display ()
D1=NewDemo(10,20,30,40)
print(' Contents of Derived Class')
D1.display()
```

### Output

```
Contents of Base Class
100 200 300
Contents of Derived Class
10 20 30 40
```

In the above program, the classes derived from the base class **Demo** were not implemented efficiently or robustly. For example, the derived class **NewDemo** explicitly initialises the value of **A**, **B** and **C**, fields of the **Base class**. The same duplication of code is found while initialising the same fields in the base class **Demo**, which is inefficient. This implies that a subclass must be granted access to the members of a super class.

Therefore, whenever a subclass needs to refer to its immediate super class, a programmer can do so by using **super**. The **super** is used to call the constructor, i.e. the `__init__` method of the super class.

### 10.20.1 Super to Call Super Class Constructor

Any subclass can call the constructor, i.e. the `__init__` method defined by its super class by making use of **super**. The syntax to call the constructor of a super class in Python 3.X is:

```
super().__init__(Parameters_of_Super_class_Constructor)
```

The syntax to call the super class constructor from its base class in Python 2.X is:

```
super(Derived_Class_Name,self).__init__(Parameters_of_Super_class_Constructor)
```

Consider the above program and use **super** to avoid duplication of code.

**PROGRAM 10.32** | Use **super()** and call the constructor of the base class.

```
class Demo:
 a = 0
 b = 0
 c = 0
```

(Contd.)

```

def __init__(self,A,B,C):
 self.a = A
 self.b = B
 self.c = C
def display(self):
 print(self.a,self.b,self.c)

class NewDemo(Demo):
 d = 0
 def __init__(self,A,B,C,D):
 self.d = D
 super().__init__(A,B,C) #Super to call Super class
 #__init__ method
 def display(self):
 print(self.a,self.b,self.c,self.d)

B1 = Demo(100,200,300)
print(' Contents of Base Class')
B1.display ()
D1=NewDemo(10,20,30,40)
print(' Contents of Derieved Class')
D1.display()

```

### Output

```

Contents of Base Class
100 200 300
Contents of Derived Class
10 20 30 40

```

**Explanation** The derived class **NewDemo()** calls the **super()** with arguments **a**, **b** and **c**. This causes the constructor **\_\_init\_\_** of the base class, i.e. **Demo** to be called. This initialises the values of **a**, **b** and **c**. The **NewDemo()** class no longer initialises these values itself.

## 10.21 METHOD OVERRIDING

In class hierarchy, when a method in a sub class has the same name and same header as that of a super class then the method in the sub class is said to override the method in the super class. When an overridden method is called, it always invokes the method defined by its subclass. The same method defined by the super class is hidden. Consider the following example to demonstrate the concept of method overriding.

**PROGRAM 10.33** | Write a program to show method overriding.

```

class A: #Base Class
 i = 0
 def display(self):
 print(' I am in Super Class')

class B(A): #Derived Class
 i = 0
 def display(self): #Overridden Method
 print(' I am in Sub Class')

D1 = B()
D1.display()

```

**Explanation** In the above program when `display()` method is invoked on an instance of B, the method `display()` defined within B is invoked. Therefore, the method `display()` overrides the method `display()` defined in the base class A.

Programmer can make use of **super** to access the overridden methods. The syntax to call the overridden method that is defined in super class is

**`super().method_name`**

The above same program 10.33 is given below except, the overridden method defined in super class is accessed by using `super()`.

```

class A: #Base Class
 i = 0
 def display(self):
 print(' I am in Super Class')

class B(A): #Super Class
 i = 0
 def display(self): #Overridden Method
 print(' I am in Sub Class')
 super().display() #Call Display method of Base class

D1 = B() #Instance of sub class
D1.display()

```

**Output:**

```

I am in Sub Class
I am in Super Class

```

From this program we have learnt how to call overridden methods using super.

## 10.22 PRECAUTION: OVERRIDING METHODS IN MULTIPLE INHERITANCE

As discussed before, in multiple inheritance there is at least one class which inherits the properties from two or more classes. Sometimes multiple inheritance can be so complex that some programming languages put restrictions on it.

Consider Program 10.34 on multiple inheritance where method named `display()` has been overridden.

**PROGRAM 10.34** | Program to override `Display()` method in multiple inheritance.

```
class A(object):
 def Display(self):
 print(" I am in A")

class B(A):
 def Display(self):
 print(" I am in B")
 A.Display(self) # call the parent class method too

class C(A):
 def Display(self):
 print(" I am in C")
 A.Display(self)

class D(B, C):
 def Display(self):
 print(" I am in D")
 B.Display(self)
 C.Display(self)

Ob = D()
Ob.Display()
```

### Output

```
I am in D
I am in B
I am in A
I am in C
I am in A
```



The problem with the above method is that **A.Display** method has been called twice. If we have a complex tree of multiple inheritance then it is very difficult to solve this problem by hand. We have to keep track of which super classes have already been called and avoid calling them a second time.

Therefore, to solve the above problem, we can make use of **super**. Consider the same program with some modifications.

```
class A(object):
 def Display(self):
 print(" I am in A")

class B(A):
 def Display(self):
 print(" I am in B")
 super().Display() # call the parent class method too

class C(A):
 def Display(self):
 print(" I am in C")
 super().Display()

class D(B, C):
 def Display(self):
 print(" I am in D")
 super().Display()

Ob = D()
Ob.Display()
```

### Output

```
I am in D
I am in B
I am in C
I am in A
```

Therefore, by using **super**, the method inside multiple inheritance hierarchy gets exactly called once and in the right order.

## MINI PROJECT Arithmetic Operations on Complex Numbers

This mini project will make use of various concepts of object-oriented programming such as **constructor**, **self-parameter**, creating **instance** of class and **overloading** of inbuilt functions.

## Explanation of Complex Numbers

Complex numbers can be written in the form  $a + bi$  where  $a$  and  $b$  are real numbers and  $i$  is the unit imaginary number, i.e.  $\sqrt{-1}$ . The values of  $a$  and  $b$  can be zero. Complex numbers contain two parts, viz. **real** and **imaginary**.

Valid examples of complex numbers are

$$2 + 6i, 1 - i, 4 + 0i$$

## Addition of Two Complex Numbers

Consider two complex numbers  $(a + bi)$  and  $(c + di)$ . In case of addition, add the real parts and then add the imaginary parts.

$$\begin{aligned}(a + bi) + (c + di) &= (a + c) + (b + d)i \\ (2 + 1i) + (5 + 6i) &= (2 + 5) + (1 + 6)i = (7 + 7i)\end{aligned}$$

## Subtraction of Two Complex Numbers

Consider two complex numbers  $(a + bi)$  and  $(c + di)$ . In case of subtraction, subtract the real parts and then subtract the imaginary parts.

$$\begin{aligned}(a + bi) - (c + di) &= (a - c) + (b - d)i \\ (2 + 1i) - (5 + 6i) &= (2 - 5) + (1 - 6)i = (-3 - 5i)\end{aligned}$$

## Multiplication of Two Complex Numbers

Consider two complex numbers  $(a + bi)$  and  $(c + di)$ . Multiplication of two such complex numbers is:

$$\begin{aligned}(a + bi) * (c + di) &= (ac - bd) + (ad + bc)i \\ (2 + 1i) * (5 + 6i) &= (2*5 - 1*6) + ((2*6)i + (1*5)i) \\ &= 4 + 17i\end{aligned}$$

**PROBLEM STATEMENT** | Write a program to perform the following operations on complex numbers.

- (a) Addition
- (b) Subtraction
- (c) Multiplication
- (d) Check if two complex numbers are equal or not
- (e) Check if  $C1 \geq C2$
- (f) Check if  $C1 \leq C2$



**Note:** Make use of the following inbuilt methods to implement the above functions.

`__add__` method to overload + Operator

`__sub__` method to overload – Operator

`__mul__` method to overload \* operator

`__le__` method to overload < operator

`__ge__` method to overload > operator

## Algorithm

- ⊙ **STEP 1:** Create a class named **Complex**.
- ⊙ **STEP 2:** Create the constructor of the class **Complex** using the **init** method. The constructor will have two parameters, viz. one each to store the real and the imaginary part.
- ⊙ **STEP 3:** Create such other methods for add, sub, mul to perform addition, subtraction and multiplication, respectively. Define all the functionalities of these operations by making use of inbuilt functions.
- ⊙ **STEP 4:** Also define the inbuilt methods to check if two complex numbers are equal or if the first complex number is greater than the second.
- ⊙ **STEP 5:** Create two **instances** of **Complex** class **C1** and **C2** to declare two complex numbers.
- ⊙ **STEP 6:** Use these two instances to perform all the operations.

## Solution

```
class Complex(object):
 def __init__(self, real, imag=0.0):
 self.real = real
 self.imag = imag

 def print_Complex_Number(self):
 print('(' , self.real, ' , ' , self.imag, ')')

 def __add__(self, other):
 return Complex(self.real + other.real,
 self.imag + other.imag)

 def __sub__(self, other):
 return Complex(self.real - other.real,
 self.imag - other.imag)

 def __mul__(self, other):
 return Complex(self.real* other.real
```

(Contd.)

```
 - self.imag * other.imag,
 self.imag* other.real +
 self.real * other.imag)

 def __eq__(self,other):
 return self.real == other.real and
 self.imag == other.imag

 def __le__(self,other):
 return self.real < other.real and self.imag < other.imag

 def __ge__(self,other):
 return self.real > other.real and self.imag > other.imag

C1 = Complex(2, 1)
print('First Complex Number is as Follows:')
C1.print_Complex_Number()

C2 = Complex(5, 6)
print('Second Complex Number is as Follows:')
C2.print_Complex_Number()

print('Addition of two complex Number is as follows:')
C3 = C1 + C2
C3.print_Complex_Number()

print('Subtraction of two Complex Number is as follows:')
C4 = C1 - C2
C4.print_Complex_Number()

print('Multiplication of two Complex Number is as follows:')
C5 = C1 * C2
C5.print_Complex_Number()

print('Compare Two Complex Numbers:')
print((C1 == C2)) #Returns true if equal
 #Returns false if not

print('Checking if C1 is Greater than C2:')
```

(Contd.)

```
print(C1 >= C2)

print('Checking if C1 is Less than C2:')
print(C1 <= C2)
```

### Output

```
First Complex Number is as Follows:
(2 , 1)
Second Complex Number is as Follows:
(5 , 6)
Addition of two complex Number is as follows:
(7 , 7)
Subtraction of two Complex Number is as follows:
(-3 , -5)
Multiplication of two Complex Number is as follows:
(4 , 17)
Compare Two Complex Numbers:
False
Checking if C1 is Greater than C2:
False
Checking if C1 is Less than C2:
True
```

Thus, in the above program we have effectively used inbuilt methods to overload various operators, such as +, -, \*, >=, <= and == operators.

## SUMMARY

- ◆ The **class** is fundamental building block of python's object-oriented programming.
- ◆ **Attributes** and **methods** can be added inside the class definition.
- ◆ **Instantiation** refers to creation of new object.
- ◆ **Self** parameter is used to distinguish between normal method defined outside the class and the method defined within the class.
- ◆ The **\_\_init\_\_** method is similar to the constructor of other programming language.
- ◆ The **\_\_del\_\_** i.e. destructor method is invoked when instance is about to be destroyed.
- ◆ Operator and method overloading have been discussed in brief in this chapter.
- ◆ The concept of inheritance is used to extend the properties of base class to its child class.

## KEY TERMS

- ⇒ **class:** Type in Python
- ⇒ **object:** Instance of class

- ⇒ **dot (.) Operator:** Access methods and attributes of a class
- ⇒ **Instantiation:** Process of creating new objects
- ⇒ **Self-parameter:** References the object itself
- ⇒ **Accessibility:** Prevents access
- ⇒ **\_\_init\_\_:** Initialiser (Constructor)
- ⇒ **\_\_del\_\_:** Destructor
- ⇒ **Operator Overloading:** Associates a special method for each operator
- ⇒ **Inheritance:** Creates a new class from an existing class
- ⇒ **Single, Multiple, Multilevel:** Types of inheritance
- ⇒ **super Keyword:** Used for method overriding

## REVIEW QUESTIONS

### A. Multiple Choice Questions

1. What is the relation between object and class?
  - a. A class is an instance of an object
  - b. An object is an instance of an object
  - c. An object is an attribute of a class
  - d. None of the above
2. Which method should be used to create default values in a class constructor?
  - a. `__doc__`
  - b. `__new__`
  - c. `__init__`
  - d. `__del__`
3. Instantiation is a process of:
  - a. Destroying an object
  - b. Initialising an object with a default value
  - c. Creating a new object with a default value
  - d. None of the above
4. What method is called when an object is created?
  - a. `Self`
  - b. `obj.self`
  - c. `init`
  - d. `__int__`
5. We have an object instance `obj` and want to call its method `calc_area()`. Which is the correct way of calling the function `calc_area()`?
  - a. `obj.calc_area(self)`
  - b. `calc_area.obj()`
  - c. `obj.calc_area()`
  - d. `calc_area.obj(self)`
6. Method overriding is \_\_\_\_\_.
  - a. A method with different name,
  - b. A method in a subclass which has the same name and same header as that of the super class
  - c. Both a and b
  - d. None of the above
7. What is used to create an object?
  - a. Constructor
  - b. Class
  - c. Method
  - d. None of the above
8. Which of the following statements are true?
  - a. Objects of the same type have the same id.
  - b. Each object has a unique id.
  - c. Both a and b
  - d. None of the above

9. \_\_\_\_\_ represents an entity in the real world which can be distinctly identified?
- Object
  - Class
  - Method
  - None of the above
10. Analyze the code given below to find the reason behind the error in the program.

```
class A:
 def __init__(self):
 self.P = 10
 self.__Q = 20

 def getY(self):
 return self.__Q

a = A()
print(a.__Q)
```

- Q is private and cannot be access outside of the class.
  - P is private and cannot be access outside of the class.
  - Both a and b
  - None of the above
11. Analyze the code given below to find the reason behind the error in the program.

```
class Base:
 def __init__(self, X):
 self.X = X
 def print(self):
 print(self.X)
```

```
Ob1 = Base()
Ob1.print()
```

- The class Base does not have a constructor.
  - X is not defined in print.
  - The constructor is invoked without arguments.
  - None of the above
12. What will be the output of the following program?

```
class A:
 def __init__(self, s):
 self.s = s
 def display():
 print(s)

a = A("Welcome")
a.display()
```

- Welcome
- Error: The self is missing in method display()
- Cannot access method display()
- None of the above

13. Which statement is correct about self?
- Self refers to the previous object.
  - Self refers to the next object.
  - Self refers to the current object.
  - None of the above
14. Which method runs as soon as an object of a class is instantiated?
- `__init__`
  - `__del__`
  - `self`
  - None of the above
15. Which of the following is not a type of inheritance?
- Single
  - Multilevel
  - Distributive
  - Multiple
16. Look at the following definition of class and determine the type of inheritance the class is using.
- ```
class A:
    Pass
class B:
    Pass
class C(A , B):
    Pass
```
- Single
 - Multilevel
 - Multiple
 - None of the above
17. Which method is used to display the attributes of a class?
- `__init__`
 - `__dict__`
 - `__del__`
 - None of the above
18. The `__del__` is executed only if all_____.
- The references to a current instance object have been removed
 - The references to a previous object have been removed
 - The references to an instance object have been removed
 - None of the above
19. Suppose B is a subclass of A. Which syntax will be used to invoke the `__init__` method defined in class A from class B?
- `super()`
 - `super().__init__(self)`
 - `super().__init__()`
 - None of the above
20. If Ob1 is an instance of class A. Which statement can be used to check whether the object Ob1 is an instance of class A?
- `Ob1.isinstance(A)`
 - `A.isinstance(Ob1)`
 - `isinstance(Ob1, A)`
 - `isinstance(A, Ob1)`

B. True or False

- Python does not permit re-use of an existing module or function.
- Indentation is not important in Python.
- A class is followed by an indented block of statements which forms the body of the class.
- In order to add methods to an existing class, the first parameter for each method should be self.
- The `directory()` function is used to see the attributes of a class.
- The `dir()` function returns a sorted list of attributes and methods belonging to an object.
- All attributes and methods in Python are public by default.

8. The `__init__` method is a special method which is used to initialise instance variable of an object.
9. We can pass an object as parameter to a method.
10. The properties of existing classes are simply extended to new classes.
11. New classes are not created from existing ones using inheritance.
12. In single inheritance, two base classes are used for the derivation of a new class.
13. When two or more base classes are used for derivation of a new class, it is called multiple inheritance.
14. Inheritance inherits features of its base class.
15. The syntax used to assign a value to an attribute on an object is `<object>.<attribute> = <Value>`

C. Exercise Questions

1. What is a class?
2. State the syntax to define a class.
3. How are attributes added to a class?
4. State the syntax to add methods in a class.
5. What is self-parameter? List its uses.
6. List the applications of special class attributes.
7. What is meant by inheritance?
8. List the different types of inheritances.
9. Explain multiple inheritance with an example.
10. What can be done with overriding a method?
11. State the syntax to override a method.
12. Complete the code given below and then perform the following tasks on the coordinate class.
 - a. Instantiate two different objects **P1** and **P2**.
 - b. Display the coordinates of P1 and P2.
 - c. Add an `__eq__` method that returns True if coordinates P1 and P2 refer to the same point in a plane.

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def getX(self):
        return self.x

    def getY(self):
        return self.y
```

13. Consider the code and answer the following questions.

```
class A(object):
    def __init__(self, Name, Gender):
        self.Name = Name
        self.Gender = Gender
```

```

def execute(self):
    print(self.Name)

class B(A):
    def __init__(self):
        A.__init__(self, 'John', 'Male')

class C(A):
    def __init__(self):
        A.__init__(self, 'Anushka', 'Female')

class D(A):
    print(A)

Ob1 = B()
Ob1.execute()

```

- Identify the parent classes present in the above code.
- Identify the child classes present in the above code.
- What will be the output of the above code?

PROGRAMMING ASSIGNMENTS

- Write a program to create a class named Demo. Define two methods **Get_String()** and **Print_String()**. Accept the string from user and print the string in upper case.
- Write a program to create a class **Circle**. Perform the following operations on it.
 - Define the attribute **radius**.
 - Define the constructor with one argument containing radius.
 - Define the method named **get_radius()** which returns the radius of the circle.
 - Define the method named **calc_area()** which returns the area of the circle.
- Write a program to create the class Point. Perform the following operations on it.
 - Initialise X and Y coordinates of the point.
 - Print the coordinates by defining the method 'Display()'
 - Define the method **Translate(X, Y)** to move the point X units in X direction and Y units in Y direction.
- Write a program to implement single inheritance.
 - Create the parent class **Circle**. Initialise the constructor with the radius of the circle.
 - Define the method **get_radius()** and **calc_area()** to know the radius and area of the circle.

(Contd.)

PROGRAMMING ASSIGNMENTS (Contd.)

- c. Create the child class named **Cylinder**. Initialise the value of the height within the constructor and call the constructor of the parent class to initialise the radius of the cylinder.
- d. Finally, define the method **Calc_area()** in the class **Cylinder** to calculate the area of the cylinder.

Note: Area of Cylinder = $2 * \pi * \text{radius} * \text{height}$

- 5. Write a program to implement the concept of multiple inheritance.
 - a. Create the parent class **Shape**. Initialise the constructor with **Shape**.
 - b. Create another class named **Rectangle** which inherits the properties of the parent class **Shape**. Define the attributes **length** and **breadth** in the **Rectangle** class. Initialise the **length** and **breadth** inside the constructor of the **Rectangle** class. Also call the constructor of the parent class to initialise the **color** of the **Rectangle**. Define the method **calc_area()** to return the area of the rectangle.
 - c. Create another class named **Triangle** which inherits the properties of the parent class **Shape**. Define the attributes **base** and **height** in the **Triangle** class. Initialise the **base** and **height** inside the constructor of the **Triangle** class. Also call the constructor of the parent class to initialise the **color** of the **Triangle**. Define the method **calc_area()** to return the area of the **Triangle**.
 - d. Also create the method **Tring_Details()** in the **Triangle** class and **Rect_Details()** in the **Rectangle** Class to return complete details about the rectangle and triangle.
 - e. Finally, create the instance of the **Rectangle** and **Triangle** classes to return the area of the **Rectangle** and **Triangle**.

Errors and Exceptions

11

CHAPTER OUTLINE

11.1 Errors and Exceptions	11.3 Handling Exception
11.2 Python Exception and Its Hierarchy	11.4 Raising Exception

LEARNING OUTCOMES

After completing this chapter, students will be able to:

- Learn how to handle exceptions using built-in Python keywords
- Handle the exceptions using **try**, **except** and **finally** keywords
- Raise exception using the **raise** keyword

11.1 ERRORS AND EXCEPTION

How often did it happen that the program you wrote ran successfully in its very first attempt? Very rarely right, isn't it? There are always some errors in the program. This is what this chapter is all about. It is common to make mistakes while typing as well as while developing a program. Such types of mistakes are called "Errors". An error is something that can produce incorrect or irrelevant output or even be responsible for the system to crash. It is, therefore, very important to find out these errors and fix them so that the program will not terminate or crash during execution. Error can be of two types, viz. the compile time errors and run-time errors. Thus, this entire chapter will help you to deal with all such kind of errors without terminating the program by the concept called "exception handling". Details of the exception handling are explained in the next section.

11.1.1 Exception

“An exception is an error that occurs during the run-time”.

or

“An exception is an erroneous condition which arises while program is running”.

Run-time error occurs while program is in execution or program is running. Examples of such condition include dividing a number by zero produces “ZeroDivisionError: division by zero exception”.

Example: Division by Zero Exception

```
>>> a=10
>>> b=0
>>> a/b
Traceback(most recent call last):
  File "<pyshell#5>", line 1, in <module>
    a/b
```

ZeroDivisionError: division by zero

If the programmer attempts to open a file that doesn't exist, it produces “FileNotFoundError”, similarly if you try to access an index of a list which is out-of-bound, i.e. to access the index of an array more than the size of the list, it produces then “IndexError: list index out of range” exception and many more other exceptions are raised.

Thus, an exception is an object that represents an error. Also it is a condition that prevents execution of statements in Python to proceed normally. Thus, if the exceptions are not handled properly, the programs will terminate abnormally. In order to run the program without termination, there is always need of exception handling.

11.2 PYTHON EXCEPTION AND ITS HIERARCHY

In Python, all of the exceptions are the instances of a class which are derived from BaseException class. Each exception type is of a separate class. Python organizes the exceptions in a hierarchical way. The following Table 11.1 contains all in-built exceptions where the indentation indicates how those exceptions are structured hierarchically.

Table 11.1 Python exception hierarchy

```
BaseException    #The base class for all built-in exceptions
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception    #All built-in exceptions and user-defined exceptions are derived
from this class.
    +-- StopIteration
    +-- StandardError
        |    +-- BufferError
```

(Contd.)

```

|      +-- ArithmeticError #The base class for those built-in exceptions that
are raised for various arithmetic operations given as follows.
|      |      +-- FloatingPointError
|      |      +-- OverflowError
|      |      +-- ZeroDivisionError
|      +-- AssertionError
|      +-- AttributeError
|      +-- EnvironmentError
|      |      +-- IOError
|      |      +-- OSError
|      |      +-- WindowsError(Windows)
|      |      +-- VMSError(VMS)
|      +-- EOFError #This exception is raised when input() function hits an
end-of-file condition (EOF) without reading any data.
|      +-- ImportError
|      +-- LookupError
|      |      +-- IndexError#
|      |      +-- KeyError
|      +-- MemoryError
|      +-- NameError
|      |      +-- UnboundLocalError
|      +-- ReferenceError
|      +-- RuntimeError
|      |      +-- NotImplementedError
|      +-- SyntaxError #Raised when parser encounters a syntax error.
|      |      +-- IndentationError
|      |      +-- TabError
|      +-- SystemError
|      +-- TypeError
|      +-- ValueError
|      |      +-- UnicodeError
|      |      +-- UnicodeDecodeError
|      |      +-- UnicodeEncodeError
|      |      +-- UnicodeTranslateError
+-- Warning#
|      +-- DeprecationWarning
|      +-- PendingDeprecationWarning
|      +-- RuntimeWarning
|      +-- SyntaxWarning
|      +-- UserWarning
|      +-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning

```

11.3 HANDLING EXCEPTION

Exception handling in Python is managed by three keywords, viz. **try**, **except** and **finally**. The working of all of these keywords is explained below.

Program statements that you want to monitor for exception should be in try block. If an exception occurs within the try block, then the particular exception is raised (or thrown). The syntax for exception handling that might raise or throw an exception in a try block is as follows (Figure 11.1).

```
try:
#Body of try block
exceptException_Type1
#Perform of some kind of exception handling
#It is Exception Handler for Exception Type 1, Alerts the user
exceptException_Type2
#Exception Handler for Exception Type 2
.....
.....
.....
.....
exceptException_TypeN
#Exception Handler for Exception Type N
```

Figure 11.1 Syntax—exception handling

Thus, as the Figure 11.1 explains, when an exception is raised (thrown), it is caught by its corresponding except statement, which then processes the exception. As the general form shows, there can be more than one except statement associated with a try. The type of the exception determines which except statement is executed. That is, if the exception type being specified by a except statement matches that of the exception, then that except statement is executed (and all others are then by-passed). If no exception is raised or thrown, then a try block ends normally, and all of its except statements are by-passed.

Points to Remember

When error occurs in a piece of code, exception mechanism performs the following tasks.

1. Finds the problem. (Get the exception.)
2. Informs an error has occurred. (Raise the exception.)
3. Receives the error information. (The except statement)
4. Takes the remedial actions. (Handles the exception.)

11.3.1 Executing the ‘Divide by Zero’ Program Without Exception Handling

Let us consider the following small programming example which demonstrates the scenario being happened when errors arise in those programs not using the exception handling.

