PYTHON PROGRAMMING
USING PROBLEM SOLVING APPROACH

Reema Thareja
Assistant Professor
Department of Computer Science
Shyama Prasad Mukherji College for Women
University of Delhi.

OXFORD UNIVERSITY PRESS

© Oxford University Press. All rights reserved.
Preface

Computers are so widely used in our day-to-day lives that imagining a life without them has become almost impossible. They are not only used by professionals but also by children for interactively learning lessons, playing games, and doing their homework. Applications of the computer and its users are increasing by the day. Learning computer and programming basics is a stepping stone to having an insight into how the machines work. Once the reader is aware of the basic terminologies and problem solving strategies that are commonly used in computer science, he/she can then go on to develop efficient and effective computer programs that may help solve a user’s problems.

Since computers cannot understand human languages, special programming languages are designed for this purpose. Python is one such language. It is an open-source, easy, high-level, interpreted, interactive, object-oriented and reliable language that uses English-like words. It can run on almost all platforms including Windows, Mac OS X, and Linux. Python is also a versatile language that supports development of a wide range of applications ranging from simple text processing to WWW browsers to games. Moreover, programmers can embed Python within their C, C++, COM, ActiveX, CORBA, and Java programs to give ‘scripting’ capabilities to the users.

Python uses easy syntax and short codes as well as supports multiple programming paradigms, including object oriented programming, functional Python programming, and parallel programming models. Hence, it has become an ideal choice for the programmers and even the novices in computer programming field find it easy to learn and implement. It has encompassed a huge user base that is constantly growing and this strength of Python can be understood from the fact that it is the most preferred programming language in companies such as Nokia, Google, YouTube, and even NASA for its easy syntax and short codes.

About the Book

This book is designed as a textbook to cater to the requirements of the first course in Python programming. It is suited for undergraduate degree students of computer science engineering and information technology as well as postgraduate students of computer applications. The objective of this book is to introduce the students to the fundamentals of computers and the concepts of Python programming language, and enable them to apply these concepts for solving real-world problems.

The book is organized into 12 chapters that provide comprehensive coverage of all the relevant topics using simple language. It also contains useful annexures to various chapters including for additional information. Case studies and appendices are also provided to supplement the text.

Programming skill is best developed by rigorous practice. Keeping this in mind, the book provides a number of programming examples that would help the reader learn how to write efficient programs. These programming examples have already been compiled and tested using Python 3.4.1 version and can be also executed on Python 3.5 and 3.6 versions. To further enhance the understanding of the subject, there are numerous chapter-end exercises provided in the form of objective-type questions, review questions, and programming problems.

Key Features of the Book

The following are the important features of the book:

• Offers **simple and lucid** treatment of concepts supported with illustrations for easy understanding.
• Contains **separate chapters** on Strings, Files, Exception Handling, and Operator Overloading.
• Provides **numerous programming examples** along with their outputs to help students master the art of writing efficient Python programs.

• Includes **notes and programming tips** to highlight the important concepts and help readers avoid common programming errors.

• Offers **rich chapter-end pedagogy** including plenty of objective-type questions (with answers), review questions, programming and debugging exercises to facilitate revision and practice of concepts learnt.

• Includes **7 Annexures and 5 appendices** covering types of operating systems, differences between Python 2.x and 3.x, installing Python, debugging and testing, iterators, generators, getters, setters, @property, @deleter, Turtle graphics, plotting graphs, multi-threading, GUI and Web Programming provided to supplement the text. Exercises are also added at the end of several annexures and appendices.

• Provides **case studies** on creating calculator, calendar, hash files, compressing strings and files, tower of Hanoi, image processing, shuffling a deck of cards, and mail merge that are linked to various chapters to demonstrate the application of concepts.

• **Point-wise summary and glossary** of key terms to aid quick recapitulation to concepts.

**Organization of the Book**

The book contains 12 chapters, 7 annexures, 8 case studies, and 5 appendices. The details of the book are presented as follows.

**Chapter 1** provides an introduction to computer hardware and software. It covers the concept of memory and its storage units, application software, and system software. The chapter provides an insight into the different stages of software development life cycle and discusses the various strategies used for problem solving. Topics such as algorithms, flowcharts, and pseudocodes are discussed in this chapter.

**Annexure 1** given after Chapter 1 discusses the classification of operating systems.

**Chapter 2** discusses about programming languages and their evolution through generations. It describes different programming paradigms, features of OOP, and merits and demerits of object oriented programming languages. The chapter also gives a comparative study Python and other OOP languages, and highlights the applications of OOP paradigm.

**Chapter 3** details the history, important features and applications of Python. It also presents the various building blocks (such as keywords, identifiers, constants variables, operators, expressions, statements and naming conventions) supported by the language.

The chapter is followed 3 annexures – **Annexure 2** provides instructions for installing Python. **Annexure 3** provides the comparison between Python 2.x and Python 3.x versions. **Annexure 4** discusses testing and debugging of Python programs using IDLE.

**Chapter 4** deals with the different types of decision control statements such as selection/ conditional branching, iterative, break, continue, pass, and else statements.

**Case studies 1 and 2** on simple calculator and generating a calendar show the implementation of concepts discussed in Chapters 3 and 4.

**Chapter 5** provides a detailed explanation of defining and calling functions. It also explains the important concepts such as variable length arguments, recursive functions, modules, and packages in Python.

**Annexure 5** explains how functions are objects in Python. **Case studies 3 and 4** on tower of Hanoi and shuffling a deck of cards demonstrates the concepts of functions as well as recursion.

**Chapter 6** unleashes the concept of strings. The chapter lays special focus on the operators used with strings, slicing operation, built-in string methods and functions, comparing and iterating through strings, and the string module.

**Chapter 7** discusses how data can be stored in files. The chapter deals with opening, processing (like reading, writing, appending, etc.), and closing of files though a Python program. These files are handled in text mode as well as binary mode for better clarity of the concepts. The chapter also explains the concept of file, directory, and the os module.

© Oxford University Press. All rights reserved.
Case studies 5, 6, and 7 on creating a hash file, mail merge, and finding the resolution of an image demonstrate the applications of concepts related strings and file handling.

Chapter 8 details the different data structures (such as list, tuple, dictionary, sets, etc.) that are extensively used in Python. It deals with creating, accessing, cloning, and updating of lists as well as list methods and functions. It also describes functional programming and creating, accessing, and updating tuples. It also includes the concepts related to sets, dictionaries, nested lists, nested tuples, nested sets, nested dictionaries, list comprehensions, and dictionary comprehensions.

Annexure 6 discusses the concepts of iterator and generator.

Chapter 9 introduces the concept of classes, objects, public and private classes, and instance variables. It also talks about special methods, built-in attributes, built-in methods, garbage collection, class method, and static method.

Annexure 7 discusses the getter and setter methods as well as @property and @deleter decorators facilitate data encapsulation in Python.

Chapter 10 introduces inheritance and its various forms. It gives a detail explanation on method overriding, containership, abstract class, interface, and metaclass.

Chapter 11 is all about overloading arithmetic and logical operators. It also discusses reverse adding and overriding __getitem__(), __setitem__(), and __call__() methods, in operator, as well as other miscellaneous functions.

