

Python Programming

A MODULAR APPROACH

With Graphics, Database, Mobile, and Web Applications

Based on Python 3.x



Sheetal Taneja
Naveen Kumar

About Pearson

Pearson is the world's learning company, with presence across 70 countries worldwide. Our unique insights and world-class expertise comes from a long history of working closely with renowned teachers, authors and thought leaders, as a result of which, we have emerged as the preferred choice for millions of teachers and learners across the world.

We believe learning opens up opportunities, creates fulfilling careers and hence better lives. We hence collaborate with the best of minds to deliver you classleading products, spread across the Higher Education and K12 spectrum.

Superior learning experience and improved outcomes are at the heart of everything we do. This product is the result of one such effort.

Your feedback plays a critical role in the evolution of our products and you can contact us - reachus@pearson.com. We look forward to it.

PYTHON PROGRAMMING: A MODULAR APPROACH

*A Modular Approach With Graphics, Database, Mobile,
and Web Applications*

Sheetal Taneja

*Department of Computer Science
Dyal Singh College
University of Delhi
Delhi, India
and*

Naveen Kumar

*Department of Computer Science
University of Delhi
Delhi, India*



Contents

Foreword

Preface

About the Authors

1. Python Programming: An Introduction

1.1 IDLE – An Interpreter for Python

1.2 Python Strings

1.3 Relational Operators

1.4 Logical Operators

1.5 Bitwise Operators

1.6 Variables and Assignment Statements

Assignment Statement

Shorthand Notation

1.7 Keywords

1.8 Script Mode

Summary

Exercises

2. Functions

2.1 Built-in Functions

input Function

eval Function

Composition

print Function

type Function

round Function

Type Conversion

min and max Functions

pow Function

Random Number Generation

Functions from math Module

Complete List of Built-in Functions

2.2 Function Definition and Call

Fruitful Functions vs Void Functions

Function help

Default Parameter Values

Keyword Arguments

2.3 Importing User-defined Module

2.4 Assert Statement

2.5 Command Line Arguments

Summary

Exercises

3. Control Structures

3.1 if Conditional Statement

General Form of if Conditional Statement

Conditional Expression

General Form of if-else Conditional Statement

General Form of if-elif-else Conditional Statement

Nested if-elif-else Conditional Statement

3.2 Iteration (for and while Statements)

3.2.1 for Loop

General Format of for Statement

3.2.2 while Loop

General Format of while Statement

Infinite Loops

3.2.3 while Statement vs. for Statement

3.2.4 Example: To Print Some Pictures

3.2.5 Nested Loops

3.2.6 break, continue, and pass Statements

3.2.7 Example: To Compute sin(x)

3.2.8 else Statement

Summary

Exercises

4. Debugging

4.1 Testing

An Example: Finding Maximum of Three Numbers

4.2 Debugging

Summary

Exercises

5. Scope

5.1 Objects and Object ids

5.2 Scope of Objects and Names

5.2.1 Namespaces

5.2.2 Scope

LEGB Rule

Summary

Exercises

6. Strings

6.1 Strings

6.1.1 Slicing

6.1.2 Membership

6.1.3 Built-in Functions on Strings

Function count

Functions find and rfind

Functions capitalize, title, lower, upper, and swapcase

Functions islower, isupper, and istitle

Function replace

Functions strip, lstrip, and rstrip

Functions split and partition

Function join

Functions isspace, isalpha, isdigit, and isalnum

Function startswith and endswith

Functions encode and decode

List of Functions

6.2 String Processing Examples

6.2.1 Counting the Number of Matching Characters in a Pair of Strings

6.2.2 Counting the Number of Common Characters in a Pair of Strings

6.2.3 Reversing a String

6.3 Pattern Matching

6.3.1 Some Important Definitions

Summary

Exercises

7. Mutable and Immutable Objects

7.1 Lists

7.1.1 Summary of List Operations

7.1.2 Function list

7.1.3 Functions append, extend, count, remove, index, pop, and insert

7.1.4 Function insert

7.1.5 Function reverse

7.1.6 Functions sort and reverse

7.1.7 List Functions

7.1.8 List Comprehension

7.1.9 Lists as Arguments

7.1.10 Copying List Objects

7.1.11 map, reduce, and filter Operations on a Sequence

7.2 Sets

7.2.1 Set Functions add, update, remove, pop, and clear

7.2.2 Set Functions union, intersection, difference, and symmetric_difference

7.2.3 Function copy

7.2.4 Subset and Superset Test

7.2.5 List of Functions

7.2.6 Finding Common Factors

7.2.7 Union and Intersection Operation on Lists

7.3 Tuples

7.3.1 Summary of Tuple Operations

7.3.2 Functions tuple and zip

7.3.3 Functions count and index

7.4 Dictionary

7.4.1 Dictionary Operations

7.4.2 Deletion

7.4.3 Function get

7.4.4 Function update

7.4.5 Function copy

7.4.6 List of Functions

7.4.7 Inverted Dictionary

Summary

Exercises

8. Recursion

8.1 Recursive Solutions for Problems on Numeric Data

8.1.1 Factorial

Iterative Approach

Recursive Approach

8.1.2 Fibonacci Sequence

Iterative Approach

Recursive Approach

8.2 Recursive Solutions for Problems on Strings

8.2.1 Length of a String

8.2.2 Reversing a String

8.2.3 Palindrome

8.3 Recursive Solutions for Problems on Lists

8.3.1 Flatten a List

8.3.2 Copy

8.3.3 Deep Copy

8.3.4 Permutation

8.4 Problem of Tower of Hanoi

Summary

Exercises

9. Files and Exceptions

9.1 File Handling

9.2 Writing Structures to a File

9.3 Errors and Exceptions

9.4 Handling Exceptions Using try...except

9.5 File Processing Example

Summary

Exercises

10. Classes I

10.1 Classes and Objects

10.2 Person: An Example of Class

10.2.1 Destructor

10.3 Class as Abstract Data Type

10.4 Date Class

Summary

Exercises

11. Classes II

11.1 Polymorphism

11.1.1 Operator Overloading

Comparing Dates

11.1.2 Function Overloading

11.2 Encapsulation, Data Hiding, and Data Abstraction

11.3 Modifier and Accessor Methods

11.4 Static Method

11.5 Adding Methods Dynamically

11.6 Composition

11.7 Inheritance

11.7.1 Single Inheritance

Scope Rule

Extending Scope of int Class Using a User Defined Class

11.7.2 Hierarchical Inheritance

11.7.3 Multiple Inheritance

11.7.4 Abstract Methods

11.7.5 Attribute Resolution Order for Inheritance

11.8 Built-in Functions for Classes

Summary

Exercises

12. List Manipulation

12.1 Sorting

12.1.1 Selection Sort

12.1.2 Bubble Sort

12.1.3 Insertion Sort

12.2 Searching

12.2.1 Linear Search

12.2.2 Binary Search

12.3 A Case Study

12.3.1 Operations for Class Section

Complete Script

12.4 More on Sorting

12.4.1 Merge Sort

12.4.2 Quick Sort

Summary

Exercises

13. Data Structures I: Stack and Queues

13.1 Stacks

13.1.1 Evaluating Arithmetic Expressions

13.1.2 Conversion of Infix Expression to Postfix Expression

13.2 Queues

Summary

Exercises

14. Data Structures II: Linked Lists

14.1 Introduction

14.2 Insertion and Deletion at the Beginning of a Linked List

14.3 Deleting a Node with a Particular Value from a Linked List

14.4 Traversing a Linked List

14.5 Maintaining Sorted Linked List While Inserting

14.6 Stack Implementation Using Linked List

14.7 Queue Implementation Using Linked List

Summary

Exercises

15. Data Structures III: Binary Search Trees

15.1 Definitions and Notations

15.2 Binary Search Tree

15.3 Traversal of Binary Search Trees

15.3.1 Inorder Traversal

15.3.2 Preorder Traversal

15.3.3 Postorder Traversal

15.3.4 Height of a Binary Tree

15.4 Building Binary Search Tree

Summary

Exercises

16. More on Recursion

16.1 Pattern within a Pattern

16.2 Generalized Eight Queens Problem

16.3 Knight's Tour Problem

16.4 Stable Marriage Problem

16.5 Fractal (Hilbert Curve and Sierpinski Triangle)

16.6 Sudoku

16.7 Guidelines on Using Recursion

Summary

Exercises

17. Graphics

17.1 2D Graphics

17.1.1 Point and Line

Axis, Title, and Label

Plotting Multiple Functions in the Same Figure

Multiple Plots

Saving Figure

17.1.2 Histogram and Pi Chart

17.1.3 Sine and Cosine Curves

17.1.4 Graphical Objects: Circle, Ellipse, Rectangle,
Polygon,
and Arrow

Circle

Ellipse

Rectangle

Polygon

Arrow

17.2 3D Objects

Box

Sphere

Ring

Cylinder

Arrow

Cone

Curve

17.3 Animation – Bouncing Ball

Summary

Exercises

18. Applications of Python

18.1 Collecting Information from Twitter

Open Authentication

An Example – Collecting User Information

Collecting Followers, Friends, and Tweets of a User

Collecting Tweets Having Specific Words

18.2 Sharing Data Using Sockets

Client-Server Communication on the Same Machine – A Simple Example

An Echo Server

Accessing Web Data (Downloading a Pdf File)

18.3 Managing Databases Using Structured Query Language (SQL)

18.3.1 Database Concepts

18.3.2 Creating Database and Tables

18.3.3 Inserting Data into Table

18.3.4 Retrieving Data from Table

18.3.5 Updating Data in a Table

18.3.6 Deleting Data from Table/Deleting Table

18.4 Developing Mobile Application for Android

18.4.1 A Simple Application Containing Registration Interface

18.4.2 Tic-Tac-Toe Game

18.4.3 Running Kivy Applications on Android

18.5 Integrating Java with Python

18.5.1 Accessing Java Collections and Arrays in Python

18.5.2 Converting Python Collections to Java Collections

[18.5.3 Invoking a Parameterized Java Method from Python](#)

[18.5.4 Invoking Parameterized Python Method from Java](#)

[18.6 Python Chat Application Using Kivy and Socket Programming](#)

[Summary](#)

[Exercises](#)

[Index](#)

[Colour Illustrations](#)

Foreword

Since the early days of teaching programming languages, the science of programming has changed substantially. Since the times of COBOL, FORTRAN, ALGOL, C, and ADA, Python reflects a major paradigm shift. The classical book '*The Art of Computer Programming*' by D. E. Knuth changed the landscape by making programming independent of any language. As programmers found it hard to design structured code in languages like FORTRAN and COBOL, subsequent languages like C, C++, and Java provided a natural framework for structured programming. While C++ introduced object-oriented programming as an add on feature, Java was designed to be an object-oriented platform independent language with extensive support for web development. Although implemented efficiently, Java lacks the conciseness and flexibility that the amateur and professional programmers cherish so much. This is where Python steps in as a language of choice.

This book '*Python Programming: A Modular Approach*' is ideally suited for the undergraduate students who do not have any prior exposure to programming. The experience has shown that when the students are taught the good programming practices later in the course, they tend to ignore them in the programs/software they develop subsequently. For example, it is my considered opinion that the use of functions should be introduced as early as possible in a programming course, and this is where this book scores over typical programming texts. Also, the book stresses the use of named constants and documentation right from the beginning.

The book contains several examples that illustrate the use of syntactic features of Python in the context of the problem at hand. It uses Python Tutor to present the concept of recursion and data structures in a simplified manner. The chapter on applications is particularly

inviting as it includes examples from databases,
networking, web, and mobile technologies.

Prof. S K Gupta

*Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India*

Preface

This book introduces programming concepts through Python language. The simple syntax of Python makes it an ideal choice for learning programming. Because of the availability of extensive standard libraries and third-party support, it is rapidly evolving as the preferred programming language amongst the application developers.

As old habits die hard, the book encourages the reader to follow good programming practices right from the beginning. We have tried to introduce the programming constructs in the context of the examples that justify the use of those constructs. We have devoted the first twelve chapters of the book to introduce the fundamental programming concepts. These chapters would serve as a foundation for the advanced concepts covered later in the book. The first three chapters cover the basic building blocks of the Python language. We introduce the concepts of modularity early in the book (Chapter 2) so that it gets integrated with the student's approach to problem-solving. In the same spirit, we emphasize the use of the named objects in preference to the ones hardcoded. Further, as sound documentation is a key to the success of any software engineering project, we consistently document the code throughout the book, describing the objective of each piece of the code and how it interfaces with rest of the system under discussion.

As the real-life problems usually require non-sequential and repetitive execution of instructions, the discussion on modularity in Chapter 2 has been followed by a detailed study of control structures in Chapter 3. As debugging is an essential skill for discovering the bugs and making the program error-free, we have devoted a full chapter (Chapter 4) to it. The parameter passing mechanisms and the scope of names are fundamental concepts in the study of a programming language, and in Chapter 5, we discuss these concepts in detail. Chapters 6 and 7 cover the mutable and immutable

objects for storing data in a program. Often, we need to save the data permanently in the form of files and retrieve it for use at a later point in time. As many errors creep in during the input/output process, in Chapter 9, we discuss the concepts of file processing and error handling side by side.

In Chapter 10, we introduce classes that lie at the heart of an important programming methodology called object-oriented programming (OOP). In Chapter 11, we continue the discussion on classes in the context of OOP concepts like polymorphism, encapsulation, data hiding, data abstraction, and inheritance.

In Chapter 12, we cover basic sorting and searching techniques, namely, insertion sort, selection sort, bubble sort, merge sort, quick sort, and linear and binary search. Recursion being a valuable tool for problem-solving, we devote two chapters (Chapters 8 and 16) to it. Beginning with the recursive functions for computing the factorial of a number and the terms in a Fibonacci sequence, we move on to more sophisticated problems like the tower of Hanoi, permutation generation, list manipulation, 8-queens problem, stable marriage problem, and plotting Hilbert curves that demonstrate the power of recursion. We introduce the basic data structures – stacks, queues, linked lists, and trees (Chapters 13, 14, and 15), and show how these data structures may be implemented efficiently using Python.

As several applications require visualization of information, we devote a chapter to 2D and 3D graphics, and animation (Chapter 17). We conclude the book with a chapter (Chapter 18) that briefly covers the application areas such as twitter, web, database management, and mobile app development. As a lot of application code is developed in Java, in this chapter we also discuss, how Python code may be seamlessly integrated with Java code and vice-versa. Recently, Philip J. Guo of the University of California, San Diego has made a Python Tutor available as an opensource tool for visualizing the execution of Python scripts, and we make use of it to illustrate various concepts throughout the book.

ACKNOWLEDGEMENTS

We are grateful to Hitarth, first-year undergraduate student at Dyal Singh College of University of Delhi, for going through the entire book with great care and

suggesting dozens of subtle corrections and modifications. We are thankful to Arpit of Amazon Inc. for his help in identifying the suitable level of abstraction in several chapters of the book. We are also grateful to Prof. S K Gupta, IIT Delhi, Mr. Suraj Prakash, Director Training, Bal Bharati group of institutions, Dr Anamika Gupta of SS College of Business Studies, Dr Sharanjeet Kaur of Acharya Narendra Dev College, Mr. Ashok Jain of Sugal Infotech, and our former students Abhishek, Ajay, Kanika, Mitul and Sunil, for their comments on various sections of the book.

We are grateful to the entire Pearson team, especially Ms. Neha Goomer and Mr. M. Balakrishnan who were readily available for any help during the preparation of the book. Ms. Neha's suggestion to include margin notes in the book has significantly improved the readability of the book. Last but not the least, we would like to express our gratitude to our family members for their support and patience.

**Sheetal Taneja
Naveen Kumar**

About the Authors



Sheetal Taneja is currently working as an Assistant Professor at Department of Computer Science, Dyal Singh College, University of Delhi, Delhi, India. She earned her B.Sc. (Honours) and M.Sc. degrees in Computer Science from University of Delhi. Being the topper in the M.Sc. programme, she was awarded a gold medal by University of Delhi. Being passionate about teaching, she qualified the National Eligibility Test (NET) for university teachers while studying for the M.Sc. degree, and joined a faculty position immediately on completing her postgraduate studies. She is currently pursuing her Ph.D. in Computer Science from University of Delhi. She is a co-author of two books on information technology, published by Central Board of Secondary Education (CBSE). She has authored several research articles, and has attended many national and international workshops and conferences. She is a

member of the Association of Computing Machinery (ACM), USA.



Naveen Kumar is a Professor at Department of Computer Science, University of Delhi, Delhi, India. He holds a Ph.D. degree in Computer Science from IIT, Delhi. Prior to his doctoral studies, he earned his M.Tech. (Computer Science) and M.Sc. (Mathematics) degrees from IIT Delhi, and an honours degree in Mathematics from University of Delhi. He has taught a varied array of courses to undergraduate and postgraduate students and has teaching experience of about 35 years. He has supervised many Ph.D. students in areas such as soft computing and social networks. His current areas of interest include data analysis, evolutionary algorithms, and parallel computing. He has contributed several articles in national and international journals and conferences. He has also attended many conferences, seminars and workshops. He has been a member of the curriculum design committees of various universities and the CBSE. He has also been an organizing chair/program committee member of several workshops, conferences, and symposia. He is a member of professional bodies like ACM, Institution of

Electronics and Telecommunication Engineers (IETE),
and Computer Society of India.

CHAPTER 1

PYTHON PROGRAMMING: AN INTRODUCTION

CHAPTER OUTLINE

[1.1 IDLE – An Interpreter for Python](#)

[1.2 Python Strings](#)

[1.3 Relational Operators](#)

[1.4 Logical Operators](#)

[1.5 Bitwise Operators](#)

[1.6 Variables and Assignment Statements](#)

[1.7 Key words](#)

[1.8 Script Mode](#)

Python is an interactive programming language. Simple syntax of the language makes Python programs easy to read and write. Python was developed by Guido Van Rossum in 1991 at the National Research Institute for Mathematics and Computer Science in the Netherlands. Guido Van Rossum named Python by getting inspired from his favourite comedy show *Monty Python's Flying Circus*.

Ever since the language was developed, it is becoming popular day by day amongst the programmers. Various versions of Python have been released till date ranging from version 1.0. This book uses Python 3.6. Python is used in various application areas such as the web, gaming, scientific and numeric computing, text processing, and network programming.

1.1 IDLE – AN INTERPRETER FOR PYTHON

IDLE stands for Integrated Development and Learning Environment. Python IDLE comprises Python shell and Python Editor. While Python shell is an interactive interpreter, Python Editor allows us to work in script

mode. While using Python shell, we just need to type Python code at the `>>>` prompt and press the `enter` key, and the shell will respond with the result of the computation. For example, when we type

Python shell displays the prompt `>>>` to indicate that it is waiting for a user command

```
>>> print('hello world')
```

the text to be printed is enclosed between apostrophe marks

and press enter key, Python shell will display:

```
hello world
```

the apostrophe marks are not displayed in the output

Python shell may also be used as a calculator, for example, when we type `18 + 5` followed by `enter`, Python shell outputs the value of the expression, i.e. `23`, as shown below:

```
>>> 18 + 5
```

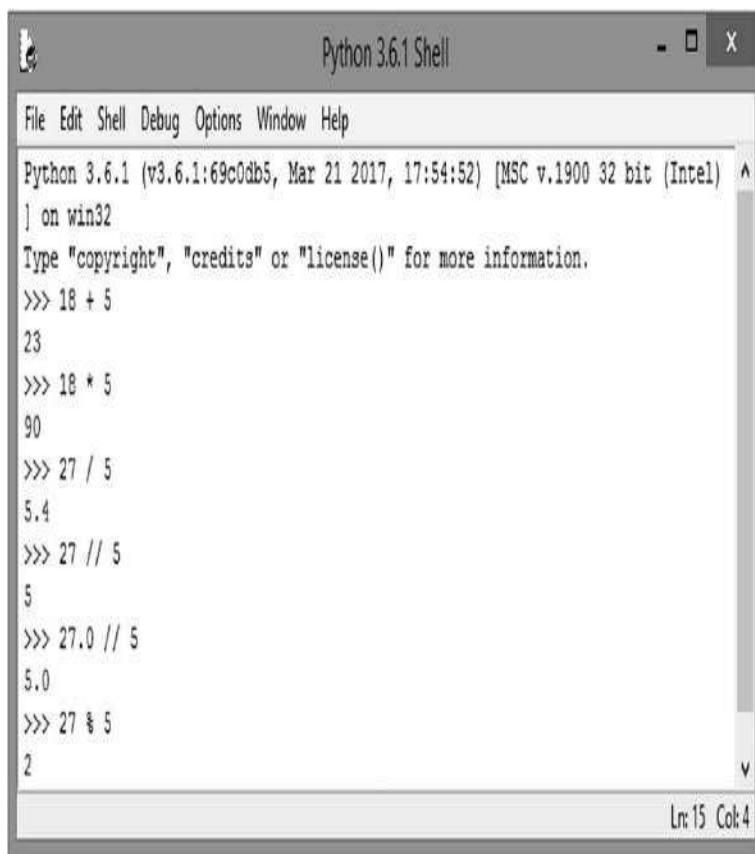
```
23
```

operator `+` acts on the operands `18` and `5`

In the expression `18 + 5`, `+` is called the operator. The numbers, `18` and `5` on which the operator `+` is applied, are called the operands. In general, an expression is a valid combination of operators and operands. In Python, all forms of data are called values or objects. For example, `18`, `5`, and `23` are objects. The expression `18 + 5` yields the value `23`. Similarly, the expression `18 - 5` yields the value `13`. The expression `18 * 5` yields `90`, as Python uses `*` as the symbol for the multiplication

operation. The expression `27 / 5` yields `5.4`. Note that the result of division `27 / 5` is real number. In Python, result of division is always a real number. However, if you wish to obtain an integer result (for example, `floor(5.4)`), Python operator `//` (two slashes without any intervening blank) can be used. For example, `27 // 5` yields `5`. However, while using the operator `//`, if at least, one of the two operands is a real number; the result is a real number, for example, the expression `27.0 // 5` yields `5.0`. Real numbers are called floating point numbers in Computer Science terminology in view of the floating point representation used to represent real numbers. However, the number representation is not the subject matter of this book. Python operator `%` (percentage sign), known as modulus, yields the remainder of the division, for example, `27 % 5` yields `2` because on dividing `27` by `5` we get `2` as the remainder. In Fig. 1.1, we illustrate these examples in Python shell.

operators: `+`, `-`, `*`, `/`, `//`, `%`, and `**`



The screenshot shows the Python 3.6.1 Shell window. The title bar reads "Python 3.6.1 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python session:

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
] on win32
Type "copyright", "credits" or "license()" for more information.