PROGRAM 11.1 | Write a program to read two numbers from user and find the quotient by dividing first number by second number.

```
a = int(input('Enter the first number:'))
b = int(input('Enter the second number:'))
c = a/b
print('a = ',a)
print('b = ',b)
print('a/b = ',c)
```

Output

#Case 1

Enter the first number: 10

Enter the second number: 2

a = 10

b = 2

a/b = 5

#Case 2

Enter the first number: 10

Enter the second number: 0

Traceback(most recent call last):

File "C:\Python34\Excel.py", line 3, in <module>

c = a/b

ZeroDivisionError: division by zero

Explanation The Program 11.1 above asks the user to enter two numbers and calculate the quotient by dividing first number by second number. But in output, we can see two different sample outputs. In the first sample execution, the user enters two numbers 10 and 2 and it shows the output as 5. Thus, for the first case, it shows a successful division.

But in second case, the user enters two numbers, viz. 10 and 0. Since the second number entered by user is 0, a run-time error occurs. All of us know that we cannot divide an integer by 0. The following is the exception as generated by Python, if we try to divide a number by 0.

Traceback(most recent call last):

File "C:\Python34\Excel.py", line 3, in <module>

c = a /b

ZeroDivisionError: division by zero



Note: 1. All of the above information generated as errors are known as "Stacktrace".

2. This includes the name of Exception "ZeroDivisionError".

Thus, the next section will demonstrate how to handle these exceptions and enable the program to run till its normal completion by the help of the various programs.

11.3.2 The try and except Blocks to Handle Various Exceptions

Handling Arithmetic Exception

PROGRAM 11.2 | Demonstrate the uses of try and except blocks in order to handle “ZeroDivisionError” in division by zero.

```
try:
    a = int(input('Enter the first number:'))
    b = int(input('Enter the second number:'))
    c = a/b
    print('a = ',a)
    print('b = ',b)
    print('a/b = ',c)
except ZeroDivisionError:
    print('You cannot divide number by zero')
```

Output

#Case 1

```
Enter the first number: 10
Enter the second number: 2
a = 10
b = 2
a/b = 5.0
```

#Case 2

```
Enter the first number: 10
Enter the second number: 0
You cannot divide number by zero
```

Explanation In the above program, the statements that produce exception are written within the try block. Thus, the try block asks the user to enter two numbers and divide first number by second number. If the user enters the second number as 0 then Python generates a run-time exception, i.e. ZeroDivisionError. As soon as the exception has occurred, the exception is raised and it is caught by except keyword.

Multiple except Blocks

In Program 11.2 above, we have seen that the code inside the try block generates only one type of exception and that is handled by a single except block. But, sometimes the code generates multiple exceptions. In such cases, we need to define multiple except blocks to handle those exceptions. The Program 11.3 below illustrates the implementation of multiple except blocks.

PROGRAM 11.3 | Write a Python program to include multiple exceptions.

```
try:
    n1 = int(input('Enter the number:'))
    print(n1)
    q = 200/n1
except ValueError:
    print('Entered string is not of type int')
except ZeroDivisionError:
    print('Number cannot be divided by zero')
```

Output**#Case 1:**

```
Enter the number: svsvadbs
svsvadbs
Entered string is not of type int
```

#Case 2:

```
Enter the number: 0
0
Number cannot be divided by zero
```

Explanation In Program 11.3 above, the statements which we want to monitor for exceptions are kept in try block. Initially, through input the number is the prompt from the user. If user fails to enter the value of type int, then Python raises an error, i.e. "ValueError". Even if user tries to enter the second value as 0, Python interpreter will raise exception "ZeroDivisionError" while executing the statement `q = 200/n1`. Thus, the try statement may contain more than one except block to specify the handlers for different exceptions.

The try-except-finally Block

The finally block sometime referred to as finally clause. The finally block consists of finally keywords. The finally block is placed after the last except block. If there is no except block, the finally block should immediately follow the try block. The finally block will execute whether or not an exception is going to be thrown from the try block. If an exception is thrown, the finally block will execute even if no except statement matches the exception. The syntax to define finally block in exception handling is as follows (Figure 11.2).

```

try:
#Body of try block
exceptException_Type1
#Perform of some kind of exception handling
#It is Exception Handler for Exception Type 1, Alerts the user
exceptException_Type2
#Exception Handler for Exception Type 2
.....
.....
exceptException_TypeN
#Exception Handler for Exception Type N
finally:
.....
.....

```

Figure 11.2 Syntax try-catch finally

As we have seen in previous section, a try statement can have more than one except clauses to handle different exceptions. Thus, along with try and except statements in exception handling block, we can also have an optional finally statement.

PROGRAM 11.4 | To demonstrate the use of finally keyword using try-except block.

```

L1 = [1, 2, 3, 4, 5]
try:
    print(L1)
    n = int(input('Enter the index to retrieve the element:'))
    print('index = ',n,' Element = ',L1[n])
except IndexError:
    print('Please check the index')
    print('Index out of bounds')
finally:
    print('No one can Stop Me from Running');

```

Output

```

[1, 2, 3, 4, 5]
Enter the index to retrieve the element: 10
Please check the index
Index out of bounds
No one can Stop Me from Running

```

Explanation In Program 11.4 above, initially the list of five numbers is created. In try block, the index is prompt from the user to retrieve the element stored at index 'n'. If the entered index is greater than the size of the list, the exception block is executed. At last, the finally block is executed no matters what has been happened in try and except blocks.

11.4 RAISING EXCEPTION

In previous section of this chapter, we have seen Python interpreter raises the exception whenever it tries to execute the invalid code. Raising an exception is also like to stop running the code in the function and move the program execution to the `except` statement. All these exceptions are wrapped in objects and are created from the classes. Therefore, the programmer can raise this exception using the `raise` statement. The syntax to raise an exception is as follows.

Syntax

```
raise Exception(value)
```

Where, `exception` is the exception type.

Example

```
raise ArithmeticError('Something is wrong')
```

PROGRAM 11.5 | Write a Program to raise the exception.

```
A = 10
b = 0
try:
    raise ArithmeticError('Cannot divide number by zero')
    c = a/b
except ZeroDivisionError:
    print('Something is Wrong')
    raise
```

Output

```
Traceback (most recent call last):
  File "C:\Python34\asacs.py", line 4, in <module>
    raise ArithmeticError('Something is Wrong')
ArithmeticError: Cannot divide number by zero
```

Explanation In Program 11.5 above, the `raise` statement allows the programmer to force the specified exception, i.e. `ArithmeticError: Cannot divide number by zero` to occur.

SUMMARY

- ♦ An **exception** is an error that occurs during run time.
- ♦ All of the exceptions are the instances of a class which is derived from **BaseException** class.
- ♦ The three keywords, viz. **try**, **except** and **finally** are used to handle the exceptions.
- ♦ The **raise** keyword is used to raise the exceptions.

KEY TERMS

- ⇒ **Exception:** It is an erroneous condition that arises while program is running.
- ⇒ **try:** It is a sort of program statements which is placed within the try block to look after the exceptions.
- ⇒ **except:** It is a raised exception which is caught by except keyword.
- ⇒ **finally:** The statement within finally block will execute even if no except matches the exception.

REVIEW QUESTIONS

A. Multiple Choice Questions

1. What is the output of the following Program?

```
def getMonth(month):
    if month<1 or month>12:
        raise ValueError("Invalid")
    print(month)
getMonth(13)
```

- a. 13
 - b. 13 and Invalid
 - c. ValueError: Invalid
 - d. None of the above
2. How many except statements does a try block have?
 - a. More than one
 - b. More than zero
 - c. one
 - d. two
 3. When finally block is executed?
 - a. When there is an exception
 - b. When there is no exception
 - c. Always
 - d. It never executes
 4. What is the output of the following program?

```
def demo():
    try:
        return 1
    finally:
        return 2
```

- ```
n = demo()
print(n)
```
- a. 1
  - b. 2
  - c. None
  - d. 1 and 2
5. Which of the following is not a standard exception in Python?
    - a. IOError
    - b. Wrong Assignment Error
    - c. NameError
    - d. ValueError
  6. Which error will be occurred when the following code is executed?

```
Area = 3.14 * radius
```

- a. ValueError
- b. NameError
- c. SyntaxError
- d. KeyError

7. Which error will be occurred when the following code is executed?

```
a = 6
```

```
b = 6 - 12
```

```
c = A/(A + B)
```

a. `NameError`

b. `ValueError`

c. `ZeroDivisionError`

d. None of the above

## B. Exercise Questions

1. Explain the needs of exception handling.
2. Explain the mechanisms to handle exception handling.
3. Write a program to handle the arithmetic exception.
4. Can we have multiple except blocks while handling the exceptions? If yes? How?
5. What are the uses of `raise` keyword? Explain with a simple program.

## PROGRAMMING ASSIGNMENTS

1. Write a program that will ask the user for an input and tries to handle the error when the programmer will try to type cast the input to an `int` using `try` and `except` blocks.
2. Write a function which will take two arguments to concatenate two strings. Assume the type of first argument is unknown but the second argument is the string. Write a Program to handle errors if the programmer will try to concatenate unknown input (unknown input can be of integer types or `float`) type to the string using `try` and `except` blocks.
3. Write a function which will take two parameters to perform the certain task. The first argument is `float` but you are unaware about the second argument. Write a Program that will try to handle the error when you will try to divide the `float` as the unknown input being entered by the user using `try-except` block.
4. Write a function which will take three parameters. The first parameter to the function will be list, the second parameter will be the index position and the third parameter will be string. The function will insert the string at the given index position within the list. In case of a failure, your function should return to the original list. Write a function which will perform this task using the `try` and `except` blocks.



# UNIT 6

---

- A Brief Tour of Standard Library
- Graphics Programming: Drawing with Turtle Graphics
- Unit Testing

# A Brief Tour of Standard Library

## 12

### CHAPTER OUTLINE

|                                               |                                   |
|-----------------------------------------------|-----------------------------------|
| 12.1 Introduction                             | 12.4 The Date and Time Module     |
| 12.2 Operating System Interface—The OS Module | 12.5 Mathematics—The Maths Module |
| 12.3 String Pattern Matching                  | 12.6 Internet Access              |
|                                               | 12.7 Data Compression             |

### LEARNING OUTCOMES

---

*After completing this chapter, students will be able to:*

- Learn interfacing with files and directories of operating system using OS module of Python
  - Learn string matching using re module, i.e. regular expression of Python
  - Learn accessing URL by using urllib module
  - Learn about compression and decompression
  - Access date and time related functionalities based on date and time module
- 

## 12.1 INTRODUCTION

Python standard library contains a large number of useful modules. It is always good for the developers to become familiar with the Python standard library, since many problems can be solved quickly. At the same time, the Python libraries can cut your development time and reduce your frustration with respect to the coding. This chapter will describe a couple of fundamental standard library modules. Most of Python projects make use these modules either directly or indirectly.





- Note:** • Python comes with two basic modules which are primary than all other modules. These two basic modules are **`_builtin_`** and **`exception`** modules.
- The **`_builtin_`** module defines the built-in functions like `range`, `len`, `int` and so on.
  - The `exception` module defines all built-in exceptions.

## 12.2 OPERATING SYSTEM INTERFACE — THE OS MODULE

The **OS** module allows us to use the operating system's dependent functionalities. It also allows us to interact with files and directories. It is used to perform the tasks, such as, to find the name of present working directory, to change current working directory, to check if certain files or directories exist at location, to create and delete the directories, to walk through a directory aiming to perform certain operations on specific files and so on. The OS modules include many functions to interact with the file system. The frequently used functions supported by OS module are listed as follows in Table 12.1.

**Table 12.1** Methods related to OS Interface

| <i>In-built Function Name</i>                                                                                                                                                                                                                                                  | <i>Meaning</i>                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <b>getcwd()</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import os &gt;&gt;&gt; os.getcwd() 'C:\\Python27'</pre>                                                                                                                                                                | Gets the complete path of current working directory.                                         |
| <b>listdir()</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import os &gt;&gt;&gt; os.listdir("C:\\Python27\\Python Programs") ['Hello_World_Python_Program.py', 'PythonProjectDemo.py']</pre>                                                                                    | The <code>listdir()</code> function returns the names of all the files in a given directory. |
| <b>chdir(path)</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; os.getcwd()    #Returns current working directory 'C:\\Python27' &gt;&gt;&gt;os.chdir("C:\\Python27\\Scripts") #Change current working directory &gt;&gt;&gt; os.getcwd() &gt;&gt;&gt;'C:\\Python27\\Scripts'</pre> | Changes the current working directory to that of the given path.                             |
| <b>makedirs(path)</b><br><b>Example:</b><br><pre>&gt;&gt;&gt;import os &gt;&gt;&gt;os.makedirs("C:\\Python_Programming_Practice") #The above command creates new directory of name "Python_Programming_Practice" on the "C" drive.</pre>                                       | Creates a directory at a given path.                                                         |

(Contd.)

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <pre>os.remove(path)</pre> <p><b>Example:</b></p> <pre>&gt;&gt;&gt;import os &gt;&gt;&gt;os.remove("C:\\Python27\\a.py") #Removes the file a.py from the given location.</pre>                                                                                                                                                                                                                                                                                                                         | Removes file from the given path.                                                                            |
| <pre>os.rename(old_name,new_name)</pre> <p><b>Example:</b></p> <pre>&gt;&gt;&gt;import os &gt;&gt; os.rename("E:\\My_Programs","E:\\My_Progs")</pre>                                                                                                                                                                                                                                                                                                                                                   | Changes the name of path from old path to new path.                                                          |
| <pre>rmdir(path)</pre> <p><b>Example:</b></p> <pre>&gt;&gt;&gt;import os &gt;&gt;&gt;os.rmdir("E:\\My_Programs\\Practice")</pre> <p><b>Note:</b> The rmdir() directory will delete directory only when the directory is empty, otherwise the operating system raises the error. In order to remove whole directory trees, shutil.rmtree() can be used.</p>                                                                                                                                             | Removes the directory from the given path.                                                                   |
| <pre>walk()</pre> <p><b>Example:</b></p> <pre>&gt;&gt;&gt;import os &gt;&gt;&gt; Dir = os.chdir('E:') &gt;&gt;&gt; for x in os.walk('.'):         print(x)</pre> <p><b>#The above code returns the files present within the given directory.</b></p> <pre>(('.', [], ['Anaconda3-5.1.0-Windows-x86_64.exe', 'jdk-10_windows-x64_bin.exe', 'jdk-8u161-nb-8_2-windows-x64.exe', 'netbeans-8.2-windows.exe', 'npp.7.5.6. Installer.exe', 'python-3.6.4.exe', 'vlc-3.0.1-win32.exe', 'wrar550.exe'])</pre> | Generates the files names in a directory tree by walking the tree either top to bottom or bottom to upwards. |

### PROGRAM 12.1 | Write program to count ".txt" files.

```
import os
count = 0
Dir1 = os.chdir('C:\Python27\Python Programs')
for (dirname, dirs, files) in os.walk('.'):
 for filename in files:
 if filename.endswith('.txt'):
 print(filename)
 count = count + 1
print('Text Files:',count)
```

#### Output

```
text1.txt
text2.txt
('Text Files:', 2)
```

**Explanation** In Program 12.1 above, `os.chdir()` function is used to change the directory. The Program first generates all of the filenames in given directory using `os.walk()` function. Finally, it just counts `.txt` files and gives total count of `.txt` files being present within the specified directory.



**Note:** The command `dir(os)` when executed on interactive mode will display the list of variables and in-built functions supported by OS module for Python 3.6.

```
>>> dir(os)
['DirEntry', 'F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY',
'O_CREAT', 'O_EXCL', 'O_NOINHERIT', 'O_RANDOM', 'O_RDONLY', 'O_RDWR',
'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEMPORARY', 'O_TEXT', 'O_TRUNC',
'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', 'P_OVERLAY',
'P_WAIT', 'PathLike', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET',
'TMP_MAX', 'W_OK', 'X_OK', 'X_OK', 'Environ', 'all', 'builtins',
'cached', 'doc', 'file', 'loader', 'name', 'package',
'spec', 'execvpe', 'exists', 'exit', 'fspath',
'get_exports_list', 'putenv', 'unsetenv', 'wrap_close', 'abc',
'abort', 'access', 'altsep', 'chdir', 'chmod', 'close', 'closerange',
'cpu_count', 'curdir', 'defpath', 'device_encoding', 'devnull', 'dup',
'dup2', 'environ', 'errno', 'error', 'execl', 'execle', 'execlp',
'execlpe', 'execv', 'execve', 'execvp', 'execvpe', 'extsep', 'fdopen',
'fsdecode', 'fsencode', 'fspath', 'fstat', 'fsync', 'ftruncate',
'get_exec_path', 'get_handle_inheritable', 'get_inheritable',
'get_terminal_size', 'getcwd', 'getcwd', 'getenv', 'getlogin',
'getpid', 'getppid', 'isatty', 'kill', 'linesep', 'link', 'listdir',
'lseek', 'lstat', 'makedirs', 'mkdir', 'name', 'open', 'pardir',
'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 'readlink',
'remove', 'removedirs', 'rename', 'renames', 'replace', 'rmdir',
'scandir', 'sep', 'set_handle_inheritable', 'set_inheritable',
'spawnl', 'spawnle', 'spawnv', 'spawnve', 'st', 'startfile', 'stat',
'stat_float_times', 'stat_result', 'statvfs_result', 'strerror',
'supports_bytes_environ', 'supports_dir_fd', 'supports_effective_ids',
'supports_fd', 'supports_follow_symlinks', 'symlink', 'sys', 'system',
'terminal_size', 'times', 'times_result', 'truncate', 'umask',
'uname_result', 'unlink', 'urandom', 'utime', 'waitpid',
'walk', 'write']
```

## 12.3 STRING PATTERN MATCHING

If you are searching a sub-string from a string, then you can use in-built methods of string, such as `find()`. But if you are searching for something that is not quite specific, then in such cases, you can't use `find()` method of string. Thus, in order to handle complex task of searching and extracting text, Python provides **re (Regular Expression)** module. Regular Expressions are patterns that we can specify as strings and use to search for and replace text in other strings. Python as well as many other languages includes a module to perform various operations on these regular expressions. The most common uses of regular expressions are to search a string, i.e. to search and match, to find a string, to break a string into a substring (`split`) and to replace a part of string (`sub`).

### 12.3.1 The re Module

The re module provides various methods to perform operation on input string. The most commonly used methods of re module are as follows.

a. `re.match(pattern, string)`

The `match()` method returns zero or more characters which match the pattern at the beginning of string. The `match()` function returns none if the string does not matches the pattern.

#### Example 1

```
>>> import re
>>> match_obj = re.match("Python", "Python Programming Language")
>>> match_obj
<_sre.SRE_Match object at 0x05CAA6E8>
```

In Example 1 above, it shows the pattern match has been found. But in order to print the match string, we will use the method `group()`. The method `group()` on match returns the matching string.

#### Example 2

```
>>> import re
>>> match_obj = re.match("Python", "Python Programming Language")
>>> match_obj
<_sre.SRE_Match object at 0x05CAA6E8>
>>> match_obj.group(0) #group() method on match
'Python'
```

In Example 2 above, we have string "Python Programming Language" and from the given string, we are going to search string pattern, i.e. "Python". Thus, the `match()` function returns the matched pattern. What if developer is interested to know the index position of match string from where it starts and ends? Thus, in such cases, we use of the methods, such as, `start()` and `end()`.

#### Example 3

```
>>> match_obj = re.match("Python", "Python Programming Language")
>>> match_obj.start()
0
>>> match_obj.end()
6
```



**Note:** The `match()` method returns none if pattern to be matched from the given string is not present at the beginning of the string.

```
>>> import re
>>> match_obj = re.match("Programming", "Python Programming Language")
>>> print(match_obj)
None
```

b. `re.search(pattern, string)`

The `search()` method same as that of `match()` method of regular expression. The main advantage of `search()` method is that it does not restrict us to find matches at the beginning of the string only.

#### Example 4

```
>>> search_obj = re.search(r"Programming", "Python Programming Language")
>>> print(search_obj)
<_sre.SRE_Match object at 0x053BEAA0>
>>> search_obj.group(0)
'Programming'
>>> search_obj.start()
7
>>> search_obj.end()
18
```



**Note:** The `search()` method is able to find a pattern from any position of the string. But it returns only first occurrence of matched string.

#### Example:

```
>>> search_obj = re.search(r"Good", "If Good Programmer then you can be
Good Developer")
>>> search_obj.group(0)
'Good'
>>> search_obj.start()
3
>>> search_obj.end()
7
```

c. `re.findall(pattern, string)`

The string is scanned from left to right and the pattern is to be searched if matched ones are returned in the order they are found. Finally, it returns the list of all matching patterns from the string.

#### Example 5

```
>>> import re
>>> re.findall("Python", "Programming in Python.I like Python")
['Python', 'Python']
```

d. `re.split(pattern, string, [maxsplit=0]):`

Let us split the string by the occurrences of pattern.

**Example 6**

```
>>> re.split("y","Python")
['P', 'thon']
```

In Example 6 above, we have split the string “Python” by ‘y’. The `split()` method has another argument, i.e. `maxsplit`. The default value of `maxsplit` is ‘0’. If `maxsplit` is ‘0’, then `split()` method does the maximum split. But if we give value to `maxsplit`, then the `maxsplit` will split the string up to the given number of splits.

**Example 7**

```
>>> re.split("-", "978-92-95055-02-5") #maxsplit = 0
['978', '92', '95055', '02', '5']

>>> re.split("-", "978-92-95055-02-5", maxsplit=3)
['978', '92', '95055', '02-5']
```

e. `re.sub(pattern, repl, string)`

The `sub()` method helps us to search a pattern and replace with new sub-strings. If the pattern to be searched is not found, then the original string will remain unchanged.

**Example 8**

```
>>> re.sub("Python", "India", "I Love Python")
'I Love India'
```

f. `re.compile(pattern)`

Let us compile the regular expression pattern into a regular expression object which later can be used for matching by utilizing its `match()`, `search()` and `findall()` methods.

The sequence of using `compile()` and `findall()` is as follow as follows.

```
Comp_obj = re.compile(pattern)
result = Comp_obj.findall(string)
```

**Example 9**

```
>>>Comp_obj = re.compile("Python")
>>>result = Comp_obj.findall("I Love Python")
>>> result
['Python']
```

**12.3.2 Regular Expression Containing Operators and Symbols**

As discussed above, the regular expressions are in general used to search particular pattern from given string. In some cases, it is also used to replace particular pattern within a string based on the set of rules. The rules to search and extract the patterns are defined by making uses of set of operators. The following Table 12.2 contains regular expression patterns.

**Table 12.2** Regular expression containing operators

| Symbol or Pattern | Meaning                                                  |
|-------------------|----------------------------------------------------------|
| .                 | Matches any character except a new line.                 |
| ^                 | Matches beginning of the string.                         |
| \$                | Matches the end of the string.                           |
| x?                | Matches x 0 or 1 time.                                   |
| +                 | Matches x 1 or more times.                               |
| x*                | Matches x 0 or more times.                               |
| []                | Uses to indicate a set of characters.                    |
| \w                | Matches any alphanumeric characters and the underscores. |
| \D                | Matches any non-digit characters.                        |
| \d                | Matches any numeric digit.                               |
| \A                | Matches only at the start of the string.                 |
| a b               | Matches either a or b.                                   |
| x{m}              | Matches exactly 'm' copies of x.                         |
| {m,n}             | It will match m to n repetitions.                        |

### 12.3.3 Basic Example of Regular Expressions

**PROBLEM 1:** Extract each character from the given string, i.e. “Python Programming”

**Solution**

```
>>> ans = re.findall('.', 'Python Programming')
>>> ans
['P', 'Y', 't', 'h', 'o', 'n', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g']
```

**Note** In Problem 1 above, space is also extracted as a character. To avoid it \w is used instead of “.”.

```
>>> ans = re.findall('\w', 'Python Programming')
>>> ans
['P', 'Y', 't', 'h', 'o', 'n', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g']
```

**PROBLEM 2:** Extract words from given string, i.e. “I Love Python”

**Solution**

```
>>> ans = re.findall('\w*', 'I Love Python')
>>> ans
['I', '', 'Love', '', 'Python', '']
```

(Contd.)

In Problem 2 above, it returns the space as word because "\*" returns zero or more different matches. In order to remove space, we will go with '+'.  
 .. .. .

```
>>> ans = re.findall('\w+', 'I Love Python')
>>> ans
['I', 'Love', 'Python']
```

**PROBLEM 3: Extract only first word from the sentence, i.e. "Incredible India"**

**Solution**

```
>>> ans = re.findall('^\w+', 'Incredible India')
>>> ans
['Incredible']
```

**PROBLEM 4: Extract last word from the sentence, i.e. "Incredible India"**

**Solution**

```
>>> ans = re.findall('\w+$', 'Incredible India')
>>> ans
['India']
```

**PROBLEM 5: Extract date from the string, i.e. 'India 15-08-1947'**

**Solution**

```
>>> ans = re.findall('\d{2}-\d{2}-\d{4}', 'India 15-08-1947')
>>> ans
['15-08-1947']
```

**PROBLEM 6: Extract domain names from the given e-mail ids, "abc@gmail.com" and "mit@standford.edu"**

```
>>> re.findall("@\w+(\.\w+)", "abc@gmail.com", "mit@standford.edu")
['com', 'edu']
```

## 12.4 THE DATE AND TIME MODULE

Python provides rich functionalities to deal with date and time data. The standard libraries contain the modules **datetime** which provides number of functions to deal with dates, time and time intervals. Python datetime module supplies classes for manipulating date and times in both simple and complex ways. In general, all the arithmetic operations are supported on date and time. Types in datetime module are given in Table 12.3 as follows.



**Table 12.3** Types in datetime module

| Type      | Description                                                       |
|-----------|-------------------------------------------------------------------|
| Date      | Stores calendar date (year, month, day) using Gregorian calendar. |
| Time      | Stores time of days as hours, minutes, seconds and microseconds.  |
| Datetime  | Stores both date and time.                                        |
| Timedelta | Represents the difference between two date time values.           |
| Tzinfo    | This is a basic type for storing time-zone information.           |

The datetime class allows us to get the current date and time. The statement `'datetime.date.today()'` allows you to get today's date. Similarly, we can save the return date into a variable. The following example demonstrates the same.

**PROGRAM 12.2** | Write Program to store date in variable `Current_date`.

```
import datetime
current_date= datetime.date.today()
print(current_date)
```

**Output**

```
2018-04-22
```

Similarly, you can access different parts of the date, i.e. month, year and day separately. The following program demonstrates the same.