Chapter 12 elucidates the concepts of exception handling that can be used to make your programs robust. Concepts such as try, except, and finally blocks, raising and re-raising exceptions, built-in and user-defined exceptions, assertions, and handling invoked functions, used for handling exceptions are demonstrated in this chapter.

Case study 8 shows how to compress strings and files using exception handling concepts.

The 5 appendices included in the book discuss about multi-threading, GUI programming, usage of Turtle graphics, plotting graphs and web programming in Python.

Online Resources
For the benefit of faculty and students reading this book, additional resources are available online at india.oup.com/orcs//9780199480173

For Faculty
• Solutions manual (for programming exercises)
• Chapter-wise PPTs

For Students
• Lab exercises
• Test generator
• Projects
• Solutions to find the output and error exercises
• Model question papers
• Extra reading material on number systems, unit testing in Python, sorting and searching methods, network programming, event-driven programming and accessing databases using Python

Acknowledgements
The writing of this textbook was a mammoth task for which a lot of help was required from many people. Fortunately, I have had wholehearted support of my family, friends, and fellow members of the teaching staff and students at Shyama Prasad Mukherji College, New Delhi.

© Oxford University Press. All rights reserved.
My special thanks would always go to my parents, Mr Janak Raj Thareja and Mrs Usha Thareja, and my siblings, Pallav, Kimi, and Rashi, who were a source of abiding inspiration and divine blessings for me. I am especially thankful to my son, Goransh, who has been very patient and cooperative in letting me realize my dreams. My sincere thanks go to my uncle, Mr B.L. Theraja, for his inspiration and guidance in writing this book.

I would like to acknowledge the technical assistance provided to me by Mr Mitul Kapoor. I would like to thank him for sparing out his precious time to help me to design and test the programs.

I would like to express my gratitude to the reviewers for their valuable suggestions and constructive feedback that helped in improving the book.

Prof. M V S V Kiranmai
University College of Engineering, JNTU Kakinada

Dr Nagender Kumar Suryadevara
Geethanjali College of Engineering and Technology, Hyderabad

Dr Vipul Kumar Mishra
School of Engineering, Bennett University, Greater Noida, U.P.

Dr G Shobha
R V College of Engineering, Bengaluru

Prof. Priyang P Bhatt
GH Patel College of Engineering and Technology, Vallabh Vidyanagar, Gujarat

Prof. Karthick Nanmaran
School of Computing, SRM University, Chennai

Last but not the least, I would like to thank the editorial team at Oxford University Press, India for their help and support over the past few years.

Comments and suggestions for the improvement of the book are welcome. Please send them to me at reemathareja@gmail.com.

Reema Thareja
Brief Contents

Preface  iv
Detailed Contents  ix

1 Introduction to Computers and Problem Solving Strategies  1
   Annexure 1 — Types of Operating Systems  52
2 Introduction to Object Oriented Programming (OOP)  63
3 Basics of Python Programming  83
   Annexure 2 — Installing Python  125
   Annexure 3 — Comparison between Python 2.x and Python 3.x Versions  127
   Annexure 4 — Testing and Debugging  130
4 Decision Control Statements  137
   Case Study 1 — Simple Calculator  180
   Case Study 2 — Generating a Calendar  183
5 Functions and Modules  185
   Annexure 5 — Functions as Objects  237
   Case Study 3 — Tower of Hanoi  239
   Case Study 4 — Shuffling a Deck of Cards  241
6 Python Strings Revisited  242
7 File Handling  289
   Case Study 5 — Creating a Hash File (or a message digest of a file)  317
   Case Study 6 — Mail Merge Program  319
   Case Study 7 — Finding Resolution of an Image  321
8 Data Structures  322
   Annexure 6 — Iterator and Generator  392
9 Classes and Objects  400
   Annexure 7 — Getters, Setters, @property, and @deleter  432
10 Inheritance  436
11 Operator Overloading  460
12 Error and Exception Handling  480
   Case Study 8 — Compressing String and Files  505

Appendix A — Multi-threading  508
Appendix B — GUI Programming With tkinter Package  514
Appendix C — Simple Graphics Using Turtle  524
Appendix D — Plotting Graphs in Python  530
Appendix E — CGI/Web Programming Using Python  536

© Oxford University Press. All rights reserved.
1 Introduction to Computers and Problem Solving Strategies  1

1.1 Introduction  1
1.2 What is a Computer?  1
1.3 History of Computers  2
1.4 Characteristics of Computers  4
1.5 Classification of Computers  5
1.6 Basic Applications of Computers  9
1.7 Stored Program Concept  11
   1.7.1 Types of Stored Program Computers  12
1.8 Components and Functions of a Computer System  12
1.9 Concept of Hardware and Software  14
   1.9.1 Hardware  14
   1.9.2 Software  14
1.10 Central Processing Unit (CPU): Basic Architecture  14
1.11 Input and Output devices  16
1.12 Computer Memory  17
   1.12.1 Memory Hierarchy  18
   1.12.2 Primary Memory  18
   1.12.3 Secondary Storage Devices  20
1.13 Classification of Computer Software  20
   1.13.1 System Software  22
   1.13.2 Application Software  27
1.14 Representation of Data: Bits and Bytes  29
1.15 Problem Solving Strategies  31
1.16 Program Design Tools: Algorithms, Flowcharts, Pseudocodes  32
   1.16.1 Algorithms  32
   1.16.2 Flowcharts  37
   1.16.3 Pseudocodes  39
1.17 Types of Errors  41
1.18 Testing and Debugging Approaches  42
Annexure I — Types of Operating Systems  52

2 Introduction to Object Oriented Programming (OOP)  63

2.1 Computer Programming and Programming Languages  63
2.2 Generations of Programming Languages  64
   2.2.1 First Generation: Machine Language  64
   2.2.2 Second Generation: Assembly Language  65
   2.2.3 Third Generation: High-level Language  66
   2.2.4 Fourth Generation: Very High-level Languages  67
   2.2.5 Fifth Generation Programming Language  68
2.3 Programming Paradigms  69
   2.3.1 Monolithic Programming  69
   2.3.2 Procedural Programming  69
   2.3.3 Structured Programming  70
   2.3.4 Object Oriented Programming (OOP)  71
2.4 Features of Object Oriented Programming  72
   2.4.1 Classes  72
   2.4.2 Objects  73
   2.4.3 Method and Message Passing  73
   2.4.4 Inheritance  74
   2.4.5 Polymorphism  75
   2.4.6 Containership  75
   2.4.7 Reusability  75
   2.4.8 Delegation  76
   2.4.9 Data Abstraction and Encapsulation  76
2.5 Merits and Demerits of Object Oriented Programming Language  77
2.6 Applications of Object Oriented Programming  77
2.7 Differences Between Popular Programming Languages  78
Python Programming