>>> 18 + 5
23
>>> 18 * 5
90
>>> 27 / 5
5.4
>>> 27 // 5
5
>>> 27.0 // 5
5.0
>>> 27 % 5
2
```

The status bar at the bottom right indicates "Ln:15 Col:4".

Fig. 1.1 Computation of arithmetic expressions

Python shell window

The exponentiation operator (`**`) yields *raise to the power* operation, i.e. a^b . For example, `3 ** 2` is expressed as `3 ** 2`, and yields the value 9. Similarly, `2 ** -1` yields 0.5 as expected. The expressions in each of the examples mentioned above involved only single mathematical operator, but in practice, we would require more complex expressions. Note that whereas the expression `6 / 3 / 2` yields 1.0 (= `(6 / 3) / 2`), the expression `2 ** 3 ** 2` yields 512 (= `2 ** (3 ** 2)`). This is so because whereas the operator `/` (as also `+`, `-`, `*`, `//`, `%`) is evaluated left to right, the exponentiation operator is evaluated right to left. We say that the operators, `+`, `-`, `*`, `/`, `//`, and `%`, are left associative operators and the operator `**` is right associative. Python operator `-` (negation) yields negative of the operand. For example, `10 * -5` multiplies 10

and -5 (negative of 5), yielding -50 as a result. Next, we evaluate the foregoing expressions in Python shell:

```
left associative operators: +, -, *, /, //, %
right associative operator: **
```

```
>>> 3 ** 2
```

```
9
```

```
>>> 6 / 3 / 2
```

```
1.0
```

```
>>> 2 ** 3 ** 2
```

```
512
```

```
>>> 10 * -5
```

```
-50
```

$-$ operator works as negation operator

When we enter the expression $7 + 4 * 3$, the system responds with value 19 as a result. Note that in the expression $7 + 4 * 3$, Python evaluates the operator $*$ before $+$ as if the input expression was $7 + (4 * 3)$ because the operator $*$ has higher precedence than $+$. In Table 1.1, we list the operators in decreasing order of their precedence. The expressions within parentheses are evaluated first; for example, given the expression $(7 + 3) * 10 / 5$, the system would yield 20.0 . Parentheses are often used to improve the readability of an expression.

Table 1.1 Precedence of arithmetic operators

() (parentheses)	decreasing order
** (exponentiation)	
- (negation)	
/ (division) // (integer division) * (multiplication) % (modulus)	
+ (addition) - (subtraction)	

while the parentheses have the highest precedence, addition and subtraction are at the lowest level

An error occurs when the input expression is not correctly formed. For example, the expression `7 + 3(4 + 5)` generates an error because the operator between 3 and (is missing:

Python complains when it encounters a wrongly formed expression

```
>>> 7 + 3(4 + 5)

Traceback (most recent call last):

File "<pyshell#17>", line 1, in <module>

    7 + 3(4 + 5)

TypeError: 'int' object is not callable
```

Similarly, the expression `7 / 0` yields a zero division error since division by zero is a meaningless operation.

```
>>> 7 / 0

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>
```

```
7 / 0
```

```
ZeroDivisionError: division by zero
```

division by zero is a meaningless operation

1.2 PYTHON STRINGS

A string is a sequence of characters. To specify a string, we may enclose a sequence of characters between single, double, or triple quotes. Thus, 'Hello world', 'Ajay ', "what's ", '''today's "action" plan''' are the examples of strings. A string enclosed in single quotes may include double quote marks and vice versa. A string enclosed in triple quotes (also known as docstring, i.e. documentation string) may include both single and double quote marks and may extend over several lines, for example:

```
>>> 'Hello World'  
'Hello World'  
  
>>> print('Hello World')  
  
Hello World  
  
>>> """Hello  
  
What's  
  
happening"""  
  
"Hello\nWhat's\nhappening"  
  
>>> print("""Hello  
  
What's  
  
happening""")  
  
Hello
```

What's

happening

a Python string may be enclosed within single, double, or triple quotes

Note that enclosing quote marks are used only to specify a string, and they do not form part of the string. When we print a string using the `print` instruction, the value of the string, i.e. the sequence of characters within quotes is printed. Also, note that, within a string, `\n` denotes the beginning of a new line. When a string having `\n` is printed, everything after `\n` (excluding `\n`) is printed on the next line. In this section, we shall study basic operations on strings. Let us begin with the concatenation operator (`+`), which is used to produce a string by concatenating two strings; for example, the expression `'hello' + '!!!'` yields the string

`'hello !!!'`. Similarly, the expression `'how' + 'are' + ' you?'` yields `'how are you?'` (`=('how' + ' are') + ' you')` as shown below:

```
>>> 'hello' + '!!!'  
'hello !!!'  
  
>>> 'how' + ' are' + ' you?'  
'how are you?'
```

escape sequence `\n` marks a new line

use of `+` as the concatenation operator

The operator `*` (multiplication) is used to repeat a string a specified number of times; for example, the expression `'hello' * 5` yields the string

`'hellohellohellohellohello'`:

```
>>> 'hello' * 5  
'hellohellohellohellohello'
```

use of * operator for repeating a string several times

1.3 RELATIONAL OPERATORS

Relational operators are used for comparing two expressions and yield `True` or `False`. In an expression involving both arithmetic and relational operators, the arithmetic operators have higher precedence than the relational operators. In [Table 1.2](#), we give a list of relational operators

Table 1.2 Relational operators

<code>==</code>	(equal to)
<code><</code>	(less than)
<code>></code>	(greater than)
<code>< =</code>	(less than or equal to)
<code>> =</code>	(greater than or equal to)
<code>! =</code>	(not equal to)

relational operators for comparing expression values

A relational operator applied on expressions takes the following form:

expression <comparison operator> expression

Thus, the expressions, `23 < 25`, `23 != 23`, and `23 - 2.5 >= 5 * 4`, yield `True`, `False`, and `True`, respectively. As arithmetic operators have higher precedence than the relational operators, the expression `23 - 2.5 >= 5 * 4` is evaluated as if it were `(23 - 2.5) >= (5 * 4)`. When the relational operators are applied to strings, strings are compared left to right, character by character, based on their ASCII codes, also called ASCII values. For example, ASCII codes of '`A`'-'`Z`', '`a`'-'`z`', and '`0`'-'`9`' lie in the range [65, 90], [97, 122], and [48, 57], respectively. Thus, '`h`' > '`H`' yields `True` as ASCII value of '`h`' (= 104) is greater than ASCII value of '`H`' (= 72). Also, '`hello`' > '`Hello`' yields `True` since '`h`' is greater than '`H`'. Similarly, '`hi`' > '`hello`' yields `True` because the first character in the two strings is identical and ASCII code of '`i`' is greater than ASCII code of '`e`'. Also, if a string is a prefix of another string, the longer string will be considered larger.

relational operators yield values: `True`, `False`

ASCII values of characters are used for string comparison

Python 3 does not allow string values to be compared with numeric values

1.4 LOGICAL OPERATORS

The logical operators `not`, `and`, and `or` are applied to logical operands `True` and `False`, also called Boolean values, and yield either `True` or `False`. The operator `not` is a unary operator, i.e. it requires only one operand. The expressions `not True` and `not False` yield `False` and `True`, respectively. An expression involving two Boolean operands and the `and` operator yields `True` if both the operands are `True`, and `False` otherwise. Similarly, an expression involving two Boolean operands and the `or` operator yields `True` if at

least one operand is `True`, and `False` otherwise. This is shown in [Table 1.3](#).

combining expressions using logical operators

logical operators `not`, `and`, `or` yield values: `True`, `False`

Table 1.3 `not`, `and`, `and or` operators

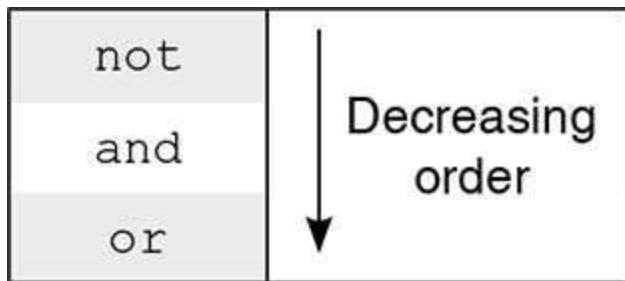
<code>not</code>	
<code>True</code>	<code>False</code>
<code>False</code>	<code>True</code>

<code>and</code>		
	<code>True</code>	<code>False</code>
<code>True</code>	<code>True</code>	<code>False</code>
<code>False</code>	<code>False</code>	<code>False</code>

<code>or</code>		
	<code>True</code>	<code>False</code>
<code>True</code>	<code>True</code>	<code>True</code>
<code>False</code>	<code>True</code>	<code>False</code>

Precedence of logical operators is shown in [Table 1.4](#).

Table 1.4 Precedence of logical operators



While evaluating an expression involving an `and` operator, the second sub-expression is evaluated only if the first sub-expression yields `True`. For example, in the expression `(10 < 5) and ((5 / 0) < 10)`, since the first sub-expression `(10 < 5)` yields `False`, and thus determines the value of the entire expression as `False`, Python does not evaluate the second sub-expression. This is called short-circuit evaluation of boolean expression. However, the expression `(10 > 5)` and `((5 / 0) < 10)` yields an error since the first sub-expression yields `True`, and Python attempts to evaluate the second sub-expression `((5 / 0) < 10)` that yields an error because division by zero is an illegal operation:

short-circuit evaluation of a boolean expression

```
>>> (10 < 5) and ((5 / 0) < 10)
False
>>> (10 > 5) and ((5 / 0) < 10)
Traceback (most recent call last):
  File "<pyshell#104>", line 1
    (10 > 5) and ((5 / 0) < 10)
ZeroDivisionError: division by zero
```

Similarly, while evaluating an expression involving an `or` operator, the second sub-expression is evaluated only if the first sub-expression yields `False`. Thus, whereas the expression `(10 > 5) or ((5 / 0) < 10)` yields `True`, the expression `(10 < 5) or ((5 / 0) < 10)` yields an error.

To evaluate an expression comprising arithmetic, relational, and logical operators, the operators are applied according to precedence order for the operators given in [Table 1.5](#). For example, the expression `not 9 == 8 and 7 + 1 != 8 or 6 < 4.5` is evaluated as if it were the following equivalent parenthesized expression:

expressions involving arithmetic, relational, and logical operators

`((not (9 == 8)) and ((7 + 1) != 8)) or (6 < 4.5)`

i.e. `((not False) and ((7 + 1) != 8)) or (6 < 4.5)`

i.e. `(True and ((7 + 1) != 8)) or (6 < 4.5)`

i.e. `(True and (8 != 8)) or (6 < 4.5)`

i.e. `(True and False) or (6 < 4.5)`

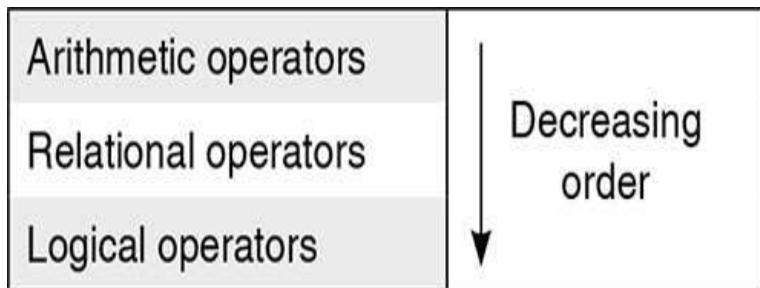
i.e. `False or (6 < 4.5)`

i.e. `False or False`

i.e. `False`

expression evaluation using precedence rules

Table 1.5 Precedence of operators



1.5 BITWISE OPERATORS

Bitwise operators are the operators that operate on integers interpreted as strings of binary digits 0 and 1, also called bits. In Table 1.6, we list bitwise operators, their description, along with examples. For the sake of brevity, we assume that all numbers are eight-bit long.

bitwise operators operate bit by bit

Table 1.6 Boolean operators

Bitwise operator	Description	Example
Bitwise AND $x \& y$	A bit in $x \& y$ is 1 if the corresponding bit in each of x and y is 1, and 0 otherwise.	10 & 6 yields 2: 10: 00001010 6: 00000110 ----- 2: 00000010
Bitwise OR $x y$	A bit in $x y$ is 1 if at least one of the corresponding bits in x and y is 1, and 0 otherwise.	10 6 yields 14: 10: 00001010 6: 00000110 ----- 14: 00001110
Bitwise Complement $\sim x$	A bit in $\sim x$ is 1 if and only if the corresponding bit in x is 0. The result obtained from	~ 11 yields -12: 11: 00001011 ----- ~11: 11110100

	this operation is $-x-1$.	Note that 11110100 is the two's complement of 12: 00001100.
Bitwise Exclusive OR $x \wedge y$	A bit in $x \wedge y$ is 1 if exactly one of the corresponding bits in x and y is 1, and 0 otherwise.	10 \wedge 6 yields 12: 10: 00001010 6: 00000110 ----- 12: 00001100
Left Shift $x \ll y$	Bits in the binary representation of x are shifted left by y places. Rightmost y bits of the result are filled with zeros. The result obtained from this operation is $x \cdot 2^y$.	5 \ll 2 yields 20: 5: 00000101 ----- 20: 00010100
Right Shift $x \gg y$	Bits in the binary representation of x are shifted right by y places. Leftmost y bits of the result are filled with sign bit. The result obtained from this operation is $x / 2^y$.	5 \gg 2 will yield 1 as the result as shown below: 5: 00000101 ----- 1: 00000001

Table 1.7 shows precedence order of different operators in Python

Table 1.7 Precedence of operators

()	
**	
+X, -X, ~X	
/, //, *, %	
+, -	
<<, >>	
&	
^	
==, <, >, <=, >=, !=	
not	
and	
or	

Decreasing
order

parentheses have the highest precedence while logical operators have the lowest precedence

1.6 VARIABLES AND ASSIGNMENT STATEMENTS

Variables provide a means to name values so that they can be used and manipulated later on. Indeed, a program essentially carries out variable manipulations to achieve the desired objective. For example, variables `english`, `maths`, and `commerce` may be used to deal with the marks obtained in these subjects, and the variables `totalMarks` and `percentage` may be used to deal with aggregate marks and the overall percentage

of marks, respectively. If a student has obtained 57 marks in english, this may be expressed in Python using the following statement:

```
>> english = 57
```

a variable is a name given to a value

When the above statement is executed, Python associates the variable english with value 57. We can also say that the variable english is assigned the value 57, or that the variable english refers to value 57. Such a statement that assigns value to a variable is called an assignment statement. In Fig. 1.2, we see that the variable english refers to the value 57. A figure such as Fig. 1.2 is known as reference diagram or state diagram as it shows the current state of the variable.

assignment statement

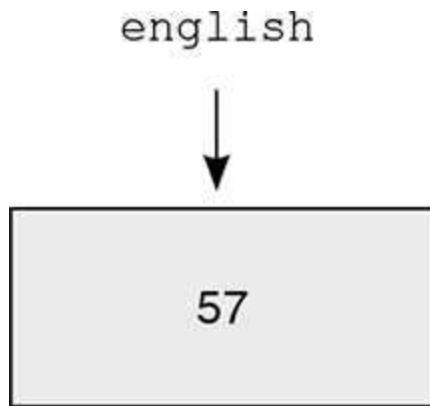


Fig. 1.2 Current state diagram of variable english

variable english refers to value 57

In Python style (often called Pythonic style, or Pythonic way), variables are called names, and an assignment is called an association or a binding. Thus, we say that the name english has been associated

with the data object 57, or that the variable `english` has been bound to the data object 57. Note that, unlike an expression, on the execution of an assignment statement, IDLE does not respond with any output. Of course, the value of the variable `english` can be displayed as follows:

```
>>> english
```

```
57
```

in Python, variables are often called names

an assignment statement binds a variable to an object

In the above example, `english` is a variable whose value is 57. Similarly, the following statements will assign values 64 and 62 to the variables `maths` and `commerce`, respectively.

```
>>> maths = 64
```

```
>>> commerce = 62
```

In this example, we assume that maximum mark in each subject is 100. Total marks in these three subjects and overall percentage can be computed using another sequence of statements:

```
>>> totalMarks = english + maths +  
commerce
```

```
>>> percentage = (totalMarks / 300) * 100
```

```
>>> percentage
```

```
61.0
```

computing overall percentage from marks in different subjects

Assignment Statement

As discussed above, values are assigned to variables using an assignment statement. The syntax for assignment statement is as follows:

variable = expression

syntax for assignment statement

where expression may yield a value such as numeric, string, or Boolean. In an assignment statement, the *expression* on the *right-hand side* of the equal to operator (=) is evaluated and the value so obtained is assigned to the variable on the *left-hand side* of the = operator. We have already seen some examples of variables. In general, we must follow the following rules while using the variables:

rules for naming variables

- A variable must begin with a letter or _ (underscore character)
- A variable may contain any number of letters, digits, or underscore characters. No other character apart from these is allowed.

Thus, marks, name, max_marks, _emp, maxMarks, n1, and max_of_3_numbers are valid variables. However, note that the following are not valid variables:

total_no.	#use of dot (.)
1st_number	#begins with a digit
AmountIn\$	#use of dollar symbol (\$)
Total Amount	#Presence of blank between two words

It is a good programming practice to make sure that the variables are meaningful as carefully chosen variables

make your code readable. For example, to denote the total marks obtained, it is easy to understand what the variable `totalMarks` represents than a variable `a`, `x`, `marks` or `total` used for the same purpose. There are several styles of forming variables, for example:

```
total_marks
```

```
TotalMarks
```

```
totalMarks
```

```
TOTAL_MARKS
```

always use meaningful variable names

However, one must be consistent in following a style. In this book, we prefer to use the lowerCamelCase style in which the first letter of every word is capitalized, except the first word which begins with a lowercase letter, for example, `totalMarks`. While forming variables, it is important to note that Python is case-sensitive. Thus, `age` and `Age` are different variables. So, if we assign a value to the variable `age` and display the value of `Age`, the system will respond with an error message.

follow a consistent style in choosing variables

```
>>> age = 24
```

```
>>> Age
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#1>", line 1, in <module>
```

```
    Age
```

```
NameError: name 'Age' is not defined
```

Python is case-sensitive.
age and Age are different names

Next, let us examine the following statements:

```
>>> a = 5
```

```
>>> b = a
```

```
>>> a = 7
```

more than one variable may refer to the same object

The first statement assigns name `a` to an integer value 5 as shown in Fig. 1.3(a). The next statement assigns another name `b` to the same value 5 (Fig. 1.3 (b)). Finally, the third statement assigns a different value 7 to `a` (Fig. 1.3 (c)). Now the variables `a` and `b` refer to different values.

Shorthand Notation

By now, we have seen several examples of assignment statements. The following assignment statement updates value of the variable `a` by adding 5 to the existing value.

```
>> a = a + 5
```

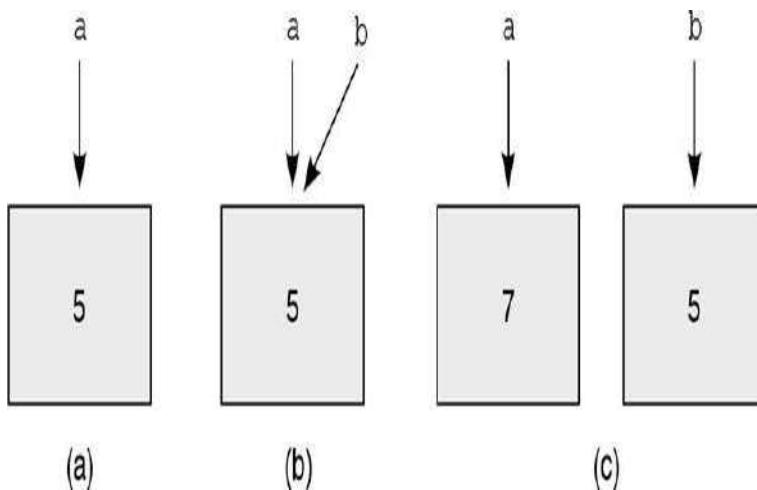


Fig. 1.3 Current state diagram

Alternatively, we can write the above statement as

```
>>> a += 5
```

a shorthand notation

Thus, the operator `+=` serves as shorthand notation in assignment statement. Similarly, the statements

```
>>> a = a + b + c
```

```
>>> a = a ** (b + c)
```

can be written as follows, respectively:

```
>>> a += b + c
```

```
>>> a **= b + c
```

As the shorthand notation works for all binary mathematical operators,

`a = a <operator> b`

is equivalent to

`a <operator>= b`

Python also allows multiple assignments in a single statement; for example, the following sequence of three assignment statements

```
>>> msg = 'Meeting'
```

```
>>> day = 'Mon'
```

```
>>> time = '9'
```

may be replaced by a single statement:

```
>>> msg, day, time = 'Meeting', 'Mon', '9'
```

multiple assignments in a single statement

Thus, we can specify more variables than one on the left side of an assignment statement, and the corresponding number of expressions, like $\langle expr_1 \rangle$, $\langle expr_2 \rangle$, ..., $\langle expr_K \rangle$, on the right side. Formally, the syntax for an assignment statement may be described as follows:

```
 $\langle name_1 \rangle, \langle name_2 \rangle, \dots, \langle name_K \rangle = \langle expr_1 \rangle,$ 
 $\langle expr_2 \rangle, \dots, \langle expr_K \rangle$ 
```

This notation can be used to enhance the readability of the program. We may also assign a common value to multiple variables, for example, the assignment statement

```
>>> totalMarks = count = 0
```

assigning same value to multiple variables in a single statement

may be used to replace the following two assignments:

```
>>> totalMarks = 0
>>> count = 0
```

Suppose we wish to swap values of two variables, say, num1 and num2. For this purpose, we use a variable temp as follows:

```
>>> num1, num2 = 10, 20
>>> temp = num1
>>> num1 = num2
>>> num2 = temp
>>> print(num1, num2)
```

20 10

Note that before executing the assignment statement

```
num1 = num2
```

we save the value of the variable `num1` in the variable `temp`, so that the same can be assigned to the variable `num2` in the following assignment statement. A variable that is used to hold the value of another variable temporarily is often called temporary variable.

Alternatively, we can use the following statement to swap the values of two variables:

```
>>> num1, num2 = num2, num1
```

1.7 KEYWORDS

Keywords are the reserved words that are already defined by the Python for specific uses. They cannot be used for any other purpose. Hence, make sure that none of the names used in a program should match any of the keywords. The list of the keywords in Python 3.6 is given in Fig. 1.4.

key words have special meaning

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

Fig. 1.4 List of Python key words

keywords cannot be used for naming objects

1.8 SCRIPT MODE

So far, we have done all work using Python shell, an interactive environment of IDLE. The definitions of objects, names, etc., exist only during an IDLE session. When we exit an IDLE session, and start another IDLE session, we must redo all computations. This is not a convenient mode of operation for most of the computational tasks. Python provides another way of working called script mode. All the instructions necessary for a task are stored in a file often called a source file script, program, source code, or a module. Python requires that such a file must have extension .py or .pyw. To create a script, Python IDLE provides a New Window option in the File menu. On clicking the New Window option, a Python Editor window opens, where we can write a sequence of instructions, collectively called a program or a script. A script can be saved using the Save As option in the File menu. It may be executed in shell by selecting Run Module option in the Run menu.

in script mode, instructions are written in a file

a script should have extension .py or .pyw

As a trivial example, the script `readNshow` in Fig. 1.5 takes two numbers from the user and outputs these numbers. On executing the first line of the script, the system waits for the user to enter a number which is assigned to the variable `number1`. On executing the second line of the script, the system waits for the user to enter another number which is assigned to the variable `number2`. The values of variables `number1` and `number2` are displayed on executing the third line of the script.

The figure consists of two screenshots of a Windows desktop environment. The top screenshot shows a code editor window titled 'readNshow.py - F:/Python Programs/Chapter-1/readNshow.py (3.6.1)'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code area contains three lines of Python code:

```
number1 = input()  
number2 = input()  
print(number1, number2)
```

The status bar at the bottom right indicates 'Ln: 5 Col: 0'. The bottom screenshot shows a 'Python 3.6.1 Shell' window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell displays the Python version and build information, followed by a prompt 'Type "copyright", "credits" or "license()" for more information.' A series of inputs and outputs are shown:
```  
==== RESTART: F:/Python Programs/Chapter-1/readNshow.py ======  
3  
6  
3 6  
```

The status bar at the bottom right indicates 'Ln: 13 Col: 4'.

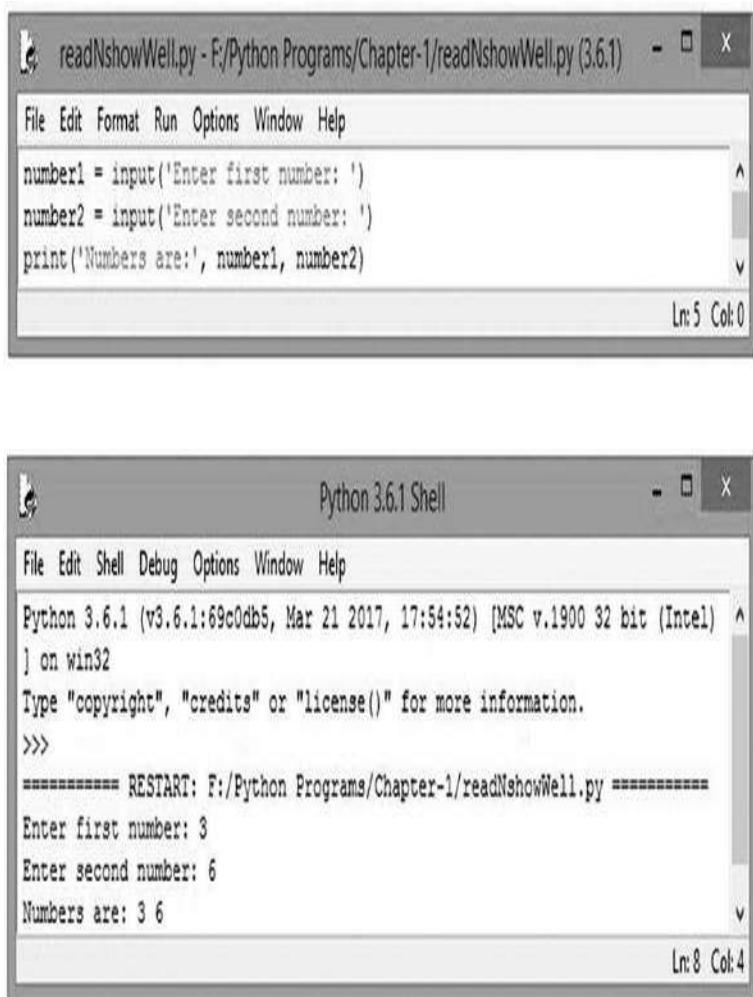
Fig. 1.5 Input–output program

a simple script

output on executing the script

The problem with the above script is that as we have developed it just now, we remember that the program requires two numbers as the input. However, if we wish to use it a month later, we may forget what inputs are required by the program, and we will need to examine the program again before running it. Surely, we would like to avoid this effort. A good programming practice is to display to the user what data is to be entered. The modified script `readNshowWell` (Fig. 1.6) achieves this

by displaying a message to the user as the system waits for user input.



The image shows two windows side-by-side. The top window is titled 'readNshowWell.py - F:/Python Programs/Chapter-1/readNshowWell.py (3.6.1)' and contains the following Python code:

```
number1 = input('Enter first number: ')
number2 = input('Enter second number: ')
print('Numbers are:', number1, number2)
```

The bottom window is titled 'Python 3.6.1 Shell' and shows the execution of the program. It starts with the Python version information, then prompts for input, and finally prints the result:

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
] on win32
Type "copyright", "credits" or "license()" for more information.

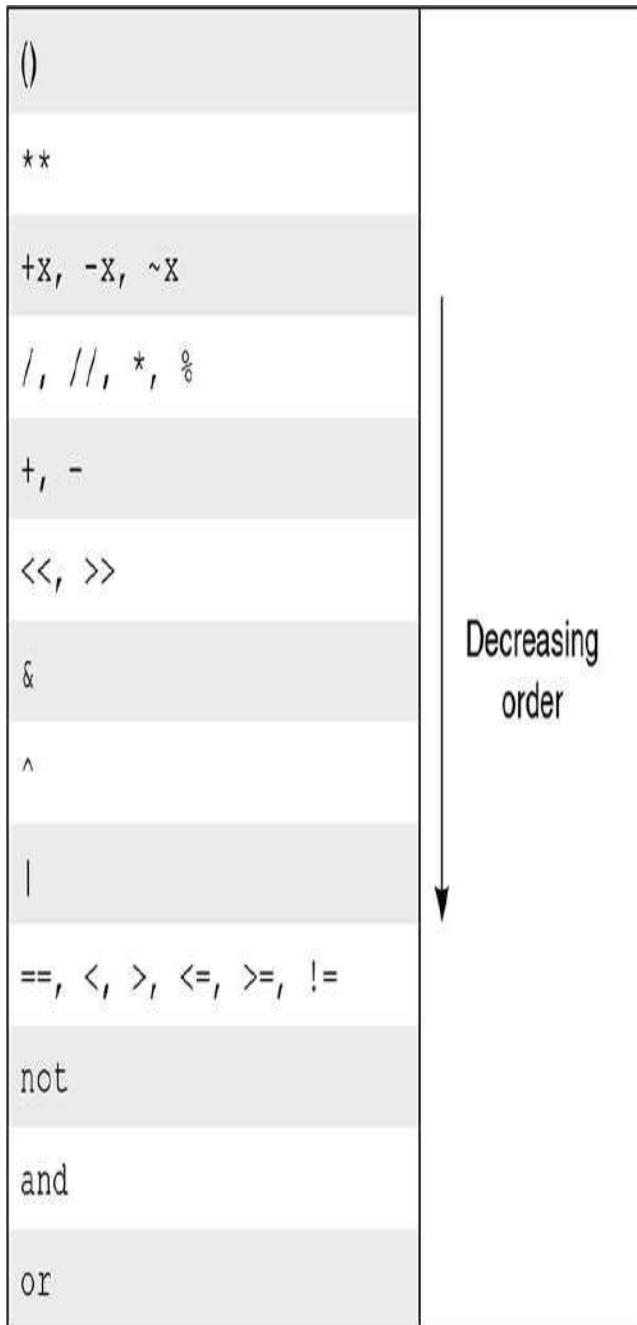
>>>
===== RESTART: F:/Python Programs/Chapter-1/readNshowWell.py =====
Enter first number: 3
Enter second number: 6
Numbers are: 3 6
```

Fig. 1.6 Input–output program

SUMMARY

1. Python interpreter executes one expression (command) at a time.
2. An expression is a valid combination of operators and operands. Operands are the objects on which operators are applied.
3. Arithmetic, relational, logical, and bitwise operators are used for performing computations in a program.
4. A string is a sequence of characters. To specify a string, we may use single, double, or triple quotes.
5. The operator + (addition) applied on two string operands is used for concatenating them. Similarly, the operator * (multiplication) is used to repeat a string a specific number of times.

6. When relational operators are applied on strings, strings are compared left to right character by character according to their ASCII values.
7. While evaluating a Boolean expression involving `and` operator, the second sub-expression is evaluated only if the first sub-expression yields `True`.
8. While evaluating an expression involving `or` operator, the second sub-expression is evaluated only if the first sub-expression yields `False`.
9. Precedence order for the operators is as follows:



10. A variable is a name that refers to a value. We may also say that a variable associates a name with a data

- object such as number, character, string, or Boolean.
11. Values are assigned to variables using assignment statement. The syntax for assignment statement is as follows:

```
variable = expression
```

where expression may yield any value such as numeric, string, or Boolean.

 12. In an assignment statement, the *expression* on the *right-hand side* of the equal to operator (=) is evaluated and the value so obtained is assigned to the variable on the *left-hand side* of the = operator.
 13. A variable name must begin with a letter or _ (underscore character). It may contain any number of letters, digits or underscore characters. No other character apart from these is allowed.
 14. Python is case-sensitive. Thus, `age` and `Age` are different variables.
 15. The shorthand notation for `a = a <operator> b` is
`a <operator>= b`
 16. In Python, multiple assignment statements can be specified in a single line as follows:
`<name1>, <name2>, ... = <expression1>, <expression2>, ...`
 17. A keyword is a reserved word that is already defined by Python for a specific use. Keywords cannot be used for any other purpose. The list of the Python keywords is given below:

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

18. Python programming can be done in an interactive and script mode.

EXERCISES

1. Evaluate the following expressions:

$$(x < y) \text{ or } (\text{not}(z == y) \text{ and } (z < x))$$
 1. $x = 0, y = 6, z = 10$
 2. $x = 1, y = 1, z = 1$
2. Evaluate the following expressions involving arithmetic operators:
 1. $-7 * 20 + 8 / 16 * 2 + 54$
 2. $7 ** 2 // 9 \% 3$
 3. $(7 - 4 * 2) * 10 / 5 ** 2 + 15$
 4. $5 \% 10 + 10 - 25 * 8 // 5$
 5. $'hello' * 2 - 5$

3. Evaluate the following expressions involving relational and logical operators:

1. 'hi' > 'hello' and 'bye' < 'Bye'
2. 'hi' > 'hello' or 'bye' < 'Bye'
3. $7 > 8$ or $5 < 6$ and 'I am fine' > 'I am not fine'
4. $10 \neq 9$ and $29 \geq 29$
5. $10 \neq 9$ and $29 \geq 29$ and 'hi' > 'hello' or 'bye' < 'Bye' and $7 \leq 2.5$

4. Evaluate the following expressions involving arithmetic, relational, and logical operators:

1. $5 \% 10 + 10 < 50$ and $29 \geq 29$
2. $7 ** 2 \leq 5 // 9 \% 3$ or 'bye' < 'Bye'
3. $5 \% 10 < 10$ and $-25 > 1 * 8 // 5$
4. $7 ** 2 // 4 + 5 > 8$ or $5 \neq 6$
5. $7 / 4 < 6$ and 'I am fine' > 'I am not fine'
6. $10 + 6 * 2 ** 2 \neq 9 // 4 - 3$ and $29 \geq 29 / 9$
7. 'hello' * 5 > 'hello' or 'bye' < 'Bye'

5. Evaluate the following expressions involving bitwise operators:

1. $15 \& 22$
2. $15 | 22$
3. $-15 \& 22$
4. $-15 | 22$
5. ~ 15
6. ~ 22
7. $\sim\sim 20$
8. $15 ^ 22$
9. $8 \ll 3$
10. $40 \gg 3$

6. Differentiate between the following operators with the help of examples:

1. = and ==
2. / and %
3. / and //
4. * and **

7. What output will be displayed when the following commands are executed in Python shell in sequence:

```
1. >>> a = 6
>>> a == 6
>>> a < 5.9
>>> a > 5.9
2. >>> b = 7
>>> b / 6
>>> b // 6
>>> b / 4
>>> b % 4
>>> b % 7
>>> b * 2
>>> b ** 2
```

8. Construct logical expressions for representing the following conditions:

1. marks scored should be greater than 300 and less than 400.
2. Whether the value of grade is an uppercase letter.
3. The post is engineer and experience is more than four years.

9. Identify Python key words from the following list of words:

And	class	PASS	if	exec	PRINT	Not
-----	-------	------	----	------	-------	-----

10. Write Python statements for the following equations:

$$1. \text{root1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$2. \text{result} = \frac{2xy - 9y}{2xy^3} - \frac{4yx^2}{2y}$$

$$3. \text{result} = 2 \cos \frac{1}{2}(x+y) \cos \frac{1}{2}(x-y) + e^x - 1 - \frac{x}{4} + \tan x - \log(v)$$

11. How does the effect of the following two statements differ?

1. `x += x + 10`
2. `x = x + 10`

CHAPTER 2

FUNCTIONS

CHAPTER OUTLINE

- [2.1 Built-in Functions](#)
- [2.2 Function Definition and Call](#)
- [2.3 Importing User-defined Module](#)
- [2.4 assert Statement](#)
- [2.5 Command Line Arguments](#)

In this chapter, we will learn how simple statements can be put together in the form of functions to do useful tasks. We can easily deal with problems whose solutions can be written in the form of small Python programs. As a problem becomes complex, the program also increases in size and complexity, and it is impossible for a programmer to keep track of the data and control statements in the whole program. Indeed, experience has shown that the most programmers cannot keep track of more than ten statements or so at a time. Therefore, we cannot apply the crude method of attacking the whole problem at the same time. Instead, we opt for a more systematic way of problem solving by dividing the given problem into several sub-problems, finding their individual solutions in the form of functions (also called sub-programs) and integrating them to form the final program. If necessary, the sub-problems may be further divided into still smaller problems, and the process of dividing a problem into a set of problems of smaller size can be continued up to any appropriate level. Later on, solutions of the small problems are merged to form programs aimed at solving the complex problem. This approach to problem solving is called stepwise refinement method or modular approach.

to solve a problem, divide it into simpler sub-problems

In this chapter, we shall begin by giving examples of built-in functions that are already made available in Python for use by the programmers. Subsequently, we shall learn how to develop functions for the specific need of the problems at hand. The chapter concludes with a description of *assertions* that serve as an error-checking mechanism.

2.1 BUILT-IN FUNCTIONS

Built-in functions are predefined functions that are already available in Python. Recall that we have already used the built-in functions `input` and `print`. Next, we will discuss some more examples of built-in functions.

input Function

The function `input` enables us to accept an input string from the user without evaluating its value. The function `input` continues to read input text from the user until it encounters a newline, for example:

```
invoking input function to take user input

>>> name = input('Enter a name: ')

Enter a name: Alok

>>> name

'Alok'
```

The variable `name` now refers to the string value '`Alok`' entered by the user. Making use of a function is called *calling* the function, or invoking the function. In the above statement, the string '`Enter a name: '` specified within the parentheses is called an argument. So, we say that the function `input` has been called with the argument '`Enter a number:` '. Further, we say that the function returns the string entered by the user (`'Alok'`) which is subsequently assigned to the variable `name`.

eval Function

The function `eval` is used to evaluate the value of a string, for example:

```
evaluating a string
```

```
>>> eval('15')
```

```
15
```

```
>>> eval('15+10')
```

```
25
```

Composition

The value returned by a function may be used as an argument for another function in a nested manner. This is called composition. For example, if we wish to take a numeric value or an expression as an input from the user, we take the input string from the user using the function `input`, and apply `eval` function to evaluate its value, for example:

```
nested calls: output of inner function serves as input  
argument to outer function
```

```
>>> n1 = eval(input('Enter a number: '))
```

```
Enter a number: 234
```

```
>>> n1
```

```
234
```

```
>>> n2 = eval(input('Enter an arithmetic  
expression: '))
```

```
Enter an arithmetic expression: 12.0 +  
13.0 * 2
```

```
>>> n2
```

38.0

Note that the inputs 234 and $12.0 + 13.0 * 2$ were correctly evaluated as **int** and **float** values respectively.

print Function

Recall from the previous chapter that the following statement enables us to produce the output in Python.
On execution of the statement:

```
print('hello')
```

the system will output hello:

```
>>> print('hello')
```

```
hello
```

Statement `print('hello')` calls the function `print` with the argument '`hello`'. Note that the output of `print` function does not include any apostrophe marks as it displays the value of the string that comprises the sequence of characters: `hello`. Apostrophe marks are used just to tell Python where a string begins and ends and do not form part of the string. Any number of comma-separated expressions may be used while invoking a `print` function, for example:

printing multiple values in a single call to `print` function

```
>>> print(2, 567, 234)
```

```
2 567 234
```

```
>>> name = 'Raman'
```

```
>>> print('hello', name, '2 + 2 =', 2 + 2)
```

```
hello Raman 2 + 2 = 4
```

Note that when several values are included in a call to the `print` function separated by commas, they are displayed on the same line, separated by single spaces between them. It is important to point out that after printing the values of the expressions included as arguments while invoking the `print` function, the print control moves to the beginning of the next line. Thus, the output of a sequence of `print` function calls appears on separate lines. In the next example, the output of a single call to the `print` function is displayed on two lines:

```
>>> print('hello', name, '\n2 + 2 =', 2 +  
2)
```

```
hello Raman
```

```
2 + 2 = 4
```

The backslash character `\` has a special meaning. The character sequence `\n` serves as a line feed character and transfers the print control to the beginning of the next line. The character sequences like `\n` and `\t` are called escape sequences. Next, we give an example of the escape sequence `\t` which is interpreted as a tab character:

```
use of Python escape sequences
```

```
>>> print('hello', name, '\t2 + 2 =', 2 +  
2)
```

```
hello Raman 2 + 2 = 4
```

If we do not want Python to interpret an escape sequence, it should be preceded by another backslash. Another way of achieving the same thing is to use `R` or `r` before the string containing escape symbol, for example:

```
>>> print('Use \\n for newline')
```

```
Use \n for newline
```

```
>>> print(R'Use \n for newline')

Use \n for newline

>>> print(r'Use \n for newline')

Use \n for newline

>>> print('Use', R'\n', 'for newline')

Use \n for newline
```

Python also supports various other escape sequences such as \a (ASCII bell), \b (ASCII backspace), \f (ASCII form feed), and \r (ASCII carriage return). However, some of these are not supported by Python IDLE.

type Function

Values or objects in Python are classified into types or classes, for example, 12 is an integer, 12.5 is a floating point number, and 'hello' is a string. Python function type tells us the type of a value, for example:

```
determining data type

>>> print(type(12), type(12.5),
type('hello'), type(int))

<class 'int'> <class 'float'> <class
'str'>

<class 'type'>
```

round Function

The round function rounds a number up to specific number of decimal places, for example:

```
rounding to nearest value
```

```
>>> print(round(89.625,2), round(89.635),
round(89.635,0))

89.62 90 90.0

>>> print(round(34.12, 1), round(-34.63))

34.1 -35
```

When calling the function `round`, the first argument is used to specify the value to be rounded, and the second argument is used to specify the number of decimal digits desired after rounding. In a call to function `round`, if the second argument is missing, the system rounds the value of first argument to an integer, for example, `round(89.635)` yields 90.

Type Conversion

Let the variables `costPrice` and `profit` denote cost price and desired profit for a grocery item in a shop. We wish to compute the selling price for the item (Fig. 2.1).

```
1 costPrice = input('Enter cost price: ')
2 profit = input('Enter profit: ')
3 sellingPrice = costPrice + profit
4 print('Selling Price: ', sellingPrice)
```

Fig. 2.1 Program to find selling price

While executing the above script, if we enter 50 and 5 as the values for `costPrice` and `profit` respectively, the system will respond as follows:

undesirable use of + operator

Enter cost price: 50

Enter profit: 5

```
Selling Price: 505
```

Note that the `input` function considers all inputs as strings. Therefore, the value of `sellingPrice` shown above as 505 is the concatenation of the input values of `costPrice` and `profit`, i.e., '50' and '5', respectively. To convert the input strings to equivalent integers, we need to use the function `int` explicitly (Fig. 2.2):

conversion from str to int

```
1 costPrice = int(input('Enter cost price: '))
2 profit = int(input('Enter profit: '))
3 sellingPrice = costPrice + profit
4 print('Selling Price: ', sellingPrice)
```

Fig. 2.2 Program to find selling price

On executing the above script, the system will respond as follows:

```
Enter cost price: 50
```

```
Enter profit: 5
```

```
Selling Price: 55
```

We may also use `float` function if we wish to take decimal value as input from the user. The function `str` can be used to convert a numeric value to a `str` value. Next, we give some examples, illustrating the use of some functions for type conversion:

conversion from int to str

```
>>> str(123)
```

```
'123'

>>> float(123)

123.0

>>> int(123.0)

123

>>> str(123.45)

'123.45'

>>> float('123.45')

123.45

>>> int('123.45')

Traceback (most recent call last):

  File "<pyshell#3>", line 1, in <module>
    int('123.45')

ValueError: invalid literal for int() with
base 10: '123.45'

conversion from str to float
```

string incompatible for conversion

Note that last example yields an error since function `int` cannot be used for converting a string containing decimal value to its integer equivalent. We already know that the `eval` function converts the value of the argument to the appropriate type, for example:

```
>>> eval('50+5')
```

min and max Functions

The functions `max` and `min` are used to find maximum and minimum value respectively out of several values. These functions can also operate on string values. Next, we illustrate the use of these functions.

```
>>> max(59, 80, 95.6, 95.2)
95.6

>>> min(59, 80, 95.6, 95.2)
59

>>> max('hello', 'how', 'are', 'you')
'you'

>>> min('hello', 'how', 'are', 'you',
'Sir')

'Sir'
```

Note that the integer and floating point values are compatible for comparison. However, numeric values cannot be compared with string values.

operands must be compatible for comparison

pow Function

The function `pow(a, b)` computes a^b . Thus, given the side of a cube, if we want to find its volume, we may just write `pow(side, 3)`.

computing power

Random Number Generation

Imagine a two-player (say, player A and player B) game in which we want to decide who will play the first turn. Python provides a function `random` that generates a random number in the range $[0, 1]$. Python module `random` contains this function and needs to be imported for using it. Let us agree that player A will play the first turn if the random number generated falls in the range $[0, 0.5]$. Otherwise, player B will play the first turn. The following sequence of statements achieves this:

```
import random

if random.random() < 0.5:

    print('Player A plays the first turn.')

else:

    print('Player B plays the first turn.')

generating a random number in the range [0,1)
```

The `if-else` statement used here will be discussed in detail in the next chapter. In the case of multiple players, using the above approach for deciding who will play the first turn will make the program complex. The `random` module provides another function `randint` that randomly chooses an integer in the specified range; for example, `randint(1, n)` will randomly generate a number in the range 1 to n (both 1 and n included).