**PROGRAM 12.3** | Write a program to access different parts of date.

```
import datetime
current_date= datetime.date.today()
print('Current Month = ',current_date.month)
print('Date = ',current_date.day)
print('Year = ',current_date.year)
```

**Output**

```
('Current Month = ', 4)
('Date = ', 22)
('Year = ', 2018)
```

In Program 12.3 above, we have made use of the function, i.e. `datetime.date.today()` to stored today's date into variable `current_date`. Similarly, we can display current time by making use of function `datetime.datetime.now()`. The Program 12.4 below demonstrates the use of `datetime.datetime.now()` function to store current time into variable `current_time`.

**PROGRAM 12.4** | Write Program to store current time in variable `Current_time`.

```
import datetime
current_time = datetime.datetime.now()
print(current_time)
```

**Output**

```
2018-04-22 20:48:05.324000
```

**12.4.1 Date Formats**

Different users and different countries make use of different date formats. So, there is always scope to handle different date formats by the same time. The default date format is "YYYY-MM-DD". Python makes the use of `strftime()` to format dates. Following Table 12.4 contains some of the format specifiers used with `strftime()` to format dates according to the user requirements.

**Table 12.4** Format date specifiers

| <i>Format Specifier</i> | <i>Meaning</i>                          |
|-------------------------|-----------------------------------------|
| %d                      | Represents the day of month.            |
| %B                      | Represents full month's name.           |
| %b                      | Represents the month.                   |
| %Y                      | Represents the year.                    |
| %y                      | Represents two digit years.             |
| %a                      | Represents abbreviated day of the week. |
| %A                      | Represents full name of the day.        |

**PROGRAM 12.5** | Demonstrate the uses of `strftime()` function to format date.

```
import datetime
current_date = datetime.date.today()
print(current_date.strftime('Date Formatting: %d %b %y'))
```

**Output**

```
Date Formatting: 22 Apr 18
```

**12.4.2 Time Module**

The time module provides the number of functions that deals with date and time within a day. When the command `dir(time)` is executed on interactive mode, it gives the list of functions available under time module.

```

>>> dir(time)
['__doc__', '__name__', '__package__', 'accept2dyear', 'altzone', 'asctime',
'clock', 'ctime', 'daylight', 'gmtime', 'localtime', 'mktime', 'sleep',
'strftime', 'strptime', 'struct_time', 'time', 'timezone', 'tzname']

```

The following Table 12.5 gives details about the some of the frequently used functions of the time module.

**Table 12.5** Functions related to time module

| <i>In-built functions</i>                                                                                                                                                                                                           | <i>Meaning</i>                                                                                                                                                                                                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>clock()</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import time &gt;&gt;&gt; time.clock() 366.2016504055217</pre>                                                                                                                | In Unix operating system, the function <code>clock()</code> returns the current processor time as floating point being expressed in seconds.<br>On Windows, this function returns wall-clock seconds as elapsed. |
| <b>ctime()</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import time &gt;&gt;&gt; time.ctime() 'Sun Apr 22 19:05:05 2018'</pre>                                                                                                       | Returns the current time.                                                                                                                                                                                        |
| <b>localtime()</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import time &gt;&gt;&gt; time.localtime() time.struct_time(tm_year=2018, tm_mon=4, tm_mday=22, tm_hour=19, tm_min=6, tm_sec=3, tm_wday=6, tm_yday=112, tm_isdst=0)</pre> | Returns the current time as that of <code>ctime</code> but in different way.                                                                                                                                     |
| <b>strptime(string[,format])</b><br><b>Example:</b><br><pre>time.strptime("21 Apr 18", "%d %b %y") time.struct_time(tm_year=2018, tm_mon=4, tm_mday=21, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=5, tm_yday=111, tm_isdst=-1)</pre>   | Parses a string representing a time according to a format.                                                                                                                                                       |
| <b>sleep(secs)</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import time &gt;&gt;&gt; time.sleep(5) #Stops the execution for 5 seconds</pre>                                                                                          | Suspends execution of the current thread for the given number of seconds.                                                                                                                                        |

## 12.5 MATHEMATICS—THE MATHS MODULE

Python provides additional list of functions defined under Python's math module to solve problems related to mathematical calculations. The following Table 12.6 describes the list of function under math module.

**Table 12.6** The in-built functions of maths module

| Function                       | Example                                                         | Description                                                       |
|--------------------------------|-----------------------------------------------------------------|-------------------------------------------------------------------|
| ceil(X)                        | >>>import math<br>>>>math.ceil(10.23)<br>11                     | Rounds X to nearest integer and returns that integer.             |
| floor(X)                       | >>>import math<br>>>> math.floor(18.9)<br>18                    | Returns the largest value not greater than X.                     |
| exp(X)                         | >>>import math<br>>>> math.exp(1)<br>2.718281828459045          | Returns the exponential value for $e^x$ .                         |
| log(X)                         | >>>import math<br>>>> math.log(2.71828)<br>0.999999327347282    | Returns the natural logarithmic of X (to base e).                 |
| log(X,base)                    | >>>import math<br>>>> math.log(8,2)<br>3.0                      | Returns the logarithmic of X to the given base.                   |
| sqrt(X)                        | >>>import math<br>>>>math.sqrt(9)<br>3.0                        | Returns the square root of X.                                     |
| <i>Trigonometric Functions</i> |                                                                 |                                                                   |
| sin(X)                         | >>>import math<br>>>> math.sin(3.14159/2)<br>0.9999999999991198 | Returns the sine of X. Where X is the value in radians.           |
| asin(X)                        | >>>import math<br>>>> math.asin(1)<br>1.5707963267948966        | Returns the angle in radians for the inverse of sine.             |
| cos(X)                         | >>>import math<br>>>> math.cos(0)<br>1.0                        | Returns the sine of X. Where X is the value in radians.           |
| aCos(X)                        | >>>import math<br>>>> math.acos(1)<br>0.0                       | Returns the angle in radians for the inverse of cosine.           |
| tan(X)                         | >>>import math<br>>>> math.tan(3.14/4)<br>0.9992039901050427    | Returns the tangent of X. Where X is the value in radians.        |
| atan(x)                        | >>>import math<br>>>> math.atan(0.999)<br>0.784897913314115     | Returns the arc tangent of x in radians.                          |
| hypot(x,y)                     | >>>import math<br>>>> math.hypot(3,4)<br>5.0                    | Returns the length of the vector from the origin to point (x, y). |

(Contd.)

| <i>Angular Conversion Functions</i> |                                                                    |                                                |
|-------------------------------------|--------------------------------------------------------------------|------------------------------------------------|
| degrees(X)                          | >>>import math<br>>>> math.degrees(1.57)<br>89.95437383553924      | Converts angle X from radians to degrees.      |
| Radians(X)                          | >>>import math<br>>>> math.radians(89.99999)<br>1.5707961522619713 | Converts angle X from degrees to radians.      |
| <i>Constants</i>                    |                                                                    |                                                |
| pi                                  | >>> import math<br>>>> math.pi<br>3.141592653589793                | Returns the constant value of pi, i.e. 3.14.   |
| e                                   | >>> import math<br>>>> math.e<br>2.718281828459045                 | Returns the constant value of e, i.e. 2.71828. |

## 12.6 INTERNET ACCESS

Python has number of modules for accessing the Internet and processing Internet protocols. Following are the various modules of Internet access discussed in brief.

### 12.6.1 The urllib Module

The **urllib** is a package. It is a collection of multiple modules for working with URLs (Uniform Resource Locators). This module is used to interact with websites to perform any of the tasks related to website such as get data, post data or parse data. Table 12.7 contains modules within urllib.

**Table 12.7** Modules within urllib module

| <i>Module Name</i> | <i>Purpose</i>                                                                         |
|--------------------|----------------------------------------------------------------------------------------|
| urllib.request     | To open and read the URLs                                                              |
| urllib.errors      | To identify exceptions raised by urllib.request<br>exceptions raised by urllib.request |
| urllib.parse       | Used for parsing URL's.                                                                |
| urllib.robotparse  | Used for parsing robots.txt files                                                      |

#### **The urllib.request Module**

The urllib module is used for opening and fetching the URLs. The most commonly used method of urllib.request is `urlopen(url)`. The method `urlopen(url)` opens and fetches the requested URL.

#### **Example**

```
>>> url_name = urllib.request.urlopen('https://www.google.com/')
>>> url_name
<http.client.HTTPResponse object at 0x05CA3250>
>>>url_name.geturl() #geturl() returns name of requested URL
'https://www.google.com/'
```

## The urllib.parse Module

The urllib.parse module provides functions for manipulating URLs. This module provides a standard interface to break the URL string into components, such as, path, network location, addressing scheme etc. It is used to combine the broken components back to URL string and also to convert the “relative” URL to an “absolute” URL.

The function and working example of urllib.parse are given below in brief as follows.

### a. The urllib.parse.urlparse(url) Method:

The general structure of any URL is “scheme://netloc/path;parameters?query#fragment”.

Where,

scheme: A URL scheme specifier  
 netloc: Network location part  
 path: Hierarchical path  
 params: Parameters for last path element  
 query: Query component  
 fragment: Fragment identifier

So, in order to parse any URL, the function urlparse() is used. The function parses a URL into six components and returns to six tuples.

### Example

```
>>>from urllib.parse import urlparse
>>> url = "https://en.wikipedia.org/wiki/National_symbols_of_India"
>>> parsed = urlparse(url)
>>> parsed
ParseResult(scheme='https', netloc='en.wikipedia.org',
path='/wiki/National_symbols_of_India', params='', query='', fragment='')
```

**PROGRAM 12.6** | Write simple program to make use of urlparse() function and store the ans of all returned components into different variable.

```
from urllib.parse import urlparse
url = 'https://en.wikipedia.org/wiki/National_symbols_of_India'
parsed = urlparse(url)
print('scheme:', parsed.scheme)
print('netloc:', parsed.netloc)
print('path:', parsed.path)
print('params:', parsed.params)
print('query:', parsed.query)
print('fragment:', parsed.fragment)
print('username:', parsed.username)
print('password:', parsed.password)
print('hostname:', parsed.hostname)
print('port:', parsed.port)
```

(Contd.)

**Output**

```
scheme: https
netloc: en.wikipedia.org
path: /wiki/National_symbols_of_India
params:
query:
fragment:
username: None
password: None
hostname: en.wikipedia.org
port: None
```

## 12.7 DATA COMPRESSION

In general, there are two different types of data compression techniques, i.e. lossless and lossy. The lossless compression allows original data to be perfectly reconstructed from the compressed data. The lossless compression technique is used in many applications, e.g. in **ZIP** file format. Where, in contrast, the lossy compression is a representation that allows you to reproduce something much similar to original data set. In case of lossy compressions, some amount of data may lose during the process.

Thus, in short, the lossless compression is used where original and decompressed data are identical. For example, source code and executable program are one of the best examples of lossless compressions. Compression of audio file is an example of lossy compression, where we can play lossy audio files on other portable player.

### 12.7.1 The `zlib` Library

Python “**zlib**” library allows compression and decompression for applications that requires data compression. The most frequently used in-built functions supported by **zlib** library are as follows.

**Table 12.8** In-built functions supported by **zlib** library

| <i>In-built Functions</i>              | <i>Meaning</i>                                                                                                                                                                                                                              |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>compress(string, [level])</code> | Compress data in a string. Returns string containing compressed data. The level indicates the level of compressions from 0 to 9. Level 1 is fastest and produces the least compression, while level 9 is the slowest and produces the most. |
| <code>decompress(data)</code>          | Decompress bytes in data. It returns bytes object containing the uncompressed data.                                                                                                                                                         |

(Contd.)

|                                                                    |                                            |
|--------------------------------------------------------------------|--------------------------------------------|
| <code>Compressobj</code>                                           | Decompress bytes in data. It returns bytes |
| <code>(level=1, method=DEFLATED, wbits=MAX_WBITS,</code>           | object containing the uncompressed data.   |
| <code>memLevel=DEF_MEM_LEVEL, strategy=Z_</code>                   |                                            |
| <code>DEFAULT_STRATEGY[, zdict])</code>                            |                                            |
| <code>memLevel</code> : Amount of memory used for internal         |                                            |
| compression state                                                  |                                            |
| <code>strategy</code> : Used to tune the compression algorithm     |                                            |
| <code>wbits</code> : Used to control window size while compressing |                                            |
| data. It decides whether header and trailer to be included         |                                            |
| in output or not. It can take range of values limited to the       |                                            |
| 15, i.e. MAX_WBITS                                                 |                                            |

**PROGRAM 12.7** | Write a Program to show the compression and decompression using `zlib` library.

```
import zlib
Message = "Launch Satellite from Earth"
compressed_message = zlib.compress(Message)
print('Message before Compression (Original):')
print(compressed_message)
decompressed_message = zlib.decompress(compressed_message)
print('Decompressed Message:', decompressed_message)
```

### Output

```
Message before Compression (Original):
xœóI,MîP.IÍÊÊ,IUH+ÊÏÛPM,*É
('Decompressed Message:', 'Lauch Stellite from Earth')
```

**Explanation** In Program 12.7 above, we have used in-built functions for the compression and decompression of `zlib` library. Programmer should also make note that the compression rate varies depending upon the contents of file.

## 12.7.2 Compressing Large Data Streams

In Program 12.7 above, we have seen `compress()` function used to compress a string of data. But, if we have a large data streams, then compressions of large data streams can be achieved by the in-built function, i.e. `compressobj()`. The syntax of `compressobj()` is as follows.

```
compressobj(level=-1, method=DEFLATED, wbits=15, memLevel=8, strategy=Z_DEFAULT_
STRATEGY[, zdict])
```

The details of above syntax are given in Table 12.9 where the `wbits` controls the window size. It also decides whether the header and trailer will be included in the output. The possible values of `wbits` are as follows.



**Table 12.9** Details of `wbits()`

| Value      | Window Size (Logarithm)              | Output                                     |
|------------|--------------------------------------|--------------------------------------------|
| +9 to 15   | Base 2                               | Includes zlib header and trailer           |
| -9 to -15  | Absolute value of <code>wbits</code> | No header and trailer                      |
| +25 to +31 | Low 4 bits of the value              | Includes gzip header and trailing checksum |

**PROGRAM 12.8** | Write a Program to demonstrate the use of `compressobj()`.

```
import zlib
original_data = open("C:\NPKey.txt", 'rb').read()
print(original_data)
compressed = zlib.compressobj(zlib.Z_DEFAULT_COMPRESSION, zlib.DEFLATED, -15)
compressed_data = compressed.compress(original_data)
compressed_data += compressed.flush()
print('Compressed Data ', (compressed_data))

compress_ratio = (float(len(original_data)) - float(len(compressed_data))) /
float(len(original_data))

print('Compressed: %d%%' % (100.0 * compress_ratio))
```

**Output**

```
E-18CEED3258
HR-6957-1F5E-8A29-87AE-D8B7
JWZN-ENGP-VNKW-PKPK-NPJJ-KGVN
26/09/2017
Dealer Code:
Dealer Name:
Dealer Mob.:
'Compressed Data ', 's\xd55\xb4pvuu162\xb5\xe0\xe5\xf2\x08\xd25\xb345\xd75t3u\
xd5\xb5p4\xb2\xd4\xb50wt\xd5u\xb1p4\xe7\xe5\xf2\n\x8f\xf2\xd3u\xf5s\x0f\xd0\r\
xf3\xf3\x0e\xd7\r\xf0\x0e\xf0\xd6\xf5\x0b\xf0\xf2\xd2\xf5v\x0f\xf3\xe3\xe522\
xd37\xb0\xd4720\x04\x9a\xc3xcb\xe5\x92\x9a\x98\x93Z\xa4\xe0\x9c\x9f\x92\xaa`\
xa5\x00\xe7\xfb%\xe6\xa2\xf0}\xf3\x93\xf4\xc0|\x00'
Compressed: 15%
```

**Explanation** In Program 12.8 above, we have seen the data within the file is compressed. Generally, the `compressobj()` is used to compress when data streams would not fit into the memory at once. The function `flush()` is used to go towards next chunk while leaving previous chunk cleaned up by the garbage collection.

**Decompressing Large Data Streams**

Decompressing large data streams requires large amount of memory to decompress. But, sometime, it is not possible to use all of the available memories for performing this task. So zlib library

contains another function named `decompressobj()`, which performs decompression of data in chunks. The syntax of `decompressobj()` is as follows.

```
decompressobj(wbits=15[, zdict])
```

The function decompresses the bytes where the `wbits` controls the window size. It also asks for the header and trailer of the compressed file. The `wbits` of `decompressobj()` can take following possible values (see Table 12.10).

**Table 12.10** The `wbits` of `decompressobj()` – Possible values

| <i>Value</i>                | <i>Window Size Logarithm</i>                                  | <i>Input</i>                           |
|-----------------------------|---------------------------------------------------------------|----------------------------------------|
| 0                           | Automatically determines the window size from the zlib header | zlib header                            |
| +8 to +15                   | Base 2                                                        | Includes zlib header and trailers      |
| –8 to –15                   | Absolute value of <code>wbits</code>                          | Raw stream with no header and trailers |
| +24 to +31 = 16 + (8 to 15) | Lower 4 bits                                                  | Includes gzip header and trailers      |
| +40 to +47 = 32 + (8 to 15) | Lower 4 bits                                                  | zlib or gzip formats                   |

**PROGRAM 12.9** | Write a Program to demonstrate the use of `decompressobj()`.

```
import zlib
original_data = "HELLO Welcome to Python"
print(original_data)

compressed = zlib.compressobj(zlib.Z_DEFAULT_COMPRESSION, zlib.DEFLATED, +15)
compressed_data = compressed.compress(original_data)
compressed_data+ = compressed.flush()
print('compressed_data = ', compressed_data)

f = open('compressed_Original_Data.txt', 'w')
f.write(compressed_data)
f.close()

CHUNKSIZE = 1024
data_to_decompress = zlib.decompressobj()
my_file = open('compressed_Original_Data.txt', 'rb')
buf = my_file.read(CHUNKSIZE)

while buf:
 decompressed_data = data_to_decompress.decompress(buf)
 buf = my_file.read(CHUNKSIZE)
```

(Contd.)

```
decompressed_data+ = data_to_decompress.flush()
print('Decompressed data: ' + decompressed_data)
my_file.close()
```

### Output

```
HELLO Welcome to Python
('compressed_data = ', '\x\x9c\xf3p\xf5\xf1\xf1W\x08O\xcdI\xce\xcfMU(\xc9W\x08\
xa8\xc9\xc8\xcf\x03\x00Y\xee\x08\x06')
Decompressed data: HELLO Welcome to Python
```

**Explanation** In Program 12.9 above, initially the original message is compressed and compressed data is placed into the file named 'compressed\_Original\_Data.txt'. The data is then compressed by making use of `compressobj()` function. Finally, the compressed data is read from the file, viz. 'compressed\_Original\_Data.txt'. To decompress the compressed file, the function `decompressobj()` is used.

## SUMMARY

- ◆ Python comes with two basic modules, such as, `_builtin_` and `exception`.
- ◆ The `os` module allows you to interact with files and directories of operating system.
- ◆ Advanced searching and matching patterns can be done by using regular expressions.
- ◆ The date and time module can be accessed and manipulated using `datetime` module.
- ◆ Various modules of Internet are used to interact with websites to perform any of the tasks related to website, such as, get data, post data or parse data.
- ◆ The lossless and lossy are two different types of compression techniques.

## KEY TERMS

- ⇒ **os module:** It allows to use operating system dependent functionality.
- ⇒ **re module:** The regular expression module is used to handle complex task of searching and extracting text.
- ⇒ **Datetime:** This module contains the classes to manipulate date and time.
- ⇒ **urllib:** It is the collection of multiple modules to work with the URLs.
- ⇒ **zlib:** It allows the compression and decompression.

### A. Multiple Choice Questions.

1. What is the output of following code?  

```
import datetime
res = datetime.date(2018,5,17)
print(res)
```

  - a. 2018-05-17
  - b. 2018-17-05
  - c. 05-17-2018
  - d. None of the above
2. What is the output of following code?  

```
tday=datetime.date.today()
print(tday.year)
```

#Note: Assume Current Year as 2018

  - a. 2018
  - b. 2017
  - c. 2019
  - d. None of the above
3. What is the output of the following code as shown below while the system date is 05-07-2018?  

```
tday=datetime.date.today()
tdelta=datetime.timedelta(days=20)
print(tday+tdelta)
```

  - a. 2018-05-17
  - b. 2018-05-27
  - c. 2018-05-7
  - d. 2018-27-05
4. Which of the following codes is used to create a directory?
  - a. `os.mkdir(path)`
  - b. `os.mdir(path)`
  - c. `os.createdir(path)`
  - d. `os.create_directory(path)`
5. Which of the following methods does return the current working directory?
  - a. `os.getd()`
  - b. `os.getcwd()`
  - c. `os.getpwd()`
  - d. `os.pwd()`

### B. Exercise Questions

1. Explain trigonometric functions supported by `maths` module.
2. Explain `match()` and `search()` functions being supported by `re`, i.e. regular expression module.
3. Explain the needs of regular expression.
4. Explain how to compress the large data streams.
5. Enlist at least six different parameters of `urllib.parse()`.

## PROGRAMMING ASSIGNMENTS

1. Write a Program to count “.pdf” files from any directory.
2. Write a Program to delete “.txt” files from any directory.
3. Write a Program to compress and decompress the text  
“Welcome to Python Programming”.
4. Write a Program to input two sides of a right-angled triangle and calculate hypotenuse using the in-built trigonometric functions of maths module.
5. Write a Program to extract all of those words that begin with vowels.

**Example:** “Abc def efg”

**Output:** [“Abc , efg”]

# Graphics Programming: Drawing with Turtle Graphics

## 13

### CHAPTER OUTLINE

|      |                                                       |      |                                       |
|------|-------------------------------------------------------|------|---------------------------------------|
| 13.1 | Introduction                                          | 13.6 | Drawing with Colors                   |
| 13.2 | Getting Started with the Turtle Module                | 13.7 | Drawing Basic Shapes using Iterations |
| 13.3 | Moving the Turtle in any Direction                    | 13.8 | Changing Color Dynamically using List |
| 13.4 | Moving Turtle to Any Location                         | 13.9 | Turtles to Create Bar Charts          |
| 13.5 | The color, bgcolor, circle and Speed Method of Turtle |      |                                       |

### LEARNING OUTCOMES

*After completing this chapter, students will be able to:*

- Create simple graphics using the Turtle module
- Draw different geometric figures, such as lines, circles, rectangles, squares and polygons using the Turtle
- Draw basic shapes using iterations
- Draw simple charts

### 13.1 INTRODUCTION

A simple way to start learning graphics programming is to use the inbuilt **Turtle** module in Python. The Turtle module is a graphics package for drawing lines, circles and various other shapes, including text. In short, the **Turtle** is a **cursor** on the screen to draw graphics related things. Importing the Turtle module helps a programmer to access all graphics functions in Python.

## 13.2 GETTING STARTED WITH THE TURTLE MODULE

To start, a programmer can use **interactive mode (command line)** or **script mode** of Python. The steps required to start graphics programming using the Turtle module in interactive mode of Python are given as follows:

- ⊙ **STEP 1:** Launch Python by pressing the start button in Windows and writing Python in the search box. Click on Python IDLE to start the interactive mode. The following window will then appear (Figure 13.1).

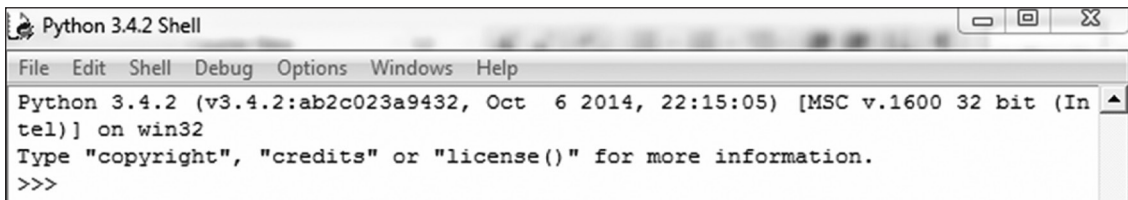


Figure 13.1

- ⊙ **STEP 2:** At the Python's statement prompt >>> type the following command to import the Turtle module.  

```
>>> import Turtle #import Turtle module
```
- ⊙ **STEP 3:** Type the following command to show the current location and direction of the Turtle.  

```
>>> Turtle.showTurtle()
```

After the execution of the above statement, Python's Turtle graphics window will be displayed as shown in Figure 13.2.

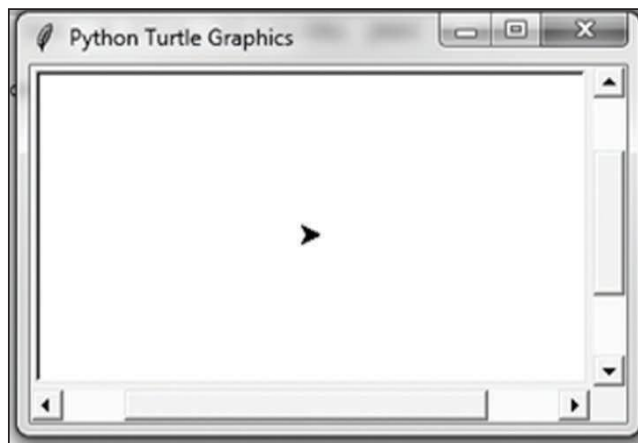


Figure 13.2 Python's Turtle Graphics Window

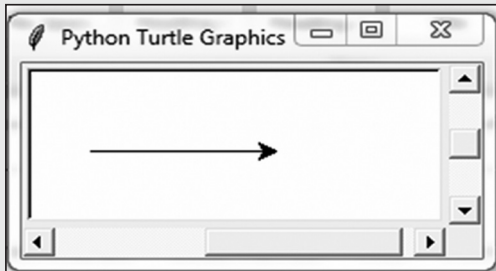
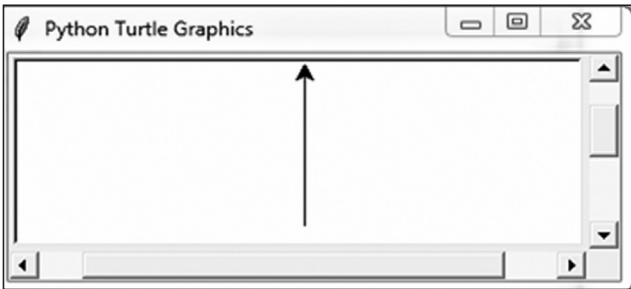
The Turtle is like a **pen**. The arrowhead indicates the current position and direction of the pen. Initially, the **Turtle** is positioned at the center of the window.