3 Basics of Python Programming 83
3.1 Features of Python 83
3.2 History of Python 85
3.3 The Future of Python 87
3.4 Writing and Executing First Python Program 87
3.5 Literal Constants 88
   3.5.1 Numbers 88
   3.5.2 Strings 90
3.6 Variables and Identifiers 94
3.7 Data Types 94
   3.7.1 Assigning or Initializing Values to Variables 94
   3.7.2 Multiple Assignment 96
   3.7.3 Multiple Statements on a Single Line 97
   3.7.4 Boolean 97
3.8 Input Operation 97
3.9 Comments 98
3.10 Reserved Words 98
3.11 Indentation 98
3.12 Operators and Expressions 99
   3.12.1 Arithmetic Operators 99
   3.12.2 Comparison Operators 100
   3.12.3 Assignment and In-place or Shortcut Operators 101
   3.12.4 Unary Operators 102
   3.12.5 Bitwise Operators 102
   3.12.6 Shift Operators 103
   3.12.7 Logical Operators 103
   3.12.8 Membership Operators 104
   3.12.9 Identity Operators 104
   3.12.10 Operators Precedence and Associativity 105
3.13 Expressions in Python 106
3.14 Operations on Strings 106
   3.14.1 Concatenation 107
   3.14.2 Multiplication (or String Repetition) 107
   3.14.3 Slice a String 108
3.15 Other Data Types 109
   3.15.1 Tuples 109
   3.15.2 Lists 109
   3.15.3 Dictionary 110
3.16 Type Conversion 110
Annexure 2 — Installing Python 125

Annexure 3 — Comparison between Python 2.x and Python 3.x
Versions 127
Annexure 4 — Testing and Debugging 130

4 Decision Control Statements 137
4.1 Introduction to Decision Control Statements 137
4.2 Selection/Conditional Branching Statements 138
   4.2.1 if Statement 138
   4.2.2 if-else Statement 139
   4.2.3 Nested if Statements 142
   4.2.4 if-elif-else Statement 142
4.3 Basic Loop Structures/Iterative Statements 147
   4.3.1 while loop 147
   4.3.2 for Loop 155
   4.3.3 Selecting an appropriate loop 156
4.4 Nested Loops 164
4.5 The break Statement 167
4.6 The continue Statement 168
4.7 The pass Statement used with Loops 172
Case Study 1 — Simple Calculator 180
Case Study 2 — Generating a Calendar 183

5 Functions and Modules 185
5.1 Introduction 185
   5.1.1 Need for Functions 186
5.2 Function Definition 187
5.3 Function Call 189
   5.3.1 Function Parameters 189
5.4 Variable Scope and Lifetime 191
   5.4.1 Local and Global Variables 192
   5.4.2 Using the Global Statement 193
   5.4.3 Resolution of Names 195
5.5 The return statement 196
5.6 More on Defining Functions 198
   5.6.1 Required Arguments 198
   5.6.2 Keyword Arguments 198
   5.6.3 Default Arguments 199
## Detailed Contents

### 6 Python Strings Revisited 242

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>242</td>
</tr>
<tr>
<td>6.1 Concatenating, Appending, and Multiplying Strings</td>
<td>243</td>
</tr>
<tr>
<td>6.2 Strings are Immutable</td>
<td>245</td>
</tr>
<tr>
<td>6.3 String Formatting Operator</td>
<td>246</td>
</tr>
<tr>
<td>6.4 Built-in String Methods and Functions</td>
<td>249</td>
</tr>
<tr>
<td>6.5 Slice Operation</td>
<td>254</td>
</tr>
<tr>
<td>6.5.1 Specifying Stride While Slicing Strings</td>
<td>256</td>
</tr>
<tr>
<td>6.6 ord() and chr() Functions</td>
<td>257</td>
</tr>
<tr>
<td>6.7 in and not in operators</td>
<td>257</td>
</tr>
<tr>
<td>6.8 Comparing Strings</td>
<td>258</td>
</tr>
<tr>
<td>6.9 Iterating String</td>
<td>259</td>
</tr>
<tr>
<td>6.10 The String Module</td>
<td>265</td>
</tr>
<tr>
<td>6.11 Regular Expressions</td>
<td>268</td>
</tr>
<tr>
<td>6.11.1 The match() Function</td>
<td>268</td>
</tr>
<tr>
<td>6.11.2 The search() Function</td>
<td>269</td>
</tr>
<tr>
<td>6.11.3 The sub() Function</td>
<td>270</td>
</tr>
</tbody>
</table>

### 7 File Handling 289

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>289</td>
</tr>
<tr>
<td>7.2 File Path</td>
<td>289</td>
</tr>
<tr>
<td>7.3 Types of Files</td>
<td>290</td>
</tr>
<tr>
<td>7.3.1 ASCII Text Files</td>
<td>291</td>
</tr>
<tr>
<td>7.3.2 Binary Files</td>
<td>291</td>
</tr>
<tr>
<td>7.4 Opening and Closing Files</td>
<td>292</td>
</tr>
<tr>
<td>7.4.1 The open() Function</td>
<td>292</td>
</tr>
<tr>
<td>7.4.2 The File Object Attributes</td>
<td>293</td>
</tr>
<tr>
<td>7.4.3 The close() Method</td>
<td>294</td>
</tr>
<tr>
<td>7.5 Reading and Writing Files</td>
<td>295</td>
</tr>
<tr>
<td>7.5.1 write() and writelines()</td>
<td>295</td>
</tr>
<tr>
<td>7.5.2 append() Method</td>
<td>295</td>
</tr>
<tr>
<td>7.5.3 The read() and readline()</td>
<td>296</td>
</tr>
<tr>
<td>7.5.4 Opening Files using with Keyword</td>
<td>298</td>
</tr>
<tr>
<td>7.5.5 Splitting Words</td>
<td>299</td>
</tr>
<tr>
<td>7.5.6 Some Other Useful File Methods</td>
<td>300</td>
</tr>
<tr>
<td>7.6 File Positions</td>
<td>300</td>
</tr>
<tr>
<td>7.7. Renaming and Deleting Files</td>
<td>303</td>
</tr>
<tr>
<td>7.8 Directory Methods</td>
<td>304</td>
</tr>
<tr>
<td>7.8.1 Methods from the os Module</td>
<td>307</td>
</tr>
<tr>
<td>Case Study 5 — Creating a Hash File (or a message digest of a file)</td>
<td>317</td>
</tr>
<tr>
<td>Case Study 6 — Mail Merge Program</td>
<td>319</td>
</tr>
<tr>
<td>Case Study 7 — Finding Resolution of an Image</td>
<td>321</td>
</tr>
</tbody>
</table>

### 8 Data Structures 322

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Sequence</td>
<td>322</td>
</tr>
<tr>
<td>8.2 Lists</td>
<td>322</td>
</tr>
<tr>
<td>8.2.1 Access Values in Lists</td>
<td>323</td>
</tr>
</tbody>
</table>
Python Programming

8.2.2 Updating Values in Lists 323
8.2.3 Nested Lists 325
8.2.4 Cloning Lists 326
8.2.5 Basic List Operations 326
8.2.6 List Methods 327
8.2.7 Using Lists as Stack 330
8.2.8 Using Lists as Queues 331
8.2.9 List Comprehensions 332
8.2.10 Looping in Lists 334