Functions from math Module

There are various operations such as `floor`, `ceil`, `log`, `square root`, `cos`, and `sin` that may be required in different applications. The `math` module provides these functions, among several others. In [Table 2.1](#), we describe some important mathematical functions.

Table 2.1 Functions from math module

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
floor(x)	Returns the largest integer less than or equal to x.
fabs(x)	Returns the absolute value of x.
exp(x)	Returns the value of expression $e^{**}x$.
log(x, b)	Returns the $\log(x)$ to the base b. In the case of absence of the second argument, the logarithmic value of x to the base e is returned.
log10(x)	Returns the $\log(x)$ to the base 10. This is equivalent to specifying <code>math.log(x, 10)</code> .
pow(x, y)	Returns x raised to the power y, i.e., $x^{**}y$.
sqrt(x)	Returns the square root of x.
cos(x)	Returns the cosine of x radians.
sin(x)	Returns the sine of x radians.
tan(x)	Returns the tangent of x radians.

Function	Description
acos(x)	Returns the inverse cosine of x in radians.
asin(x)	Returns the inverse sine of x in radians.

atan(x)	Returns the inverse tangent of x in radians.
degrees(x)	Returns a value in degree equivalent of input value x (in radians).
radians(x)	Returns a value in radian equivalent of input value x (in degrees).

In order to make these functions available for use in a script, we need to import the `math` module. The `import` statement serves this purpose:

```
import math
```

import a module before using it

The `math` module also defines a constant `math.pi` having value `3.141592653589793`. Next, we illustrate the use of the functions in the `math` module discussed above:

name of the module, followed by the separator dot, should precede function name

```
>>> import math

>>> math.ceil(3.4)

4

>>> math.floor(3.7)

3

>>> math.fabs(-3)

3.0
```

```
>>> math.exp(2)
7.38905609893065
>>> math.log(32, 2)
5.0
>>> math.log10(100)
2.0
>>> math.pow(3, 3)
27.0
>>> math.sqrt(65)
8.06225774829855
>>> math.cos(math.pi)
-1.0
>>> math.sin(math.pi/2)
1.0
>>> math.tan(math.pi/4)
0.9999999999999999
>>> math.acos(1)
0.0
>>> math.asin(1)
1.5707963267948966
>>> math.atan(1)
0.7853981633974483
```

```
>>> math.degrees(math.pi)
```

```
180.0
```

```
>>> math.radians(180)
```

```
3.141592653589793
```

Note that the name of a function in a module is preceded by the name of the module and the separator '.'.

Complete List of Built-in Functions

If we want to see the complete list of built-in functions, we can use the built-in function `dir` as

`dir(__builtins__)`. To know the purpose of a function and how it is used, we may make use of the function `help`, for example, to display help on `cos` function in the `math` module, we may give the following instructions:

getting help on a function

```
>>> import math
```

```
>>> help(math.cos)
```

Help on built-in function `cos` in module
`math`:

`cos(...)`

`cos(x)`

Return the cosine of `x` (measured in radians).

2.2 FUNCTION DEFINITION AND CALL

We have already mentioned that the functions provide a systematic way of problem solving by dividing the given problem into several sub-problems, finding their individual solutions, and integrating the solutions of individual problems to solve the original problem.

Suppose we wish to print a picture that consists of a triangle followed by a square as shown in Fig. 2.3.

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

Fig. 2.3 Picture

A program segment to achieve this would require the following steps:

outline of a solution

print a triangle

print a blank line

print a square

If we replace the above outline with Python code, our job would be done. We call this script `picture` and store it in the file `picture.py` as per Python convention (Fig. 2.4).

```
01 def main():
02     # To print a triangle
03     print(' *')
04     print(' ***')
05     print(' *****')
06     print('*****')
07
08     # To print a blank line
09
10    print()
11
12    # To print a square
13    print('* * * *')
14    print('* * * *')
15    print('* * * *')
16    print('* * * *')
```

Fig. 2.4 Program to print a triangle followed by square
(picture.py)

In Fig. 2.4 (and elsewhere in the book), line numbers are not part of Python program but used only for the purpose of explanation. In this program, line number 2 begins with # (hash). This line is a comment. A comment is a non-executable text in the program that is not processed by the system. Comments are ignored by the Python interpreter as if they are not present. They are placed to make the intent of Python code clear to the reader. In general, if # appears anywhere in a line in the Python program, rest of the line is ignored by the system. For example:

comments enhance readability of the code

```
# side of an equilateral triangle
```

single line comments start with #

is a comment in the statement,

```
side = 6 # side of an equilateral triangle
```

Another way of writing a comment is to use a docstring (documentation string). As mentioned earlier, a docstring may extend to several lines. In Fig. 2.4, note that the function definition begins in line 1 with keyword `def`, followed by the name of the function `main`, empty parenthesis, and a colon. The following statements (lines 2–16) that form the body of the function `main` do not begin in column number 1, just below `def`, but begin with four spaces. This is called indentation. We have used four spaces for indentation as per advice in Google's coding guidelines. However, one may choose a different number of spaces, say, two or three, for indentation. But it is desirable to be consistent, across all the programs that we write. In general, the syntax for a function definition is as follows:

multi-line comment

Python insists on strict indentation rules

```
def function_name
(comma_separated_list_of_parameters) :
    statements
```

syntax for function definition

Python also requires that a *function_name* should not be a Python keyword. The list of parameters (if any) specifies the information required for using the function. The statements inside the function have to be indented from the left margin as shown in Fig. 2.4. It is important to emphasize that apart from the fact that indented code looks elegant, it is also a requirement of Python that the code following a colon must be indented.

Python code following colon must be indented, i.e., shifted right

Having developed the function `main` in the script `picture`, we would like to execute it. To do this, we need to invoke the function `main`. This can be done by performing the following two steps:

1. In Run menu, click on the option Run Module.
2. Using Python shell, invoke (call) the function `main` by executing the following command:
`>>> main()`

We can eliminate the need to call function `main` explicitly from the shell, by including in the script `picture`, the following call to function `main`:

```
if __name__=='__main__':
    main()
```

invoking the function `main` in the script

The revised script is shown in Fig. 2.5. Every Python module has a built-in variable called `__name__` containing the name of the module. When the module itself is being run as the script, this variable `__name__` is assigned the string '`__main__`' designating it to be a `__main__` module. The `__main__` module, i.e. script `picture` in this case (Fig. 2.5), comprises:

built-in variable `__name__`

1. the definition of function `main`
2. an `if` statement

When a script is executed, Python creates a run-time environment, called global frame. In the script `picture`, when the definition of the function `main` is encountered, Python makes a note of its definition in the global frame. Next, on encountering the `if` statement, Python checks whether the name of the current module is `__main__`. This being true, the expression `__name__ == '__main__'` evaluates as `True`, and the function `main` is invoked. The check `__name__ == '__main__'` is performed to prevent the accidental calling of a function from an imported module. The `if` statement is used in a script to specify the starting point of execution for the module being run. So, if we have a code that should only be executed when the module is run, and not when it is imported, we need to execute the code using `if` clause as shown in lines 18-19. For the module being imported, variable `__name__` contains the name of imported module.

you may read this paragraph casually for now, and come back to it later when you have some experience in programming

```
01 def main():
02     # To print a triangle
03     print(' *')
04     print(' ***')
05     print(' *****')
06     print('*****')
07
08     # To print a blank line
09
10    print()
11
12    # To print a square
13    print('* * * *')
14    print('* * * *')
15    print('* * * *')
16    print('* * * *')
17
18 if __name__=='__main__':
19     main()
```

Fig. 2.5 Program to print a triangle followed by square
(picture.py)

Now, the statements in the `main` function are executed in a sequence. Thus, a triangle and a square, separated by a blank line get printed. In Fig. 2.5, we note that printing a triangle and printing a square are independent activities, having nothing to do with each other. We can make the above program more elegant by first developing independent functions to print a square

and a triangle and then making use of these functions in the `main` function to print a picture that comprises a triangle and a square separated by a blank line. In Fig. 2.6, we give the modified script `picture1`.

developing functions for printing a square and triangle

The script in Fig. 2.6 works as follows: Python makes a note of the definition of functions `triangle`, `square`, and `main` as it sees their definitions. As described above, on the execution of the `if` statement (line 22), the control is transferred to the function `main` (line 15). Line 16 being a comment, is ignored by Python. In line 17, the function `triangle` is called, and the body of function `triangle` gets executed. As the `main` function calls the function `triangle`, it is said to be the *caller* function, and the function `triangle` that is called is said to be the *callee* function or *called* function.

function `main` serves as a caller function for the called functions `triangle` and `square`

```
01 def triangle():
02     # To print a triangle
03     print('*')
04     print(' **')
05     print(' ****')
06     print('*****')
07
08 def square():
09     # To print a square
10     print('* * * *')
```

```
11     print('* * * *')
12     print('* * * *')
13     print('* * * *')
14
15 def main():
16     # To print a triangle
17     triangle()
18     print()
19     # To print a square
20     square()
21
22 if __name__=='__main__':
23     main()
24 print('\nEnd of program')
```

Fig. 2.6 Program to print a triangle followed by square
(picture1.py)

Again, line 2 of the function `triangle` being a comment, Python ignores it. Next, lines 3–6 are executed in sequence. On completing execution of the function `triangle`, the control is transferred to the statement immediately following the one that called the function `triangle`, i.e. at line 18 that prints a blank line. In line 20, we call the function `square`. On execution of the body of function `square`, the control returns to `main` function (line 21). As there are no more statements to be executed in the `main`, the control now moves to the line following the statement that invoked the `main` function, i.e., call to `print` function at line 24 in the global frame, which now executes, and the program comes to an end.

```
triangle
```

```
square
```

invoking the function `triangle`

invoking the function `square`

We would like to emphasize that a function definition appears in a program as a provision to do something and can come into action only when invoked (called). So, defining a function has no effect until it is invoked, for example, the function `triangle` in Fig. 2.7 will produce no output as the function `triangle` has only been defined and not invoked.

a function definition has no effect unless it is invoked

Unlike the scripts `picture` (Fig. 2.5) and `picture1` (Fig. 2.6), often, the caller and the called functions need to share some information between themselves. To demonstrate this, we develop a program to print the area of a rectangle that takes `length` and `breadth` of the rectangle as inputs from the user.

```
01 def triangle():
02     # To print a triangle
03     print(' *')
04     print(' ***')
05     print(' *****')
06     print('*****')
07
08 def main():
09     # To print a triangle
10     print('Triangle')
11
12 if __name__ == '__main__':
13     main()
```

Fig. 2.7 The script shows that function `triangle` is not being invoked

We can describe this task of computing area of a rectangle as a sequence of three steps:

1. take length and breadth of the rectangle as inputs
2. compute the area of the rectangle
3. print the area of the rectangle

Such an informal description of a program (to be developed) is called a pseudocode. It serves as the basis of the program to be developed. When solving a complex problem, a pseudocode serves as the first hand description of the solution strategy. In the above description, we have omitted how to compute the area of a rectangle. Let us now refine step 2 for the computing

area of a rectangle using the following pseudocode. To compute the area of a rectangle, we develop a function that accepts length and breadth as arguments, computes the area, and outputs it as return value. We may describe this using the following pseudocode:

```
areaRectangle:  
inputs: length, breadth  
output: area  
computations:  
    area <- length * breadth
```

The above mentioned process is known as stepwise refinement process, for example, in the above description, we refined step 2 of the pseudocode. In this book, we have described the solution strategy in detail as part of the text. So, we have avoided giving a formal pseudocode while solving the programming problems. Let us now develop a function `areaRectangle` to compute the area of a rectangle. To compute the area of a rectangle, this function would need to know the length and breadth of the rectangle. The main function will communicate this information to function `areaRectangle` at the time of invoking it. Further, the function `areaRectangle` (Fig. 2.8) would need to communicate to the caller function `main`, the area computed by it.

computing area of a rectangle

```
1 def areaRectangle(length, breadth):  
2     '''  
3         Objective: To compute the area of rectangle  
4         Input Parameters: length, breadth - numeric value  
5         Return Value: area - numeric value  
6     '''  
7     area = length * breadth  
8     return area
```

Fig. 2.8 Function `areaRectangle`

Line 1 of the function definition includes the name of the function (i.e., `areaRectangle`). The names `length` and `breadth` within parenthesis in the function are called parameters. Lines 2–6 constitute a docstring describing the function. Line 3 describes the overall objective of the function. Line 4 describes each of the inputs to the function and the type of value it takes. Line 5 indicates that the function would return the area of a rectangle to the calling function. Area of the rectangle is computed in line 7. Finally, line 8 is a `return` statement that returns the area computed in line 7 to the caller function `main`. In general, the `return` statement returns the value of the expression following the keyword `return` to the caller function. If there is no `return` statement in a function, the function returns the value `None` to the caller function on the execution of the last statement of the function. The value being returned may be assigned to a variable.

return statement returns the value of expression following the `return` keyword in the absence of the `return` statement, default value `None` is returned

In Fig. 2.9, we give the complete script to compute the area of a rectangle that takes length and breadth of the rectangle as inputs from the user. The variables and

expressions whose values are passed to the called function are called *arguments*. At the point of call to function `areaRectangle` in line 19, values of arguments `lengthRect` and `breadthRect` are passed to parameters `length` and `breadth`, respectively. These values are used inside the function `areaRectangle`, for computing `area`. While the parameters `length` and `breadth` are called formal parameters, or dummy arguments, `lengthRect` and `breadthRect` used for invoking the function `areaRectangle` are called actual parameters, or arguments. The arguments in call to the function must appear in the same order as that of parameters in the function definition.

arguments: variables/expressions whose values are passed to called function parameters: variables/expressions in function definition which receives value when the function is invoked

Suppose values entered by the user for variables `lengthRect` and `breadthRect` are 5 and 4, respectively.

arguments must appear in the same order as that of parameters

```
01 def areaRectangle(length, breadth):  
02     '''  
03     Objective: To compute the area of rectangle  
04     Input Parameters: length, breadth - numeric value  
05     Return Value: area - numeric value  
06     '''  
07     area = length * breadth  
08     return area  
09  
10 def main():
```

```
10  Q:\PY\area.py
11  """
12  Objective: To compute the area of rectangle based on user input
13  Input Parameter: None
14  Return Value: None
15  """
16  print('Enter the following values for rectangle:')
17  lengthRect = int(input('Length : integer value: '))
18  breadthRect = int(input('Breadth : integer value: '))
19  areaRect = areaRectangle(lengthRect, breadthRect)
20  print('Area of rectangle is', areaRect)
21
22 if __name__ == '__main__':
23     main()
24 print('\nEnd of program')
```

Fig. 2.9 Program to find area of a rectangle (area.py)

Enter the following values for rectangle:

Length : integer value: 5

Breadth : integer value: 4

Area of rectangle is 20

End of program

Figure 2.10 shows how the values of the arguments `lengthRect` and `breadthRect` are passed to the parameters `length` and `breadth` at the point of call to function `areaRectangle`. The figure showing the correspondence between arguments and parameters is sometimes called a memory map.

We could have used `length` and `breadth` in both the functions `areaRectangle` and `main` to denote length and breadth of the rectangle. On returning the value of `area` to the function `main` in line 8, the function `areaRectangle` terminates, and the control returns to line 19 in the script and the value returned by the function `areaRectangle` is assigned to the variable `areaRect`.

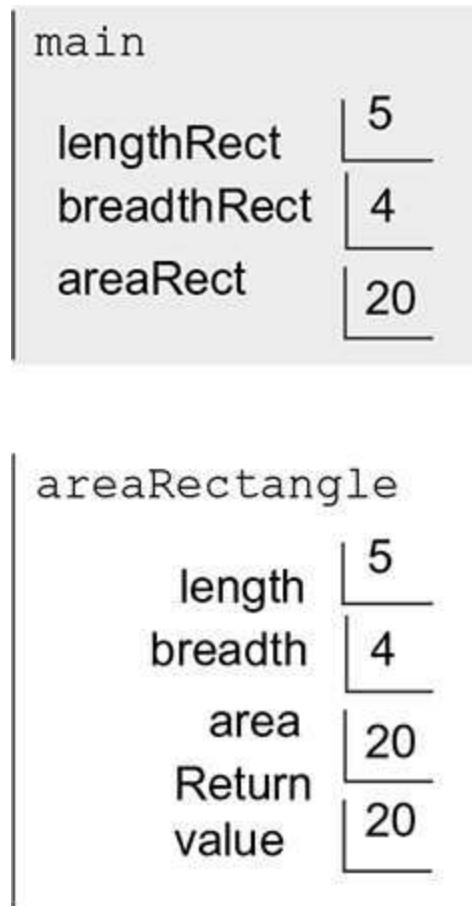


Fig. 2.10 Call to function `areaRectangle`

correspondence between arguments and parameters

Next, suppose we wish to compute the area of a square. To do this, we may use the function `areaRectangle` defined above as follows:

`areaRectangle(side, side)`

use of function `areaRectangle` for computing area of the square

However, while using function `areaRectangle` to find the area of a square, we assume that the user is aware of the fact that square is a rectangle with equal length and breadth. A better way to do this would be to define a function `areaSquare` that accepts side of the square as an input parameter. The user interface for the function `areaSquare` makes no assumption about the relationship between `areaSquare` and `areaRectangle`. However, there is nothing wrong in using the function `areaRectangle` in the body of the function `areaSquare` (Fig. 2.11). Indeed, this illustrates a software engineering principle of reusing the already developed functions.

```
1 def areaSquare(side):
2     """
3     Objective: To compute area of square
4     Input Parameter: side - numeric value
5     Return Value: area - numeric value
6     """
7     area = areaRectangle(side, side)
8     return area
```

Fig. 2.11 Function `squareArea`

In light of above discussion, we present a complete script for computing the area of a rectangle and a square (Fig. 2.12).

```
01 def areaRectangle(length, breadth):
02     """
03     Objective: To compute area of rectangle
04     """
```

```

04     Input Parameters: length, breadth - numeric value
05     Return Value: area - numeric value
06     """
07     area = length * breadth
08     return area
09
10 def areaSquare(side):
11     """
12     Objective: To compute area of square
13     Input Parameter: side - numeric value
14     Return Value: area - numeric value
15     """
16     area = areaRectangle(side, side)
17     return area
18
19 def main():
20     """
21     Objective: To compute area of rectangle and square based on
22     user input
23     Input Parameter: None
24     Return Value: None
25     """
26     print('Enter the following values for rectangle:')
27     lengthRect = int(input('Length : integer value: '))
28     breadthRect = int(input('Breadth : integer value: '))
29     areaRect = areaRectangle(lengthRect, breadthRect)
30     print('Area of rectangle is', areaRect)
31     sideSqr = int(input('Enter side of square: integer
32     value: '))
33     areaSqr = areaSquare(sideSqr)
34     print('Area of square is', areaSqr)
35
36 if __name__=='__main__':
37     main()

```

Fig. 2.12 Program to find area of square and rectangle (area.py)

Fruitful Functions vs Void Functions

A function that returns a value is often called a fruitful function, for example, the built-in function `sin`, and `abs`, and the functions `areaRectangle`, and `areaSquare` defined earlier in the chapter. A function that does not return a value is called a void function, for example, the built-in function `print`, and the functions `triangle`, and `square`, defined earlier in the chapter.

Function help

Recall that function `help` can be used to provide a description of built in functions. Function `help` can also be used to provide description of the function defined by user. All one needs to do is to add a multi-line comment. Function `help` works by retrieving the contents specified using multi-line comments. For example, consider the function given in Fig. 2.12.

help on user defined functions

On executing the command

`help(areaRectangle)`, contents specified using multi-line comment will be retrieved.

```
>>> help(areaRectangle)
```

```
Help on function areaRectangle in module  
__main__:
```

```
areaRectangle(length, breadth)
```

Objective: To compute area of rectangle

Input Parameters: length, breadth -
numeric value

Return Value: area - numeric value

function `help` retrieves first multi-line comment from the function definition

If there are more than one multi-line comments, only the first multi-line comment is displayed.

Default Parameter Values

The function parameters may be assigned initial values also called default values as shown in Fig. 2.13. When the function `areaRectangle` is called without specifying the second argument `breadth`, default value 1 is assumed for it, for example:

```
>>> areaRectangle(5)
```

```
5
```

```
01 def areaRectangle(length, breadth = 1):  
02     '''  
03     Purpose: To compute area of rectangle  
04     Input Parameters:  
05         length - int  
06         breadth (default 1) - int  
07     Return Value: area - int  
08     '''  
09     area = length * breadth  
10     return area
```

Fig. 2.13 Default parameter values

However, if the default parameters are specified in a function call, the default values are ignored, for example:

function call uses default value if value for a parameter is not provided

```
>>> areaRectangle(5, 2)
```

10

It is important to mention that the default parameters must not be followed by non-default parameters, for example:

non-default arguments should not follow default arguments
in a function definition

```
>>> def areaRectangle(length = 10,  
breadth):  
  
    return length * breadth
```

SyntaxError: non-default argument follows
default argument

Keyword Arguments

In the programs developed so far, the order of arguments always matched the parameters in the function definition. However, Python allows us to specify arguments in an arbitrary order in a function call, by including the parameter names along with arguments. The arguments specified as

parameter_name = value

syntax for keyword arguments

are known as keyword arguments. For example, in the following call to the function `areaRectangle`, the order of arguments is different from the one in the function definition.

```
areaRect = areaRectangle(breadth = 2,  
length = 5)
```

Indeed, in situations involving a large number of parameters, several of which may have default values,

keyword arguments can be of great help, for example:

```
>>> def f(a = 2, b =3, c = 4, d =5, e = 6,
f = 7, g = 8, h = 9):
    return a + b + c + d + e + f + g + h

>>> f(c = 10, g = 20)
```

62

2.3 IMPORTING USER-DEFINED MODULE

Recall from Section 2.1 that to access functions such as `floor`, `ceil`, and `log` given in Table 2.1, we need to import `math` module that includes the definitions of these functions. Similarly, to access a function from a user-defined module (also known as program or script that may comprise functions, classes, and variables), we need to import it from that module. To ensure that the module is accessible to the script, we are currently working on, we append to the system's path, the *path* to the folder containing the module. Once this is done, we can import the module by using an instruction like:

specifying system path

```
import name-of-the-module
```

syntax for importing a module

Once this is done, we can access all the functions defined in it by using the following notation: module name, followed by a dot, followed by the function name as shown in line 14 in Fig. 2.14.

```
01 import sys
02 sys.path.append('F:\PythonCode\Ch02')
03 import area
04
05 def main():
06     """
07     Purpose: To compute area of floor
08     Input Parameter: None
09     Return Value: None
10    """
11    print('Enter the following values for floor')
12    length = int(input('Length: '))
13    breadth = int(input('Width: '))
14    print(area.areaRectangle(length, breadth))
15
16 if __name__=='__main__':
17     main()
```

Fig. 2.14 Program to find area of floor (`floorArea.py`)

The script in Fig. 2.14 may be used from IDLE as follows:

Enter the following values for floor:

Length: 500

Width: 400

200000

2.4 ASSERT STATEMENT

The `assert` statement is used for error checking.