### 13.3 MOVING THE TURTLE IN ANY DIRECTION

As discussed above, the Turtle is an object which is created when we import the Turtle module. As soon as the object is created its position is set at **(0, 0)**, i.e. at the center of the Turtle graphics window. Also by default its direction is set to go straight to the right.

The imported Turtle module uses a pen to draw shapes. It can be used to move and draw lines in any direction. Python contains methods for **moving the pen, setting the pen's size, lifting and putting the pen down**. By default, the pen is down, i.e. it draws a line from the current position to the new position. Table 13.1 shows a list of methods to move the Turtle in specified directions.

**Table 13.1** Turtle methods related to directions

| Method                                                                                                                                              | Meaning                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Turtle.forward(P)</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import Turtle &gt;&gt;&gt; Turtle.forward(100)</pre>                               | Moves the Turtle P pixels in the direction of its current heading.<br><b>Output:</b>                                                                                                                                                                                                                       |
| <b>Turtle.left(angle)</b><br><b>Example:</b><br><pre>&gt;&gt;&gt; import Turtle &gt;&gt;&gt; Turtle.left(90) &gt;&gt;&gt; Turtle.forward(100)</pre> | Rotates the Turtle left by the specified angle.<br><b>Output:</b>  <p><b>Explanation:</b> Initially the Turtle is placed at the centre by default. The command <b>Turtle.left(90)</b> changes the direction of the Turtle to the left by 90 degrees. Finally, the arrowhead moves 100 pixels forward.</p> |

(Contd.)

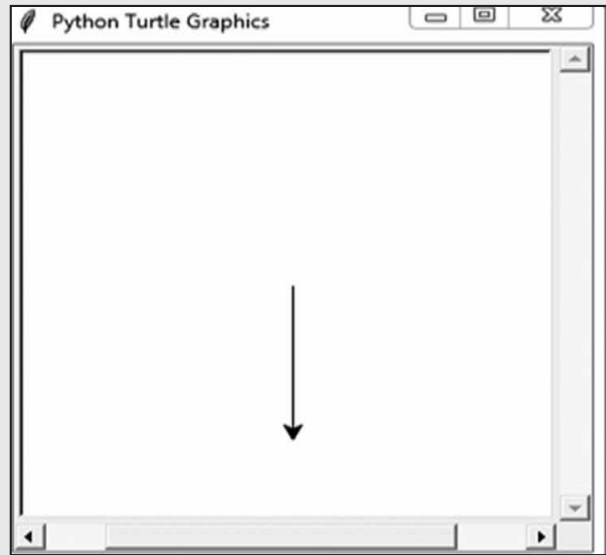


**right(P)**

Rotates the Turtle in place a degree clockwise.

**Example:****Output:**

```
>>> import Turtle
>>> Turtle.right(90)
>>> Turtle.forward(100)
```



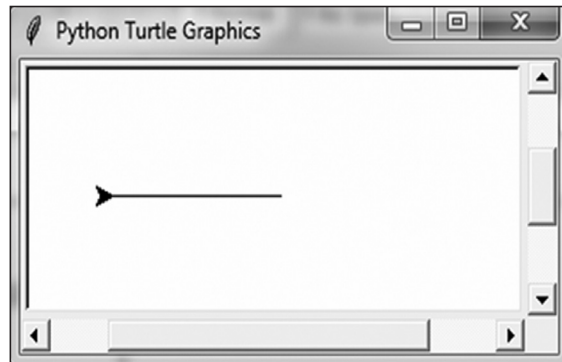
**Explanation:** Initially the Turtle is placed at the centre by default. The command **Turtle.right(90)** changes the direction of the Turtle to the right by 90 degrees. Finally, the arrowhead moves 100 pixels forward.

**backward(P)**

Moves the Turtle P pixels in a direction opposite to its current heading.

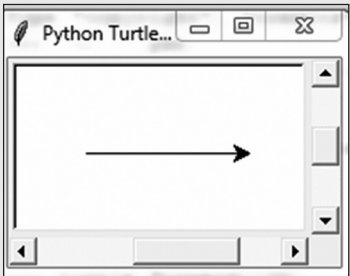
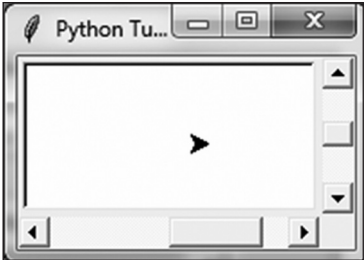
**Example:****Output:**

```
>>> import Turtle
>>> Turtle.backward(100)
```



In Table 13.1, we used various methods to move the Turtle from one position to the other. As discussed above, the Turtle draws a line from one position to the other with the help of the pen. Table 13.2 illustrates various methods related to the state of a pen.

**Table 13.2** Methods related to the state of a pen

| Method                                                                                                                                                           | Meaning                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><code>Turtle.pendown()</code></p> <p><b>Example:</b></p> <pre>&gt;&gt;&gt; import Turtle &gt;&gt;&gt; Turtle.pendown() &gt;&gt;&gt; Turtle.forward(100)</pre> | <p>Pulls the pen down. Draws when it moves from one place to the other.</p> <p><b>Output:</b></p>  <p><b>Explanation:</b> In the above example, the method <b>Turtle.pendown()</b> <b>draws different</b> shapes when it moves from one place to the other.</p>                                                                                                                                                                                                                                                                          |
| <p><code>Turtle.penup()</code></p> <p><b>Example:</b></p> <pre>&gt;&gt;&gt; import Turtle &gt;&gt;&gt; Turtle.penup() &gt;&gt;&gt; Turtle.forward(100)</pre>     | <p>Pulls the pen up. In this state, it just moves from one place to the other without drawing anything.</p> <p><b>Output:</b></p>  <p><b>Explanation:</b> The import Turtle method places the pen at the center of the circle. The <b>Turtle.penup()</b> doesn't allow a programmer to draw things, it just moves from one place to the other. When the statement <code>Turtle.forward(100)</code> is executed immediately after the <code>penup()</code> statement, it moves 100 pixels forward without drawing any line or shape.</p> |

(Contd.)

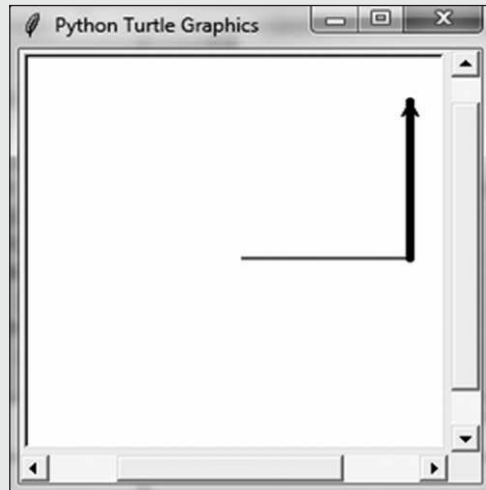
```
Turtle.pensize(width)
```

Sets the line thickness to the specified width.

**Example:**

**Output:**

```
>>> import Turtle
>>> Turtle.forward(100)
>>> Turtle.pensize(5)
>>> Turtle.pensize(5)
>>> Turtle.left(90)
>>> Turtle.forward(100)
```

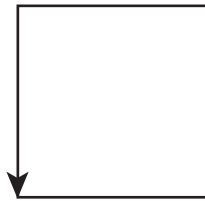


**Explanation:** In the above code, initially the line is drawn 100 pixels ahead in the forward direction. The statement **Turtle.pensize(5)** increases the thickness to draw the figure from here onwards.

### 13.3.1 Programs to Draw Different Shapes

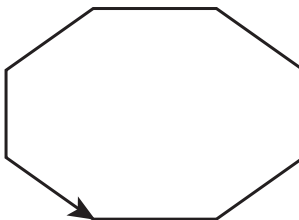
The following programs make use of methods discussed above to draw different shapes.

**PROGRAM 13.1** | Write a program to draw the square shown as follows using Python's Turtle module.



```
import Turtle #import Turtle module
Turtle.forward(100) #Move Turtle in forward direction
Turtle.left(90) #Change the direction of Turtle to left by 90 degree
Turtle.forward(100)
Turtle.left(90)
Turtle.forward(100)
Turtle.left(90)
Turtle.forward(100)
```

**PROGRAM 13.2** | Write a program to display the polygon shown as follows:

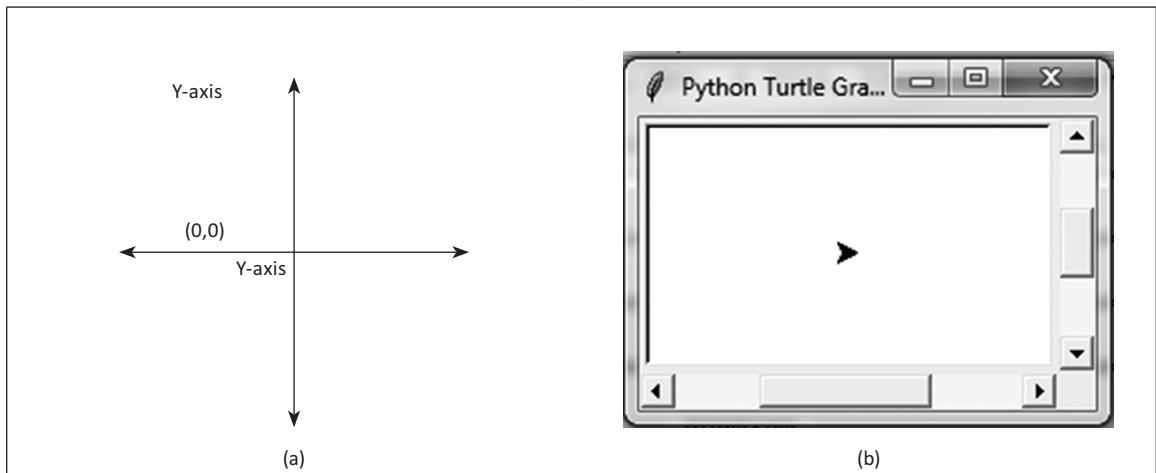


```
import Turtle #import Turtle module
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
Turtle.left(45)
Turtle.forward(50)
```

## 13.4 MOVING TURTLE TO ANY LOCATION

When a programmer tries to run Python's Turtle graphics program by default, the Turtle's arrowhead (**Cursor** or **Pen**) is at the center of the graphics window at coordinate(0, 0) as shown in Figure 13.3.

```
>>> import Turtle #import Turtle module
>>> Turtle.showTurtle ()
```



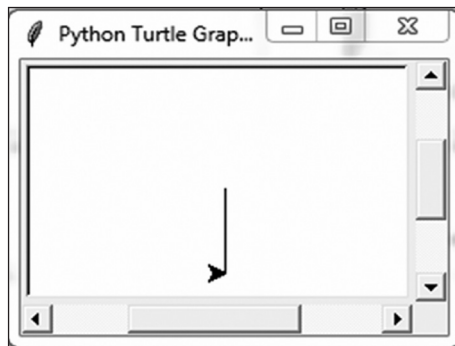
**Figure 13.3** (a) Representation of coordinate system (b) Centre of Turtle graphics at (0, 0)

The method **goto(x, y)** is used to move the Turtle at specified points (x, y). The following example illustrates the use of **goto(X, Y)** method.

### Example

```
>>> import Turtle
>>> Turtle.showTurtle ()
>>> Turtle.goto(0, -50)
```

### Output



**Figure 13.4**

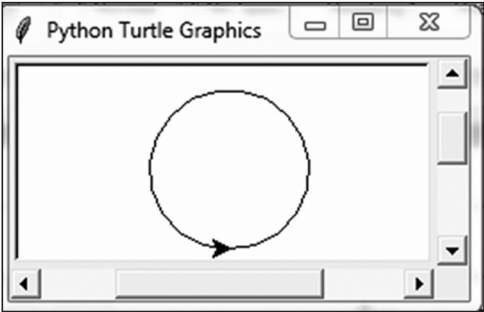
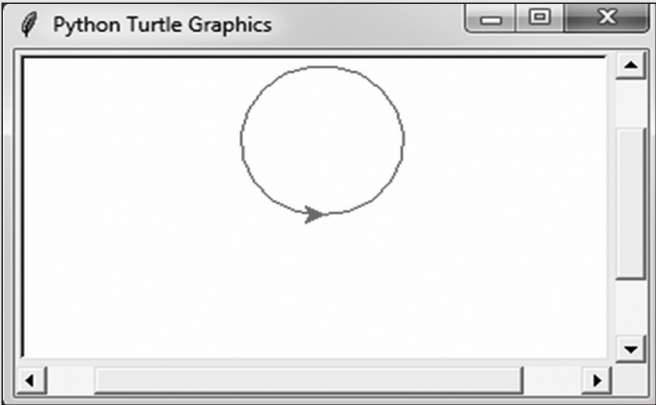
### Explanation

In the above example, the statement **goto(0,-50)** will move towards coordinate (0, -50).

# 13.5 THE COLOR, BGCOLOR, CIRCLE AND SPEED METHOD OF TURTLE

Table 13.3 gives more details about color, bgcolor, circle and speed method of the Turtle.

**Table 13.3** Turtle methods related to color and speed of the Turtle

| Method                                                                                                                                           | Meaning                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Turtle.speed(integer _<br>paramter)                                                                                                              | The drawing speed of the Turtle must be in the range int 1 (slowest) to 10 (fastest) or 0 (instantaneous).                                                                                                                                                                                         |
| Turtle.circle(radius,<br>extent=None)<br><b>Example:</b><br><pre>&gt;&gt;&gt; import Turtle &gt;&gt;&gt; Turtle.circle (45)</pre>                | Draws a circle with the given radius. The center is radius units left of the Turtle. The extent determines which part of the circle is drawn. If it is not given, the entire circle is drawn.<br><b>Output:</b>  |
| Turtle.color(*args)<br><b>Example:</b><br><pre>&gt;&gt;&gt; import Turtle &gt;&gt;&gt; Turtle.color("red") &gt;&gt;&gt; Turtle.circle (45)</pre> | The color method is used to draw colorful animations.<br><b>Output:</b>                                                                                                                                        |
| <b>Explanation:</b> The above statement draws a circle in red color.                                                                             |                                                                                                                                                                                                                                                                                                    |

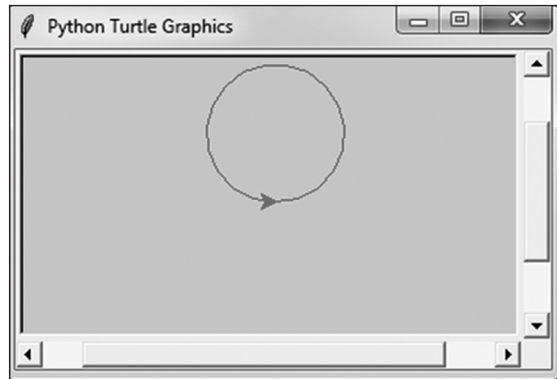
(Contd.)

```
Turtle.bgcolor(*arg)
```

**Example:**

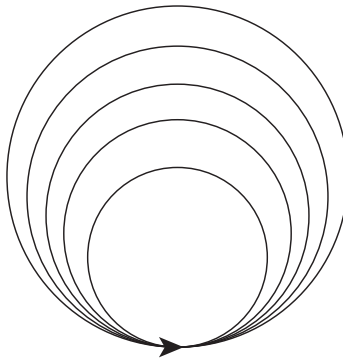
```
>>> import Turtle
>>> Turtle.color("red")
>>> Turtle.
bgcolor("pink")
```

Returns the background color of the Turtle screen.

**Output:**

**Explanation:** Changes the background color of the Turtle graphics window to pink.

**PROGRAM 13.3** | Write a program to display the circles shown. You can consider any radius.



```
import Turtle
Turtle.circle (45)
Turtle.circle (55)
Turtle.circle (65)
Turtle.circle (75)
Turtle.circle (85)
```

**Explanation** In the above program, the 5 circles are drawn with different radius, viz. 45, 55, 65, 75 and 85, respectively.

## 13.6 DRAWING WITH COLORS

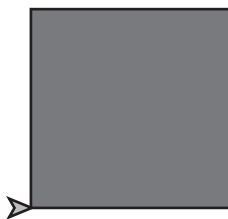
A Turtle object contains methods for setting a color. In the above section, we have learnt how to draw different shapes. Table 13.4 lists methods to draw different shapes with different colors.

**Table 13.4** More methods of Turtle related to color

| Method                                                                         | Meaning                                                                           |
|--------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>Turtle.color(c)</code>                                                   | Sets the pen's color                                                              |
| <code>Turtle.fillcolor(C)</code>                                               | Sets the pen's fill color to 'C'                                                  |
| <code>Turtle.begin_fill()</code>                                               | Calls this method before filling a shape                                          |
| <code>Turtle.end_fill()</code>                                                 | Fills the shape drawn before the last call to <b>begin_fill</b>                   |
| <code>Turtle.filling()</code>                                                  | Returns the fill state. True is filling, False if not filling.                    |
| <code>Turtle.clear()</code>                                                    | Clears the window. The state and position of window is not affected.              |
| <code>Turtle.reset()</code>                                                    | Clears the window and resets the state and position to its original default value |
| <code>Turtle.screensize()</code>                                               | Sets the width and height of the canvas                                           |
| <code>Turtle.showTurtle()</code>                                               | Makes the Turtle visible                                                          |
| <code>Turtle.hideTurtle()</code>                                               | Makes the Turtle invisible                                                        |
| <code>Turtle.write(msg, move, align, font=fontname, fontsize, fonttype)</code> | Writes a message on the Turtle graphics window                                    |

Program 13.4 demonstrates the use of **begin\_fill()** and **end\_fill()** method to fill a shape.

**PROGRAM 13.4** | Write a program to draw a color filled square box as shown.



```
import Turtle
Turtle.fillcolor ("gray") #Fill gray color inside the square
Turtle.begin_fill ()
Turtle.forward(100)
Turtle.left (90)
```

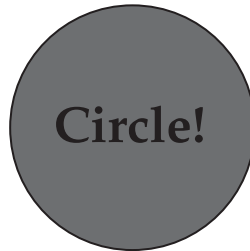
(Contd.)



```
Turtle.forward(100)
Turtle.left(90)
Turtle.forward(100)
Turtle.left(90)
Turtle.forward(100)
Turtle.left(90)
Turtle.end_fill()
```

**PROGRAM 13.5** | Write a program to create a circle with specifications as:

- (a) Fill circle with gray color
- (b) Display the text message "Circle!" inside the circle.



```
import Turtle
Turtle.pendown()
Turtle.fillcolor ("gray")
Turtle.begin_fill()
Turtle.circle(70)
Turtle.end_fill()
Turtle.penup()
Turtle.goto(-25,50)
Turtle.hideturtle ()
Turtle.write('Circle!', font = ('Times New Roman', 20, 'bold'))
```

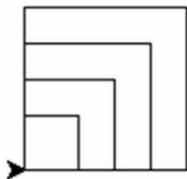
## 13.7 DRAWING BASIC SHAPES USING ITERATIONS

As shown in Program 13.4, a programmer needs to write the following six sentences to draw a simple square:

```
Turtle.forward(100)
Turtle.left(90)
Turtle.forward(100)
Turtle.left(90)
Turtle.forward(100)
Turtle.left(90)
```

However, if a programmer wants to display four different squares then it would be very cumbersome to type the above code repeatedly. Such kind of iterations can be accomplished using the **for** loop. Thus, to create four different squares, we need to create a function **square ()** and then draw the square using the for loop. Function takes one argument which is the side of a square. The following program demonstrates the use of the for loop to display multiple squares.

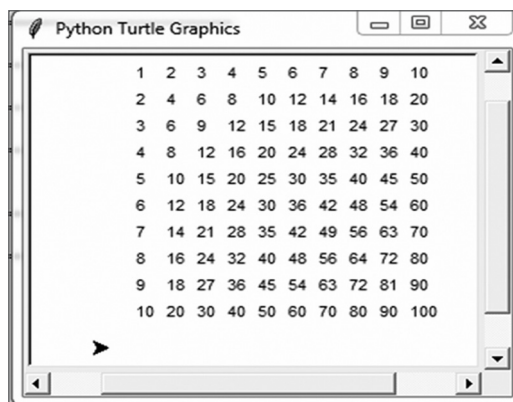
**PROGRAM 13.6** | Create a function to draw four different squares using the for loop as shown.



```
import Turtle
def square(side):
 for i in range(4):
 Turtle.forward(side)
 Turtle.left(90)

square(20)
square(30)
square(40)
square(50)
```

**PROGRAM 13.7** | Write a program to display the multiplication table from 1 to 10 in the Turtle graphics window as shown.



```
import Turtle as t
t.penup()
```

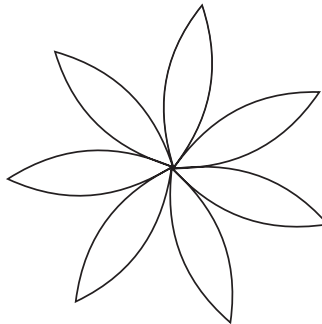
(Contd.)

```

x = -100
y = 100
t.goto(x,y) #Move pen at location x and y
t.penup()
for i in range(1,11,1): # value of i varies from 1 to 10
 y = y - 20
 for j in range(1,11,1): # Value of j varies from 1 to 10
 t.penup()
 t.speed(1)
 t.forward(20)
 t.write(i*j)
 t.goto(x, y)

```

**PROGRAM 13.8** | Write a program to draw the petals of the flower shown as follows using the circle method.



```

import Turtle as t
def petal(t, r, angle):
 """Use the Turtle (t) to draw a petal using two arcs
 with the radius (r) and angle.
 """
 for i in range(2):
 t.circle(r,angle)
 t.left(180-angle)

def flower(t, n, r, angle):
 """Use the Turtle (t) to draw a flower with (n) petals,
 each with the radius (r) and angle.
 """
 for i in range(n):
 petal(t, r, angle)
 t.left(360.0/n)

flower(t, 7, 80.0, 60.0)

```

## 13.8 CHANGING COLOR DYNAMICALLY USING LIST

As we have studied in the previous chapter, a list is a sequence of values called **items** or **elements**, where the elements can be of any type. Similarly, we can define the various colors inside a list using the syntax:

```
List_Name = ["First_Color_Name", "Second_Color_Name",, ..]
```

### Example

```
C = ["blue", "RED", "Pink"]
```

Program 13. 9 demonstrates the use of **list** and **for** loop to change a color dynamically.

**PROGRAM 13.9** | Write a program to draw and fill circles with different colors.

```
import Turtle as t
C = ["blue", "RED", "Pink"]
for i in range(3):
 t.fillcolor (C[i])
 t.begin_fill()
 t.circle(70)
 t.end_fill()
```

**Explanation** In the above program, all the names of colors are defined inside the list **C**. The **for** loop is used to iterate all the elements of the list. The statement **t.fillcolor (C[i])** is used to fill color inside the circle.

## 13.9 TURTLES TO CREATE BAR CHARTS

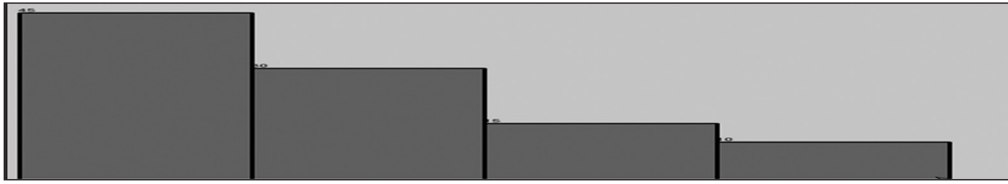
Turtles can be used to create bar charts. Bar charts can be created using various inbuilt methods discussed in the previous section of this chapter. A method such as **write ()** can be used to display the text on the canvas at a particular location. Other methods such as **begin \_ fill()** and **end \_ fill()** can be used to fill a shape with a specific color. Thus, by using various methods, we can draw bar charts in Python.

Table 13.5 shows statistics for the most downloaded browser by users in 2016–2017.

**Table 13.5** Sample data to draw a chart

| Web Browser       | Percentage |
|-------------------|------------|
| Mozilla Firefox   | 45%        |
| Google Chrome     | 30%        |
| Internet Explorer | 15%        |
| Others            | 10%        |

Corresponding to the percentage given in Table 13.5, we will draw a simple rectangle of the given height with fixed width. The bar chart for Table 13.5 will be:



**PROGRAM 13.10** | Write a program to draw a bar chart using Turtle for the sample data given in Table 13.5.

```
import Turtle
def Draw_Bar_Chart(t, height):
 t.begin_fill() # start filling this shape
 t.left(90)
 t.forward(height)
 t.write(str(height))
 t.right(90)
 t.forward(40)
 t.right(90)
 t.forward(height)
 t.left(90)
 t.end_fill() # stop filling this shape

Mozilla_Firefox = 45
Chrome = 30
IE = 15
Others = 10
S = [Mozilla_Firefox, Chrome, IE, Others] # Sample Data
maxheight = max(S)
num_ofBars = len(S)
border = 10
w = Turtle.Screen() # Setting up attributes of Window
w.setworldcoordinates(0,0,40*num_ofBars + border, maxheight + border)
w.bgcolor("pink")
T1 = Turtle.Turtle()
T1.color("#000000")
T1.fillcolor("#DB148E")
T1.pensize(3)
for a in S:
 Draw_Bar_Chart(T1,a)
```

**Explanation** In the above program, we have created a function named **Draw\_Bar\_Chart()**. Initially, the sample statistics data of a browser is given in list **S**. The function **setworldcoordinates()** is used to set the coordinates. The actual syntax and its details are:

**setworldcoordinates(LLX, LLY, URX, URY)**

where,

**LLX** - A number which indicates X coordinate of the lower left corner of the canvas.

**LLY** - A number which indicates Y coordinate of the lower left corner of the canvas.

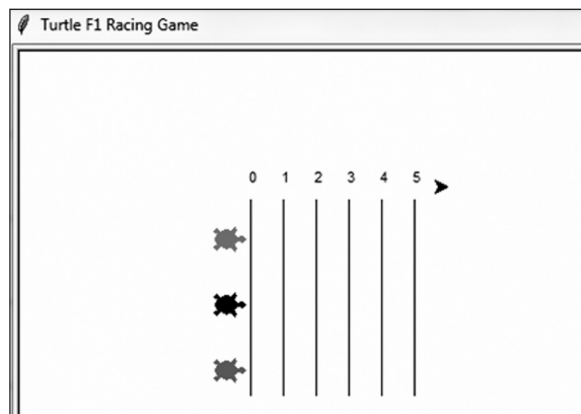
**URX** - A number which indicates X coordinate of the upper right corner of the canvas.