8.3 Functional Programming 335
8.3.1 filter() Function 335
8.3.2 map() Function 336
8.3.3 reduce() Function 337

8.4 Tuples 346
8.4.1 Creating Tuple 346
8.4.2 Utility of Tuples 347
8.4.3 Accessing Values in a Tuple 347
8.4.4 Updating Tuple 348
8.4.5 Deleting Elements in Tuple 348
8.4.6 Basic Tuple Operations 349
8.4.7 Tuple Assignment 349
8.4.8 Tuples for Returning Multiple Values 350
8.4.9 Nested Tuples 351
8.4.10 Checking the Index: index() method 351
8.4.11 Counting the Elements: count() Method 352
8.4.12 List Comprehension and Tuples 352
8.4.13 Variable-length Argument Tuples 353
8.4.14 The zip() Function 354
8.4.16 Advantages of Tupple over List 356

8.5 Sets 359
8.5.1 Creating a Set 359

8.6 Dictionaries 366
8.6.1 Creating a Dictionary 367
8.6.2 Accessing Values 368
8.6.2 Adding and Modifying an Item in a Dictionary 369
8.6.3 Modifying an Entry 369
8.6.4 Deleting Items 370
8.6.5 Sorting Items in a Dictionary 373
8.6.6 Looping over a Dictionary 373
8.6.7 Nested Dictionaries 373
8.6.8 Built-in Dictionary Functions and Methods 374
8.6.9 Difference between a List and a Dictionary 376
8.6.10 String Formatting with Dictionaries 377
8.6.11 When to use which Data Structure? 377
8.6.12 List vs Tuple vs Dictionary vs Set 377

Annexure 6 — Iterator and Generator 392

9 Classes and Objects 400
9.1 Introduction 400
9.2 Classes and Objects 400
9.2.1 Defining Classes 400
9.2.2 Creating Objects 401
9.2.3 Data Abstraction and Hiding through Classes 401
9.3 Class Method and self Argument 402
9.4 The __init__() Method (The Class Constructor) 403
9.5 Class Variables and Object Variables 403
9.6 The __del__() Method 406
9.7 Other Special Methods 407
9.8 Public and Private Data Members 408
9.9 Private Methods 409
9.10 Calling a Class Method from Another Class Method 410
9.11 Built-in Functions to Check, Get, Set, and Delete Class Attributes 412
9.12 Built-in Class Attributes 413
9.13 Garbage Collection (Destroying Objects) 414
9.14 Class Methods 425
9.15 Static Methods 426
Annexure 7 — Getters, Setters, @property, and @deleter 432

10 Inheritance 436
10.1 Introduction 436
10.2 Inheriting Classes in Python 437
10.2.1 Polymorphism and Method Overriding 438

© Oxford University Press. All rights reserved.
10.3 Types of Inheritance 441
  10.3.1 Multiple Inheritance 441
  10.3.2 Multi-level Inheritance 442
  10.3.3 Multi-path Inheritance 443
10.4 Composition or Containment or Complex Objects 444
10.5 Abstract Classes and Interfaces 446
10.6 Metaclass 447

11 Operator Overloading 460
  11.1 Introduction 460
    11.1.1 Concept Of Operator Overloading 460
    11.1.2 Advantage of Operator Overloading 461
  11.2 Implementing Operator Overloading 461
  11.3 Reverse Adding 470
  11.4 Overriding __getitem__() and __setitem__() Methods 471
  11.5 Overriding the in Operator 472
  11.6 Overloading Miscellaneous Functions 473
  11.7 Overriding the __call__() Method 474

12 Error and Exception Handling 480
  12.1 Introduction to Errors and Exceptions 480
    12.1.1 Syntax Errors 480
    12.1.2 Logic Error 481
    12.1.3 Exceptions 481
  12.2 Handling Exceptions 482
  12.3 Multiple Except Blocks 483
  12.4 Multiple Exceptions in a Single Block 484
  12.5 Except Block Without Exception 484
  12.6 The else Clause 485
  12.7 Raising Exceptions 486
  12.8 Instantiating Exceptions 486
  12.9 Handling Exceptions in Invoked Functions 487
  12.10 Built-in and User-defined Exceptions 489
  12.11 The finally Block 491
  12.12 Pre-defined Clean-up Action 493
  12.13 Re-raising Exception 494
  12.14 Assertions in Python 494
  Case Study 8 — Compressing String and Files 505

Appendix A — Multi-threading 508
Appendix B — GUI Programming with tkinter Package 514
Appendix C — Simple Graphics Using Turtle 524
Appendix D — Plotting Graphs in Python 530
Appendix E — CGI/Web Programming Using Python 536
11.1 INTRODUCTION
Till now, we have seen that Python is an interesting and easy language. You can build classes with desired attributes and methods. But just think, if you want to add two Time values, where Time is a user-defined class, then how good it would be if we write $T_3 = T_1 + T_2$, where $T_1$, $T_2$, and $T_3$ are all objects of the class Time. As of now, we need to write the same statement as $T_3 = T_1.add(T_2)$.

Basically, the meaning of operators like $+, =, \ast, /, >, <$ etc. are pre-defined in any programming language. Programmers can use them directly on built-in data types to write their programs. But, for user-defined types like objects, these operators do not work. Therefore, Python allows programmers to redefine the meaning of operators when they operate on class objects. This feature is called operator overloading. **Operator overloading** allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can be also applied to user-defined data types.

You already have a clue of operator overloading. Just give a thought, if you write $5 + 2$, then the integers are added, when you write `str1 + str2`, two strings are concatenated, when you write `List1 + List2`, the two lists are merged, so on and so forth. Thus, we see that the same operator behaves differently with different types.

11.1.1 Concept Of Operator Overloading
With operator overloading, a programmer is allowed to provide his own definition for an operator to a class by overloading the built-in operator. This enables the programmer to perform some specific computation when the operator is applied on class objects and to apply a standard definition when the same operator is applied on a built-in data type.

This means that while evaluating an expression with operators, Python looks at the operands around the operator. If the operands are of built-in types, Python calls a built-in routine. In case, the operator is being applied on user-defined operand(s), the Python compiler checks to see if the programmer has an overloaded operator function that it can call. If such a function whose parameters match the type(s) and number of the operands exists in the program, the function is called, otherwise a compiler error is generated.
Another form of Polymorphism
Like function overloading, operator overloading is also a form of compile-time polymorphism. Operator overloading, is therefore less commonly known as operator *ad hoc polymorphism* since different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

**Note** Ad hoc polymorphism is a specific case of polymorphism where different operators have different implementations depending on their arguments.

### 11.1.2 Advantage of Operator Overloading
We can easily write our Python programs without the knowledge of operator overloading, but the knowledge and use of this feature can help us in many ways. Some of them are:

- With operator overloading, programmers can use the same notations for user-defined objects and built-in objects. For example, to add two complex numbers, we can simply write \( C_1 + C_2 \).
- With operator overloading, a similar level of syntactic support is provided to user-defined types as provided to the built-in types.
- In scientific computing where computational representation of mathematical objects is required, operator overloading provides great ease to understand the concept.
- Operator overloading makes the program clearer. For example, the statement

\[
(C_1 \cdot \text{mul}(C_2) \div \text{div}(C_1 \cdot \text{add}(C_2))
\]

can be better written as

\[
C_1 \cdot C_2 \div \text{div}(C_1 + C_2)
\]

### 11.2 IMPLEMENTING OPERATOR OVERLOADING
Just consider the code given below which is trying to add two complex numbers and observe the result.