Suppose, we want to compute the percentage of marks obtained in a subject. It may happen that value of variables `maxMarks` and `marks` entered by the user may not be in proper range, for example, `marks` may be negative or greater than the `maxMarks`. Hence, we need to make sure that inputs provided by the user are in the correct range. For this purpose, we make use of an `assert` statement (Fig. 2.15, lines 18 and 20) that has the following form:

use of assert statement for error checking

`assert condition`

```
01 def percent(marks, maxMarks):  
02     '''  
03     Objective: To find percentage of marks obtained in a subject  
04     Input Parameters: marks, maxMarks - float  
05     Return Value: percentage - float  
06     '''  
07     percentage = (marks / maxMarks) * 100  
08     return percentage  
09  
10 def main():  
11     '''  
12     Objective: To find percentage of marks obtained in a subject  
13     based on user input
```

```
13     based on user input
14
15     Input Parameter: None
16
17     Return Value: None
18
19     """
20
21     maxMarks = float(input('Enter maximum marks: '))
22     assert maxMarks >=0 and maxMarks <=500
23
24     marks = float(input('Enter marks obtained: '))
25     assert marks >=0 and marks <=maxMarks
26
27     percentage = percent(marks, maxMarks)
28
29     print('Percentage is : ', percentage)
30
31
32 if __name__=='__main__':
33     main()
```

Fig. 2.15 Program to find percentage (percent.py)

If the assertions in lines 18 and 20 hold, the function displays percentage as the output. However, if these assertions fail to hold the system responds with an assertion error, for example:

```
Enter maximum marks: 150
Enter marks obtained: 155
Traceback (most recent call last):
  File "F:/PythonCode/Ch02/percent.py",
line 25,
    in <module>
        main()

  File "F:/PythonCode/Ch02/percent.py",
line 20,
```

```
in main

    assert marks >=0 and marks <=maxMarks

AssertionError
```

! 2.5 COMMAND LINE ARGUMENTS

Let us now revisit the program to find area of the rectangle. So far, we have executed the Python programs in Python IDLE. However, we may choose to run the Python script from the command line interface. For example, the script `area` (Fig. 2.9) may be run from the command prompt by executing the following steps:

1. open the directory (say, F:\PythonCode\Ch02) containing the file `area.py`
2. open command prompt window using an option in context menu (opened using shift+right click)
3. execute the command: `python area.py`

```
F:\PythonCode\Ch02>python area.py
```

```
Enter the following values for rectangle:
```

```
Length : integer value: 20
```

```
Breadth : integer value: 10
```

```
Area of rectangle is 200
```

```
End of program
```

So far, we have been taking inputs from the user in an interactive manner. However, we may also take the inputs as command line arguments while executing a script from the command prompt as follows:

```
F:\PythonCode\Ch02>python areal.py 20 10
```

```
Area of rectangle is 200
```

```
End of program
```

In above command, 20 and 10 serve as inputs for the script `areal`. Whenever, we execute a script from

command line, it takes name of the script as the first argument followed by other input arguments (if any) in string form and stores them in the list (discussed later) `sys.argv`. We access the arguments stored in `argv` using indexes `argv[0]`, `argv[1]`, `argv[2]`, etc.

Examine the script `areal` (Fig. 2.16). Since, the number of arguments including the script name should be three, we ensure this using condition `len(sys.argv) == 3`. We also assume that in the command

usually whitespace(s) are used for separating the command line arguments from each other

```
python areal.py 20 10
```

command line arguments length and breadth that follow the script name (`areal.py`) are stored in `argv[1]` and `argv[2]` respectively.

```
01 import sys
02 def areaRectangle(length, breadth):
03     """
04     Objective: To compute the area of rectangle
05     Input Parameters: length, breadth - numeric value
06     Return Value: area - numeric value
07     """
08     area = length * breadth
09     return area
10
11 def main():
12     """
13     Objective: To compute the area of rectangle based on user input
14     taken as command line arguments
15     Input Parameter: None
```

```

16     Return Value: None
17
18     """
19
20     if len(sys.argv) == 3:
21         lengthRect = int(sys.argv[1])
22         breadthRect = int(sys.argv[2])
23         areaRect = areaRectangle(lengthRect, breadthRect)
24         print('Area of rectangle is', areaRect)
25     else:
26         print('Unexpected number of command line arguments!')
27
28 if __name__=='__main__':
29     main()
30
31 print('\nEnd of program')

```

Fig. 2.16 Program to compute area of rectangle using command line inputs (area1.py)

SUMMARY

1. Functions provide a systematic way of problem solving by dividing the given problem into several sub-problems and finding their individual solutions.
2. Built-in functions are predefined functions that are provided by the Python environment.
 1. `input` function enables us to accept an input string from the user without evaluating its value.
 2. `print` function enables us to produce output in Python.
 3. `eval` function is used to evaluate the value of a string.
 4. `type` function tells us the type of value.
 5. `round` function rounds a number up to a specified number of decimal places.
 6. `int` function can be used to convert a value to its corresponding integer.
 7. `str` function can be used to convert a value to its corresponding string.
 8. `max` and `min` functions are used for finding maximum and minimum out of several values, respectively.
 9. `pow` function is used to find some power of a number.

10. `random` function of the module `random` is used for generating a random number in the range $[0, 1]$.
 11. `randint(1, n)` function of the module `random` is used for generating a random number in the range $[1, n]$.
 12. `math` module contains several mathematical functions such as `log`, `ceil`, and `sin`.
3. A comment is a non-executable text in the program that is ignored by the Python interpreter as if it is not present. The comments are used to make the intent of Python code clear to the reader. A single-line comment starts with character `#`, whereas a multi-line comment may be specified using docstring.
4. The code following colon must be indented i.e. shifted to the right.
5. The variables that appear in function definition enclosed in parentheses immediately following the name of the function are called formal parameters or dummy parameters. When the function is called, they are replaced by the values called arguments. The arguments in call to the function must appear in the same order as that of the parameters in the function definition.
6. A user-defined module is also known as program or script that may comprise functions, classes, and variables.
7. The function parameters may be assigned initial values, called default values. All non-default parameters must appear before the default parameters.
8. Python allows us to specify arguments in an arbitrary order in a function call, by including the parameter names along with arguments. An argument specified as
`parameter_name = value`
is known as keyword argument.
9. `import` statement is used for importing a module that may be user-defined or built-in.
10. `assert` statement is used for error checking. If the condition specified in an `assert` statement fails to hold, Python responds with an assertion error.

EXERCISES

1. What will be the output produced by each of the following function calls:

math.ceil(65.65)	math.ceil(65.47)
math.fabs(-67.58)	math.fabs(3)
math.exp(2.7)	math.log(45,2)
math.log10(1000)	math.pow(4,1/2)
math.sqrt(121)	math.radians(30)
math.degrees(math.pi/2)	

2. Give the range in which value of variable x may lie on execution of the following statements:

```
import random
x = random.random() + 5
```

3. Evaluate the following expressions using Python shell.
Assume that ASCII coding scheme is used for character data.

abs(-5.4)	abs(15)	chr(72)
round(-24.9)	float(57)	complex('1+2j')
divmod(5,2)	float(57)	pow(9,2)
max(97, 88, 60)	min(55, 29, 99)	
max('a', 'b', 'AB')		

4. Develop Python functions to produce the following outputs:

```
*
 *
 *
1. *
 *
 *
 *
 *
 *
```

\$ \$ \$ \$ \$
 \$ \$
 2. \$ \$
 \$ \$
 \$ \$ \$ \$ \$

5. Consider the following function:

```
def nMultiple(a = 0, num = 1):
    return a * num
```

What will be the output produced when the following calls are made:

- 1. nMultiple(5)
- 2. nMultiple(5, 6)
- 3. nMultiple(num = 7)
- 4. nMultiple(num = 6, a = 5)
- 5. nMultiple(5, num = 6)

6. Study the program segments given below. Give the output produced, if any.

```
1. def test(a, b):
    a = a+b
    b = a-b
    a = a-b
    print('a = ', a)
    print('b = ', b)
    test(5,8)
2. def func():
    pass
    a = func()
    print(a)
```

7. Write a function `areaTriangle` that takes the lengths of three sides: `side1`, `side2`, and `side3` of the triangle as the input parameters and returns the area of the triangle as the output. Also, assert that sum of the length of any two sides is greater than the third side. Write a function `main` that accepts inputs from the user interactively and computes the area of the triangle using the function `areaTriangle`.

8. Create the following scripts `importedModule` (Fig. 2.17) and `mainModule` (Fig. 2.18) in the working directory, execute the script `mainModule` and justify the output.

```
01 def test1():
02     print('test1 in imported module')
03
04 def test2():
05     print('test2 in imported module')
06
07 test1()
08 test2()
```

Fig. 2.17 (importedModule.py)

```
01 import importedModule
02 print('hello')
```

Fig. 2.18 (mainModule.py)

9. Rewrite the code in question 7 so that it takes inputs as command line arguments.

[!]
This section may be skipped on first reading without loss of
continuity.

CHAPTER 3

CONTROL STRUCTURES

CHAPTER OUTLINE

3.1 if Conditional Statement

3.2 Iteration (for and while Statements)

The functions that we have developed so far had the property that each instruction in a function was executed exactly once. Further, the instructions in these functions were executed in the order in which they appeared in the functions. Such functions are called straight line functions. However, real life problems would usually require non-sequential and repetitive execution of instructions. Python provides various control structures for this purpose. In this chapter, we will study the following control structures with suitable examples: `if`, `for`, and `while`.

control structures are used for non-sequential and repetitive execution of instructions

3.1 IF CONDITIONAL STATEMENT

Suppose a teacher grades the students on a scale of 100 marks. To pass the examination, a student must secure pass marks (say, 40). Before announcing the results, the teacher decides to moderate the results by giving maximum of two grace marks. Thus, if the student has scored 38 or 39 marks, he or she would be declared to have scored 40 marks. Let us see how script `moderate` (Fig. 3.1) achieves this.

First, we set `passMarks` equal to 40 in the function `main` (Fig. 3.1). The next statement prompts the user to enter the marks obtained by a student. As `marks` entered

by the user is of type `str`, we transform it to an integer quantity `intMarks` using the function `int`. In line 23, we invoke the function `moderate` defined in lines 1–12 with the arguments `intMarks` and `passMarks`. Inside the function `moderate` (line 10), we check whether the value of the input parameter `marks` is less than `passMarks` by one or two using the condition `marks == passMarks-1` or `marks == passMarks-2`. The assignment statement in line 11 gets executed only if the condition evaluates to `True`. Next, the function `moderate` returns `marks` (possibly modified) to the main function. The value returned by the function `moderate` is assigned to the variable `moderatedMarks` (line 23). Finally, we print `moderatedMarks` in line 24. If we run the script in Fig. 3.1, the system responds by asking the `marks` and displays `moderatedMarks`.

```
01 def moderate(marks, passMarks):
02     """
03     Objective: To moderate result by maximum 1 or 2 marks to
04     achieve passMarks
05     Input Parameters:
06         marks - int
07         passMarks - int
08     Return Value: marks - int
09     """
10    if marks == passMarks-1 or marks == passMarks-2:
11        marks = passMarks
12    return marks
13
14 def main():
15     """
16     Objective: To moderate marks if a student just misses pass
17     marks
```

```
17 Input Parameter: None
18 Return Value: None
19 ''
20 passMarks = 40
21 marks = input('Enter marks: ')
22 intMarks = int(marks)
23 moderatedMarks = moderate(intMarks, passMarks)
24 print('Moderated marks:', moderatedMarks)
25
26 if __name__ == '__main__':
27     main()
```

Fig. 3.1 Program to moderate the results by giving maximum of two grace marks (moderate.py)

marks to be updated to passMarks based on a condition

>>>

Enter marks: 38

Moderated marks: 40

>>>

Enter marks: 39

Moderated marks: 40

>>>

Enter marks: 40

Moderated marks: 40

In Fig. 3.2, we give a representation of the function `moderate` using a flowchart.

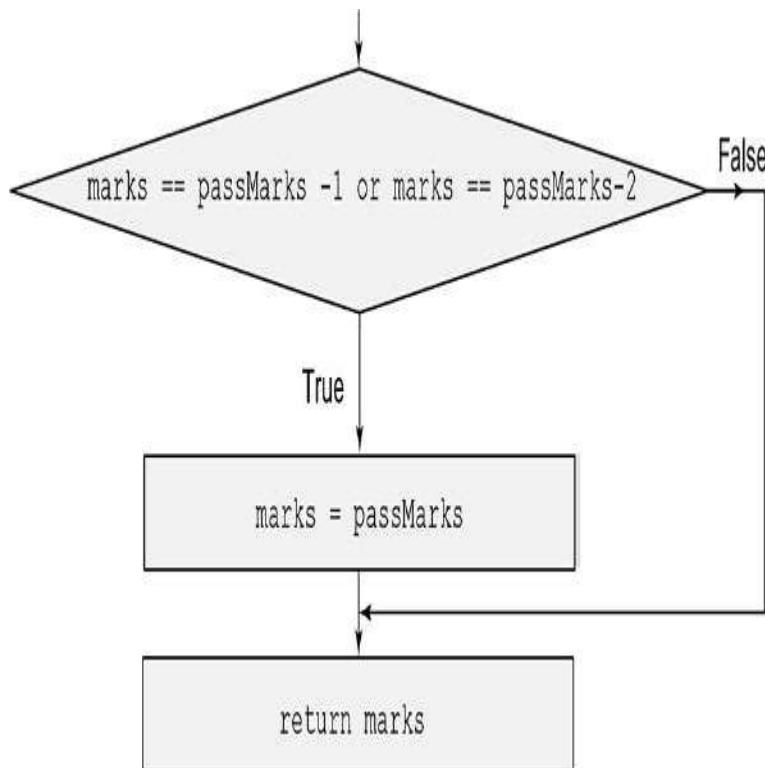


Fig. 3.2 Flow diagram of function `moderate`

`moderate` marks only if the student is 1 or 2 marks short of `passMarks`

General Form of if Conditional Statement

The general form of `if` conditional statement is as follows:

syntax for `if` conditional statement

`if < condition >:`

*< Sequence S of statements to be
executed >*

Here, *condition* is a Boolean expression, which is evaluated when the `if` statement is executed. If this condition evaluates to `True`, then the sequence *S* of statements is executed, and the control is transferred to the statement following the `if` statement. However, if the Boolean expression evaluates to `False`, the sequence *S* of statements is ignored, and the control is immediately transferred to the statement following the `if` statement. The flow diagram for execution of the `if` statement has been shown in Fig. 3.3.

statements following `if` clause are executed only if the condition in the clause evaluates to `True`

Note that *<Sequence S of statements to be executed>* following the colon is indented (i.e., shifted right). Next, suppose we want to restrict the use of a system. To keep the things simple, all the valid users are assigned a common password and password validation is the only task this program performs. We can verify the password entered by a user against the password stored in the system using an `if` statement. If both the passwords match, the program prints a welcome message, else an error message indicating password mismatch is displayed. Let us see how the script in Fig. 3.4 achieves this.

flow-diagram of `if` statement

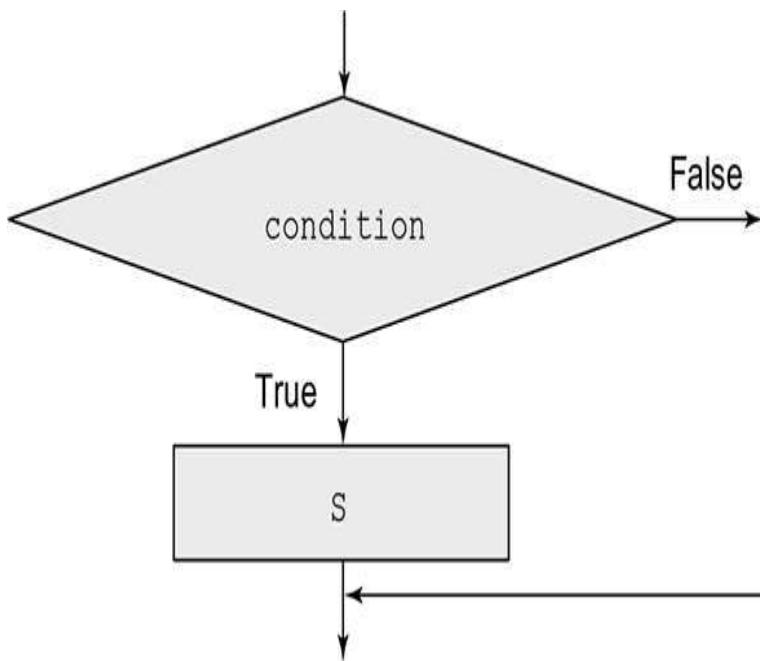


Fig. 3.3 Flow diagram of `if` statement

```

01 def authenticateUser(password):
02     """
03     Objective: To authenticate user and allow access to system
04     Input Parameter: password - string
05     Return Value: message - string
06     """
07     if password == 'magic':
08         message = ' Login Successful !!\n Welcome to system.'
09     if password != 'magic':
10         message = ' Password mismatch !!\n '
11     return message
12
13 def main():
14     """

```

```

15    Objective: To authenticate user
16    Input Parameter: None
17    Return Value: None
18    """
19    print(' \t LOGIN SYSTEM ')
20    print('-----')
21    password = input(' Enter Password: ')
22    message = authenticateUser(password)
23    print(message)
24
25 if __name__=='__main__':
26     main()

```

Fig. 3.4 Program to authenticate user and allow access to system
(authenticate.py)

The function `main` in Fig. 3.4 prompts the user to enter the password, which is stored in the variable `password`. Next, the function `authenticateUser` beginning line 1 is called in line 22 with the argument `password`. Inside the function `authenticateUser`, the value of this input parameter is matched against the password 'magic' (already known to the system), using the condition `password == 'magic'` (line 7). The assignment statement in line 8 gets executed only if the condition holds `True`. Similarly, if the condition specified in line 9 holds `True`, the assignment statement in line number 10 gets executed. String value returned by the function `authenticateUser` (line 11) is assigned to the variable `message` in line 22. Finally, we print a message indicating whether the user is authorized to use the system in line 23.

If we run the script in Fig. 3.4, the system responds by asking for the password and displays a suitable message indicating successful or unsuccessful login attempt.

```
LOGIN SYSTEM
=====
Enter Password: hello
Password mismatch !!

LOGIN SYSTEM
=====
Enter Password: magic
Login Successful !!
Welcome to system.
```

In Fig. 3.5, we represent the function authenticateUser using a flowchart:

flow chart to validate the password entered by the user

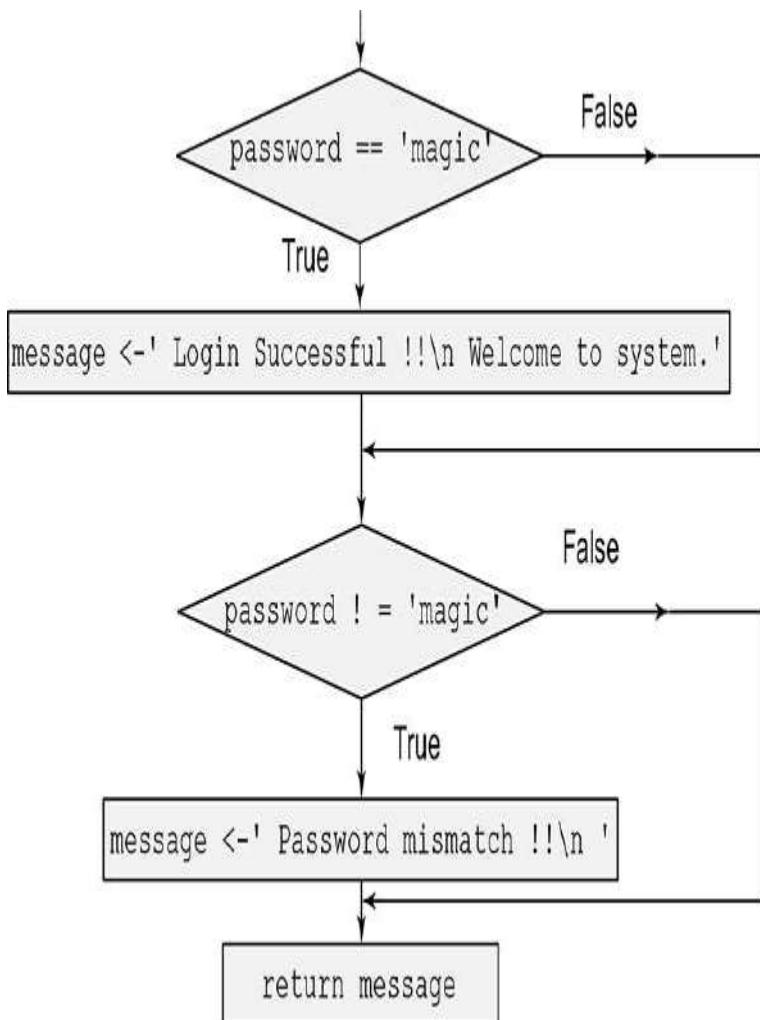


Fig. 3.5 Flow diagram of function `authenticateUser`

You must have noted in the previous program that there were two `if` statements for checking the correctness and incorrectness of the password entered by a user. However, it is obvious that if `password` is not correct, it must be incorrect. Indeed, Python provides an `if-else` statement to handle such situations. Let us have a look at the modified script given in Fig. 3.6.

```

01 def authenticateUser(password):
02     """
03     Objective: To authenticate user and allow access to system
04     Input Parameter: password - string

```

```

01     Input Parameter: password - string
02
03     Return Value: message - string
04
05     """
06
07     if password == 'magic':
08         message = ' Login Successful !!\n Welcome to system.'
09     else:
10         message = ' Password mismatch !!\n '
11
12     return message
13
14
15 def main():
16     """
17
18     Objective: To authenticate user
19     Input Parameter: None
20     Return Value: None
21
22     """
23
24     print(' \t LOGIN SYSTEM ')
25     print('-----')
26     password = input(' Enter Password: ')
27     message = authenticateUser(password)
28
29     print(message)
30
31
32 if __name__=='__main__':
33     main()

```

Fig. 3.6 Program to authenticate user and allow access to system
(authenticate.py)

If the condition in line 7 holds True, the statement in line number 8 (body of `if` statement) gets executed. If the condition in line 7 fails to hold, control is transferred

to the assignment statement in line 10 (body of the `else` part). In Fig. 3.7, we illustrate the `if-else` statement with the help of a flowchart.