**URY** - A number which indicates X coordinate of the upper right corner of the canvas.

Thus, `setworldcoordinates()` sets the coordinates position to draw the chart.

## MINI PROJECT Turtle Racing Game

Create **three** different Turtles of colors **red**, **green** and **black**. Design one track for all of them to run over the track and win the competition. The track and the Turtle before the start of the completion should look as shown.



Turtle Racing Track

To solve this case study, the for loop and Turtle's inbuilt functions such as `penup()`, `pendown()`, `forward()`, `right()`, `goto()`, `color()`, `shape()`, `speed()`, and `left()` will be used.

### Algorithm

- ⊙ **STEP 1:** Design the track.
- ⊙ **STEP 2:** Place all the Turtles at the appropriate position to start the race.
- ⊙ **STEP 3:** Use the for loop to run over the track and a random number to move the Turtle forward by x pixels.
- ⊙ **STEP 4:** End.

### Part 1: Design the track.

- (a) First place the Turtles at the starting position `position(x,y)`.

`goto(-240,240)`

- (b) To draw the lines vertically, change the direction of the Turtle facing towards the right.

```
penup()
right(90)
```

- (c) Move the Turtle 10 pixels ahead.

```
forward(10)
```

- (d) Now move the Turtle forward by 150 pixels.

```
pendown()
forward(150)
```

Thus, up to this step we have successfully created the starting line of the track. To draw the second line, move backwards by 160 pixels.

```
backward(160)
```

Now, the Turtle has come to its starting position in the first line but is facing up. Therefore, to draw the second line, change the direction towards the left by 90 degrees and forward by some 'y' distance.

```
left(90)
forward(Y)
```

Repeat all the above steps to draw the remaining lines.

The code to create track is given as follows:

```
from Turtle import*
title('Turtle F1 Racing Game')
speed(10)
penup()
goto(-240,240) #Initial Position of track
z=0
y=25

for x in range(6): #Iterate to draw six lines
 write(x) #Mark distance at the top of line
 right(90) #change direction facing downwards
 forward(10) #Move 10 steps ahead
 pendown() #Open Pen to draw
 forward(150) #Move 150 Steps ahead
 penup() #Close pen
 backward(160) #Move 160 steps backward
 left(90) #Change direction towards left
 forward(y) #Move by y distance
```

**Part II:** Write the code to create the three Turtles and place them at the proper position before the start of the first line.

```
t1 = Turtle() #create Turtle object t1
t1.penup() #pen up to place Turtle at x y position
```

```

t1.goto(X,Y)
t1.color('color_name') #Change the color of Turtle
t1.shape('Turtle') #Give proper shape to it

```

Repeat the above steps three times to create three Turtles.

Code to create three Turtles and place them at the proper position before the start of the first line is given as follows:

```

t1 = Turtle() #First Turtle - Red Colored
t1.penup()
t1.goto(-260,200)
t1.color('red')
t1.shape('Turtle')

t2 = Turtle() #Second Turtle - Black Colored
t2.penup()
t2.goto(-260,150)
t2.color('Black')
t2.shape('Turtle')

t3 = Turtle() #Third Turtle - Green Colored
t3.penup()
t3.goto(-260,100)
t3.color('Green')
t3.shape('Turtle')

```

### Part III: Moving the Turtles randomly.

Use randint from the random module to move a Turtle by x position randomly.

```
Turtleobject.forward(randint(1,5))
```

Perform the above steps for all the three Turtles.

Code to move the Turtles is given as follows:

```

from random import*
for t in range(50):
 t1.forward(randint(1,5))
 t2.forward(randint(1,5))
 t3.forward(randint(1,5))

```



## Solution

Merging all codes, viz. part I, II and III, we get

```
from Turtle import*
from random import*
title('Turtle F1 Racing Game')
speed(10)
penup()
goto(-240,240)
z=0
y=25

for x in range(6):
 write(x)
 right(90)
 forward(10)
 pendown()
 forward(150)
 penup()
 backward(160)
 left(90)
 forward(y)

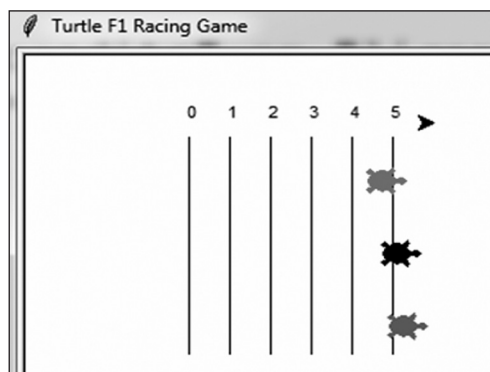
t1 = Turtle()
t1.penup()
t1.goto(-260,200)
t1.color('red')
t1.shape('Turtle')

t2 = Turtle()
t2.penup()
t2.goto(-260,150)
t2.color('Black')
t2.shape('Turtle')

t3 = Turtle()
t3.penup()
t3.goto(-260,100)
t3.color('Green')
t3.shape('Turtle')

for t in range(50):
 t1.forward(randint(1,5))
 t2.forward(randint(1,5))
 t3.forward(randint(1,5))
```

## Output



## SUMMARY

- ◆ Turtle is Python's inbuilt graphics module for drawing various shapes such as lines, circle etc.
- ◆ The Turtle is like a pen.
- ◆ Initially the Turtle is positioned at the center of the window.
- ◆ Various methods such as `forward()` and `backward()` are used to move the Turtle forward and backward by x pixels.
- ◆ The Turtle `left(angle)` and `right(angle)` is used to rotate the Turtle left or right by some angle.
- ◆ The Turtle `goto(x, y)` method is used to move the Turtle to specified points(x, y).

## KEY TERMS

- ⇒ **turtle()**: Graphics package to draw objects
- ⇒ **forward()**, **left()**, **right()** and **backward()**: Direction to move the Turtle in the given direction
- ⇒ **penup()** and **pendown()**: Draw depends on the status of the pen
- ⇒ **color()**, **fillcolor()**, **end\_fill()**, **begin\_fill()**: Methods to color Turtle objects
- ⇒ **setworldcoordinates()**: Coordinate the position to draw objects
- ⇒ **goto(x, y)**: Move the Turtle at location x,y.

## REVIEW QUESTIONS

## A. Multiple Choice Questions

1. Which instruction is used to set the pen size to 10 pixels?
  - a. `Turtle.size(10)`
  - b. `Turtle.pensize(10)`
  - c. `Turtle.setsize(10)`
  - d. All of them

2. Which instruction is used to set the position of the Turtle at 0,0?
  - a. Turtle.set(0,0)
  - b. Turtle.xy(0,0)
  - c. Turtle.goto(0,0)
  - d. Turtle.moveto(0,0)
3. Which instruction related to the pen is used to draw an object while it is moving?
  - a. pendown()
  - b. Pendown()
  - c. penDown()
  - d. PenDown()
4. Which instruction will help us draw a circle with radius 10?
  - a. Turtle.drawcircle(10)
  - b. Turtle.circledraw(10)
  - c. Turtle.c(10)
  - d. Turtle.circle(10)
5. Which inbuilt function is used change the speed of the Turtle?
  - a. Turtle.move(x)
  - b. Turtle.speed(x)
  - c. Both a and b
  - d. None of the above
6. Which instruction is used to show the current location and direction of the Turtle object?
  - a. Turtle.show()
  - b. Turtle.showdirection()
  - c. Turtle.shoedirloc()
  - d. Turtle.showTurtle()
7. Which instruction prevents the Turtle from drawing objects?
  - a. penUp()
  - b. penup()
  - c. PenUp()
  - d. PenuP()
8. Which instruction hides the Turtle?
  - a. Turtle.hide()
  - b. Turtle.noTurtle()
  - c. Turtle.invisble()
  - d. Turtle.hideall()

## B. True or False

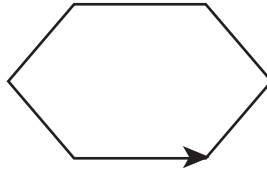
1. Interactive mode (command Line) cannot used for graphics programming in Python.
2. The turtle is an object created when module turtle is imported.
3. The imported turtle module uses pen to draw shapes.
4. The turtle is used to move and draw lines in only forward and backward direction on the screen.
5. By default the position using Turtle pen is down side.
6. It is not possible to draw complicated figures using turtle.
7. Slowest speed range for drawing a figure using a turtle is -1.
8. Maximum speed range for drawing a figure using a turtle is 0.
9. A turtle object contains methods for setting color.
10. We can't fill up the circular figure with a color.

## C. Exercise Questions

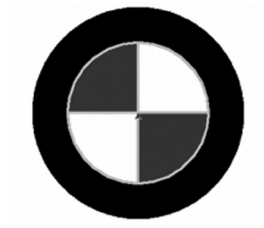
1. What is Turtle and how is it used to draw objects?
2. Explain the various inbuilt methods to change the direction of the Turtle.
3. Explain how different shapes can be drawn using iterations.
4. Explain the steps required to create bar charts.
5. How can **penup()** and **pendown()** functions be used effectively?

## PROGRAMMING ASSIGNMENTS

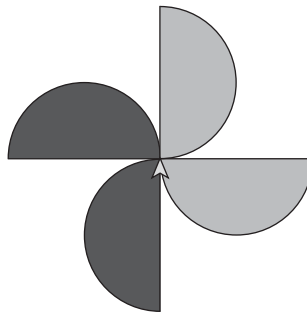
1. Write a program to display the hexagon given as follows:



2. Write a program to display the logo of BMW as given as follows:



3. Write a program to draw the figure given as follows:



4. Write a program to display the patterns of stars in the Turtle graphics window as shown.

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

# Unit Testing

## 14

### CHAPTER OUTLINE

|                               |                                                             |
|-------------------------------|-------------------------------------------------------------|
| 14.1 Why Testing Is Required? | 14.4 Unit Testing in Python—The <code>unittest</code> Class |
| 14.2 Types of Testing         | 14.5 List of <code>assert</code> Methods                    |
| 14.3 Identifying Units        | 14.6 Writing and Running Multiple Test Cases                |

### LEARNING OUTCOMES

*After completing this chapter, students will be able to:*

- Learn to write and run test cases
- Learn writing multiple test cases at once
- Learn to test the functional units in isolation
- Write the better test cases in order to verify that requirements are being met

## 14.1 WHY TESTING IS REQUIRED?

Writing programs is difficult at the beginning but it becomes easier after doing them regularly. Once you start writing programs, the size of program grows by few lines and, ultimately, there are more chances of appearing errors in the programs. In general, the best practice of testing is to debug programs and write automated test for your software. Also testing will help developers to create error-free software. Actually, the testing is the process of running software which looks for errors. It also tries to make program fail by testing it in many ways.

## 14.2 TYPES OF TESTING

Generally, there are two categories of testing, i.e. **black box** and **white box** testing. In black box testing, the testers write the test cases based on the specifications for what the code is supposed to accomplish and they are also not supposed to look at the code itself. But, in cases of white box testing, testers write the code based on the knowledge of the specifications and the code itself. Thus, in case of white box testing, testers can write the code before or while the code is written.

Following are the various types of automated tests (Table 14.1), where testers can use them to test the written software.

**Table 14.1** Types of Automated Testing

| <i>Types of Automated Testing</i> | <i>Description</i>                                                                                        |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------|
| Unit Testing                      | Here the small modules, classes or functions are being tested. Or the isolated units of codes are tested. |
| Acceptance Testing                | Test features from user's perspective.                                                                    |
| Integration Testing               | Test two or more larger components.                                                                       |
| Performance Testing               | Test execution speed, memory usage and other parameters.                                                  |
| Regression Testing                | Running test in order to check if previously built features are still working.                            |

Consequently, here we will discuss testing in general and will learn to perform unit testing in Python.

## 14.3 IDENTIFYING UNITS

Imagine, you are a group of people as working on one simple Python project. Among the teammates, suppose one of your colleagues has written following code and it is your responsibility to test the code.

As it is your turn to test the code, so in order to make the task of testing easy, you need to identify which section of the codes does contain the units. Is there only one unit which consists of the entire class or if each of the methods will act as a separate unit? The answer to question may vary upon the tester to tester. One may say that whole class is a single unit and, in some circumstances, it may be the best answer. But, it is always easy to subdivide most classes into methods as all of the methods have isolated behaviour.

## 14.4 UNIT TESTING IN PYTHON—THE `unittest` CLASS

To test the software, we must write both test cases and software. Typically, the programmer should understand the problem first and then write the test cases based on understanding of the problem and write the code. Of course, the programmer can also write the code first, but it would be better if the programmer can continuously check the code what he/she is writing against the number of test cases as already been written. Thus, first to write the test cases and then to write the code for the application will help the developers/programmers to know how much progress is being made towards passing all of the respective tests.

Following is the simple example which describes writing test cases for a Program. Here, we have used the standard Python library containing a module, named “unittest”. Where, the unittest is a unit-testing framework and it comes with Python’s standard library. The module contains the core framework classes that form the basis of test cases and other suites. It also provides the utility class for running the tests and reporting the results. The unittest module provides a class called “TestCase” from which we can create sub-classes to perform the unit testing. The following program demonstrates the steps to write test cases using TestCase class for unittest module.

**PROGRAM 14.1** | Create the file `arithmetic.py` and write the code within it.

```
def mul(X,Y):
 return X*Y
```

**PROGRAM 14.2** | Write the test case to check if `arithmetic.py` works for other numbers or not. Create a test file called “`arithmetic_test.py`”.

```
from arithmetic import mul
import unittest
class TestArithmetic(unittest.TestCase):
 ##Your test methods go here.
 def test_mul(self):
 self.assertEqual(10,mul(5,2))
unittest.main() #Outside the class which tells the framework to run
```

### Output

```
Ran 1 test in 0.005s OK
```

**Explanation** The line-by-line execution of the Program 14.2 shows that this example first imports our written program `arithmetic.py` and imports its functionality called `mul()`. The second line imports then Python’s `unittest` module. The `test_mul()` method is executed before each test in order to check if it returns the correct result. To check if the `mul()` method of `arithmetic.py` file is giving the adequate result, the `assertEqual()` method is used. Here, the `assertEqual()` method is provided by imported `unittest` module. This method checks if the first argument equals to second or not. In the Program 14.2 above, we are interested to check if the multiplication of 5 and 2 is 10 and the same should be returned by the function `mul()` of `arithmetic.py` by checking whether it calculates correctly and returns 10. Finally, after execution of the Program above, the output shows “Ran 1 test in 0.005s OK” while it indicates that your test has run successfully. What if, the test as run went wrong or was unsuccessful? Then what would be the output in such cases? Thus, we can simulate the same by breaking the test as shown below.

In order to fail the test case, we will change then the code of “`arithmetic_test.py`” as follows.

**PROGRAM 14.3** | Write a Program to fail the test case by changing the code "arithmetic\_test.py".

```
from arithmetic import mul
import unittest
class TestArithmetic(unittest.TestCase):
 ##Your test methods go here.
 def test_mul(self):
 self.assertEqual(12,mul(4,6))

unittest.main() #Outside the class, which tells the framework to run
```

### Output

```
F
=====
FAIL: test_mul (__main__.TestArithmetic)

Traceback(most recent call last):
 File "C:/Python27/Demo/arithmetic_test.py", line 6, in test_mul
 self.assertEqual(12,mul(4,6))
AssertionError: 12 != 24

Ran 1 test in 0.005s
```

In cases of the failure as the above Program shows, the line number where the error was detected, the assertion which was failed and the actual result too which was not just as been expected.

So from the test case above, we can analyse unittest modules which provide output in details even in those cases where the tests fail. In some cases, it may also happen that failure of test case can go wrong if something goes wrong in the code and then it gives you the wrong answer. So, in such cases, you need to go in and fix your code.



**Note:** Rules for writing test methods:

- The name of a test method must start with word "test", otherwise it will be ignored.
- Every test method must have exactly one parameter with "self".
- Put "self." in front of every builtin assertion method while you call.

Let us change the name of method "test\_mul" to "tmul" in the code "arithmetic\_test.py".

**PROGRAM 14.4** | Write a Program where the name of the method will be changed from "test\_mul" to "tmul" in the code "arithmetic\_test.py".

```
from arithmetic import mul
import unittest
class TestArithmetic(unittest.TestCase):
 ##Your test methods go here.
```

(Contd.)



```
def tmul(self):
 self.assertEqual(12,mul(4,6))
unittest.main() #Outside the class, which tells the framework to run
```

### Output

```
Ran 0 tests in 0.000s OK
```

Thus, we got the output “Ran 0 tests in 0.000s OK” which means that it has ignored the test case since the name of the method has not started with “test”.

## 14.5 LIST OF assert METHODS

Similar to `assertEqual()`, there are some in-built methods which you can call. Each method also has an optional “message” parameter to be printed if the corresponding test would fail. The brackets within the parentheses indicate the optional arguments that can be omitted. Following Table 14.2 contains the list of different `assert` methods.

**Table 14.2** Different methods of “assert”

|   |                                                                                                                                                                           |
|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | <b><code>self.assertEqual(expectedResult,actualResult,[message])</code></b>                                                                                               |
|   | The <code>assertEqual()</code> method is used to check if the values of expected result and actual result are exactly equal.                                              |
| 2 | <b><code>self.assertNotEqual(firstValue secondValue,[message])</code></b>                                                                                                 |
|   | The <code>assertNotEqual()</code> method is used to check if the values of expected result and actual result are different and fail if they are equal.                    |
| 3 | <b><code>self.assertAlmostEqual(expectedResult,actualResult,[places,[message]])</code></b>                                                                                |
|   | The <code>assertAlmostEqual()</code> method is used to check if the values of expected result and actual result are equal, after rounding to the definite decimal places. |
| 4 | <b><code>self.assertTrue(booleanCondition,[message])</code></b>                                                                                                           |
|   | The <code>assertTrue()</code> method is used to check if the <code>booleanCondition</code> is true.                                                                       |
| 5 | <b><code>self.assertFalse(booleanCondition,[message])</code></b>                                                                                                          |
|   | The <code>assertFalse()</code> method is used to check if the <code>booleanCondition</code> is false.                                                                     |
| 6 | <b><code>self.assertRaises(exception,functionName,parameter,...,parameter)</code></b>                                                                                     |
|   | Test that the function, <code>functionName</code> , when being called with the given (zero or more) parameters, it raises the given exception,                            |

## 14.6 WRITING AND RUNNING MULTIPLE TEST CASES

In the above section, we have seen to the processes to write the single test cases for a particular program. But what if the tester would want to test multiple test cases at once? Thus, the following Programs 14.4 and 14.5 demonstrate the single and multiple test cases in order to calculate the factorials of a number.

**PROGRAM 14.5** | Write a simple Program `factorial.py` to calculate the factorial of number.

```
def fact(n):
 if n == 0:
 return 1
 elif n == 1:
 return 1
 else:
 return n*fact(n-1)

def main(n):
 print(fact(n))

if __name__ == '__main__':
 main(5)
```

**PROGRAM 14.6** | Write the test case to check if above Program 14.5, i.e. `factorial.py`, works for other numbers. Create a test file called `Factorial_Test.py`.

```
from factorial import fact
import unittest
class TestFactorial(unittest.TestCase):

 def test_fact(self):
 res = fact(5)
 self.assertEqual(res,120)

if __name__ == '__main__':
 unittest.main()
```

### Output

```
Ran 1 test in 0.063s OK
```

**Explanation** The line-by-line execution of the Program 14.6 shows that this example first imports our written program `factorial.py` and imports its functionality called `fact()`. The second line imports Python's `unittest` module. The `test_fact()` method is executed before each test to check if it returns the correct result. To check if `fact()` method of `factorial.py` file gives the

correct result, the `assertEqual()` method is used. Here `assertEqual()` method is provided by imported `unittest` module. This method checks if the first argument equals to second or not. In the Program 14.6 above, we are interested to check if factorial of 5 is 120 and the same should be returned by the function `fact()` of `factorial.py` checking whether it calculates correctly and returns 120. Finally, after execution of the Program 14.6 above, the output is “Ran 1 test in 0.063 OK” and it indicates that our test has run successfully. What if as the test run went wrong or was unsuccessful? Then what would be the output in such cases? Thus, we can simulate the same by breaking the test as shown below.

In order to fail the test case, we will change then the code of “`factorial_test.py`” as follows.

**PROGRAM 14.7** | Write a Program to fail the test case by changing the code “`factorial_test.py`”.

```
from factorial import fact
import unittest
class TestFactorial(unittest.TestCase):

 def test_fact(self):
 res = fact(5)
 self.assertEqual(res,121)

if __name__ == '__main__':
 unittest.main()
```

### Output

```
F
=====
FAIL: test_fact (__main__.TestFactorial)

Traceback (most recent call last):
 File "C:\Python27\try.py", line 7, in test_fact
 self.assertEqual(res,121)
AssertionError: 120 != 121

Ran 1 test in 0.004s
FAILED (failures=1)
```

So from the test case above, we can analyse `unittest` module which provides the output in details even in cases where test would fail. In some cases, it may also happen that failure of test cases can go wrong if something goes wrong in the code and thus giving you the wrong answer. So in such cases, you need to go in and fix your code at first.

In the section above, we have just written only one test case, i.e.

```
def test_fact(self):
 res = fact(5)
 self.assertEqual(res, 120)
```

Thus, the above code has been able to check only one test case, i.e. in other words, it can check only if the factorial of 5 is 120 or not. But, if we want to check the factorial of 1, 2, 3, 4, 5, ..... etc. at once, what we should do then? In such scenarios, we use the example list of tuple to check if the expected output and output through the program is correct. Following Program 14.8 demonstrates the uses of writing multiple test cases at once.

**PROGRAM 14.8** | Write a Program to demonstrate the uses of writing multiple test cases at once.

```
from factorial import fact
import unittest
class TestFactorial(unittest.TestCase):

 def test_fact(self):
 L = [(1,1), (2,2), (3,6), (4,24), (5,120)]
 for n, ans in L:
 res = fact(n)
 self.assertEqual(res, ans)
if __name__ == '__main__':
 unittest.main()
```

### Output

```
Ran 1 test in 0.047s OK
```

**Explanation** In Program 14.8 above, we have imported unittest module and fact() function from our factorial.py file which was already written.

The statement is:

```
L = [(1,1), (2,2), (3,6), (4,24), (5,120)]
```

It contains list of tuples, i.e. (x, y) is used to check if for each number for x, it gives the correct output, i.e. y. Finally, the value of each tuple from the list is passed as parameter to fact() function. Then, the assert function will let you know if the test case for each tuple is successful or unsuccessful. Thus, the programs containing all of the test cases are executed successfully, and, therefore, it returns output as “Ran 1 test in 0.047s OK”.

In some cases, it may happen that while testing multiple cases, one of the test cases may go wrong. Even in such cases, the `unittest` module gives the details about the failure. The following Program 14.9 demonstrates how a unit test case fails if any of the corresponding conditions goes wrong.

```
from factorial import fact
import unittest
class TestFactorial(unittest.TestCase):

 def test_fact(self):
 L = [(1,1), (2,2), (3,6), (4,25), (5,120)]
 for n, ans in L:
 res = fact(n)
 self.assertEqual(res,ans)

if __name__ == '__main__':
 unittest.main()
```

### Output

```
FAIL: test_fact(_main_.TestFactorial)

Traceback (most recent call last):
 File "C:\Users\kamthanea\Documents\Unit Testing in Python\Test.py", line 9,
in test_fact
 self.assertEqual(res,ans)
AssertionError: 24! = 25

Ran 1 test in 0.031s

FAILED (failures=1)
```

## SUMMARY

- ◆ Testing is the process of running software which is looking for errors.
- ◆ Testing will help the developers to create error-free software.
- ◆ **Black Box** and **White Box** testing are the categories of testing.
- ◆ The `unittest` module provides a class called "**TestCase**" from which we can create sub-classes to perform unit testing.

---

**KEY TERMS**

- ⇒ **Black Box Testing:** In this type, the tester writes the test cases based on the specifications only but he or she is not supposed to look at the code while writing them.
- ⇒ **White Box Testing:** In this type, the tester writes the code based on the knowledge of the specification and looking at the code itself.
- ⇒ **Unit Testing:** The small modules, classes or functions are tested in unit testing method.
- ⇒ **The unittest module:** This is a utility class for running the tests and reporting the result.

---

**REVIEW QUESTIONS****A. Exercise Questions**

1. Explain White Box and Black Box testing.
2. Explain different types of automated testing.
3. Can we write multiple test cases? How?
4. Describe different methods of “assert” which can be used in unit testing.

**PROGRAMMING ASSIGNMENTS**

1. Write a Program and test cases to calculate the area of a circle.
2. Write a Program and multiple test cases in the same time in order to calculate the area of a circle.
3. Write a Program and test cases to calculate and verify the  $n$ th term of Fibonacci series.

# Previous Year's Solved JNTU Examination Questions

## PART A

1. a. Explain input function.

(2M)

### Solution

The input() function is used to accept an input from a user. A programmer can ask a user to input a value by making use of input(). The input() function is used to assign a value to a variable. Syntax of input() function is as follows.

### Syntax

```
Variable_Name = input()
```

OR

```
Variable_Name = input('String')
```

Simple program of input() function to read string from the keyword is as follows.

### PROGRAM A.1.1

```
Name = input('Please Enter Your Name:')
print('Hello',Name,' How are you??')
```

### Output

```
Please Enter Your Name: Steve Jobs
Hello Steve Jobs How are you ??
```

b. Give an example of `lstrip()` method.