**Example 11.1** Program to add two complex numbers without overloading the + operator

```python
class Complex:
    def __init__(self):
        self.real = 0
        self.imag = 0
    def setValue(self, real, imag):
        self.real = real
        self.imag = imag
    def display(self):
        print("(", self.real, "+", self.imag, ")")
C1 = Complex()
C1.setValue(1,2)
C2 = Complex()
C2.setValue(3,4)
C3 = Complex()
C3 = C1 + C2
C3.display()
```

**OUTPUT**
Traceback (most recent call last):
So, the reason for this error is simple. + operator does not work on user-defined objects. Now, to do the same concept, we will add an operator overloading function in our code. For example, look at the code given below which has the overloaded add function specified as __add__().

Example 11.2  Program to overload the + operator on a complex object

```python
class Complex:
    def __init__(self):
        self.real = 0
        self.imag = 0
    def setValue(self, real, imag):
        self.real = real
        self.imag = imag
    def __add__(self, C):
        Temp = Complex()
        Temp.real = self.real + C.real
        Temp.imag = self.imag + C.imag
        return Temp
    def display(self):
        print("(", self.real, " + ", self.imag, ")")
C1 = Complex()
C1.setValue(1,2)
C2 = Complex()
C2.setValue(3,4)
C3 = Complex()
C3 = C1 + C2
Print("RESULT = ")
C3.display()

OUTPUT
RESULT =  ( 4 + 6 i)
```

In the program, when we write C1 + C2, the __add__() function is called on C1 and C2 is passed as an argument. Remember that, user-defined classes have no + operator defined by default. The only exception is when you inherit from an existing class that already has the + operator defined.

Note  The __add__() method returns the new combined object to the caller.

We can also overload the comparison operators to work with class objects. But before we write further programs, let us first have a look at Table 11.1 to know the name of the function for each operator.
Table 11.1 Operators and their corresponding function names

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function Name</th>
<th>Operator</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td><strong>add</strong></td>
<td>+=</td>
<td><strong>iadd</strong></td>
</tr>
<tr>
<td>-</td>
<td><strong>sub</strong></td>
<td>-=</td>
<td><strong>isub</strong></td>
</tr>
<tr>
<td>*</td>
<td><strong>mul</strong></td>
<td>*=</td>
<td><strong>imul</strong></td>
</tr>
<tr>
<td>/</td>
<td><strong>truediv</strong></td>
<td>/=</td>
<td><strong>idiv</strong></td>
</tr>
<tr>
<td>**</td>
<td><strong>pow</strong></td>
<td>**=</td>
<td><strong>ipow</strong></td>
</tr>
<tr>
<td>%</td>
<td><strong>mod</strong></td>
<td>%=</td>
<td><strong>imod</strong></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td><strong>rshift</strong></td>
<td>&gt;&gt;=</td>
<td><strong>irshift</strong></td>
</tr>
<tr>
<td>&amp;</td>
<td><strong>and</strong></td>
<td>&amp;=</td>
<td><strong>iand</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>or</strong></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td><strong>xor</strong></td>
<td>^=</td>
<td><strong>ixor</strong></td>
</tr>
<tr>
<td>~</td>
<td><strong>invert</strong></td>
<td>~=</td>
<td><strong>iinvert</strong></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td><strong>lshift</strong></td>
<td>&lt;&lt;=</td>
<td><strong>ilshift</strong></td>
</tr>
<tr>
<td>&gt;</td>
<td><strong>gt</strong></td>
<td>&gt;=</td>
<td><strong>ge</strong></td>
</tr>
<tr>
<td>&lt;</td>
<td><strong>lt</strong></td>
<td>==</td>
<td><strong>eq</strong></td>
</tr>
<tr>
<td>&gt;=</td>
<td><strong>ge</strong></td>
<td>!=</td>
<td><strong>ne</strong></td>
</tr>
</tbody>
</table>

The program given below compares two Book objects. Although the class Book has three attributes, comparison is done based on its price. However, this is not mandatory. You can compare two objects based on any of the attributes.

Example 11.3 Program to compare two objects of user-defined class type

class Book:
    def __init__(self):
        title = ""
        publisher = ""
        price = 0
    def set(self, title, publisher, price):
        self.title = title
        self.publisher = publisher
        self.price = price
    def display(self):
        print("TITLE: ", self.title)
        print("PUBLISHER: ", self.publisher)
        print("PRICE: ", self.price)
    def __gt__(self, B):
        if self.price > B.price:
            return True
        else:
            return False
B1 = Book()
B1.set("OOP with C++", "Oxford University Press", 525)
B2 = Book()
B2.set("Let us C++", "BPB", 300)
if B1>B2:
    print("This book has more knowledge so I will buy")
    B1.display()

OUTPUT
This book has more knowledge so I will buy
TITLE :  OOP with C++ PUBLISHER :  Oxford University Press PRICE :  525

PROGRAMMING EXAMPLES

Program 11.1 Write a program that overloads the + operator on a class Student that has attributes name and marks.

class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
    def display(self):
        print(self.name, self.marks)
    def __add__(self, S):
        Temp = Student(S.name, [])
        for i in range(len(self.marks)):
            Temp.marks.append(self.marks[i] + S.marks[i])
        return Temp
S1 = Student("Nikhil", [87, 90, 85])
S2 = Student("Nikhil", [83, 86, 88])
S1.display()
S2.display()
S3 = Student("",[])
S3 = S1 + S2
S3.display()

OUTPUT
Nikhil [87, 90, 85]
Nikhil [83, 86, 88]
Nikhil [170, 176, 173]

Program 11.2 Write a program that overloads the + operator to add two objects of class Matrix.

class Matrix:
    def __init__(self, List):
Operator Overloading

```python
self.List = List
def display(self):
    print(self.List)
def __add__(self, M):
    Temp = Matrix([])
    for i in range(len(self.List)):
        for j in range(len(self.List[0])):
            Temp.List.append(self.List[i][j] + M.List[i][j])
    return Temp
M1 = Matrix([[1,2],[3,4]])
M2 = Matrix([[3,4],[5,1]])
M3 = Matrix([])
M3 = M1 + M2
print("RESULTANT MATRIX IS : ")
M3.display()
```

**OUTPUT**

RESULTANT MATRIX IS : [4, 6, 8, 5]

**Program 11.3** Write a program that overloads the + operator so that it can add two objects of class Fraction.

def GCD(num, deno):
    if(deno == 0):
        return num
    else:
        return GCD(deno, num%deno)
class Fraction:
    def __init__(self):
        self.num = 0
        self.deno = 1
    def get(self):
        self.num = int(input("Enter the numerator : "))
        self.deno = int(input("Enter the denominator : "))
    def simplify(self):
        common_divisor = GCD(self.num, self.deno)
        self.num //= common_divisor
        self.deno //= common_divisor
    def __add__(self, F):
        Temp = Fraction()
        Temp.num = (self.num * F.deno) + (F.num * self.deno)
        Temp.deno = self.deno * F.deno
        return Temp
    def display(self):
        self.simplify()
        print(self.num, "/", self.deno)
F1 = Fraction()
F1.get()
F2 = Fraction()
F2.get()
F3 = Fraction()
F3 = F1 + F2
print("RESULTANT FRACTION IS : ")
F3.display()