Conditional Expression

Python allows us to write conditional expressions of the form given below:

use of `if-else` conditional statement to validate the password entered by the user

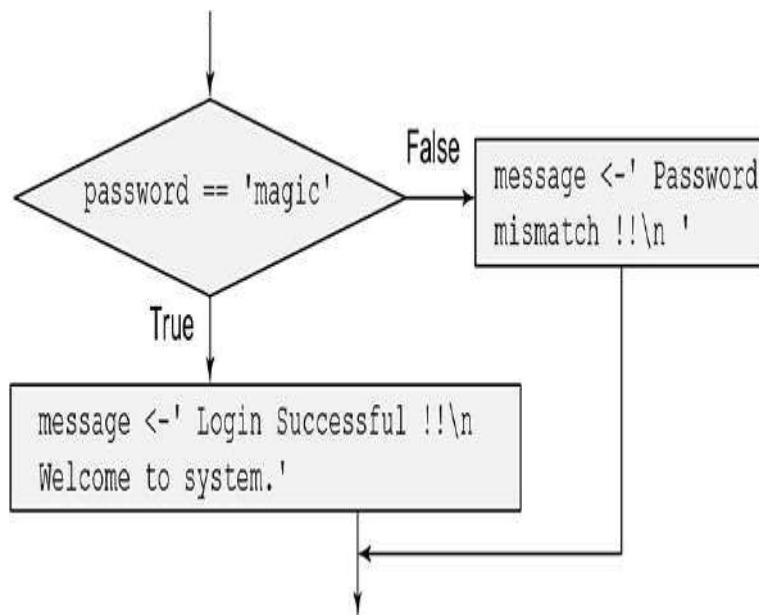


Fig. 3.7 Flow diagram of `if-else` statement in the function `authenticateUser`

`<expression1> if <condition> else < expression2>`

If the `condition` holds `True`, the conditional expression yields the value of `expression1`, otherwise, it yields the value of `expression2`. For example, the following piece of code

```
if password == 'magic':  
    message = ' Login Successful !!\\n '  
else:
```

```
message = ' Password mismatch !!\n '
```

may be replaced by

```
message = ' Login Successful !!\n ' if  
password == 'magic' else ' Password  
mismatch !!\n '
```

use of conditional expression for password validation

When there is more than one way of doing a thing, one should prefer the one which enhances the readability of the program. As you might have noted that the use of an `if-else` statement is more readable as compared to a conditional expression. Although a conditional expression happens to be more concise, we discourage the use of conditional expressions for the beginners.

General Form of if-else Conditional Statement

The general forms of `if-else` statement is as follows:

syntax of `if-else` conditional statement

```
if < condition >:  
  
    < Sequence S1 of statements to be  
    executed >  
  
else:  
  
    < Sequence S2 of statements to be  
    executed >
```

Here, `condition` is a Boolean expression. If this condition evaluates to `True`, then the sequence `S1` of statements is executed, else sequence of statements `S2` gets executed. Subsequently, the control is transferred to the statement following the `if-else` statement. The execution of `if-else` statement is illustrated in Fig. 3.8.

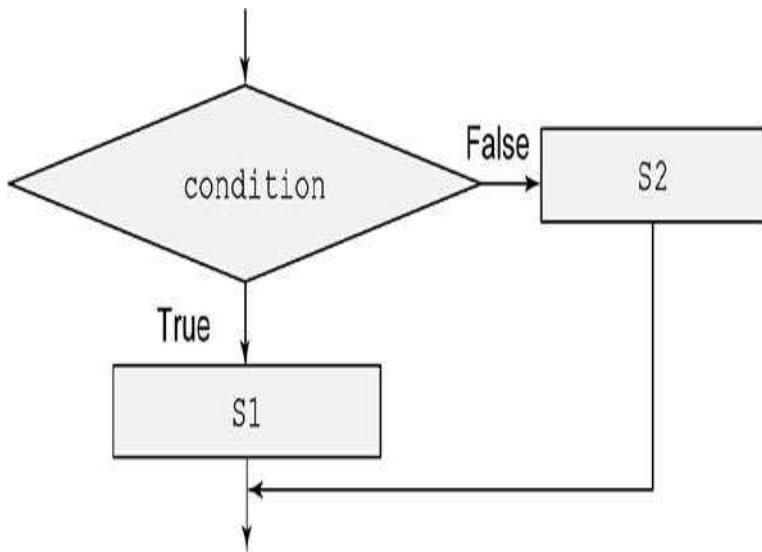


Fig. 3.8 Flow diagram of `if-else` structure

flow diagram of `if-else` conditional statement

In the next problem, we want to assign a grade to a student on the basis of marks obtained as per the criteria mentioned in Table 3.1.

Table 3.1 Criteria for assigning grades

Range	Grade
[90, 100]	A
[70, 89]	B
[50, 69]	C
[40, 49]	D
[0, 39]	F

conversion of grades to marks

The script `grade` (Fig. 3.9) takes marks of a student as input from the user and assigns a grade on the bases of marks obtained using `if-elif-else` statement. In Fig. 3.10, we illustrate the `if-elif-else` statement with the help of a flowchart.

```
01 def assignGrade(marks):
02     """
03         Objective: To assign grade on the basis of marks obtained
04         Input Parameter: marks - numeric value
05         Return Value: grade - string
06     """
07     assert marks >= 0 and marks <= 100
08     if marks >= 90:
09         grade = 'A'
```

```
10     elif marks >= 70:
11         grade = 'B'
12     elif marks >= 50:
13         grade = 'C'
14     elif marks >= 40:
15         grade = 'D'
16     else:
17         grade = 'F'
18     return grade
19
20
21 def main():
22     """
23         Objective: To assign grade on the basis of input marks
24         Input Parameter: None
```

```
25     Return Value: None
26
27     marks = float(input('Enter your marks: '))
28     print('Marks:', marks, '\nGrade:', assignGrade(marks))
29
30 if __name__=='__main__':
31     main()
```

Fig. 3.9 Program to assign grade on the basis of marks obtained
(grade.py)

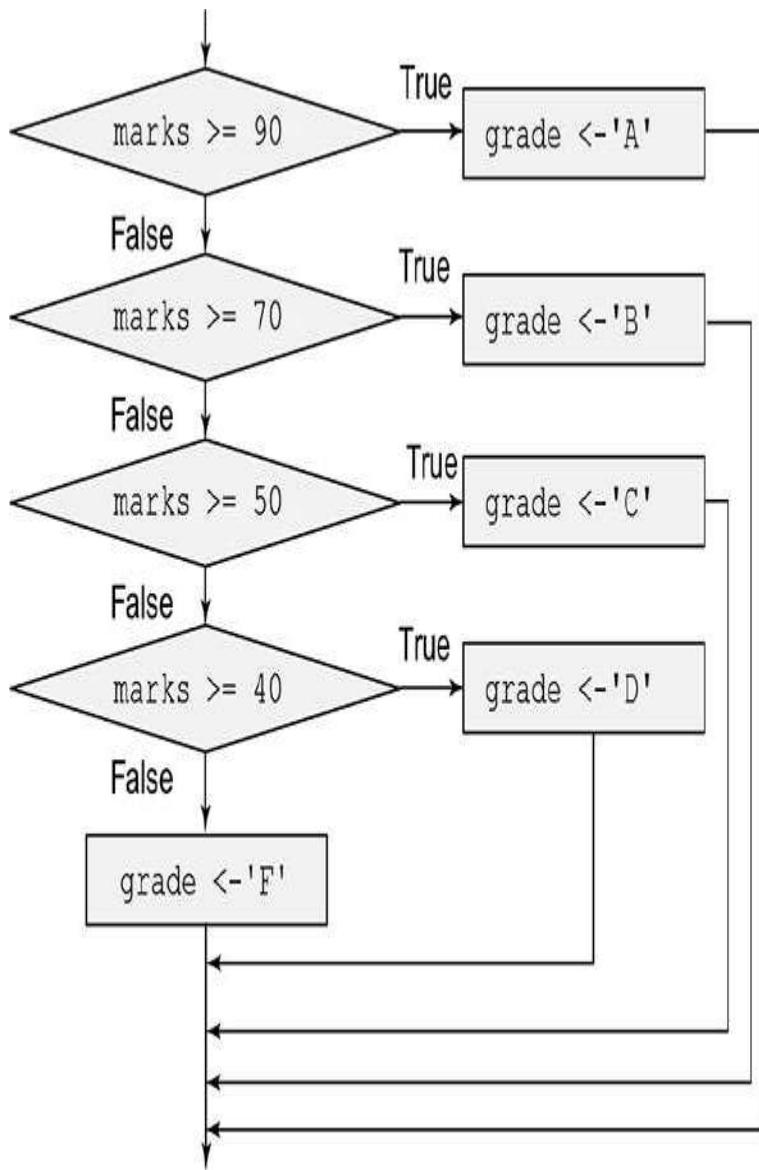


Fig. 3.10 Flow diagram of if-elif-else statement in the function assignGrade

conversion of marks to grades

General Form of if-elif-else Conditional Statement

The general form of if-elif-else statement is as follows:

```
if < condition1 >:
```

```
    < Sequence S1 of statements to be
    executed >
```

```

elif < condition2 >:
    < Sequence  $S_2$  of statements to be
executed >

elif < condition3 >:
    < Sequence  $S_3$  of statements to be
executed >

.
.
.

else:
    < Sequence  $S_n$  of statements to be
executed >

```

syntax of if-elif-else conditional statement

elif and else clauses are optional in a conditional statement

In the above description, `elif` is an abbreviation used in Python for `else if`. By now, it must be clear that the clauses `elif` and `else` of the `if` statement are optional, and that the sequence of statements defined under a clause is executed in a sequence. As the physical alignment of statements determines which statements form a block of code, one needs to be very careful about indentation while writing Python programs.

Nested if-elif-else Conditional Statement

Sometimes, we need a control structure within another control structure. Such a mechanism is called nesting. For example, the function `maximum3` (Fig. 3.12) finds the maximum of three numbers using nested structure. Here, an `if` clause has been used within another `if`

clause. First, we test whether n_1 is greater than n_2 . If so, the condition $n_1 > n_3$ is evaluated, and if True, n_1 is declared as the maximum number. The other cases are dealt with similarly. This is illustrated with the help of flow diagram in Fig. 3.11.

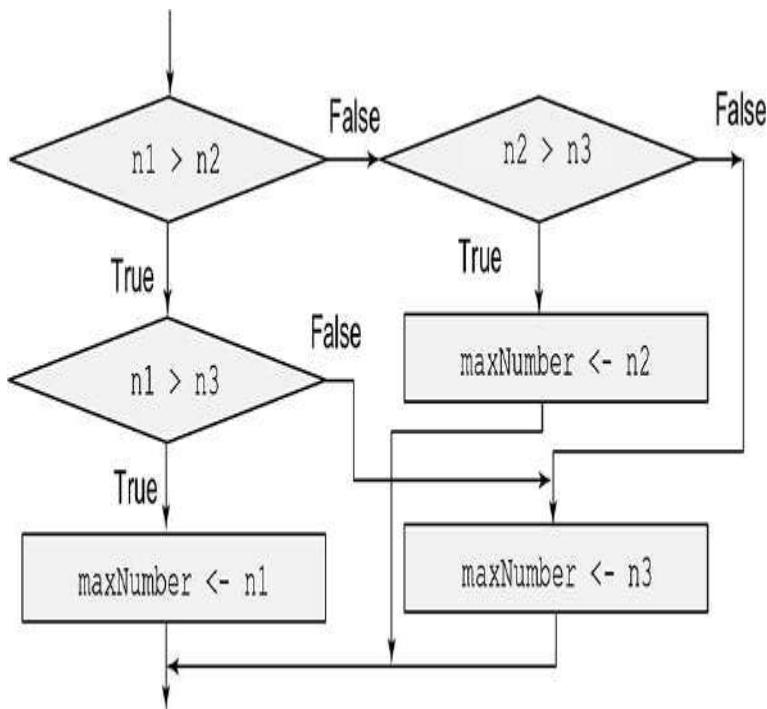


Fig. 3.11 Flow diagram of if statement in the function maximum

to find maximum of three numbers

```

01 def maximum3(n1, n2, n3):
02     """
03     Objective: To find maximum of three numbers
04     Input Parameters: n1, n2, n3 - numeric values
05     Return Value: maxNumber - numeric value
06     """
07     if n1 > n2:
08         if n1 > n3:
09             maxNumber = n1
10         else:
11             maxNumber = n3
  
```

```

12     elif n2 > n3:
13         maxNumber = n2
14     else:
15         maxNumber = n3
16     return maxNumber
17
18 def main():
19     """
20     Objective: To find maximum of three numbers provided as input
21     by user
22     Input Parameter: None
23     Return Value: None
24     """
25     n1 = int(input('Enter first number: '))
26     n2 = int(input('Enter second number: '))
27     n3 = int(input('Enter third number: '))
28     maximum = maximum3(n1, n2, n3)
29     print('Maximum number is', maximum)
30
31 if __name__=='__main__':
32     main()

```

Fig. 3.12 Program to find the maximum of three numbers
(maximum.py)

On executing the script given in Fig. 3.12, Python prompts the user to enter three numbers and responds by displaying the maximum number.

>>>

Enter first number: 78

Enter second number: 65

Enter third number: 89

```
Maximum number is 89
```

We can simplify the above program using *nested function* approach (Fig. 3.13). We define a function `max2` that takes two numbers as an input and computes their maximum. Next, we make use of `max2` to define the function `max3` that finds the maximum of three numbers. We say that the function `max2` is nested within the function `max3`. The functions `max2` and `max3` are known as inner function and outer function, respectively.

nested function approach to find maximum of three numbers

```
01 def max3(n1, n2, n3):
02     """
03     Objective: To find maximum of three numbers
04     Input Parameters: n1, n2, n3 - numeric values
05     Return Value: number - numeric value
06     """
07     def max2(n1, n2):
08         """
09         Objective: To find maximum of two numbers
10         Input Parameters: n1, n2 - numeric values
11         Return Value: maximum of n1, n2 - numeric value
12         """
13         if n1 > n2:
14             return n1
15         else:
16             return n2
17     return max2(max2(n1, n2), n3)
18
19 def main():
20     """
21     Objective: To find maximum of three numbers provided as input
```

```
22 by user
23 Input Parameter: None
24 Return Value: None
25 """
26 n1 = int(input('Enter first number: '))
27 n2 = int(input('Enter second number: '))
28 n3 = int(input('Enter third number: '))
29 maximum = max3(n1, n2, n3)
30 print('Maximum number is', maximum)
31
32 if __name__=='__main__':
33     main()
```

Fig. 3.13 Program to find maximum of three numbers
(maximum3.py)

3.2 ITERATION (FOR AND WHILE STATEMENTS)

Suppose we wish to read and add 100 numbers. Using the method discussed so far, we will have to include 100 statements for reading and an equal number of statements for addition. If the numbers to be read and added were 10,000 instead of 100, we can well imagine the gigantic appearance of the program that would run into well over a hundred pages. Surely, we cannot think of doing that. The process of repetitive execution of a statement or a sequence of statements is called a loop. Using loops, we need to write only once the sequence of statements to be repeatedly executed. Execution of a sequence of statements in a loop is known as an iteration of the loop.

loop: repetitive execution of instructions

iteration of a loop

The `for` statement in Python provides a mechanism to keep count of the number of times a sequence of

statements has been executed. Sometimes, we say that the `for` statement loops over a sequence of statements and as such is also called `for` loop. At times, there may be a situation where the number of times, a sequence of statements has to be processed is not known in advance, but depends on the fulfillment of some condition. In such situations, we use a `while` loop.

3.2.1 `for` Loop

The control statement `for` is used when we want to execute a sequence of statements (indented to the right of keyword `for`) a fixed number of times. Suppose, we wish to find the sum of first few (say n) positive integers. Further, assume that the user provides the value of n . For this purpose, we need a counting mechanism to count from 1 to n and keep adding the current value of count to the old value of `total` (which is initially set to zero). This is achieved using the control statement `for` (Fig. 3.14).

```
01 def summation(n):
02     """
03     Objective: To find sum of first n positive integers
04     Input Parameter: n - numeric value
05     Return Value: total - numeric value
06     """
07     total = 0
08     for count in range(1, n + 1):
09         total += count
10     return total
11
12 def main():
13     """
```

```
14     Objective: To find sum of first n positive integers based on user  
15     input  
16     Input Parameter: None  
17     Return Value: None  
18     '''  
19     n = int(input('Enter number of terms: '))  
20     total = summation(n)  
21     print('Sum of first', n, 'positive integers: ', total)  
22  
23 if __name__=='__main__':  
24     main()
```

Fig. 3.14 Program to find sum of first n positive integers
(sum.py)

for loop: used to execute a sequence of instructions a fixed number of times

On executing the script (Fig. 3.14), Python prompts the user to enter the value of n and responds with the sum of first n positive integers:

```
Enter number of terms: 5
```

```
Sum of first 5 positive integers: 15
```

In the function summation (script sum, Fig. 3.14), the variable total is initialized to zero. Next, for loop is executed. The function call range(1, n + 1) produces a sequence of numbers from 1 to n. This sequence of numbers is used to keep count of the iterations. In general,

```
range(start, end, increment)
```

`range` function generates a sequence of integers

returns an object that produces a sequence of integers from `start` up to `end` (but not including `end`) in steps of `increment`. If the third argument is not specified, it is assumed to be 1. If the first argument is also not specified, it is assumed to be 0. Values of `start`, `end`, and `increment` should be of type integer. Any other type of value will result in error. Next, we give some examples of the use of `range` function:

Function	Sequence of values produced
<code>range(1,11)</code>	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
<code>range(11)</code>	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
<code>range(1,11,2)</code>	1, 3, 5, 7, 9
<code>range(30, -4, -3)</code>	30, 27, 24, 21, 18, 15, 12, 9, 6, 3, 0, -3

In the script `sum`, the variable `count` takes a value from the sequence 1, 2, ..., n, one by one, and the statement

```
total += count
```

is executed for every value of `count`. On completion of the loop, the function `summation` returns the sum of first n positive numbers stored in the variable `total` (line 10). The value returned is displayed using `print` function in line 21.

Next, we develop a program to read and add the marks entered by the user in n subjects, and subsequently use the total marks so obtained to compute the overall percentage of marks (Fig. 3.15). The number of subjects n is taken as the input from the user. The variable `totalMarks` is initialized to zero in line 8 before the `for` loop. The sequence of statements at lines 11, 12, and 13 forms the body of the loop and is executed n times, once for each value of i. We say that the loop iterates for each value of i in the sequence 1, 2, ...,

n. Once the execution of the loop completes, percentage is computed.

```
01 def main():
02     """
03     Objective: To compute percentage for marks entered in n
04     subjects
05     Input Parameter: None
06     Return Value: None
07     """
08     n = int(input('Number of subjects: '))
09     totalMarks = 0
10     print('Enter marks')
11     for i in range(1, n + 1):
12         marks = float(input('Subject ' + str(i) + ': '))
13         assert marks >= 0 and marks <= 100
14         totalMarks += marks
15     percentage = totalMarks / n
16     print('Percentage is: ', percentage)
17
18 if __name__ == '__main__':
19     main()
```

Fig. 3.15 Program to compute percentage (percentage.py)

On executing the script percentage (Fig. 3.15), Python prompts the user to enter the number of subjects and the marks, and responds by displaying the percentage.

```
Number of subjects: 5
```

```
Enter marks
```

```
Subject 1: 75
```

```
Subject 2: 80
```

```
Subject 3: 85
```

```
Subject 4: 90
```

```
Subject 5: 95
```

```
Percentage is: 85.0
```

General Format of for Statement

The general form of `for` structure is as follows:

```
for variable in sequence:
```

```
<block S of statements>
```

for loop syntax

Here, `variable` refers to the control variable used for counting, which is set in the beginning to the first value in a sequence of values (e.g., `range(10)`, '`abcdef`'). Subsequently, in each iteration of the loop, the value of the control `variable` is replaced by the successor value in the sequence. The process of executing the sequence `S` and replacing the current value of control variable by its successor in sequence is repeated until the entire sequence is exhausted.