(2M)

### Solution

The `lstrip()` function is used to remove the leading white space characters. The following is the simple example related to `lstrip`.

```
>>> str1 = 'Hello How are You'
>>> str1
'Hello How are You' #Before Stripping
>>> str1.lstrip()
'Hello How are You' #After applying lstrip
```

c. How can we access the values in a dictionary?

(2M)

### Solution

The subscript[] operator is used to obtain the value associated with the key. The syntax to retrieve the value from dictionary is as follows.

### Syntax

```
Dictionary_Name[Key] = Value
```

### Example

```
>>> Founder = {'Microsoft':'Bill Gates','MAC OS':'Steve Jobs'}
>>> Founder['Microsoft']
```

d. What is Default Argument?

(2M)

### Solution

The parameters within a function's definition can have the default values. We can provide the default values to a parameter by using the assignment(=) operator. The Program A.1.2 below demonstrates the use of default argument to calculate the area of circle.

**PROGRAM A.1.2** | Show the uses of default argument to calculate the area of circle

```
def Calc_Area_Circle(radius,pi=3.14):
 print('Radius is',radius)
 print('Area of Circle is ', pi*radius*radius)
Calc_Area_Circle(5)
```

### Output

```
Radius is 5
Area of Circle is 78.5
```



e. What are the basic overloading methods?

(3M)

### Solution

The basic overloading methods are as follows.

**Table A.1.1** Basic Overloading Methods

|                    |            |                                |
|--------------------|------------|--------------------------------|
| Addition           | $E1 + E2$  | <code>E1._add_(E2)</code>      |
| Subtraction        | $E1 - E2$  | <code>E1._sub_(E2)</code>      |
| Multiplication     | $E1 * E2$  | <code>E1._mul_(E2)</code>      |
| Power              | $E1 ** E2$ | <code>E1._pow_(E2)</code>      |
| Division           | $E1 / E2$  | <code>E1._truediv_(E2)</code>  |
| Floor Division     | $E1 // E2$ | <code>E1._floordiv_(E2)</code> |
| Remainder (modulo) | $E1 \% E2$ | <code>E1._mod_(E2)</code>      |

f. Explain importing turtle graphics.

(3M)

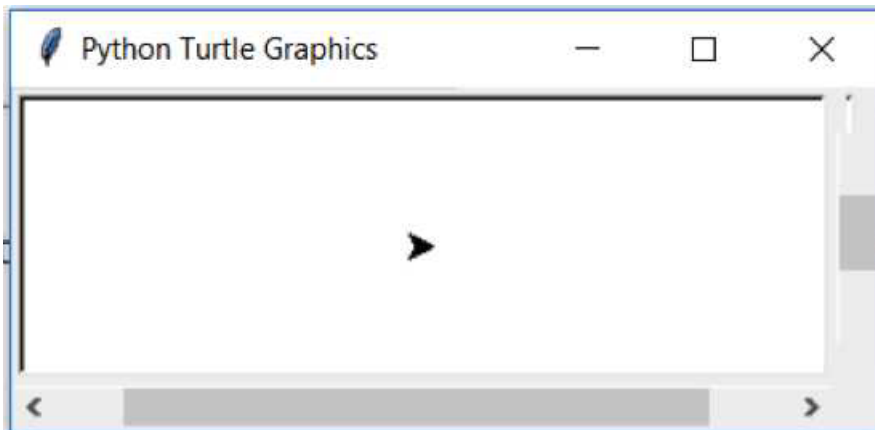
### Solution

The statement "**import turtle**" is used to import the turtle graphics or the programmer can also write the statement as "**import turtle as t**".

⊙ **STEP 1:** `import turtle as t.`

⊙ **STEP 2:** `t.showturtle()`

After importing turtle the method **showturtle()** is used to show imported turtle.



**Figure A.1.1** Python Turtle Graphics

# PART B

## 2. a. What are the IDLE usability features?

(7M)

### Solution

- Programmer can make use of '**ALT+P**' combination used to repeat prior commands in IDLE's main interactive window, i.e. to scroll backward commands with respect to the command history.
- Programmer can recall commands by positioning the cursor on them by pressing the Enter to insert text at the input prompt.
- The auto-indent is one of the advanced functionality of IDLE where on pressing of backspace, the indentation goes one level back.
- The programmer has to invoke **ctrl + space** while typing in word auto-completion feature.
- The other advanced options can help to customize IDLE.
- To change the text fonts and colours in IDLE, the programmer can select configuration option in option menu of IDLE menu.

## b. Explain about keywords used in Python.

(7M)

### Solution

The keywords used in python are as follows.

Table A.1.2

|          |         |        |          |        |
|----------|---------|--------|----------|--------|
| and      | def     | For    | is       | Raise  |
| As       | del     | from   | lambda   | return |
| assert   | elif    | global | nonlocal | Try    |
| break    | else    | If     | not      | While  |
| class    | except  | import | or       | With   |
| continue | finally | In     | pass     | Yield  |

Some of the uses of keywords are as follows.

**class:** The keyword **class** is just another name for a **type** in Python. It means that the programmers can create objects of their own class.

**global:** The keyword **global** is used to change the value of global variables.

**try:** The keyword **try** is used to monitor the exceptions for written block of the code.

**except:** The **except** keyword is used to catch already thrown exception.

**raise:** The keyword **raise** is used to raise the exceptions.

**Finally:** The **finally** block is placed after the last **except** block. If there is no **except** block, the **finally** block should immediately follow the **try** block.

**3. a. What are the four built-in numeric data types in Python? Explain.****(7M)****Solution**

The four built-in numeric data types in python are as follows.

**int (The Integer Type):** The integer literals in a program are written without commas and the leading minus sign indicates a negative value. The example of integer data type is as follows.

```
>>> a = 10
>>> type(a)
<type 'int'>
```

**float (The float Type):** The float type is used to store the decimal numbers. Examples of floating type is as follows.

```
>>> float_var = 1.23
>>> type(float_var)
<type 'float'>
```

**complex (The Complex Type):** The complex type is used to store the complex numbers. In this case, the complex number is expressed in terms of  $a + bj$  and  $a$  and  $b$  are real numbers and  $j$  is the imaginary unit. Example of complex type is as follows.

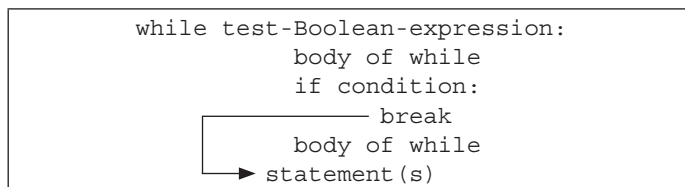
```
>>> Com_Var = 10+5j
>>> type(Com_Var)
<type 'complex'>
```

**b. Describe Python jump statements with examples.****(7M)****Solution**

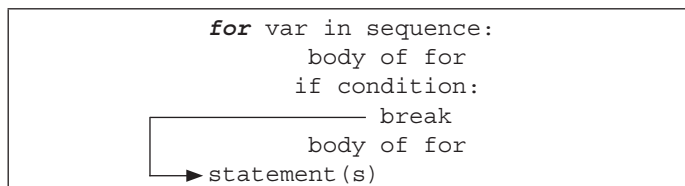
The **break**, **continue** and **pass** are three different types of **jump** statements in python. All of these statements are used to alter the flow of a loop, like where you want to skip a part of a loop or terminate a loop. The more details about each of them is as follows.

The **break** statement:

The keyword **break** allows the programmer to terminate the loop. When the break statement is encountered inside a loop, the loop is immediately terminated and program control automatically goes to the first statement following the loop. The working of break statement in **while** block is as follows.

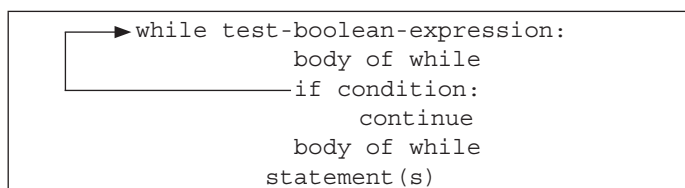


Similarly, the working of **break** in **for** loop is as follows.



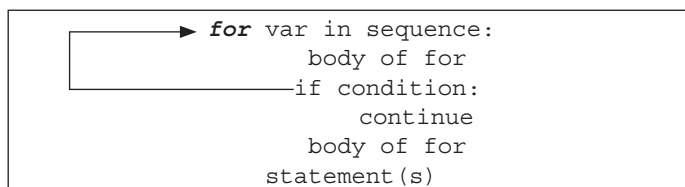
The **continue** statement:

This **continue** statement is exactly opposite of the **break** statement. When **continue** is encountered within the loop, the remaining statements within the body are skipped but the loop condition is checked there to see if the loop should continue or exited. The working of **continue** in **while** loop is shown as follows.



The working of **continue** in **while** loop:

Alternately, the working of **continue** in **for** loop:



The **pass** statement:

The **pass** Statement in Python does nothing. You use **pass** statement when you create a method that you do not want to implement.

### Example

```

>>> def Demo():
 Pass
>>> Demo()

```

#### 4. a. Explain in detail about dictionaries in Python.

(7M)

#### Solution

The dictionary is a collection that stores the values along with the keys. The sequences of key and value pairs are separated by the commas. These pairs are sometimes called **entries** or **item**. All entries are enclosed by curly braces, i.e. { and }. A colon separates a key and its value. The simple example of the dictionary is as follows.

**Phone Book Example**

```
{ "Amit": "918624986968", "Amol": "919766962920" }
```

The following example gives details about creating dictionaries in four different ways.

**Example****#Way 1:**

```
>>> D1 = { 'Name': 'Sachin', 'Age': 40 }
>>> D1
{ 'Name': 'Sachin', 'Age': 40 }
```

**#Way 2:**

```
>>> D2 = {}
>>> D2['Name'] = 'Sachin'
>>> D2['Age'] = 40
>>> D2
{ 'Name': 'Sachin', 'Age': 40 }
```

**#Way 3:**

```
>>> D3 = dict(Name='Sachin', Age=40)
>>> D3
{ 'Name': 'Sachin', 'Age': 40 }
```

**#Way 4:**

```
>>> dict([('name', 'Sachin'), ('age', 40)])
{ 'age': 40, 'name': 'Sachin' }
```

The in-built functions **keys()** is used to retrieve all of the sequences of keys within the dictionary and the **values()** function returns the sequence of values. Also the following example illustrates the use of **values** and **keys**.

```
>>> IPL_Team = { "MUMBAI INDIANS": "Rohit Sharma", "Rajasthan
Royals": "Ajinkya Rahane" }
>>> type(IPL_Team)
<type 'dict'>
>>> IPL_Team.keys()
['Rajasthan Royals', 'MUMBAI INDIANS']
>>> IPL_Team.values()
['Ajinkya Rahane', 'Rohit Sharma']
```

**b. Discuss about the tuples in Python.****(7M)****Solution**

Tuple contains the sequence of items of any types. But the elements of tuples are fixed. Once the tuple is created, you cannot add, delete, reshuffle or reorder the elements in tuple. Therefore, the tuples are **immutable**, i.e. once created, it cannot be changed.

Following example gives the details about creating tuple on python interactive mode.

### Example

```
T1 = () #Creates an Empty Tuple
T2 = (12,34,56,90) #Creates Tuple with four elements
T3 = ('a','b','c','d','e') #Creates Tuple of five characters
T4 = 'a','b','c','d','e' #Creates tuple without parentheses
```

Following are the in-built functions that can be used on tuples.

**Table A.1.3**

| Built-in Functions | Meaning                                        |
|--------------------|------------------------------------------------|
| len()              | Returns the number of elements in the tuple.   |
| max()              | Returns the element with the greatest value.   |
| min()              | Returns the element with the minimum value.    |
| sum()              | Returns the sum of all the elements of tuple.  |
| index(x)           | Returns the index of element x.                |
| count(x)           | Returns the number of occurrence of element x. |

### Example

```
>>> t1= ("APPLE")
>>> len(t1) #Return the length of tuple t1
5
>>> max(t1) #Return the element from tuple with maximum Value
'P'
>>> min(t1) #Return the element from tuple with minimum value
'A'
>>> t1.index('A')
0
>>> t1.count('P')
2
```

Following is the simple program to traverse the tuple from the list.

### PROGRAM A.1.3

```
T=[(1,"Amit"),(2,"Divya"),(3,"Sameer")]
for no, name in t:
 print(no, name)
```

#### Output

```
1 Amit
2 Divya
3 Sameer
```

**5. a. Describe the anonymous functions examples.****(7M)****Solution**

In Python, the anonymous function is a function that is defined without a name. The normal functions are defined using the `def` keyword. In Python, the anonymous functions are defined using the `lambda` keyword. The `lambda` operator or `lambda` function is used to create small anonymous functions. Hence, the anonymous functions are also called `lambda` functions. The general syntax of `lambda` function is as follows.

**`lambda argument-list: expression`**

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. The following example of a `lambda` function returns the multiplication of its two arguments.

```
>>> f = lambda x, y : x * y
>>> mul = f(2,8)
>>> mul
16
```

**b. Why do we use modules? How can we structure a program?****(7M)****Solution**

A module is a piece or part of something. In python, we can say the modules are smaller pieces of bigger program. Each module is a separate file on hard disk. The programmer can take a big program and split it up into more than one module. The reason behind using the modules is as follows.

- a. It makes the file smaller which helps the programmers to find the things easily into the written code.
- b. Once module is created, it can be used in lot of programs. This feature saves the programmers from starting all over again by the next time if he or see needs the same function.

Following example illustrates how to structure a program while using the module.

Let us create a simple file as **Demo.py**.

**#Demo.py**

```
def Display():
 print('Hello, Welcome all!')
```

If we try to execute the above code, nothing will happen, because we have just written the function and it has not been called from elsewhere to perform its action. So, let us create another file named `main.py`, so that we can import the module `Demo.py` what we have just created and then call the function `Display()` as present in file, `Demo.py`, from a new file, `main.py`. Therefore, we create another file named `main.py`. In the file `main.py`, we will make use of `import` statement to import the module, i.e. `Demo.py`. The contents of the `main.py` file are as follows.

**PROGRAM A.1.4**

```
#main.py
import Demo #Importing Module named Demo
demo.Display() #Call function Display present within Demo.py

Output

'Hello, Welcome all!'
```

In Program A.1.4 above, we are importing a module, therefore, we need to call the function by referencing the module by `""`, i.e. by the dot notation. Thus, we use the **ModuleName.FunctionName()** to reference the function as present within the module. The statement **demo.Display()** calls the function **Display()** from module named `demo.py`. The above code contains the following two lines as shown below.

```
import Demo
demo.Display()
```

We can use the **from** keyword and replace the above two lines as

```
from Demo import Display
```

**6. a. Explain the creating classes in Python with examples. (7M)**

**Solution**

Where **class** is just another name for a **type** in python. The class may contain data in the form of **fields**. The fields are also called as **attributes** and the codes in the form of **procedure** are known as **methods**.

Syntax for defining class in Python is as follows.

```
Class Class_Name:
 Initializer
 attributes
 methods()
 Statement(s)
```

Following example illustrates the simple class program.

**PROGRAM A.1.5**

```
class Demo:
 pass
D1=Demo() #Instance or Object of the class Demo
print(D1)

Output

<_main_.Demo object at 0x029B3150>
```



In Program A.1.5 above, we have created the object "D1" where the D1 is instance of the class. Thus, creation of object is called as "**instantiation**". The programmer can also add methods to the class. The following syntax is used to add methods within the class.

```
class Class_Name:
 instance variable; #instance variable with initialization
 def method_name(self,parameter_list):#Parameter List is Optional
 block_of_statements
```

As mentioned above, in order to add methods to the existing class, the first parameter for each method should be **self**. This is only one specific difference between class methods and the ordinary functions. The **self**-parameter is used in the implementation of the method, but it is not used when the method is called. Therefore, the **self**-parameter references the object itself.

The Program A.1.6 below illustrates the use of adding method to the class.

### PROGRAM A.1.6

```
class MethodDemo:
 def Display_Message(self):
 print('Welcome to Python Programming')
ob1 = MethodDemo() #Instance of a class
ob1.Display_Message() #Calling Method
```

#### Output

```
Welcome to Python Programming
```

In the Program A.1.6 above, the **Display\_Message()** method takes no parameters but still the method has the **self** in the function definition. Therefore, the **self**-parameter refers to the current object itself. Finally, the method is called and message is displayed.

**b. Define error and exception. Distinguish between these two features.**

**(7M)**

#### Solution

It is common for the programmers to make mistakes while typing as well as developing a program. Such types of mistakes are called "**Errors**". An error is something that can produce incorrect or irrelevant output or even tend the system to crash.

The **exception** is an error which occurs during run-time. Also, one can define the exception as an erroneous condition that arises while a program is running.

**7. a. Why the testing is required? Explain laborately.**

**(7M)**

#### Solution

Writing programs are difficult once you start the program. The size of program grows by few lines and ultimately there are more chances of appearing errors in the programs. In this case, the best practice of testing is to debug programs and write automated test for your own software.

Also, testing will help the developer to create error-free software. Actually, the testing is the process of running the software which looks for errors. To test the software, we must write both test cases and software. Typically, a programmer should understand problem first and then write the test cases based on understanding of the problem and the code as well. Of course, the programmer can also write the code first, but it is better if the programmer can continuously check the code what he/she has already written against the number of test cases. Thus, the writing test cases first and then writing the code for the applications will help the developers/programmers to know how much progress is being made towards the passing of all the tests. Following are the various types of automated tests where the tester can use them to test the written software.

**Table A.1.4**

| Type of Automated Test | Description                                                                                     |
|------------------------|-------------------------------------------------------------------------------------------------|
| Unit Testing           | The small modules, classes or functions are tested. Or, the isolated units of codes are tested. |
| Acceptance Testing     | Test features from the user's perspective.                                                      |
| Integration Testing    | Testing of two or more larger components.                                                       |
| Performance Testing    | Testing of the execution speed, memory usage and other parameters.                              |
| Regression Testing     | Conducting tests in order to check whether previously built features are still working or not.  |

Also in case of testing, identifying units plays a major role. Entire class can be one unit or each method will act as a separate unit. Selection of unit varies from tester to tester. One may say that whole class is a single unit and, in some circumstances, it may be the best answer. But, it is always easy to subdivide most classes into methods as all of these methods show the isolated behaviours.

**b. Explain the following:**

**(7M)**

**i. Calendar module and ii. Synchronizing threads**

### **Solution**

#### **i. Calendar Module**

The calendar module defines the calendar class. It is used to encapsulate the calculations for values, such as, the dates of the week in a given month or a year. The calendar module has the predefined **TextCalendar** and **HTMLCalendar** classes which produces the pre-formatted outputs.

The following example illustrate the use of calendar module.

#### **Example**

```
>>> import calendar
>>> Cal = calendar.TextCalendar(calendar.SUNDAY)
>>> Str1 = Cal.formatmonth(2018,1)
>>> print(str1)
```

```

..
 January 2018
Mo Tu We Th Fr Sa Su
 1 2 3 4 5 6 7
 8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

```

In the example above, `Cal.formatmonth(2018,1)` is used to print the calendar for the first month of 2018.

## ii. Synchronizing threads

Locks are the most fundamental synchronization mechanism provided by the **threading** module. A lock has only two states, i.e. **locked** and **unlocked** where the locks are typically used to synchronize access to a shared resource. The **acquire()** and **release()** are two states being used at the time of the synchronizing threads. The function **acquire()** is used when you need to access the resource. It will block the execution until the **release()** method in some other co-routine sets makes it unlocked. The **release()** method should only be called in the locked state while it sets the state to unlocked and returns immediately. The skeleton to set and acquire locks is as follows.

```

lock = Lock()
lock.acquire()
try:
.....access shared resource
finally:
lock.release() #Release Lock

```

# PART A

1. a. Explain output function.

(2M)

## Solution

A function is a group of statements that are put together to perform a specific task. The Python `print()` function will act as output function. The task of **print** function is to display the contents on to the screen. The syntax of **print** function is as follows.

Syntax of **print()** function:

```
print(argument)
```

## Example Display messages using `print()`

```
>>> print('Hello Welcome to Python Programming')
Hello Welcome to Python Programming
```

b. Give an example of `istitle()` method.

(2M)

## Solution

The method **istitle()** returns true if the string is a string in title case and there is at least one character (e.g. uppercase characters may only follow uncased characters and the lowercase characters follow the cased ones). Otherwise, it returns 'False'.

Example of **istitle()** method is as follows:

```
>> str1 = "Python is Best Programming Language!!!"
>>> str1.istitle()
False
>>> str1 = "Python Is Best Programming Language!!!"
>>> str1.istitle()
True
```

c. Describe `type()` method with example.

(2M)

### Solution

To know the exact type of any value, Python provides in-built method named "**type**". The syntax to know the type of any value is as follows.

**type(value)**

### Example

```
>>> type('Hello World')
<class 'str'>
```

```
>>> type(123)
<class 'int'>
```

Thus, when the above examples are executed on Python interactive mode, the return type of value as passed to the in-built function is **type()**.

d. What is the general form of `lambda`?

(2M)

### Solution

The `lambda` operator or `lambda` function is used to create small anonymous functions. Hence, the anonymous functions are also called `lambda` functions. The general syntax of `lambda` function is as follows.

**lambda argument-list: expression**

The argument list consists of a comma separating the list of arguments and the expression is an arithmetic expression using these arguments. The following example of a `lambda` function returns the multiplication of its two arguments as shown below.

```
>>> f = lambda x, y : x * y
>>> mul = f(2,8)
>>> mul
16
```

e. Explain about `self`-variable with example.

(3M)

### Solution

To add methods to the existing class, the first parameter for each method should be **self**. This is only one specific difference between `class` methods and ordinary functions. The **self**-parameter is used in the implementation of the method, but it is not used when the method is called. Therefore, the **self**-parameter references the object itself. The following program gives the clear idea about **self**-parameter and adding methods to the existing class.

Following program demonstrates the use of self-parameter to calculate the area of circle.

#### PROGRAM A.2.1

```
import math
class Circle:
 def Calc_Area(self,radius):
 print('radius = ',radius)
 return math.pi*radius**2
ob1 = Circle()
print('Area of circle is ',ob1.Calc_Area(5))
```

##### Output

```
radius = 5
Area of circle is 78.53981633974483
```

**Explanation** The class named as **Circle** is created. The extra parameter **radius** is passed to method as defined inside the class **Calc\_Area()**. The instance **ob1** of a class is created and then is used to call the method of existing class. Even though the method **Calc\_Area()** contains two parameters, i.e. **self** and **radius**. But, in the calling method, only one parameter should be passed, i.e. the radius of circle.

#### f. Describe `time.time()` method.

(3M)

##### Solution

The method `time()` returns the time as a floating point number as expressed in seconds. The following example illustrates the use of `time()` method.

##### Example

```
>>> import time
>>> print(time.time())
1527586504.56
>>> print (time.asctime(time.localtime(time.time())))
Tue May 29 15:05:48 2018
```

# PART B

2. a. What are the assignment operators in Python? Explain.

(7M)

## Solution

The equal '=' sign is used as the assignment operator. The statement for assigning a value to a variable is called as an **assignment statement**. The syntax to assign value to a variable or identifier is as follows.

## Syntax

```
Variable = expression
```

Let us consider the following example of assigning and displaying value of a variable on Python interactive mode.

## Example

```
>>> Z = 1 #Assign value 1 to variable Z
>>> Z #Display value of Z
1
>>> radius = 5 #Assign value 5 to the variable radius
>>> radius #Display value of variable radius
5
>>> R = radius + Z #Assign the addition of radius and Z to R
>>> R #Display value of Variable R
6
```

## More on Assignment operator

The operators +, \*, //, /, % and \*\* are used with assignment operator '=' to form the compound or augmented assignment operators.

## Example

Consider the following example, where the value of a variable **x** is increased by 1.

```
X = X + 1
```

But, Python allows the programmers to combine assignment and addition operators. Thus, the above statement `X = X + 1` can also be written as:

```
X += 1
```

The **+=** operator is called the addition operator. The list of all other compound assignment operators is shown as follows.

**Table A.2.1** The compound assignment operators

| Operators | Examples | Equivalents | Explanations                                                                   |
|-----------|----------|-------------|--------------------------------------------------------------------------------|
| +=        | Z+=X     | Z=Z+X       | Adds the value of Z to X.                                                      |
| -=        | Z-=X     | Z=Z-X       | Subtracts X from Z.                                                            |
| *=        | Z*=X     | Z=Z*X       | Multiplies the value of x, y and stores the result in Z.                       |
| /=        | Z/=X     | Z=Z/X       | Performs the floating point division operation and stores the result in Z.     |
| //=       | Z//=X    | Z=Z//X      | Performs normal integer floor division and stores the result in Z.             |
| **=       | Z**=X    | Z=Z**X      | The value of variable X is raised to Z and the result is stored in variable Z. |
| %=        | Z%=X     | Z=Z%X       | The <b>Z modulo X</b> operation is performed here.                             |

**b. Explain about iteration statements with examples.**

**(7M)**

### Solution

Python provides two types of loop statements as **while** loops and **for** loops. The **while** loop is a **condition-controlled** loop. It is controlled by true or false conditions. But, the **for** loop is a **count-controlled** loop which repeats for specified number of times.

### The while loop

The **while** loop is one of the loop-control statements in Python. It executes the sequence of statements repeatedly as long as the conditions remain true. The syntax of **while** loop is given in Figure A.2.1 below.

```
while test-condition:
 #Loop Body
 statement(s)
```

**Figure A.2.1** Syntax of **while** loop

The reserved keyword **while** begins with the **while** statement. The test-condition is a Boolean expression. The colon ':' must follow the test-condition, i.e. the **while** statement is terminated with colon (:). The statement(s) within while loop will be executed till the condition is true, i.e. the condition is evaluated and if the condition is true, then the body of the loop will be executed. When the condition becomes false, the execution will be out of the loop.

**PROGRAM A.2.2** | Write a program to print the numbers from one to five using **while** loop.

```
count=0 #initialize the counter
while count<=5: #Test condition
 print("Count = ",count) #print the value of count
 count=count+1 #Increment the value of count by 1
```

(Contd.)