OUTPUT
Enter the numerator : 4
Enter the denominator : 10
Enter the numerator : 2
Enter the denominator : 5
RESULTANT FRACTION IS :  4 / 5

Program 11.4 Write a program that overloads the + operator so that it can add a specified number of days to a given date.

def chk_Leap_Year(year):
    if (year%4 == 0 and year%100 != 0) or (year%400 == 0):
        return 1
    else:
        return 0
class Date:
    def __init__(self):
        d = m = y = 0
    def get(self):
        self.d = int(input("Enter the day : "))
        self.m = int(input("Enter the month : "))
        self.y = int(input("Enter the year : "))
    def __add__(self, num):
        self.d += num
        if self.m !=2:
            max_days = Dict[self.m]
        elif self.m == 2:
            isLeap = chk_Leap_Year(self.y)
            if isLeap == 1:
                max_days = 29
            else:
                max_days = 28
        while self.d > max_days:
            self.d -= max_days
            self.m += 1
        while self.m > 12:
            self.m -= 12
            self.y += 1
def display(self):
    print(self.d, "-", self.m,"-", self.y)
D = Date()
D.get()
num = int(input("How many days to add : "))
D + num
D.display()

OUTPUT
Enter the day : 25
Enter the month : 2
Enter the year : 2016
How many days to add : 10
6 - 3 - 2016

Program 11.5 Write a program that has an overloads the *, /, and > operators so that it can multiply, divide, and compare two objects of class Fraction.

def chk_Leap_Year(year):
    if (year%4 == 0 and year%100 != 0) or (year%400 == 0):
        return 1
    else:
        return 0
class Date:
    def __init__(self):
        d = m = y = 0
    def get(self):
        self.d = int(input("Enter the day : "))
        self.m = int(input("Enter the month : "))
        self.y = int(input("Enter the year : "))
    def __add__(self, num):
        self.d += num
        if self.m !=2:
            max_days = Dict[self.m]
        elif self.m == 2:
            isLeap = chk_Leap_Year(self.y)
            if isLeap == 1:
                max_days = 29
            else:
                max_days = 28
        while self.d > max_days:
            self.d -= max_days
            self.m += 1
        while self.m > 12:
            self.m -= 12
            self.y += 1
def display(self):
    print(self.d, "-", self.m,"-", self.y)
D = Date()
D.get()
num = int(input("How many days to add : "))
D + num
D.display()

OUTPUT
Enter the numerator : 2
Enter the denominator : 3
Enter the numerator : 4
Enter the denominator : 9
F1 > F2 True
F1 * F2 IS :  8 / 27
F1 / F2 IS :  3 / 2

Program 11.6 Write a program that overloads the + operator so that it can add two objects of class Binary.

class Binary:
    number = []
    def set(self, bnum):
        self.number = bnum
    def display(self):
        print(self. Number)
    def __add__(self, B):
        Temp = Binary()
        index = len(self.number)
        carry = []
        while len(Temp.number) != index:
            Temp.number.append(-1)
            carry.append(0)
        index -= 1
        while (index)>=0:
            if self.number[index] == 0 and B.number[index] == 0:
                Temp.number[index] = 0 + int(carry[index])
            if self.number[index] == 0 and B.number[index] == 1:
                Temp.number[index] = 1 + int(carry[index])
            if self.number[index] == 1 and B.number[index] == 0:
                Temp.number[index] = 1 + int(carry[index])
            if self.number[index] == 1 and B.number[index] == 1:
                Temp.number[index] = 0 + int(carry[index])
                carry[index-1] = 1
            if Temp.number[index] == 2:
                Temp.number[index] = 0
            if (index-1)>=0:
carry[index-1] = 1

index -= 1
return Temp

B1 = Binary()
B1.set([1,1,0,1,1])
B2 = Binary()
B2.set([0,1,1,0,1])
B3 = B1 + B2
B3.display()

OUTPUT
[0, 1, 0, 0, 0]

Program 11.7 Write a program to compare two Date objects.

class Date:
    def __init__(self):
        d = m = y = 0
    def get(self):
        self.d = int(input("Enter the day : "))
        self.m = int(input("Enter the month : "))
        self.y = int(input("Enter the year : "))
    def __eq__(self, D):
        Flag = False
        if self.d == D.d:
            if self.m == D.m:
                if self.y == D.y:
                    Flag = True
        return Flag
    def __lt__(self, D):
        Flag = False
        if self.y < D.y:
            if self.m < D.m:
                if self.d < D.d:
                    Flag = True
        return Flag

D1 = Date()
D1.get()
D2 = Date()
D2.get()
print("D1 == D2", D1 == D2)
print("D1 < D2", D1 < D2)

OUTPUT
Enter the day : 21
Enter the month : 3

Programming Tip: The __eq__ function gives NotImplemented as result when left hand argument does not know how to test for equality with given right hand argument.
Enter the year: 2017
Enter the day: 21
Enter the month: 3
Enter the year: 2017
D1 == D2 True
D1 < D2 False

**Program 11.8** Write a program to overload the -= operator to subtract two Distance objects.

class Distance:
    def __init__(self):
        self.km = 0
        self.m = 0
    def set(km, m):
        self.km = km
        self.m = m
    def __isub__(self, D):
        self.m = self.m - D.m
        if self.m < 0:
            self.m += 1000
            self.km -= 1
        self.km = self.km - D.km
        return self
    def convert_to_meters(self):
        return (self.km*1000 + self.m)
    def display(self):
        print(self.km, "kms", self.m, "mtrs")
D1 = Distance()
D1.set(21, 70)
D2 = Distance()
D2.set(18, 123)
D1 -= D2
print("D1 - D2 = ")
D1.display()
print("that is", D1.convert_to_meters(), "meters")

**OUTPUT**
D1 - D2 = 2 kms 947 mtrs that is 2947 meters

11.3 **REVERSE ADDING**

In a program, we have added a certain number of days to our Date object by writing d + num. In this case, it is compulsory that the class object will invoke the __add__(). But, to provide greater flexibility, we should also be able to perform the addition in reverse order, that is, adding a non-class object to the class object. For this, Python provides the concept of reverse adding. The function to do normal addition on Date object is discussed in the following example.
Operator Overloading

But, had we written the same statement as `num + d`, then the desired task would not have been performed. The simple reason for this is that the `__add__()` takes `self` as the first argument, so the `+` operator has to be invoked using the `Date` object. But this is not the case when you work with numbers. You can either write `10 + 20` or `20 + 10`, it means the same and the correct result is produced. So, we should also have the same result when we write `d + num` or `num + d`. Python has a solution to this. It has the feature of reverse adding. As you write the `__add__()` function, just write the `__radd__()` function which will do the same task.

**Note** To overload the `+` or `-=` operators, use the `__iadd__()` or `__isub__()` functions.

11.4 **OVERLOADING __getitem__() AND __setitem__() METHODS**

Python allows you to override `__getitem__()` and `__setitem__()` methods. We have already seen in Chapter 9 that `__getitem__()` is used to retrieve an item at a particular index. Similarly, `__setitem__()` is used to set value for a particular item at the specified index. Although they are well defined for built-in types like list, tuple, string, etc. but for user-defined classes we need to explicitly write their codes. Consider the program given below which has a list defined in a class. By default, Python does not allow you to apply indexes on class objects but if you have defined the `__getitem__()` and `__setitem__()` methods in the class, then you can simply work with indices as with any other built-in type as shown in the following example.