Next, we wish to develop a function to print a multiplication table for a given number. For example, the multiplication table for 4 should appear as follows:

```
4 * 1 = 4
```

```
4 * 2 = 8
```

```
4 * 3 = 12
```

```
4 * 4 = 16
```

```
4 * 5 = 20
```

```
4 * 6 = 24
```

```
4 * 7 = 28
```

```
4 * 8 = 32
```

```
4 * 9 = 36
```

```
4 * 10 = 40
```

By now you must have noted that Python output is aligned on the left-hand side, for example:

```
>>> print(7)
```

```
7
```

```
>>> print(100)
```

```
100
```

By default, Python aligns the output on LHS

However, the numeric output looks good, if aligned on the right-hand side. For this purpose, we need to specify how many places we would like to reserve for printing a value, for example, the format string '%5d' may be used to indicate that at least five positions are to be reserved for an integer value.

```
>>> '%5d'% 45
```

```
' 45'
```

```
>>> print('%5d'% 45)
```

```
45
```

```
>>> print('%5d' % 12345)
```

```
12345
```

formatted output using format string

To develop multiplication table for a given number, we would require two parameters: number (say, num) whose multiplication table is to be printed, and the number of multiples (say, nMultiples) of the number. In this exercise, we need to print nMultiples (= 10). The following code segment does this job (Fig. 3.16). In Fig. 3.17, we present a complete script to print the multiplication table of a given number.

```
01 for multiple in range(1, nMultiples + 1):
02     product = num * multiple
03     print(num, '*', '%2' % multiple, '=', '%5d' % product)
```

Fig. 3.16 Loop for printing multiplication table

```
01 def printTable(num, nMultiples = 10):
02     """
03     Objective: To print multiplication table of a number
04     comprising of first nMultiples
05     Input Parameters:
06         nMultiples: numeric - number of multiples of a number
07             to be printed
08         num: numeric - number whose multiplication table is to
09             be printed
10     Output: Multiplication tables of a number
11     Return Value: None
12     """
```

```

13     for multiple in range(1, nMultiples + 1):
14         product = num * multiple
15         print(num, '*', '%2d' % multiple, '=', '%5d' % product)
16
17
18 def main():
19     """
20     Objective: To print multiplication table of a number
21     Input Parameter: None
22     Return Value: None
23     """
24     num = int(input('Enter the number: '))
25     printTable(num)
26
27 if __name__=='__main__':
28     main()

```

Fig. 3.17 Program to print multiplication table of a number

3.2.2 while Loop

The `while` loop is used for executing a sequence of statements again and again on the basis of some *test condition*. If the *test condition* holds `True`, the body of the loop is executed, otherwise the control moves to the statement immediately following the `while` loop.

Suppose, we wish to find the sum of all the numbers entered by a user until a null string (empty string: '') is entered as input. As we do not know in advance the count of numbers that the user will enter before entering a null string, we make use of a `while` loop in the script `sumNumbers` (Fig. 3.18). Every time the user enters a string, its integer value is added to the old value of `total` (initialized to zero before the beginning of the loop). The process continues until the user enters a null

string as input. On encountering a null string, the *test condition* in line 10 becomes `False`, the `while` loop terminates, and the control moves to line 13, where we display the value of `total`.

while loop: to repeat execution of an instruction sequence as long as a condition holds

empty string is also known as null string

```
01 def main():
02     """
03     Objective: To compute sum of numbers entered by user until
04     user provides with null string as the input
05     Input Parameter: None
06     Return Value: None
07     """
08     total = 0
09     number = input('Enter a number: ')
10    while number != '':
11        total += int(number)
12        number = input('Enter a number: ')
13    print('Sum of all input numbers is', total)
14
15 if __name__=='__main__':
16     main()
```

Fig. 3.18 Program to compute sum of numbers (`sumNumbers.py`)

On executing the script `sumNumbers` (Fig. 3.18), Python prompts for numbers until the user enters null string as an input.

```
>>>  
  
Enter a number: 2  
  
Enter a number: 18  
  
Enter a number: 15  
  
Enter a number: 15  
  
Enter a number:  
  
Sum of all input numbers is 50
```

Next, we show the working of the `while` loop in the context of the foregoing example (Fig. 3.19).

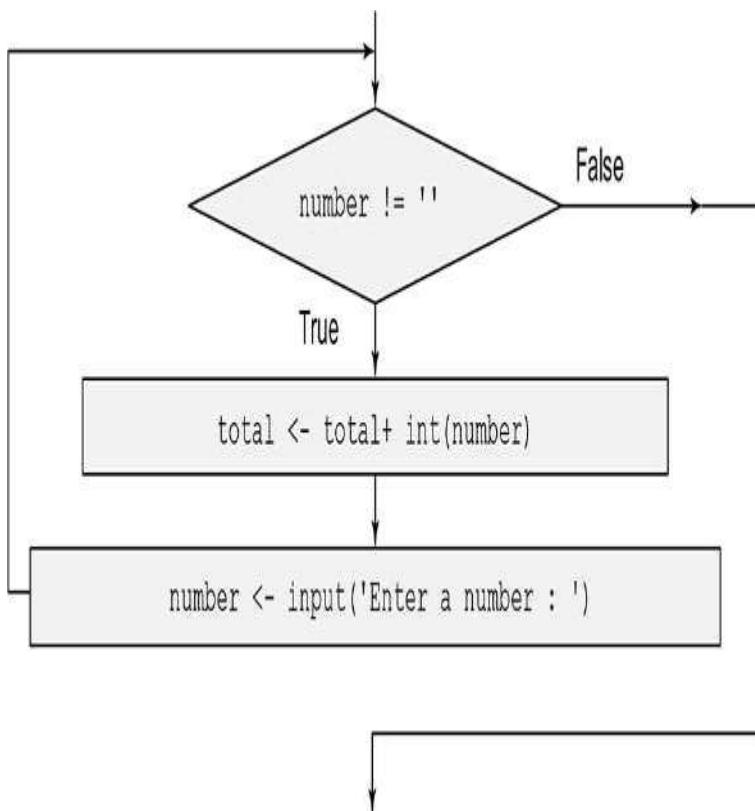


Fig 3.19 Flow diagram of `if` statement in script `sumNumbers`

General Format of while Statement

The general form of `while` statement is as follows:

```
while <test condition>:
```

```
    <Sequence of statements S>
```

syntax for while loop

Here, *test condition* is a Boolean expression, which is evaluated at the beginning of the loop. If the *test condition* evaluates to True, the sequence *S* of statements is executed, and the control moves to the evaluation of *test condition* once again. The process is repeated until *test condition* evaluates to False. The execution of the `while` loop is illustrated in Fig. 3.20.

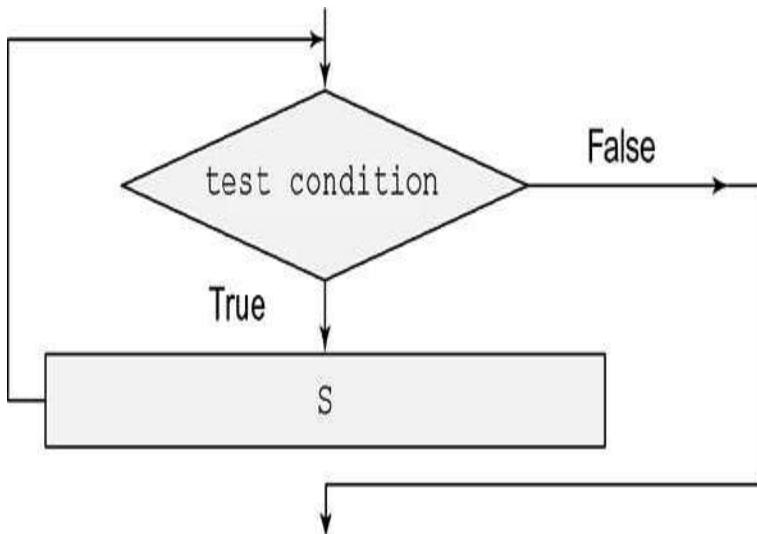


Fig. 3.20 Flow diagram of while structure

flow diagram of while loop

Infinite Loops

Sometimes we want a `while` loop to continue indefinitely until an enabling event takes place, for example, the screen of a laptop may remain off until a key is pressed. For this purpose, we make use of a `while`

loop based on a condition that always evaluates to `True`. Such a loop is known as an infinite loop, for example:

```
infinite loop: loop with a condition which always evaluates to
True

import time

while True:

    try:

        print('Loop processing....')

        print('Use ctrl+c to break')

        time.sleep(1)

    except KeyboardInterrupt:

        print('User interrupted the loop...
        exiting...')

        break
```

in the absence of a keyboard interrupt, the `while` loop will execute infinitely.

3.2.3 while Statement vs. for Statement

Having learned the use of `while` statement, let us use a `while` statement to rewrite the following piece of code:

```
for count in range(1, n + 1):

    total += count
```

The revised code is given below:

```
count = 1
```

```

while count < n+1:

    total += count

    count += 1

```

not a wise idea: rewriting `for` loop using `while`

Note that for computing the sum of first few natural numbers, the use of `for` loop is more elegant and easy as compared to the `while` loop. Out of several control structures which may be used at a place, it is the discretion of the programmer to choose the simple and elegant one.

3.2.4 Example: To Print Some Pictures

Next, we develop a program that prints a right triangle or an inverted triangle (shown in Fig. 3.21) depending on user's choice. Further, the number of rows in the triangle is also to be taken as input from the user. Let us examine the script `triangle` (Fig. 3.22) that serves this purpose. In this program, the user needs to enter his/her choice of figure 1 or 2 depending on whether he or she wants to print a right triangle or an inverted triangle. The assertion in line 11 is used to check whether the input choice is a valid choice, i.e., 1 or 2. Subsequently in line 12, the number of rows in the figure is taken as input. Next, we develop the functions to print right triangle and inverted triangle. To print a right triangle having a number of rows (equal to `nRows`), we have to generate as many rows of output. Outline of the Python code for this purpose is given below:

```

for i in range(1, nRows + 1):

    {Generate one row of output for right
    triangle}

```


Fig. 3.21 (a) Right triangle and (b) inverted triangle

```
01 def main():
02     """
03         Objective: To print right triangle or inverted triangle
04         depending on user's choice
05     Input Parameter: None
06     Return Value: None
07     """
08     choice = int(input('Enter 1 for right triangle.\n'+
09                         'Enter 2 for inverted triangle.\n'))
10
11     assert choice == 1 or choice == 2
12     nRows = int(input('Enter no. of rows: '))
13
14     if choice == 1:
15         rightTriangle(nRows)
16     else:
17         invertedTriangle(nRows)
```

```
19 if __name__=='__main__':
20     main()
```

Fig 3.22 main function to print right triangle and inverted triangle (triangle.py)

We note that in the first row, one '*' is to be output, in the second row two '*'s are to be output, and so on. So, in general, the i th row will have i '*'s. With these remarks, we modify the outline of the code given above:

```
for i in range(1, nRows + 1):
    print('*' * i)
```

for loop for printing a right triangle

Similarly, to print an inverted triangle comprising the number of rows (equal to `nRows`), we have to generate as many rows of output. The Python code may be outlined as follows:

```
for i in range(0, nRows):
    {Generate one row of output for inverted
    triangle}
```

Let `nSpaces` denotes the number of leading spaces in a row. We note that there are no leading spaces in the first row. So, we set

```
nSpaces = 0
```

For every subsequent row, the number of leading spaces `nSpaces` increases by one in each row. Next, let `nStars` denote the number of stars to be printed in a row. We note that in the first row, the number of stars to be printed is $2 * nRows - 1$. So, we set

```
nStars = 2 * nRows - 1
```

We also note that the number of stars to be printed decreases by two for every subsequent row. With these remarks, we modify the outline of the above piece of code as follows:

```
nSpaces = 0  
  
nStars = 2 * nRows - 1  
  
for i in range(1, nRows+1):  
  
    print(' ' * nSpaces + '*' * nStars)  
  
    nStars -= 2  
  
    nSpaces += 1
```

for loop for printing inverted triangle

In light of the above discussion, we give the complete script in Fig. 3.23:


```
01 def rightTriangle(nRows):
02     """
03     Objective: To print right triangle
04     Input Parameter: nRows - integer value
05     Return Value: None
06     """
07     for i in range(1, nRows + 1):
08         print('*' * i)
09
10 def invertedTriangle(nRows):
11     """
12     Objective: To print inverted triangle
13     Input Parameter: nRows - integer value
14     Return Value: None
15     """
16     nSpaces = 0
17     nStars = 2 * nRows - 1
18     for i in range(1, nRows+1):
19         print(' ' * nSpaces + '*' * nStars)
20         nStars -= 2
21         nSpaces += 1
22
23
```

```
24 def main():
```

```

25 """
26 Objective: To print right triangle or inverted triangle
27 depending on user's choice
28 Input Parameter: None
29 Return Value: None
30 """
31 choice = int(input('Enter 1 for right triangle.\n'+
32                     'Enter 2 for inverted triangle.\n'))
33
34 assert choice == 1 or choice == 2
35 nRows = int(input('Enter no. of rows: '))
36
37 if choice == 1:
38     rightTriangle(nRows)
39 else:
40     invertedTriangle(nRows)
41
42 if __name__=='__main__':
43     main()

```

Fig. 3.23 Program to print right triangle and inverted triangle
(triangle.py)

In the above script, the figures right triangle and inverted triangle comprise '*' only. However, we may make the above program more general by printing the figures made using a character provided by the user. For this purpose, the function `rightTriangle` may be modified as follows (Fig. 3.24):

```
01 def rightTriangle(nRows, char):  
02     '''  
03         Objective: To print right triangle  
04         Input Parameters:  
05             nRows - integer value  
06             char - character to be used for printing figure  
07         Return Value: None  
08     '''  
09     for i in range(1, nRows + 1):  
10         print(char * i)
```

Fig. 3.24 Function `rightTriangle`

The reader is encouraged to modify the `invertedTriangle` and `main` functions in a similar manner and experiment with different figures.

3.2.5 Nested Loops

Suppose we wish to develop a function to print multiplication tables, one for each of the first few (say 10), natural numbers. Further, in the table, multiples of a number are to be arranged vertically in a column. The output should appear as shown in Fig. 3.25.

nested loop: a loop inside another loop

nesting may continue up to any level

1 * 1 = 1	2 * 1 = 2	3 * 1 = 3	: 10 * 1 = 10
1 * 2 = 2	2 * 2 = 4	3 * 2 = 6	: 10 * 2 = 20
1 * 3 = 3	2 * 3 = 6	3 * 3 = 9	: 10 * 3 = 30
1 * 4 = 4	2 * 4 = 8	3 * 4 = 12	: 10 * 4 = 40
1 * 5 = 5	2 * 5 = 10	3 * 5 = 15	: 10 * 5 = 50
1 * 6 = 6	2 * 6 = 12	3 * 6 = 18	: 10 * 6 = 60
1 * 7 = 7	2 * 7 = 14	3 * 7 = 21	: 10 * 7 = 70
1 * 8 = 8	2 * 8 = 16	3 * 8 = 24	: 10 * 8 = 80
1 * 9 = 9	2 * 9 = 18	3 * 9 = 27	: 10 * 9 = 90
1 * 10 = 10	2 * 10 = 20	3 * 10 = 30	: 10 * 10 = 100

Fig. 3.25 Multiplication table

In order that the output of the program appears nicely, we would like that each number be printed using at least the number of positions specified in the format string. This may be done as follows (5 digits are reserved for each number which is left padded with spaces):

```
>>> for i in range(1,4):
    print('{0: >5}'.format(i*9))
9
18
27
```

by default, `print` function inserts new line after printing a line

Also, we would like that the output of several calls to `print` function may be printed on the same line. For

for this purpose, we need to use empty string (' ') in place of the default newline. This can be achieved using keyword argument `end`. For example,

```
>>> for i in range(1, 4):  
  
    print('{0: >5}'.format(i*9), end = '')  
  
9 18 27
```

using empty string as the terminator in `print` function by changing the default value of keyword argument `end`

Indeed, the code segment

```
>>> for i in range(1, 4):  
  
    print('{0: >5}'.format(i*9))
```

is equivalent to:

```
>>> for i in range(1, 4):  
  
    print('{0: >5}'.format(i*9), end = '\n')
```

To develop such a function, we would require two parameters: the number of tables, (say, `nTables`) to be printed, and the number of multiples (say, `nMultiples`) of each number to be printed. The skeleton of a function for this purpose is given in Fig. 3.26.

```

01 def printTable(nMultiples = 10, nTables = 10):
02     """
03         Objective: To print multiplication table of numbers in range
04             [1,nTables], comprising of first nMultiples
05         Input Parameters:
06             nMultiples: numeric - number of multiples of a number
07                             to be printed
08             nTables: numeric - number of tables to be printed
09         Output: Multiplication tables of numbers, beginning 1 ending
10             nTables
11         Return Value: None
12     """

```

Fig. 3.26 Skeleton of function printTable

To print the multiplication table, we need to print `nMultiples` (=10) rows, one for each multiple of the number `num`. Skeleton of the code segment for printing multiplication table is given below:

```

for multiple in range(1, nMultiples + 1):

    # Print a row of multiples of each
    number num

```

Next, to print one row of a multiple of all numbers i.e. multiples of `num` in the range `(1, nTables+1)`, we may iterate over `num` using a `for` loop as follows:

```

for num in range(1, nTables + 1):

    print('{0: >2}'.format(num), '*', \
        '{0: >2}'.format(multiple), '=', \

```

```

'{0: >3}'.format(num*multiple), '\t',
end = '')
print()

```

for loop to print one row of multiples of numbers

Note the use of the keyword argument `end` in the `print` function under the `for` loop, as we want the output of the entire loop to appear on the same line. Finally, when the `for` loop completes, invoking the `print` function moves the control to the next line for printing the next row of the multiplication table. The character backslash (\) used at the end of second and third line is a line continuation character that can be used for wrapping long lines i.e. if we wish to extend a statement over multiple lines. The complete script to print multiplication table is shown in Fig. 3.27.

line continuation character backslash(\)

```

01 def printTable(nTables = 10, nMultiples = 10):
02     """
03     Objective: To print multiplication table of numbers in range
04     [1,nTables], comprising of first nMultiples
05     Input Parameters:
06         nTables: numeric - number of tables to be printed
07         nMultiples: numeric - number of multiples of a number
08             to be printed
09     Output: Multiplication tables of numbers, beginning 1 ending
10         nTables
11     Return Value: None
12     """
13     for multiple in range(1, nMultiples + 1):
14         # Print a row of multiples of each number num

```

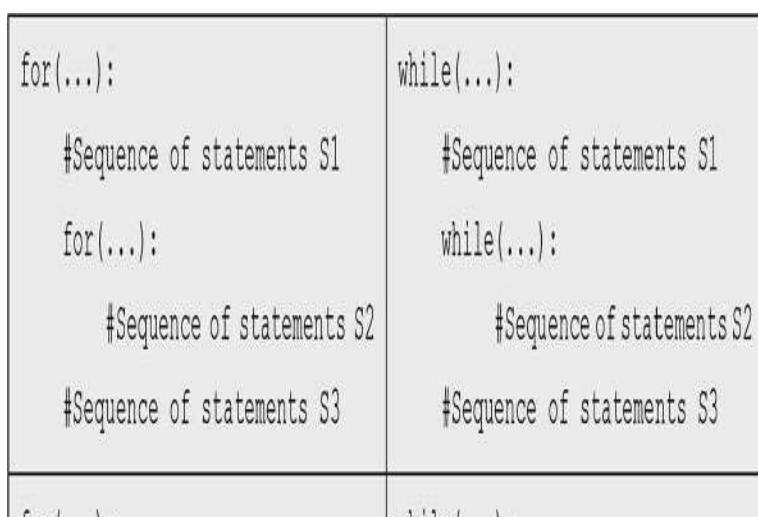
```

15     for num in range(1, nTables + 1):
16         print('{0: >2}'.format(num), '*', \
17             '{0: >2}'.format(multiple), '=', \
18             '{0: >3}'.format(num*multiple), '\t', end = '')
19     print()
20
21 def main():
22     """
23     Objective: To display table of numbers in range [1, nTables]
24     comprising of first 10 multiples
25     Input Parameter: None
26     Return Value: None
27     """
28     nTables = int(input('Enter number of multiplication
29     tables: '))
30     printTable(nTables)
31
32 if __name__ == '__main__':
33     main()

```

Fig. 3.27 Program to print multiplication table (table.py)

The control structures `for` and `while` may be nested up to any level as required. For example, see Fig. 3.28.



<pre>for(...): #Sequence of statements S1 while(...): #Sequence of statements S2 #Sequence of statements S3</pre>	<pre>while(...): #Sequence of statements S1 for(...): #Sequence of statements S2 #Sequence of statements S3</pre>
<pre>for(...): #Sequence of statements S1 for(...): #Sequence of statements S2 while(...): #Sequence of statements S3 #Sequence of statements S4 #Sequence of statements S5</pre>	<pre>while(...): #Sequence of statements S1 for(...): #Sequence of statements S2 while(...): #Sequence of statements S3 #Sequence of statements S4 #Sequence of statements S5</pre>

Fig. 3.28 Examples of nested control structures

3.2.6 break, continue, and pass Statements

Sometimes, we need to alter the normal flow of a loop in response to the occurrence of an event. In such a situation, we may either want to exit the loop or continue with the next iteration of the loop skipping the remaining statements in the loop. The `break` statement enables us to exit the loop and transfer the control to the statement following the body of the loop. The `continue` statement is used to transfer the control to next iteration

of the loop. When the `continue` statement is executed, the code that occurs in the body of the loop after the `continue` statement is skipped.

use `break` statement to exit the loop

use `continue` statement to transfer the control to next iteration of the loop

To illustrate the use of `break` statement, we examine the function `printSquares` (Fig. 3.29) intended to print squares of the integers entered by the user until a null string is encountered. This function uses a `while` loop. The condition in the `while` loop has been set as `True`. If the user enters a null string, the control exits the loop and moves to the statement following the `while` loop, i.e. line 14. Thus, the function terminates with the message '`End of input!!`'. In the other case, when the user input is not null, the input string is converted to an integer (line 12) and the square of the integer so obtained is printed (line 13).

```
01 def printSquares():
02     """
03         Objective: To print squares of positive numbers entered by
04         the user. The program terminates if user enters null string.
05         Input Parameter: None
06         Return Value: None
07     """
08     while True:
09         numStrng = input('Enter an integer, to end press Enter: ')
10         if numStrng == '':
11             break
12         number = int(numStrng)
13         print(number, '^ 2 =', number ** 2)
14     print('End of input!!')
```

Fig. 3.29 Function to print squares of positive numbers
(square.py)

In the script `percentage1` (Fig. 3.30), we wish to compute the overall percentage of marks obtained by a student. As the number of subjects on which examination was conducted is not known beforehand, we use a `while` loop. When the `while` loop is executed, the user is prompted to enter the marks obtained in different subjects. If the user responds with a null string, the `break` statement is executed which results in termination of the loop, and the control moves to line 19 for computation of `percentage`. However, if `marks` entered by the user are outside the range `[0, 100]`, the user is prompted again to enter marks. Thus in the case of invalid marks, the use of `continue` statement makes it possible to skip the statements (lines 17 and 18) that appear after the `continue` statement in the `while` loop.

and continue with the next iteration of the loop for taking the next input from the user. Sample output on executing the script `percentage` is given below:

```
01 def main():
02     """
03         Objective: To display percentage of marks scored by the
04             student
05         Input Parameter: None
06         Return Value: None
07         """
08         totalMarks = 0
09         nSubjects = 0
10         while True:
11             marks = input('Marks for subject ' + str(nSubjects + 1)
12                         + ': ')
13             if marks == '': # End of input
14                 break
15             marks = float(marks)
16             if marks < 0 or marks > 100:
17                 print('INVALID MARKS !! ')
18                 continue # Marks to be entered again
19             nSubjects = nSubjects + 1
```

```
20             totalMarks += marks
21             percentage = totalMarks / nSubjects
22             print('Total marks: ', int(totalMarks))
23             print('Number of Subjects: ', nSubjects)
24             print('Percentage: ', round(percentage, 2))
```

```
22     print('Percentage: / ',percentage/100)
23
24 if __name__=='__main__':
25     main()
```

Fig. 3.30 Program to print percentage and total marks
(percentage1.py)

Marks for subject 1: 60

Marks for subject 2: 80

Marks for subject 3: 280

INVALID MARKS !!