**Output**

```
Count = 0
Count = 1
Count = 2
Count = 3
Count = 4
Count = 5
```

**The for loop**

The for loop in Python iterates through a sequence of objects, i.e. it iterates through each value in a sequence. The sequence of object holds multiple items of data as stored one after another there.

The syntax of for loop is as follows.

```
for var in sequence:
 statement(s)


```

As described in syntax above, the keywords **for** and **in** are essential keywords to iterate the sequence of values. Then, the variable **var** takes on each consecutive value in the sequence and the statements in the body of the loop are executed once for each value. The simple example of for loop is as follows.

The general example of for loop:

```
for var in range(m,n):
 print var
```

**PROGRAM A.2.3** | Use for loop to print the numbers from 1 to 5.

```
for i in range(1,6):
 print(i)
print("End of The Program")
```

**Output**

```
1
2
3
4
5
End of The Program
```

**3. a. Discuss about immutable constraints and frozen sets.****(7M)****Solution**

Set is an unordered collection of the elements without duplicates. It is a collection of unique elements. Sets are mutable so we can easily add or remove elements. A programmer can create a set by enclosing the elements inside a pair of curly braces, i.e. {}. The elements within the set are separated by commas. The following example illustrates the creation of sets.

**Examples**

```
>>>S1 = set() #Creates an empty Set
>>>S1 #Print Set S1
set()
>>> type(S1) #Check type of S1
<class 'set'>

>>> S1 = {10,20,30,40} #Create set of four elements
>>> S1 #Print Set S1
{40, 10, 20, 30}
```

**Frozen Sets**

The frozen sets are immutable. They are like the sets but they cannot be changed. The examples of immutable frozen sets is are follows.

```
>> Countries = frozenset(["India","USA","JAPAN","UK"])
>>> Countries
frozenset(['JAPAN', 'India', 'UK', 'USA'])
>>> type(Countries)
<type 'frozenset'>
>>> Countries.add("China")
```

```
Traceback (most recent call last):
 File "<pyshell#18>", line 1, in <module>
 Countries.add("China")
AttributeError: 'frozenset' object has no attribute 'add'
```

**b. What are the built-in dictionary functions? Explain.****(7M)****Solution**

The built-in dictionary functions are as follows.

Table A.2.2

| <i>Methods of dict Class</i>                                                                                                                                                                                                                                                             | <i>What it does?</i>                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| <b>keys()</b><br><b>Example</b><br><pre>&gt;&gt;&gt; ASCII_CODE={"A":65,"B":66,"C":67,"D":68} &gt;&gt;&gt; ASCII_CODE      #Print Dictionary named ASCII_CODE {'D': 68, 'B': 66, 'A': 65, 'C': 67} &gt;&gt;&gt; ASCII_CODE.keys() #Return all keys dict_keys(['D', 'B', 'A', 'C'])</pre> | It returns the sequence of keys.                            |
| <b>Values()</b><br><b>Example</b><br><pre>&gt;&gt;&gt; ASCII_CODE={"A":65,"B":66,"C":67,"D":68} &gt;&gt;&gt; ASCII_CODE.values()      #Return Values dict_values([68, 66, 65, 67])</pre>                                                                                                 | It returns the sequences of values.                         |
| <b>items()</b><br><b>Examples</b><br><pre>&gt;&gt;&gt;ASCII_CODE={"A":65,"B":66,"C":67,"D":68} &gt;&gt;&gt;ASCII_CODE.items() dict_items([('D', 68), ('B', 66), ('A', 65), ('C', 67)])</pre>                                                                                             | It returns the sequences of Tuples.                         |
| <b>clear()</b><br><b>Example</b><br><pre>&gt;&gt;&gt;ASCII_CODE={"A":65,"B":66,"C":67,"D":68} &gt;&gt;&gt; ASCII_CODE.clear()      #Delete all entries &gt;&gt;&gt; ASCII_CODE              #Print after {}</pre>                                                                        | It deletes all entries.                                     |
| <b>get(key)</b><br><pre>&gt;&gt;&gt; Temperature={"Mumbai":35,"Delhi":40,"Chennai":54} &gt;&gt;&gt; Temperature.get("Mumbai") 35</pre>                                                                                                                                                   | It returns the values for the key.                          |
| <b>pop(key)</b><br><pre>&gt;&gt;&gt; Temperature.pop("Mumbai") 35 &gt;&gt;&gt; Temperature #Print after removing key "Mumbai". {'Delhi': 40, 'Chennai': 54}</pre>                                                                                                                        | It removes the key and returns the value if the key exists. |
| <b>clear()</b><br><pre>&gt;&gt;&gt; Temperature={"Mumbai":35,"Delhi":40,"Chennai":54} &gt;&gt;&gt; Temperature.clear() &gt;&gt;&gt; Temperature</pre>                                                                                                                                    | It removes all of the keys.                                 |

**4. a. Distinguish between local and global variables with examples.****(7M)****Solution**

The variables that are initialized within a function including parameters are said to exist in that function's local scope. The variables that exist in local scope are called "**local variables**" whereas

the variables that are assigned outside the functions are said to exist in the global scope. Therefore, the variables that exist in global scope are called “global variables”.

#### PROGRAM A.2.4 | Local Scope versus Global scope

```
p = 20 #global variable p
def Demo():
 q = 10 #Local variable q
 print('The value of Local variable q:',q)
 #Access global variable p within this function
 print('The value of Global Variable p:',p)
Demo()
#Access global variable p outside the function Demo()
print('The value of global variable p:',p)
```

##### Output

```
The value of Local variable q: 10
The value of Global Variable p: 20
The value of global variable p: 20
```

#### b. Briefly discuss about Python packages.

(7M)

##### Solution

A package is basically a directory within Python file and the file with the name ‘`_init_.py`’. In short, every directory inside the Python path which contains a file named `_init_.py` will be treated as a package by Python.

Following are the simple steps used to create a package.

- ⊙ **STEP 1:** Let us create a package (folder). Say the name of package is “**My\_First\_Package**”.
- ⊙ **STEP 2:** Create `_init_.py` file inside created package “**My\_First\_Package**”. The directory should contain a file named `_init_.py`. This file can be empty or it may contain valid Python codes. Let keep it as empty file.
- ⊙ **STEP 3:** Inside the package, let us create two different `.py` files i.e. `a.py` and `b.py` files.
- ⊙ **STEP 4:** Write Python code inside `a.py` and `b.py` files. Therefore, contents of `a.py` and `b.py` are as follows.

##### #a.py

```
def call_A():
 print("I am in File a.py")
```

##### #b.py

```
def call_B():
 print("I am in File b.py")
```

Execute both the files, i.e. `a.py` and `b.py`. Make sure that there are no errors present in them.

- ⊙ **STEP 5:** Import ‘**My\_First\_Package**’ from the interactive Python as follows.

```
>>> from My_First_Package import a, b
>>> a.call_A() #Call method Call_A() present in file a.py
```

```
..
I am in File a.py
>>> b.call_B() #Call method Call_B() present in file b.py
I am in File b.py
```

In the example above, we have created three files, `_init_.py`, `a.py` and `b.py`. But the file `_init_.py` is empty. We have used statement `from My_First_Package import a, b` to import the contents of files `a.py` and `b.py`. Alternatively, we can minimize the statement `from My_First_Package import a, b` and automatically load these modules using `_init_.py` file. Thus, open empty `_init_.py` file and insert the following code as follows.

```
import My_First_Package.a
import My_First_Package.b
```

In order to check if the imported modules work within the file `_init_.py`, execute the following instructions in the interactive mode as follows.

```
>>> My_First_Package.a.call_A()
I am in File a.py
>>> My_First_Package.b.call_B()
I am in File b.py
```

#### 5. a. Explain about the processes of the handling an exception.

(7M)

##### Solution

An exception is an error that occurs during run time. In Python, all of the exceptions are the instances of a class which are derived from **BaseException** class. Each exception type is of a separate class. Exception handling in Python is managed by three keywords, viz. **try**, **except** and **finally**. The working of all the keywords is explained below.

Program statements that you want to monitor for exception should be in **try** block. If an exception occurs within the **try** block, then the particular exception is raised (or thrown). The syntax for exception handling that might raise or (throw) an exception in a **try** block is as follows.

```
try:
#Body of try block
exceptException_Type1
#Perform of some kind of exception handling
#It is Exception Handler for Exception Type 1, Alerts the user
exceptException_Type2
#Exception Handler for Exception Type 2
.....
.....
.....
exceptException_TypeN
#Exception Handler for Exception Type N
```

Figure A.2.2 Syntax-Exception Handling

Following program demonstrates the uses of above said mechanism to handle “ZeroDivisionError” division by zero error.

#### PROGRAM A.2.5 | Show the mechanism to handle ZeroDivisionError by Zero Error

```
try:
 a = int(input('Enter the first number:'))
 b = int(input('Enter the second number:'))
 c = a/b
 print(' a = ',a)
 print(' b = ',b)
 print(' a/b = ',c)
except ZeroDivisionError:
 print('You cannot Divide number by Zero')
```

#### Output

##### #Case 1:

Enter the first number:10

Enter the second number:2

a = 10

b = 2

a/b = 5.0

##### #Case 2:

Enter the first number:10

Enter the second number:0

You cannot Divide number by Zero

#### b. Describe data hiding and constructors.

(7M)

#### Solution

The `__init__` method is known as an initializer. It will act as the constructor to a class. It is a special method that is used to initialize instance variable of an object. This method runs as soon as an object of a class is instantiated. Whenever a programmer wants to add constructor to a class, the `__init__` method is used.

The syntax of adding `__init__` method to class is given as follows.

```
class Class_Name:
 def __init__(self): #__init__ method


```

Note that `__init__` needs to be preceded and followed by two underscores. Also the `__init__` method must have `self` as first argument. Here, the `self` refers to the object itself. Therefore, it refers to the object that invokes the method.

The following is the simple program to calculate the area of circle by making use of `__init__` method.

#### PROGRAM A.2.6 | Show the program to calculate the area

```
class Circle:
 def __init__(self,pi):
 self.pi = pi
 def calc_area(self,radius):
 return self.pi*radius**2
C1=Circle(3.14)
print(' The area of Circle is ',C1.calc_area(5))
```

#### Output

```
The area of Circle is 78.5
```

#### 6. a. Explain the writing test cases and running tests.

(7M)

#### Solution

The standard Python library contains a module named “unittest”. Where, the unittest is a unit-testing framework. It comes with Python’s standard library. The module contains the core framework classes which form the basis of test cases and suites. It also provides the utility class for running the tests and reporting the result. The unittest module provides a class called “TestCase” from which we can create subclasses to perform the unit testing.

The following program demonstrates the steps to write test cases using TestCase class for unittest module.

#### PROGRAM A.2.7 | Create file arithmetic.py and write code within it as follows.

```
def mul(X,Y):
 return X*Y
```

**PROGRAM A.2.8** | Write the test case to check if the Program, i.e. `arithmetic.py` works for other numbers. Create a test file called `"arithmetic_test.py"`.

```
from arithmetic import mul
import unittest
class TestArithmetic(unittest.TestCase):
 ##Your test methods go here.
 def test_mul(self):
 self.assertEqual(10,mul(5,2))
unittest.main() #Outside the class, which tells the framework to run
```

### Output

```
Ran 1 test in 0.005s
OK
```

### Explanation

- The first line of a program imports the written program `arithmetic.py` at first and imports its functionality called `mul()` as well.
- Second line imports Python's `unittest` module.
- The `test_mul()` method is executed before each test to check if it returns the correct result.
- Where `assertEqual()` method is provided by imported `unittest` module. This method checks then if the first argument is equals to second or not.



# PART A

1. a. What is indentation?

(2M)

## **Solution**

In most other programming languages, the indentation is used only to help make the code look pretty. But in Python, it is required for indicating what block of code a statement belongs to.

b. Give an example of `isalnum()` method.

(2M)

## **Solution**

The function `isalnum()` returns true if characters in this string are alphanumeric and there is at least one character.

## **Example**

```
>>>S="Python Programming"
>>>S.isalnum()
False
```

```
>>> S="Python"
>>>S.isalnum()
True
```

```
>>> P="1Jhon"
>>>P.isalnum()
True
```

c. Describe `has_key()` method with example.

(2M)

### Solution

The function `has_key()` returns true if a given key is inside the dictionary, else it returns false.

```
>>> dict_players = {'Microsoft': 'Bill Gates', 'Apple': 'Steve Jobs'}
>>> dict_players.has_key('Apple')
True
>>> dict_players.has_key('Facebook')
False
```

d. What is a namespace?

(2M)

### Solution

A namespace is basically a system to make sure that all of the names in a program are unique and can be used without any conflict. Following are the few examples of namespaces.

1. **Local Namespace:** It includes local names inside a function. The namespace is created only when a function is called and lasts until the function returns.
2. **Global Namespace:** This namespace includes the names from various imported modules which you are using in a project. It is created when the module is included in the project and it lasts until the script ends.
3. **Built-in Namespace:** It includes built-in functions and built-in exception names.

e. Explain user-defined exceptions with examples.

(3M)

### Solution

Programmer can create user-defined or custom exceptions by creating a new exception class in Python. This exception class has to be derived from base **Exception** class. Following is the simple example of creating user-defined exceptions.

```
>>> class MyUserDefinedException(Exception):
 def __init__(self):
 Exception.__init__(self, " Please Correct it!! Something went wrong.")
>>> raise MyUserDefinedException
Traceback (most recent call last):
 File "<pyshell#22>", line 1, in <module>
 raise MyUserDefinedException
MyUserDefinedException: Please Correct it!! Something went wrong.
```

**Explanation** In the example above, we have created a user-defined exception called `MyUserDefinedException` which is derived from the `Exception` class. This new exception is raised like other exceptions using the **raise** statement with an optional error message.

The following is the complete example related to creating and throwing user-defined exceptions.

### PROGRAM A.3.1

```
class UserDefinedErrors(Exception):
 pass
class NumberTooSmallException(UserDefinedErrors):
 pass
class NumberTooLargeException(UserDefinedErrors):
 pass
#main program
#User guesses a number until you gets it right
#User need to guess this number
number = 10
while True:
 try:
 input_num = int(input("Enter a number: "))
 if input_num < number:
 raise NumberTooSmallException
 elif input_num > number:
 raise NumberTooLargeException
 break
 except NumberTooSmallException:
 print("Number is too small, try again!")
 except NumberTooLargeException:
 print("Number is too large, try again!")
print("Great Work!! You guessed number correctly.")
```

### Output

```
Enter a number: 2
Number is too small, try again!
Enter a number: 4
Number is too small, try again!
Enter a number: 15
Number is too large, try again!
Enter a number: 12
Number is too large, try again!
Enter a number: 11
Number is too large, try again!
Enter a number: 10
Great Work!! You guessed number correctly.
```

**f. What is time tuple?****(3M)****Solution**

Python's `time` function handles the time as tuple of nine numbers. The following table contains all of those nine tuple members.

**Table A.3.1**

| <i>Field</i>    | <i>Values</i>                                        |
|-----------------|------------------------------------------------------|
| Year            | 2008 (any four digit numbers)                        |
| Month           | 1 to 12                                              |
| Day             | 1 to 31                                              |
| Hour            | 0 to 23                                              |
| Minute          | 0 to 59                                              |
| Second          | 0 to 60                                              |
| Day of Week     | 0 to 6                                               |
| Day of Year     | 1 to 365 or 366                                      |
| Daylight saving | -1, 0, 1, -1 means library which determines the DST. |

# PART B

2. a. What are relational operators used in Python? Explain.

(7M)

## Solution

The relational operators are also called as **Comparison Operators**. All of the comparison operators compare the value of two operands and return true or false based upon the conditions. Following are the relational operators used in python.

**Table A.3.2**

| <i>Relational Operator</i>     | <i>Symbol</i> |
|--------------------------------|---------------|
| Less than                      | <             |
| Greater than                   | >             |
| Less than or Equal to          | <=            |
| Greater than or Equal to       | >=            |
| Equal to (Comparison Operator) | ==            |
| Not Equal to Operator          | !=            |

Examples of all of the relational operators are given as follows.

```
#Less than Operator
>>> 10 < 12
True
#Greater than Operator
>>> 10 > 12
False
#Less than or Equal to
>>> 10<=11
True
#Greater than or Equal to
>>> 12 >= 8
True
#Comparison Operator (Equal to)
>>> 10 == 9
>>> False
#Not Equal to Operator
>>> 10 != 9
True
>>> 10 != 10
False
```

**(7M)**

The syntax for `format()` method is as follows.

```
template.format(P0,P1,.....,k0=V0,K1=V1...}
```

The arguments to the `.format()` method are of two types. It consists of zero or more positional arguments, i.e.  $P_i$  followed by zero or more keyword arguments of the form like  $K_i=V_i$ .

```
>>> '{ } plus{ } equals{ }'.format(4,5,'Nine')
'4 plus 5 equals Nine'
```

Also, the programmer can make use of conversion code **s** to format a string with a specified width. But, by default, the string is left justified. Following are the examples of string formatting.

```
>>> print(format("Hello World!","25s"))#Left Justification
```

Hello World!

```
>>>print(format("HELLO WORLD!",">20s"))#String Right Justification
 HELLO WORLD!
```

In the example above, the statement `(format("HELLO WORLD!", ">20s"))` displays the output in following format.



In print function, '20' specifies the string to be formatted with a width of 20. In second print statement, '>' is used to make right justified the given string.

**(7M)**

The following table contain various built-in functions that can be used with tuples.

### Table A.3.3 Built-in functions that can be used with Tuples

(Contd.)

|             |                                                                                          |
|-------------|------------------------------------------------------------------------------------------|
| sum()       | It returns the sum of all of the elements of tuple.                                      |
| index(x)    | It returns the index of element x.                                                       |
| count(x)    | It returns the number of occurrence of element x.                                        |
| divmod(x,y) | It returns the tuple, ((x - x % y)/y, x % y)                                             |
| all()       | The all() method returns true if all the elements of an iterable are true or else false. |

**Example**

```
>>> t1= ("APPLE")
>>> len(t1) #Return the length of tuple t1
5
>>> max(t1) #Return Element from tuple with Maximum Value
'P'
>>> min(t1) #Return Element from tuple with Minimum value
'A'
>>> t1.index('A')
0
>>> t1.count('P')
2
>>> z = divmod(10,3)
>>> z
(3, 1)
>>> type(z)
<type 'tuple'>
>>> T = (1,2,3,4)
>>> all(T)
True
>>> T = [1, False, 8]
>>> all(T)
False
```

**b. Discuss about list and dictionary comprehensions.****(7M)****Solution****List Comprehension**

List Comprehension is used to create a new list from existing sequences. It is a tool for transforming a given list into another list.

The Syntax of List comprehension is based on the set-builder notation. the set builder notation is a mathematical notation for describing a set by stating the property which its member should satisfy. The syntax is as follows.

**Syntax of List Comprehension**

```
[<expression> for <element> in <sequence> if <conditional>]
```

The syntax is designed to read like English: “Compute the expression for each element in the sequence if the conditional is true.”

**Example** (Using List Comprehension)

```
>>> List1= [10, 20, 30, 40, 50]
>>> List1
[10, 20, 30, 40, 50]
>>>for i in range(0,len(List1)):
 List1[i]=List1[i]+10
>>>List1
[20, 30, 40, 50, 60]
```

#### Without List Comprehension

With reference to the above example, we can say the list comprehension contains

- An input sequence
- A variable referencing the input sequence,
- An optional expression and
- An output expression or output variable

```
>>> List1=[10, 20, 30, 40, 50]
>>> List1=[x+10 for x in List1]
>>> List1
[20, 30, 40, 50, 60]
```

#### Using List Comprehension

#### Example

```
List1= [20, 30, 40, 50, 60]
List1=[x+10 for x in List1]
```

(An Output Variable)

(A variable Referencing An input Sequence)

(An Input Sequence)

#### Dictionary Comprehensions

On the top of list comprehensions, Python also supports dictionary comprehensions. It is a method for transforming one dictionary into another dictionary. It allows programmer to express the creation of dictionaries during run-time. The syntax of create dictionaries using dictionary comprehensions is as follows.

#### Syntax

```
{key: value for (key, value) in iterable}
>>> dict1 = {'a': 1, 'b': 2, 'c': 3}
>>> Cube_dict1 = {k:v**3 for (k,v) in dict1.items()}
>>> cube_dict1
```



**4. a. Explain about required and variable-length arguments.****(7M)****Solution**

Variable is the number of parameters to a function. An argument which begins with \* in function definition gathers arguments into a tuple.

**PROGRAM A.3.2** | Write a Program to create a function `create_tup()` which accepts the variable as the number of arguments and print all of them.

```
def create_tup(*args):
 print(args)
```

Run the above program from interactive mode of Python.

```
>>> create_tup(1,2,3,4)
(1, 2, 3, 4)
>>> create_tup('a','b')
('a', 'b')
```

**b. Discuss in detail about the import statement.****(7M)****Solution**

A module is a piece or part of something. In Python, we can say modules are smaller pieces of bigger programs. Each module is a separate file on hard disk. Programmer can take a big program and split it up into more than one module. In short, modules are Python's **.py** files which contain the Python code. In order to use something that is in a module, you first have to tell Python which module you want to use. The Python can **import** keyword to let you include other modules in your program.

Writing a module is like writing a simple Python program in a file and saving it in **.py** extension. Let us create `Demo.py`.

```
def Display():
 print('Hello, Welcome all!')
```

Let us create another file named `main.py`, so that we can import the module `Demo.py` what we have just created and then call the function `Display()` in order to present in file `Demo.py` from a new file named as `main.py`. The contents of the `main.py` file are as follows.

**PROGRAM A.3.3**

```
#main.py
import Demo #Importing Module named Demo
demo.Display() #Call function Display present within Demo.py
```

## Output

```
'Hello, Welcome all!'
```

### 5. a. Explain inheritance class with suitable example.

(7M)

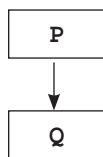
#### Solution

The new classes are created from existing one. The properties of existing classes are simply extended to the new classes. The new classes created using such methods are known as **derived classes** or **sub classes** and the existing classes are known as **base classes** or **super classes**.

The inheritance is classified as follows.

- Single Inheritance
- Multi-level Inheritance
- Multiple Inheritance

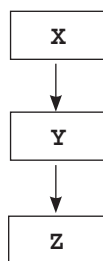
#### a. Single Inheritance



P is a base class. Q is a derived class. This type involves one base and one derived class. Further, no class is derived from Q.

Figure A.3.2 Single Inheritance

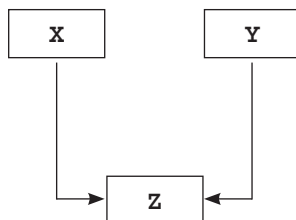
#### b. Multi-level Inheritance



X is a base class. Y is derived from X. Further, Z is derived from Y. Here, Y is not only a derived class but also a base class for Z. Further, Z can be used as a base class.

Figure A.3.3 Multi-Level Inheritance

#### c. Multiple Inheritances



X and Y are base classes. Z is a derived class. Class Z inherits properties of both X and Y. Further, Z is not used as a base class.

Figure A.3.4 Multiple Inheritances

**Syntax (To Inherit Multiple bases classes)**

```
Class Derived_Class_Name(Comma_Seperated_Base_Class_Names):
 Body_of_Derived_Class
```

**PROGRAM A.3.4** | Write a simple program to demonstrate the Concept of Multiple Inheritances.

```
class A: #Base Class A
 a = 0

class B: #Other Base Class B
 b = 0

class C(A,B): #Inherit A and B to create New Class C
 c = 0
 def Read(self):
 self.a=(int(input('Enter the Value of a:')))
 self.b=(int(input('Enter the value of b:')))
 self.c=(int(input('Enter the value of c:')))
 def display(self):
 print(' a = ',self.a)
 print(' b = ',self.b)
 print(' c = ',self.c)

Ob1 = C() #Instance of Child Class
Ob1.Read()
Ob1.display()
```

**Output**

```
Enter the Value of a: 10
Enter the value of b: 20
Enter the value of c: 30
a = 10
b = 20
c = 30
```

**b. Discuss about try-except block with example.****(7M)****Solution**

Exception handling in python is managed by three keywords, viz. **try**, **except** and **finally**. Program statements that you want to monitor for exception should be in **try** block. If an exception occurs within the **try** block, then the particular exception is raised (or thrown). The syntax for exception handling that might raise or (throw) an exception in a **try** block is as follows.

```

try:
#Body of try block
exceptException_Type1
#Perform of some kind of exception handling
#It is Exception Handler for Exception Type 1, Alerts the user
exceptException_Type2
#Exception Handler for Exception Type 2
.....
.....
.....
.....
exceptException_TypeN
#Exception Handler for Exception Type N

```

**Explanation** When an exception is raised (thrown), it is caught by its corresponding **except** statement which then processes the exception.

**PROGRAM A.3.5** | Write a Program to use of **try** and **except** blocks to handle “ZeroDivisionError” division by zero.

```

try:
 a = int(input('Enter the first number:'))
 b = int(input('Enter the second number:'))
 c = a /b
 print(' a = ',a)
 print(' b = ',b)
 print(' a/b = ',c)
except ZeroDivisionError:
 print('You cannot Divide number by Zero')

```

### Output

#### #Case 1:

Enter the first number:10

Enter the second number:2

a = 10

b = 2

a/b = 5.0

#### #Case 2:

Enter the first number:10

Enter the second number:0

You cannot Divide number by Zero

**7. a. Explain about unit-testing in Python.****(7M)****Solution**

The standard Python library containing a module named "unittest". Here the unittest is a unit-testing framework and it comes with Python's standard library. The module contains the core framework classes that form the bases of test cases and suites. It also provides the utility class for running the tests and reporting the results. The unittest module provides a class called "TestCase" from which we can create sub-classes to perform unit-testing.

The following program demonstrates steps to write the test cases using TestCase classes for unittest module.