**Example 11.4** Program to illustrate adding on `Date` object

```python
def __add__(self, num):
    self.d += num
    if self.m != 2:
        max_days = Dict[self.m]
    elif self.m == 2:
        isLeap = chkLeapYear(self.y)
        if isLeap == 1:
            max_days = 29
        else:
            max_days = 28
    while self.d > max_days:
        self.d -= max_days
        self.m += 1
    while self.m > 12:
        self.m -= 12
        self.y += 1
```

**Example 11.5** Program that overrides `__getitem__()` and `__setitem__()` methods in a class

```python
class myList:
    def __init__(self, List):
        self.List = List
    def __getitem__(self, index):
        return self.List[index]
```

© Oxford University Press. All rights reserved.
11.5 **OVERRIDING THE in OPERATOR**

We have seen that in is a membership operator that checks whether the specified item is in the variable of built-in type or not (like string, list, dictionary, tuple, etc.). We can overload the same operator to check whether the given value is a member of a class variable or not. To overload the in operator we have to use the function __contains__(). In the program given in the following example, we have created a dictionary that has name of the subjects as key and their maximum weightage as value. In the main module, we are asking the user to input a subject. If the subject is specified in our dictionary, then its maximum weightage is displayed.

### Example 11.6  Program to override the in operator

```python
class Marks:
    def __init__(self):
        self.max_marks = {"Maths":100, "Computers":50, "SST":100, "Science":75}
    def __contains__(self, sub):
        if sub in self.max_marks:
            return True
        else:
            return False
    def __getitem__(self, sub):
```

© Oxford University Press. All rights reserved.
11.6 OVERLOADING MISCELLANEOUS FUNCTIONS

Python allows you to overload functions like `long()`, `float()`, `abs()`, and `hex()`. Remember that we have used these functions on built-in type variables to convert them from one type to another. We can use these functions to convert a value of one user-defined type (object) to a value of another type.

Example 11.7  Program to overload `hex()`, `oct()`, and `float()` functions

class Number:
    def __init__(self, num):
        self.num = num
    def display(self):
        return self.num
    def __abs__(self):
        return abs(self.num)
    def __float__(self):
        return float(self.num)
    def __oct__(self):
        return oct(self.num)
    def __hex__(self):
        return hex(self.num)
    def __setitem__(self, num):
        self.num = num

M = Number(-14)
print(str(M))
sub = input("Enter the subject for which you want to know extra marks : ")
if sub in M:
    print("Social Studies paper has maximum marks allotted = ", M[sub])

OUTPUT

The Dictionary has name of subjects and maximum marks allotted to them
Enter the subject for which you want to know extra marks : Computers
Social Studies paper has maximum marks allotted =  50
Let us take another example in which we have two classes for calculating the distance. One has distance specified in meters and the other has distance in kilometers. There are two functions–km() and mts(), which takes the argument of class Distance and then converts the distance into kilometers and meters respectively.

```python
class Distance_m:
    def __init__(self, m):
        self.m = m
    def display(self):
        print("Distance in meters is : ", self.m)
    def mts(D):
        return D.km*1000

class Distance_km:
    def __init__(self, km):
        self.km = km
    def display(self):
        print("Distance in kilometers is : ", self.km)
    def km(D):
        return D.m/1000

Dm = Distance_m(12345)
Dm.display()
print("Distance in kilo metres = ", km(Dm))
Dkm = Distance_km(12.345)
Dkm.display()
print("Distance in metres = ", mts(Dkm))
```

**OUTPUT**
N IS :  -14
ABS(N) IS :  14
Converting to float....., N IS :  14.0
Hexadecimal equivalent of N IS :  0xe
Octal equivalent of N IS :  016

11.7 **OVERRIDING THE __call__() METHOD**
The __call__() method is used to overload call expressions. The __call__() method is called automatically when an instance of the class is called. It can be passed to any positional or keyword arguments. Like other functions, the __call__() method also supports all of the argument-passing modes. The __call__() method can be declared as, def __call__(self, [args...])
Example 11.9  Program to overload the __call__() method

```python
class Mult:
    def __init__(self, num):
        self.num = num
    def __call__(self, O):
        return self.num * O
x = Mult(10)
print(x(5))
```

OUTPUT
```
50
```

Summary

- The meaning of operators like +, =, *, /, >, <, etc. are pre-defined in any programming language. So, programmers can use them directly on built-in data types to write their programs.
- Operator overloading allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can also be applied to user-defined data types.
- With operator overloading, a programmer is allowed to provide his own definition for an operator to a class by overloading the built-in operator.
- Operator overloading is also known as operator ad hoc polymorphism since different operators have different implementations depending on their arguments.
- The __add__() method returns the new combined object to the caller.
- By default, Python does not allow you to apply indexes on class objects but if you have defined the __getitem__() and __setitem__() in the class, then you can simply work with indices as with any other built-in type.

Glossary

Ad hoc polymorphism  A specific case of polymorphism where different operators have different implementations depending on their arguments.

Membership operator  An operator that checks whether the specified item is present in the instance of an object or not.

Operator Overloading  Redefining the meaning of operators when they operate on class objects.

Exercises

Fill In The Blanks

1. _____ allows programmers to redefine the meaning of existing operators.
2. Operator overloading is also known as _____ polymorphism.
3. _____ is a specific case of polymorphism where different operators have different implementations depending on their arguments.
4. The name of the function to overload ** operator is _____.
5. The __add__() method returns _____.
6. To overload the * operator you will use _____ function.
7. The \_\_eq\_ function gives _____ as result when left hand argument does not know how to test for equality with given right hand argument.
8. The _____ method is used to overload call expressions.

State True Or False

1. You can have overload only one operator per class.
2. Operator overloading allows you to create new operators.
3. With operator overloading, a programmer is allowed to provide his own definition for an operator to a class.
4. Operator overloading makes the program simple to understand.
5. To overload the \<\=\> operator, you will write the code for \_\_lshift\_ function.

Multiple Choice Questions

1. Which function will be written to overload the \+ operator?
   (a) \_\_call\_
   (b) \_\_contains\_
   (c) \_\_member\_
   (d) \_\_add\_
2. Which of the following function will help you to retrieve an item at a particular index?
   (a) slice
   (b) \_\_getitem\_
   (c) \_\_setitem\_
   (d) in
3. Which of the following function is used to set value for a particular item at the specified index?
   (a) slice
   (b) \_\_getitem\_
   (c) \_\_setitem\_
   (d) in
4. Membership operator when overloaded is invoked on _____.
   (a) object
   (b) class
   (c) method
   (d) attribute
5. Which conversion function cannot be overloaded in a class?
   (a) long()
   (b) hex()
   (c) str()
   (d) None of these

Review Questions

1. Define the term operator overloading.
2. Assume that you have overloaded the + operator in your program. Illustrate the cases in which the operator overloaded function will be called and the cases in which the default function will be called.
3. Define the term ad hoc polymorphism.
4. Give the advantages of operator overloading.
5. Differentiate between \_\_add\_, \_\_radd\_, and \_\_iadd\_ functions.
6. Which functions will you use to index a class object? Explain with the help of an example.
7. Which operator is used to check whether a value is present in the object or not? Can you overload this object on user-defined types? If yes, how?
8. Is it possible to convert a class object into a floating type value? If yes, how?
9. With the help of an example explain how you can convert a value of one class type into a value of another class type.

Programming Problems
1. Write a class Money with attributes Rupees and Paise. Overload operators +=, -=, and >= so that they may be used on two objects. Also write functions so that a desired amount of money can be either added or subtracted from Money.
2. Use the class defined in the previous functions to calculate the amount of money to be paid by multiplying it with a specified quantity.
3. Again, using class Money, find the price of one item given the total amount paid and number of units of item bought.
4. Write a class INR with attributes Rupees and Paise. Write another class USD with attributes dollars and cents. Write a function to convert USD into INR and vice versa.
5. Write a program that overloads the + operator to add two objects of class Time.
6. Write a menu driven program to overload +=, -=, and *= operators on the Matrix class.
7. Write a menu driven program to overload +=, -=, ==, >=, and <= operators on the Distance class.
8. Write a menu driven program to overload +=, -=, ==, >=, and <= operators on the Time class.
9. Write a menu driven program to overload +=, -=, ==, >=, and <= operators on the Height class.
10. Write a menu driven program to overload +=, -=, ==, >=, and <= operators on the Polynomial class.
11. Write a menu driven program to overload +=, -=, ==, >=, and <= operators on the Fraction class.
12. Write a menu driven program to overload +=, -=, *=, /=, ==, and >=, and <= operators on the String class.
13. Write a menu driven program to overload +=, -=, ==, >=, and <= operators on the Array class.
14. Write a menu driven program to overload +=, -=, ==, >=, and <= operators on the Polynomial class.
15. Write a program to convert minutes into class Time with data members—hrs and mins.
16. Write a program to convert class Time with data members—hrs and mins into minutes.
17. Write a program that performs conversion to and from Array class.
18. Write a program that performs conversion to and from String class.
19. Write a program that performs conversion from a Square to Rectangle class.
20. Write a program to convert data of class Student having members—roll no and marks in three subjects to another class Student that stores just the roll number and the average.
21. Write a program to convert Polar co-ordinates specified in one class into Rectangular co-ordinates.
22. Write a program to convert temperature specified in Celsius in one class into Fahrenheit in another class.
23. Write a program to implement a timer using increment operator overloading.

Find the Output
1. class Point:
   def __init__(self, x, y):
       self.x = x
       self.y = y
   def __abs__(self):
       return (self.x**2 + self.y**2)**0.5
   def __add__(self, P):
       return Point(self.x + P.x, self.y + P.y)
   def display(self):
       print(self.x, self.y)

P1 = Point(12, 25)
P2 = Point(21, 45)
print(abs(P2))
P1 = P1+ P2
P1.display()

2. class A(object):
   def __init__(self, num):
       self.num = num
   def __eq__(self, other):
       return self.num == other.num
   def __call__(self):
       return self.num

P1 = Point(12, 25)
P2 = Point(21, 45)
print(abs(P2))
P1 = P1+ P2
P1.display()
class B(object):
    def __init__(self, num):
        self.num = num
    print(A(5) == B(5))
3. class Circle:
    def __init__(self, radius):
        self.__radius = radius
    def getRadius(self):
        return self.__radius
    def area(self):
        return 3.14 * self.__radius ** 2
    def __add__(self, C):
        return Circle(self.__radius + C.__radius)
C1 = Circle(5)
C2 = Circle(9)
C3 = C1 + C2
print("RADIUS : ", C3.getRadius())
print("AREA : ", C3.area())
4. class Circle:
    def __init__(self, radius):
        self.__radius = radius
    def __gt__(self, another_circle):
        return self.__radius > another_circle.__radius
    def __lt__(self, C):
        return self.__radius < C.__radius
    def __str__(self):
        return "Circle has radius " + str(self.__radius)
C1 = Circle(5)
C2 = Circle(9)
print(C1)
print(C2)
print("C1 < C2 : ", C1 < C2)
print("C2 > C1 : ", C1 > C2)
5. class One:
    def __init__(self):
        num = 10
    def __eq__(self, T):
        if isinstance(T, One):
            return True
        else:
            return NotImplemented
class Two:
    def __init__(self):
        num = 100
    print(One() == Two())
6. class A:
    def __bool__(self):
        return True
X = A()
if X:
    print('yes')
7. class String(object):
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        return self.val + '....' + other.val
    def __str__(self):
        return "This is String representation of " + self.val
S1 = String("Hello")
S2 = String("World")
print(S1 + S2)
print(S1 - S2)
8. class String(object):
    def __init__(self, val):
        self.val = val
    def __str__(self):
        return "This is String representation of " + self.val
    def __repr__(self):
        return repr(self) + "This is String representation of " + self.val
S = String("Hi")
print(str(S))
9. class A:
    def __len__(self):
        return 0
X = A()
if not X:
    print('no')
else:
    print('yes')
10. class A:
    def __init__(self):
        self.str = "abcdef"
    def __getitem__(self, i):
        return self.str[i]
x = A()
for i in x:
    print(i, end=" ")
11. class A:
    str = "Hi"
    def __gt__(self, str):
        return self.str > str
X = A()
print(X > 'hi')
Find the Error

1. class Matrix:
   def __init__(self):
       Mat = []
   def setValue(self, number):
       self.number = number
   def display(self):
       print(self.number)
M1 = Matrix()
M1.setValue(((1,2),(3,4)))
M2 = Matrix()
M2.setValue(((5,6),(2,3)))
M3 = Matrix()
M3 = M1 + M2
M3.display()

2. class A(object):
   def __init__(self, num):
       self.num = num
   def __eq__(self, other):
       return self.num == other.num
class B(object):
   def __init__(self, val):
       self.val = val
print(A(5) == B(5))

3. class Point:
   def __init__(self, x, y):
       self.x = x
       self.y = y
   def __mul__(self, num):
       return self.x * num + self.y * num
P1 = Point(3, 4)
print(2*P1)

4. class String(object):
   def __init__(self, val):
       self.val = val
S1 = String("Hello")
print(S1[5])

5. class Number:
   def __init__(self, num):
       self.num = num
   def __sub__(self, N):
       return Number(self.num - N)
   def __sub__(N, self):
       return Number(N - self.num)
x = Number(4)
y = x - 4

6. class A:
   def __init__(self):
       self.str = "abcdef"
   def __setitem__(self, i, val):
       self.str[i] = val
x = A()
x[2] = 'X'

Answers

Fill in the Blanks

1. operator overloading 4. __pow__ 6. __mul__ 9. argument-passing
2. compile time or ad hoc 5. the new combined 7. NotImplemented 10. __rsub__()
3. ad hoc polymorphism 8. __call__()

State True or False


Multiple Choice Questions

1. (b) 2. (b) 3. (c) 4. (a) 5. (d) 6. (c) 7. (c) 8. (a)