**PROGRAM A.3.6** | Create file named **arithmetic.py** and write code within it as -follows.

```
def mul(X,Y):
 return X*Y
```

**PROGRAM A.3.7** | Write the test case to check if the Program, i.e. **arithmetic.py** works for other numbers. Create a test file called "**arithmetic\_test.py**".

```
from arithmetic import mul
import unittest
class TestArithmetic(unittest.TestCase):
 ##Your test methods go here.
 def test_mul(self):
 self.assertEqual(10,mul(5,2))
unittest.main() #Outside the class which tells the framework to run
```

**Output**

```
Ran 1 test in 0.005s
OK
```

**Explanation**

- The first line of a program imports the written program, i.e. **arithmetic.py** first and imports its functionality called **mul()**.
- Second line imports Python's **unittest** module.
- The **test\_mul()** method is executed then before each test to check if it returns the correct result or else.
- Then, the **assertEqual()** method is provided by imported **unittest** module. This method checks if first argument is equal to the second one.

**b. Explain the following.**i) **zlib module**ii) **Paned Window****(7M)****Solution****zlib Module**

Python “**zlib**” library allows compression and decompression for the applications which require the data compression. Frequently used in-built functions supported by **zlib** library are as follows.

**Table A.3.4** In-built functions supported by **zlib** library

| <i>In-built Functions</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | <i>Meaning</i>                                                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>compress(string,[level])</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | It compresses data in a string. It returns string containing compressed data. Level indicates the levels of compression from 0 to 9. Level 1 is the fastest and produces the least compression while level 9 is the slowest and produces the most. |
| <code>decompress(data)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | It decompresses the bytes in data. It returns the bytes objects containing the uncompressed data.                                                                                                                                                  |
| <code>Compressobj(level=-1, method=DEFLATED, wbits=MAX_WBITS, memLevel=DEF_MEM_LEVEL, strategy=Z_DEFAULT_STRATEGY[, zdict])</code><br><b>memLevel:</b> It is the amount of memory used for internal compression state.<br><b>Strategy:</b> It is used to tune the compression algorithm.<br><b>wbits:</b> It is used to control window size while compressing data. It decides whether header and trailer to be included in output or not. It can take the range of values limited to 15, i.e. MAX_WBITS | It is used to compress the data streams that would not fit into memory at once. It returns the compressed object.                                                                                                                                  |

**PROGRAM A.3.8** | Write a Program to show the compression and decompression using **zlib** library.

```
import zlib
Message = "Launch Satellite from Earth"
compressed_message = zlib.compress(Message)
print(' Message before Compression(Original):')
print(compressed_message)
decompressed_message = zlib.decompress(compressed_message)
print(' Decompressed Message :',decompressed_message)
```

(Contd.)

**Output**

```
Message before Compression(Original):
xæóI,MîP.IíÉÉ,IUH+ÊiUpM,*É
('Decompressed Message :', 'Launch Satellite from Earth')
```

**Explanation** In the program above, we have used in-built functions, viz. compression and decompression of zlib library. Programmer should also make note that the compression rate varies depending upon the contents of file.

**Paned Window**

The panel window is a container widget which may contain any number of panes horizontally or vertically. Syntax to create this widget is as follows.

```
Paned_Window = PanedWindow(master,option,.....)
```

Please note that the **master** represents the parent window.

The list of commonly used options on the paned window are as follows.

- Bg: This is the colour of the sliders and arrowheads when mouse is not over them.
- borderwidth: It defines border width which is by default 2.
- orient: It is default as horizontal.
- height: No such default value it has.
- width: No such default value it has.

**PROGRAM A.3.9** | Write a simple Program on Panel Window.

```
from Tkinter import *

p1 = PanedWindow()
p1.pack(fill=BOTH, expand=1)

Left_Pane = Label(p1, text="left pane")
p1.add(Left_Pane)

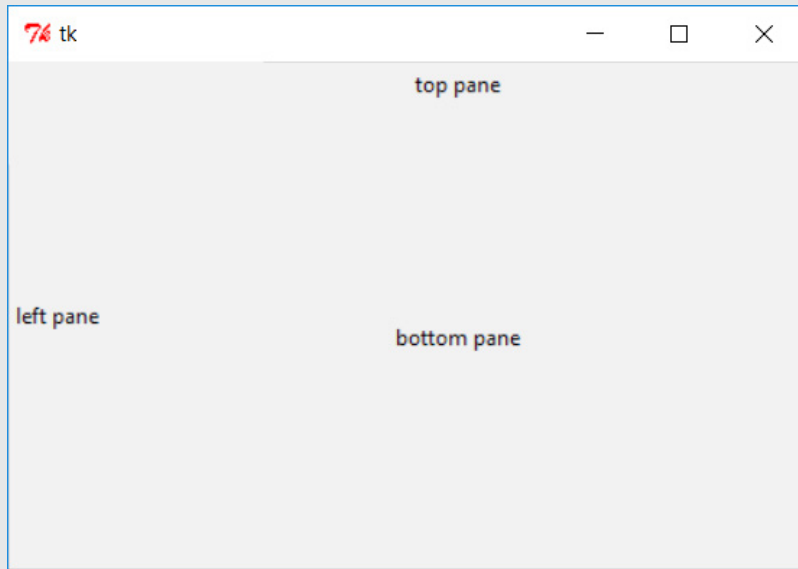
p2 = PanedWindow(p1, orient=VERTICAL)
p1.add(p2)

Top_Pane = Label(p2, text="top pane")
p2.add(Top_Pane)

Bottom_Pane = Label(p2, text="bottom pane")
p2.add(Bottom_Pane)

mainloop()
```

(Contd.)

**Output****Figure A.3.5**



# PART A

1. a. What is byte code?

(2M)

## Solution

If there is no error, i.e. if python instruction/code is in well-formatted order, then the interpreter translates it into its equivalent form in low-level language called “**Byte Code**”.

b. Give an example of `endswith()` method.

(2M)

## Solution

The function `endswith(str)` returns true if the string ends with the substring `Str1`.

## Example

```
>>> S="Python Programming"
>>>S.endswith("Programming")
True
```

c. What is a set? Why the sets are required?

(2M)

## Solution

- Set is an unordered collection of elements without any duplicate element.
- It is a collection of unique elements.
- The sets are mutable, so we can easily add or remove any element.
- The set data structure in python is used to support mathematical set operations.

**Example**

The following example creates two different sets A and B and we will perform intersection of both the sets and store result into set C.

```
>>> A = {1, 2, 3}
>>> B = {2, 3, 4}
>>> C = A.intersection(B)
>>> C
set([2, 3])
>>> type(A)
<type 'set'>
>>> type(B)
<type 'set'>
>>> type(C)
<type 'set'>
```

**d. Define anonymous function.****(2M)****Solution**

- In Python, the anonymous function is defined without a name.
- The normal functions are defined using the `def` keyword. In Python, the anonymous functions are defined using the `lambda` keyword.
- The `lambda` operator or `lambda` function is used to create small anonymous functions.
- Hence, the anonymous functions are also called `lambda` functions. The general syntax of `lambda` function is as follows.

**`lambda argument-list: expression`**

- The argument list consists of a comma separating list of arguments and the expression is an arithmetic expression using these arguments.
- The following example of a `lambda` function returns the multiplication of its two arguments.

```
>>> f = lambda x, y : x * y
>>> mul = f(2,8)
>>> mul
16
```

**e. Give an example for raising an exception.****(3M)****Solution**

- Raising an exception is also like stop running the code in the function and move the program execution to the **`except`** statement.
- All these exception are wrapped in objects and are created from the classes.
- Therefore, the programmer can raise this exception using the **`raise`** statement.
- The syntax to raise an exception is as follows.

**Syntax**

```
raise Exception(value)
```

Here, the Exception is the exception type.

**Example**

```
raise ArithmeticError('Something is wrong')
```

**PROGRAM A.4.1** | Write a Program to raise the exception.

```
A = 10
b = 0
try:
 raise ArithmeticError('Cannot divide number by zero')
 c = a/b
except ZeroDivisionError:
 print('Something is Wrong')
 raise
```

**Output**

```
Traceback (most recent call last):
 File "C:\Python34\asacs.py", line 4, in <module>
 raise ArithmeticError('Something is Wrong')
ArithmeticError: Cannot divide number by zero
```

**f. What is tick?****(3M)****Solution**

- Python has in-built **time** module.
- The time module provides functions for working with times.
- When it comes to the concept of ticks, time is expressed in seconds, since 12:00 am, January 1, 1970.
- In these cases, the time intervals are nothing but the floating point numbers expressed in seconds.
- The function time.time() returns the current system time in ticks.

**PROGRAM A.4.2**

```
import time
ticks = time.time()
print("Number of ticks since 12:00 am, January 1, 1970:", ticks)
```

**Output**

```
('Number of ticks since 12:00 am, January 1, 1970:', 1528005429.106)
```

# PART B

## 2. a. Briefly discuss about running Python scripts.

(7M)

### Solution

All of the steps below are explained with respect to Python IDLE 3.5.

- Running python program from a script file is known as running python in “**script mode**”.
- You can write sequence of instructions in one file and execute them.

The steps required to write python programs in python IDLE’S script mode.

- In python IDLE’s shell window, click on **File** and then click on **New File** or just click **CTRL + N**.
- Thus, after opening the above window you can write series of instructions and run it to view the output.
- Now we have to write the Python code in script mode.
- Save the code written in script mode by some Name. You can do this by simple Click on File and then click on **Save Button** or press **Ctrl + S**.
- Thus, after opening the save As window, write the name of your own python program.
- Thus, to run the written and saved program, click on **Run** from the menu bar and then click on **run module** or you can directly press **ctrl + F5** in order to see the output if the program is error-free or else.

## b. Write the history of Python.

(7M)

### Solution

- Python was developed by Guido van Rossum at National Research Institute for Mathematics and computer science in Netherlands in 1990.
- Guido van Rossum was inspired by Monty Python’s *flying circus*, a BBC-aired comedy series and he wanted the name of his new language to be short, unique and mysterious.
- Hence, he named it as Python.
- Rossum developed python and it became a popular programming language as widely used in industry and academia because of its simple, concise and extensive support of libraries.
- It is a general purpose, interpreted and object-oriented programming language. Python source code is available under the General Public License (**GPL**) and maintained by a core development and maintained at the National Research Institute.

**3. a. Explain Python bitwise operators with example.****(7M)****Solution**

Python has six bitwise operators. Python supports bitwise operators for the bit-wise manipulations. The Bitwise operator permits the programmer to access and manipulate individual bits within a piece of data.

**Table A.4.1**

| <i>Operator</i> | <i>Meaning</i> |
|-----------------|----------------|
| &               | Bitwise AND    |
|                 | Bitwise OR     |
| ^               | Bitwise XOR    |
| >>              | Right Shift    |
| <<              | Left Shift     |
| ~               | Bitwise NOT    |

**The Bitwise AND (&) operator**

the truth table of **And (&)** Operator is as follows.

**Table A.4.2**

| <i>Input</i> |          | <i>Output</i>    |
|--------------|----------|------------------|
| <b>X</b>     | <b>Y</b> | <b>X &amp; Y</b> |
| 0            | 0        | 0                |
| 0            | 1        | 0                |
| 1            | 0        | 0                |
| 1            | 1        | 1                |

**The Bitwise OR (|) operator**

The truth table of **OR (|)** Operator is as follows.

**Table A.4.3**

| <i>Input</i> |          | <i>Output</i> |
|--------------|----------|---------------|
| <b>X</b>     | <b>Y</b> | <b>X   Y</b>  |
| 0            | 0        | 0             |
| 0            | 1        | 1             |
| 1            | 0        | 1             |
| 1            | 1        | 1             |

### The Bitwise XOR (^) Operator

The truth table of XOR operator is as follows.

**Table A.4.4**

| Input |   | Output       |
|-------|---|--------------|
| X     | Y | $X \wedge Y$ |
| 0     | 0 | 0            |
| 0     | 1 | 1            |
| 1     | 0 | 1            |
| 1     | 1 | 0            |

### The Right Shift (>>) Operator

The right shift operator is represented by >>. It also needs two operands. It is used to shift bits to right by  $n$  position.

#### Example

```
>>>4 >> 2 #The input data 4 is to be shifted by 2 bits right side
1
>>>8 >> 2
2
```

### The Left Shift (<<) Operator

The left shift operator is represented by <<. It also needs two operands. It is used to shift bits to left by  $n$  position.

```
>>> 4<< 2 #The input data 4 is to be shifted by 2 bits left side
16

>>> 8<< 2 #The input data 8 is to be shifted by 2 bits towards left
3
```

### b. Discuss about Python operators precedence with example.

(7M)

#### Solution

The operator precedence determines the order in which the python interpreter evaluates the operators in an expression.

For example, consider the expression  $4+5*3$ .

So, how does Python know which operation needs to perform first? From the above example, i.e.  $4+5*3$ , it is important to know whether  $4+5*3$  evaluates to 19 (where the multiplication has been done first) or 27 (where the addition has been done first). But the default order of precedence determines that multiplication is computed first so the result will be 19. As an expression may contain lot of operators, the operations on the operands are carried out according to the priority which is called the precedence of the operator. The operator having higher priority is evaluated first.

Also, when an expression contains operators with equal precedence, then the associative property decides which operation is to be performed first. The associative property or associativity depicts the direction of execution. Associativity is of two types as follows.

**i. Left to Right:** In this type of expression, the evaluation starts from left to right direction.

**Example**  $4 + 6 - 3 + 2$

In the example as shown above, all operators are having the same precedence, therefore, the associativity rule is followed (i.e. direction of execution is from left to right always).

The evaluation of the expression  $4+6-3+2$  is equivalent to

$$\begin{aligned} &= ((4+6)-3)+2 \\ &= ((10)-3)+2 \\ &= (7)+2 \\ &= 9 \end{aligned}$$

**ii. Right to Left:** In this type of expression, the evaluation starts from right to left.

**Example**

$X = Y = Z = \text{Value}$

In the example as shown above, the assignment operators are used. The value of Z is assigned to Y and then to X. Thus, the evaluation starts from right to left direction.

The precedence and associativity table for arithmetic operators has been shown in table below.

**Table A.4.5** Associativity Rules for Arithmetic Operators

| Precedence | Operators   | Associativity          |
|------------|-------------|------------------------|
| Highest    | ( )         | Innermost to Outermost |
|            | **          | Highest                |
| ↓          | *, /, //, % | Left to Right          |
| Lowest     | + -         | Left to Right          |

**4. a. What are built-in dictionary functions? Explain.**

**(7M)**

**Solution**

The built-in dictionary functions are as follows.

**Table A.4.6**

| Methods of dict Class                                                                                                                                                                                                                                        | What it does?                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| <b>keys()</b>                                                                                                                                                                                                                                                | It returns the sequence of keys. |
| <b>Example</b>                                                                                                                                                                                                                                               |                                  |
| <pre>&gt;&gt;&gt; ASCII_CODE={"A": 65, "B": 66, "C": 67, "D": 68} &gt;&gt;&gt; ASCII_CODE      #Print Dictionary named ASCII_CODE {'D': 68, 'B': 66, 'A': 65, 'C': 67} &gt;&gt;&gt; ASCII_CODE.keys() #Return all keys dict_keys(['D', 'B', 'A', 'C'])</pre> |                                  |

(Contd.)

|                                                                                                                                                                                                                |                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
| <b>Values()</b>                                                                                                                                                                                                | It returns the sequence of values.                         |
| <b>Example</b><br><pre>&gt;&gt;&gt; ASCII_CODE={"A": 65, "B": 66, "C": 67, "D": 68} &gt;&gt;&gt; ASCII_CODE.values()          #Return Values dict_values([68, 66, 65, 67])</pre>                               |                                                            |
| <b>items()</b>                                                                                                                                                                                                 | It returns the sequence of Tuples.                         |
| <b>Example</b><br><pre>&gt;&gt;&gt;ASCII_CODE={"A": 65, "B": 66, "C": 67, "D": 68} &gt;&gt;&gt;ASCII_CODE.items() dict_items([('D', 68), ('B', 66), ('A', 65), ('C', 67)])</pre>                               |                                                            |
| <b>clear()</b>                                                                                                                                                                                                 | It deletes all entries.                                    |
| <b>Example</b><br><pre>&gt;&gt;&gt;ASCII_CODE={"A": 65, "B": 66, "C": 67, "D": 68} &gt;&gt;&gt; ASCII_CODE.clear()          #Delete all entries &gt;&gt;&gt; ASCII_CODE                  #Print after {}</pre> |                                                            |
| <b>get(key)</b>                                                                                                                                                                                                | It returns the value for the key.                          |
| <b>Example</b><br><pre>&gt;&gt;&gt; Temperature={"Mumbai": 35, "Delhi": 40, "Chennai":54} &gt;&gt;&gt; Temperature.get("Mumbai") 35</pre>                                                                      |                                                            |
| <b>pop(key)</b>                                                                                                                                                                                                | It removes the key and returns the value if the key exist. |
| <b>Example</b><br><pre>&gt;&gt;&gt; Temperature.pop("Mumbai") 35 &gt;&gt;&gt; Temperature #Print after removing key "Mumbai". {'Delhi': 40, 'Chennai': 54}</pre>                                               |                                                            |
| <b>clear()</b>                                                                                                                                                                                                 | It removes all the keys.                                   |
| <b>Example</b><br><pre>&gt;&gt;&gt; Temperature={"Mumbai": 35, "Delhi": 40, "Chennai":54} &gt;&gt;&gt; Temperature.clear() &gt;&gt;&gt; Temperature</pre>                                                      |                                                            |

## b. Explain about the importance of lists in Python.

(7M)

### Solution

- We need to store variables of same data type on many occasions.
- For example, the currency notes used in daily life lives India are of denominations Rs. 5, 10, 20, 100, 500 and 2000.
- In case, if the programmer wants to display all these six currency notes, then by the normal way of programming programmer may print it by reading all the currency notes in six different variables.



- But by making use of the list, the programmer can use just single variable to store all of the elements of same type or different types and print them using single variable.
- Similarly, in day-to-day life applications, we can make use of lists to display the list of top 100 countries in the world, the list of students qualifying GRE exams, a shopping list of grocery store and so on.
- The list is not only able to store same type of values, it can also store values of different types.
- The simple working examples of list are as follows.

### Example

- a. Create a list with any three integer elements, such as, 10, 20 and 30.

```
L1=[10,20,30]
```

- b. List can contain the elements of Mixed Type.

```
L3=["Jhon","Male",25,5.8]
```

### 5. a. Write a brief note on PIP. Explain installing packages via PIP.

(7M)

### Solution

The PIP is a tool for installing python packages from "Python Package Index" (PyPI). The PyPI is repository of software for python programming language. It helps you to find and install software developed and shared by the python community.

### PIP Directory Structure

The following is the directory structure where you can find the PIP.

```
Python27 (Base Directory)
|
+-----Scripts (Sub-directory)
|
+-----PIP
```

The Python software packages can easily be installed by making use of command line interface and just writing one single command. The syntax to install any package is as follows.

```
pip install some-package-name
```

Similarly, the user can uninstall the installed packages using PIP. The syntax is as follows.

```
pip uninstall some-package-name
```

Following are the some of the options provided by PIP command and they are as follows.

**Table A.4.7**

|                            |                                                    |
|----------------------------|----------------------------------------------------|
| <b>list</b>                | List install packages.                             |
| <b>list-format columns</b> | List install packages with formatted column.       |
| <b>legacy</b>              | Uses legacy formatting.                            |
| <b>json</b>                | Uses json formatting to list installed packages.   |
| <b>freeze</b>              | Uses freeze formatting to list installed packages. |

(Contd.)

| Outdated                                              | List outdated packages. |
|-------------------------------------------------------|-------------------------|
| <b>Example:</b>                                       |                         |
| C:\Python27\Scripts>pip list outdated--format columns |                         |
| Package                                               | Version                 |
| -----                                                 | -----                   |
| backports.functools-lru-cache                         | 1.5                     |
| cycler                                                | 0.10.0                  |
| Cython                                                | 0.28.2                  |
| matplotlib                                            | 2.2.2                   |
| numpy                                                 | 1.14.2                  |
| pandas                                                | 0.22.0                  |
| pdfminer                                              | 20140328                |
| Pillow                                                | 5.1.0                   |
| pip                                                   | 9.0.1                   |

**b. Explain about keyword and default arguments. (7M)**

**Solution**

**Default Arguments**

like other languages, Python provides support for default argument values. It is a function arguments, that can either be specified by the caller or left blank to automatically receive a pre-defined value.

**PROGRAM A.4.3**

```
def greet(name,msg="Welcome to Python!!"):
 print(" Hello ",name,msg)
greet("Sachin")
```

**Output**

Hello Sachin Welcome to Python!!

In the Program above, the function greet(), has the parameter **name**. The parameter **name** does not have default value and is required mandatory during a function call. On the other hand, the parameter **msg** has a default value of "Welcome to Python!!". So, during a function call, it will be optional. If a value is provided, it will overwrite the default value. Here are some valid function calls to this function.

**Keyword Arguments**

The programmer can pass a keyword argument to a function by using its corresponding parameter's name rather than its position. Simply type **Parameter\_name = value** in the function call.

Syntax to call function using keyword argument is as follows.

```
Name_of_Function(pos_args,keyword1=value,keyword2=value2.....)
```

**PROGRAM A.4.4** | Write a simple Program on Keyword Argument.

```
def Display(Name,age):
 print("Name = ",Name,"age = ",age)
Display(age=25,Name="Jhon") #Call function using keyword arguments
```

**Output**

```
Name = Jhon age = 25
```

**6. a. Give an overview of OOP terminology.****(7M)****Solution**

Object-oriented languages help the programmer to reduce complexity of programs. It also encourages the programmer to re-use the existing module or functions. The general concept of object-oriented programming language is about **class**. Here, the **class** is just another name for a **type** in python. It means programmer can create objects of their own class.

Overall, in any object-oriented languages, the **class** defines what object of its type looks like and how it behaves. For an example, we know how an integer looks like and its behaviour. Behaviour is the operation where one can perform on it.

The **class** may also contain data in the form of **fields**. Fields are also called as **attributes** and codes in the form of **procedure** been known as **methods**. Finally, the programmer has to create object of their own **class**. Here, object represents an entity which can be easily identified. For an example, the person, vehicle, fan, books and so on represents the real-time objects. One important thing regarding the object is that each object has a unique identity, state and behaviour. The object state is also called as **properties** or **attributes**.

Syntax for defining class in Python is as follows.

```
Class Class_Name:
 Initializer
 attributes
 methods()
 Statement(s)
```

**b. Explain about except clause with multiple exceptions.****(7M)****Solution**

If an exception occurs within the **try** block, then the particular exception is raised (or thrown). When an exception is raised (thrown), it is caught by its corresponding **except** statement. Sometimes, the code generates the multiple exceptions. In such case, multiple **except** blocks are used to handle multiple exceptions. The following program demonstrates using multiple **except** clause in a program.

**PROGRAM A.4.5** | Write a Python program to include multiple exceptions.

```

try:
 n1 = int(input('Enter the number:'))
 print(n1)
 q = 200/n1
except ValueError:
 print('Entered string is not of type int')
except ZeroDivisionError:
 print('Number cannot be divided by Zero')

```

**Output**

```

#Case 1:
Enter the number:svsvadbs
Entered string is not of type int

#Case 2:
Enter the number:0
0
Number cannot be divided by Zero

```

**7. a. What is multithreading? Discuss about starting a new thread.****(7M)****Solution**

- The multi-threading library is lightweight, it shares memory and been responsible for responsive UI. It is used well for Input/Output-bound applications.
- Multiple threads live in the same process in the same space.
- Each thread will do a specific task as it has its own code and shares the heap memory.
- Python contains two modules to implement threads in the programs.
  - thread module: It implements a thread as a function.
  - threading module: It provides object-oriented approach to enable the thread creation.
- To start new thread, call the method `start_new_thread(function, args[,kargs])` which is present in thread module.
- This method starts with a new thread and returns its identifier.
- It will invoke the function specified as the “function” parameter with the passed list of arguments.
- Following is the simple program which can be used to calculate square and cube of a number by creating two different threads.
- Running two different threads will reduce the time of execution of them in parallel.

**PROGRAM A.4.6** | Create two different threads to calculate square and cube of a number using threading module.

```
import threading
def calc_square(number):
 print('Square:', number * number)

def calc_Cube(number):
 print('Cube:' , number * number * number)

if __name__ == "__main__":
 number = 7

 thread1 = threading.Thread(target=calc_square, args=(number,))
 thread2 = threading.Thread(target=calc_Cube, args=(number,))

 #Will execute both threads in parallel
 thread1.start()
 thread2.start()

 thread1.join()
 thread2.join()
```

**Output**

```
('Square:', 49) ('Cube:', 343)
```

**b. Explain about colours and filled shapes.****(7M)****Solution**

A turtle object contains the methods for setting colours. The following table describes the list of methods to draw different shapes with different colours.

**Table A.4.8** List of methods to draw different shapes with different colours.

| Method                           | Meaning                                                                 |
|----------------------------------|-------------------------------------------------------------------------|
| <code>turtle.color(c)</code>     | It sets the pen colour.                                                 |
| <code>turtle.fillcolor(C)</code> | It sets the pen fill colour to "C".                                     |
| <code>turtle.begin_fill()</code> | It calls this method before filling a shape.                            |
| <code>turtle.end_fill()</code>   | It fills the shape drawn before the last call to <b>begin_fill()</b> .  |
| <code>turtle.filling()</code>    | It returns the fill state. True is filling and False if not-filling.    |
| <code>turtle.clear()</code>      | It clears the window. The state and position of window is not affected. |

(Contd.)

|                                                                              |                                                                                      |
|------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <code>turtle.reset()</code>                                                  | It clears the window and reset the state and position to its original default value. |
| <code>turtle.screensize()</code>                                             | It sets the width and height of the canvas.                                          |
| <code>turtle.showturtle()</code>                                             | It makes turtle visible.                                                             |
| <code>turtle.hideturtle()</code>                                             | It makes the turtle invisible.                                                       |
| <code>turtle.write(msg,move,align,font=font name, fontsize, fonttype)</code> | They write messages on the turtle graphics window.                                   |

The following Program demonstrate the uses of **begin\_fill()** and **end\_fill()** method to fill a shape.

#### PROGRAM A.4.7

```
import turtle
turtle.fillcolor ("grey") #Fill grey colour inside the square
turtle.begin_fill ()
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.end_fill()
```

The above Program A.4.7 has created square and fill square with colour, "grey".