Marks for subject 3: 70

Marks for subject 4: 800

INVALID MARKS !!

Marks for subject 4: 80

Marks for subject 5:

Total marks: 290

Number of Subjects: 4

Percentage: 72.5

Sometimes we may want to leave out the details of the computation in a function body, to be filled in at a later point in time. The `pass` statement lets the program go through this piece of code without executing any code. It is just like a null operation. Often `pass` statement is used as a reminder for some code, to be filled in later. For example, let us think of a merchant wanting to sell clothes, who is thinking of allowing some discounts, but not as of now. So, as of now, he does not want his IT team to develop the code for discounting. In the script

sellingPrice, we develop the function sellingPrice (Fig. 3.31) that invokes the function discount. As the code for the function discount just comprises a pass statement, it produces None as the return value. Accordingly, the function sellingPrice ignores the value None returned by the function discount (lines 16–17) and returns price (that was passed on to it as an argument) as the selling price.

pass statement: execute no code

```
01 def discount(price):
02     """
03     Objective: To compute discount
04     Input Parameter: price - numeric value
05     Return Value: None
06     """
07     pass
08
09 def sellingPrice(price):
10     """
11     Objective: To compute selling price
12     Input Parameter: price - numeric value
13     Return Value: numeric value
14     """
15     discountedPrice = discount(price)
16     if discountedPrice == None:
17         return price
18     else:
19         return discountedPrice
20
21 def main():
22     """
```

```

23     Objective: To compute selling price
24     Input Parameter: None
25     Return Value: None
26     ''
27     price = float(input('Enter price: '))
28     print('Selling Price is', sellingPrice(price))
29
30 if __name__=='__main__':
31     main()

```

Fig. 3.31 Program to compute selling price (`sellingPrice.py`)

3.2.7 Example: To Compute $\sin(x)$

The value of $\sin(x)$ may be computed as the sum of the following series:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \dots$$

In the above series, the first term is $x/1$. The second term can be computed by multiplying the first term by $-x^2/(2*3)$. Similarly, the third term in the series can be computed by multiplying the second term by $-x^4/(4*5)$, and so on. In general, we can obtain a new term of the series by multiplying the previous term by $-x^2$ and dividing this product by the product of the next two terms in the sequence 1, 2, 3, 4, 5, 6, Thus, we can write first few terms of the series as follows:

$$x/1$$

$$x/1 * (-x^2) / (2*3) = -x^3 / 3!$$

$$-x^3 / 3! * (-x^2) / (4*5) = x^5 / 5!$$

To code the above idea, we set `multBy` equal to $-x^2$ and initialize `nxtInSeq` (used to compute `divBy`) equal to 2. We compute `divBy = nxtInSeq * (nxtInSeq+1)`. Every time we compute a new term, we increment the value of `nxtInSeq` by 2. Table 3.1 illustrates these computations:

Table 3.1 Sine terms computations

term	multBy	nxtInSeq	divBy	newTerm
$x/1$	$-x^2$	2	$2*3$	$-x^3/3!$
$-x^3/3!$	$-x^2$	4	$4*5$	$x^5/5!$
$x^5/5!$	$-x^2$	6	$6*7$	$-x^7/7!$

Since the given series is an infinite one, and we can do only finite computations on a computer, we need to decide when to stop. We keep on adding more terms until the absolute value of `term` becomes smaller than a predefined value, say, `epsilon`. In Fig. 3.32, we present the complete function `mySine` that computes `sine(x)` as sum of the above series.

```
01 def mySine(x):
02     """
03         Objective: To find sum of sine series until the absolute value
04         of newTerm becomes smaller than epsilon
05         Input Parameter: x - numeric value in radians
06         Return Value: total - numeric value
07     """
08     epsilon = 0.00001
09     multBy = -x**2
10     term = x
11     total = x
12     nxtInSeq = 2.0
```

```
13
14     while abs(term) > epsilon:
15         divBy = nxtInSeq*(nxtInSeq+1)
16         term = term * multBy / divBy
17         total += term
18         nxtInSeq += 2
19     return total
```

Fig. 3.32 Function mySine (sine.py)

3.2.8 else Statement

The `else` clause is useful in such situations where we may want to perform a task only on successful execution

of a `for` loop or a `while` loop. The statements specified in `else` clause are executed on normal termination of the loop. However, if the loop is terminated forcibly using `break` statement, then the `else` clause is skipped. We demonstrate the use of the `else` clause for testing whether a given number is prime. Recall that a number is said to be prime if and only if it has no divisor other than one and itself. Given a number `n`, we need to check for each number in the range `(2, n)` whether it is a divisor of `n`. If the entire sequence is exhausted and no divisor of `n` is found, it is a prime number. In the function `prime` (Fig. 3.33) if `n` is 1, `flag` is set equal to `False` indicating that the number `n` is not prime. Inside the `for` loop, if we find a divisor `i` of `n` in the range `(2, n)`, the condition `n % i == 0` becomes `True`, indicating that the number `n` is not a prime number. So we set `flag` equal to `False` and exit the `for` loop. However, if the `for` loop executes smoothly (not on the execution of `break` statement), `flag` is set equal to `True` indicating that the number `n` is prime. Finally, `flag` is returned as the value of the function.

`else` clause: to execute a task only on successful completion of the loop

```
01 def prime(n):
02     """
03     Objective: To check if a number is prime or not
04     Input Parameter: n - numeric value
05     Return Value: message - boolean value
06     """
07     if n == 1: # 1 is not prime
08         return False
```

```
09     for i in range(2, n ):
10         if n % i == 0:
11             flag = False # n is not prime
12             break
13     else:
14         flag = True # n is prime
15     return flag
16
17 def main():
18     """
19     Objective: To check if the number entered by the user is a
20     prime number
21     Input Parameter: None
22     Return Value: None
23     """
24     n = int(input('Enter number: '))
25     result = prime(n )
26     if result == True:
27         print(n, 'is a prime number')
28     else:
29         print(n, 'is not a prime number')
30
31 if __name__=='__main__':
32     main()
```

Fig. 3.33 Program to check if a number is prime number or not
(prime.py)

SUMMARY

1. Control statements (also called control structures) are used to control the flow of program execution by allowing non-sequential or repetitive execution of instructions. Python supports `if`, `for`, and `while` control structures. In a control statement, the Python code following the colon (`:`) is indented.
2. `if` statement allows non-sequential execution depending upon whether the *condition* is satisfied. The general form of `if` conditional statement is as follows:

```
if < condition >:  
    < Sequence S of statements to be executed >
```

Here, *condition* is a Boolean expression which is evaluated at the beginning of the `if` statement. If *condition* evaluates to `True`, then the sequence *S* of statements is executed, and the control is transferred to the statement following `if` statement. However, if *condition* evaluates to `False`, then the sequence *S* of statements is ignored, and the control is immediately transferred to the statement following `if` statement. The general form of `if-else` statement is as follows:

```
if < condition >:  
    < Sequence S1 of statements to be executed >  
else:  
    < Sequence S2 of statements to be executed >
```

Here, *condition* is a Boolean expression. If *condition* evaluates to `True`, then the sequence *S1* of statements gets executed, otherwise, the sequence *S2* of statements gets executed.

The general form of `if-elif-else` statement is as follows:

```
if < condition1 >:  
    < Sequence S1 of statements to be executed >  
elif < condition2 >:  
    < Sequence S2 of statements to be executed >  
elif < condition3 >:  
    < Sequence S3 of statements to be executed >  
.  
. .  
else:  
    < Sequence Sn of statements to be executed >
```

The clauses `elif` and `else` of `if` control structure are optional.

3. When a control structure is specified within another control structure, it is called nesting of control structures.
4. The process of repetitive execution of a statement or a sequence of statements is called a loop. Execution of a sequence of statements in a loop is known as an iteration of the loop.
5. The control statement `for` is used when we want to execute a sequence of statements a fixed number of

times. The general form of `for` statement is as follows:

```
for variable in sequence:  
    {Sequence S of statements}
```

Here, `variable` refers to the control variable. The sequence `S` of statements is executed for each value in `sequence`.

6. The function call `range(1, n + 1)` generates a sequence of numbers from 1 to `n`. In general, `range(start, end, increment)` produces a sequence of numbers from `start` up to `end` (but not including `end`) in steps of `increment`. If third argument is not specified, `increment` is assumed to be 1.
7. A `while` loop is used for iteratively executing a sequence of statements again and again on the basis of a `test-condition`. The general form of a `while` loop is as follows:

```
while <test-condition>:  
    <Sequence S of statements>
```

Here, `test-condition` is a Boolean expression which is evaluated at the beginning of the loop. If the `test-condition` evaluates to `True`, the control flows through the `Sequence S of statements` (i.e., the body of the loop), otherwise the control moves to the statement immediately following the `while` loop. On execution of the body of the loop, the `test-condition` is evaluated again, and the process of evaluating the `test-condition` and executing the body of the loop is continued until the `test-condition` evaluates to `False`.

8. The `break` statement is used for exiting from the loop to the statement following the body of the loop.
9. The `continue` statement is used to transfer the control to next iteration of the loop without executing rest of the body of loop.
10. The `pass` statement lets the program go through this piece of code without performing any action.
11. The `else` clause can be used with `for` and `while` loop. Statements specified in `else` clause will be executed on smooth termination of the loop. However, if the loop is terminated using `break` statement, then the `else` clause is skipped.

EXERCISES

1. Write an assignment statement using a single conditional expression for the following `if-else` code:

```
if marks >=70:  
    remarks = 'good'  
else:  
    remarks = 'Average'
```

2. Study the program segments given below. In each case, give the output produced, if any.

```
1. total = 0  
count = 20  
while count > 5:  
    total += count
```

```

        count -= 1
        print(total)
2.total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        total += 1
    print(total)
3.total = 0
N = 10
for i in range(1, N+1):
    for j in range(1, N+1):
        total += 1
    print(total)
4.total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        total += 1
    total -=1
    print(total)
5.total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, N+1):
        total += i
    print(total)
6.total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        total += j
    print(total)
7.total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, N+1):
        total += i + j
    print(total)
8.total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        for k in range(1, j+1):
            total += 1
    print(total)
9.number = 72958476
a, b = 0, 0
while (number > 0):
    digit = number % 10
    if (digit % 2 != 0):
        a += digit
    else:
        b += digit
    number /= 10
print(a, b)

```

3. Write a function to determine whether a given natural number is a perfect number. A natural number is said to be a perfect number if it is the sum of its divisors. For example, 6 is a perfect number because $6=1+2+3$, but 15 is not a perfect number because $15 \neq 1+3+5$.

4. Write a function that takes two numbers as input parameters and returns their least common multiple.
 5. Write a function that takes two numbers as input parameters and returns their greatest common divisor.
 6. Write a function that accepts as an input parameter the number of rows to be printed and prints a figure like:

(a)		(b)
1		1
1 2		2 1 2
1 2 3		3 2 1 2 3
1 2 3 4		4 3 2 1 2 3 4
1 2 3 4 5		
(c)		(d)
5 4 3 2 1		1
4 3 2 1		2 2
3 2 1		3 3 3
2 1		4 4 4 4
1		5 5 5 5 5
(e)		(f)
1 2 3 4 5		*
2 3 4 5		*
3 4 5		*
4 5		*
5		*
(g)		(h)
*	*	*
*	*	*
*	*	*
*	*	*
*	*	*

(i)	
(j)	
(k)	
(l)	
(m)	
(n)	

7. Write a function that finds the sum of the n terms of the following series:
- $$1. \frac{1}{1!} - \frac{x^4}{2!} + \frac{x^6}{3!} - \frac{x^n}{n!} + \dots$$

$$2. e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

8. Write a function that returns True or False depending on whether the given number is a palindrome.
9. Write a function that returns the sum of digits of a number, passed to it as an argument.
10. Write a program that prints Armstrong numbers in the range 1 to 1000. An Armstrong number is a number whose sum of the cubes of the digits is equal to the number itself. For example, $370 = 3^3 + 7^3 + 0^3$.
11. Write a function that takes two numbers as input parameters and returns True or False depending on whether they are co-primes. Two numbers are said to be co-prime if they do not have any common divisor other than one.
12. Write a function sqrt that takes a non-negative number as an input and computes its square root. We can solve this problem iteratively. You will recall from high school mathematics that to find the square root of a number, say 2, we need to solve the equation $f(x) = x^2 - 2 = 0$. To begin with, choose two numbers a and b so that $f(a) < 0$ and $f(b) > 0$. Now, for the equation $f(x) = x^2 - 2 = 0$, $f(1) < 0$ and $f(2) > 0$. So, the root of the equation must lie in the interval $[a, b]$ (i.e. $[1, 2]$). We find the midpoint, say, mid of the interval $[a, b]$. If $f(a) < 0$ and $f(mid) > 0$, we know that the root of the equation $f(x) = 0$ lies in the interval $[a, mid]$. However, in the other case, ($f(mid) < 0$ and $f(mid) > 0$), the root of the equation $f(x) = 0$ must lie in the interval $[mid, b]$. Thus, for the next iteration, we have reduced the search interval for the root of the equation to half, i.e. from $[a, b]$ to $[a, mid]$ or $[mid, b]$. Proceeding in this way, we find a good approximation to the root of the equation when the length of the search interval becomes sufficiently small, say, 0.01. The following table depicts the steps for computing square root approximation for the number 2.

a	b	mid = (a + b)/2	f(a)	f(b)	f(mid)
1	2	1.5	-1	2	0.25
1	1.5	1.25	-1	0.25	-0.4375
1.25	1.5	1.375	-0.4375	0.25	-0.1093
1.375	1.5	1.4375	-0.1093	0.25	0.0664
1.375	1.4375	1.4062	-0.1093	0.0664	-0.0226
1.4062	1.4375	1.4218	-0.0226	0.0664	0.0215
1.4062	1.4218	1.4140	-0.0226	0.0215	-0.0006
1.4140	1.4218	-	-0.0006	0.0215	-

13. Write a function to multiply two non-negative numbers by repeated addition, for example, $7 * 5 = 7 + 7 + 7 + 7 + 7$.

CHAPTER 4

DEBUGGING

CHAPTER OUTLINE

[4.1 Testing](#)

[4.2 Debugging](#)

When a program fails to yield the desirable result, we say that it contains a bug. The bug could be an error such as division by zero, invalid type conversion, using a variable not defined, wrong initialization, or some other unintended operation being performed. The process of discovering the bugs and making the program bug-free is called debugging. In this chapter, we discuss some debugging techniques.

debugging: making the program error free

4.1 TESTING

Program testing aims to expose the presence of bugs in the programs. To find the bugs in a program, we test it on various inputs. We correct the errors found in this process and feel confident that the program will run smoothly.

An Example: Finding Maximum of Three Numbers

Let us examine the script `max3` (Fig. 4.1) intended to find the maximum of three numbers. The function `max3` is intended to return the maximum of three numbers. Since the input numbers can be either positive or negative, we may have test cases containing various combinations of positive and negative numbers. Further, given a set of numbers, we will need to test the code for all possible 6 (= 3!) permutations of three numbers as enumerated below:

```
1. n1 > n2 and n2 > n3 : Maximum : n1
2. n1 > n2 and n2 < n3 and n1 > n3 : Maximum : n1
3. n1 > n2 and n2 < n3 and n1 < n3 : Maximum : n3
4. n1 < n2 and n2 > n3 and n1 > n3 : Maximum : n2
5. n1 < n2 and n2 > n3 and n1 < n3 : Maximum : n2
6. n1 < n2 and n2 < n3 : Maximum : n3
```

test cases for finding maximum of three numbers

We test this script on various permutations of the inputs:

10, 20, 30, namely:

```
1. 30, 20, 10
2. 30, 10, 20
3. 20, 10, 30
4. 20, 30, 10
5. 10, 30, 20
6. 10, 20, 30
```

```
01 def max3(n1, n2, n3):
02     """
03     Objective: To find maximum of three numbers
04     Input Parameters: n1, n2, n3 - numeric values
05     Return Value: maxNumber - numeric value
06     """
07     maxNumber = 0
08     if n1 > n2:
09         if n1 > n3:
10             maxNumber = n1
11         elif n2 > n3:
12             maxNumber = n2
13         else:
14             maxNumber = n3
15     return maxNumber
16
17 def main():
18     """
```

```
19  Objective: To find maximum of three numbers
20  Input Parameter: None
21  Return Value: None
22  ''
23  n1 = int(input('Enter first number: '))
24  n2 = int(input('Enter second number: '))
25  n3 = int(input('Enter third number: '))
26  maximum = max3(n1, n2, n3)
27  print('Maximum number is', maximum)
28
29 if __name__=='__main__':
30     main()
```

Fig. 4.1 Program to find the maximum of three numbers
(max3.py)

It so turns out that the result obtained is correct for all input permutations, except for the permutations 20, 10, and 30:

incorrect output indicates bug in the program

>>>

Enter first number: 20

Enter second number: 10

Enter third number: 30

Maximum number is 0

Thus, a bug persists in the program that needs to be detected and removed.

There are various Python debugging tools such as `pdb`, `pudb`, `pydb`, and `pydbgr`. In this section, we will discuss Python's built-in debugger `pdb`, which is an interactive tool that helps in examining the code execution step by step. It allows us to stop the execution of a program at a chosen instruction called a break point, and evaluate the various expressions in the current context. The debugger also allows us to examine the current vicinity of the code, as well as the status of various objects in the current function being executed which collectively constitute the stack frame corresponding to that function.

```
pdb: python debugger
```

In order to debug a program in Python shell, we need to import the module `pdb` as well as the script to be debugged.

```
>>> import pdb  
  
>>> import max3  
  
>>> pdb.run('max3.main()')  
  
> <string>(1)<module>() ->None  
  
(Pdb)
```

importing `pdb` module

(`pdb`) : prompt to indicate that program is executing in debugging mode

Python debugger uses the prompt (`Pdb`) to indicate that the program is executing in the debugging mode. Another way to execute a program in debugging mode is by including the following two lines in the script:

```
import pdb
```

```
pdb.set_trace()
```

Note that including above two lines at the beginning of the program in Fig. 4.1 will increase every line number by two. By invoking the function `set_trace()` at the very beginning, the program would run in debugging mode right from the first instruction to be executed. However, since we know that the bug exists in the function `max3`, we may invoke `set_trace()` within the function body. Before proceeding further, let us have a look at debugging commands (Table 4.1).

invoke the function `set_trace()` at right place

Table 4.1 Debugging commands

Command	Explanation
<code>h or help</code>	It lists all the available commands.
<code>h commandName</code>	It prints the description about a command.
<code>w or where</code>	Prints the stack trace (sequence of function calls currently in execution, most recent function call being at the beginning). Also shows the statement to be executed next.
<code>u or up</code>	Moves to the stack frame one level up in the stack trace.
<code>d or down</code>	Moves to the stack frame one level down in the stack trace.
<code>s or step</code>	Executes the current statement and moves the control to either next statement, or the function being invoked in the current statement.
<code>n or next</code>	Executes the current statement and moves the control to the next statement. Unlike step command, if a function is being invoked in the current statement, the invoked function gets executed completely.
<code>r or return</code>	Continue execution until the current function returns.
<code>j(ump) lineno</code>	Jumps to the given line number for the next statement to be executed.

l or list	Lists 11 lines in the vicinity of the current statement.
b or break [[f i l e :] line func[,cond]]	Sets the breakpoint at the specified line. Name of the file is one of the optional arguments. When the argument func is used, the breakpoint is set at the first executable statement of the specified function. The second argument may be used to denote a condition which must evaluate to True for setting the breakpoint.
tbreak [[f i l e :] line func[,cond]]	Similar to break command. However, the break point being set is automatically removed once it is reached.
c or clear [[f i l e :] line func[,cond]]	Clears the specified breakpoint. In the absence of an argument, clears all the breakpoints.
c or continue	Continue execution until the breakpoint is reached.
a or args	Prints the argument list of the current function along with their values.

Command	Explanation
p <i>expression</i> print(expression)	Prints the value of the specified <i>expression</i> in the current context.
q or quit	Quits from the Python debugger

commands for debugging

Execution of the program in Fig. 4.1 in debugging mode for the inputs 20, 10, and 30 is shown in Figs. 4.2 (a), (b1), and (b2).

```

01
02 > f:\pythoncode\ch04\max3.py(3)<module>()
03 -> def max3(n1, n2, n3):
04 (Pdb) h
05
06 Documented commands (type help <topic>):
07 =====

```

```
08 EOF  c    d    h    list   q    rv    undisplay
09 a     cl   debug  help  ll    quit s    unt
10 alias clear disable ignore longlist r    source until
11 args commands display interact n    restart step up
12 b    condition down j    next return tbreak w
13 break cont enable jump p    retval u    whatis
14 bt   continue exit l    pp    run   unalias where
15
16
17
18 Miscellaneous help topics:
19 =====
20 exec pdb
21
22
23
24 (Pdb) h s
25 s(tep)
26 Execute the current line, stop at the first possible
27 occasion (either in a function that is called or in the
28 current function).
29 Pdb) w
30 <string>(1)<module>()
31 c:\users\st\appdata\local\programs\python\python36-
32 32\lib\idlelib\run.py(142)main()
33 -> ret = method(*args, **kwargs)
34 c:\users\st\appdata\local\programs\python\python36-
35 32\lib\idlelib\run.py(460)runcode()
```



```
36 -> exec(code, self.locals)
37 > f:\pythoncode\ch04\max3.py(3)<module>()
38 -> def max3(n1, n2, n3):
39 (Pdb) l
40   1 import pdb
41   2 pdb.set_trace()
42   3 ->     def max3(n1, n2, n3):
43   4     """
44   5     Objective: To find maximum of three numbers
45   6     Input Parameters: n1, n2, n3 - numeric values
46   7     Return Value: maxNumber - numeric value
47   8     """
48   9     maxNumber = 0
49  10    if n1 > n2:
50  11        if n1 > n3:
51 (Pdb) s
52 > f:\pythoncode\ch04\max3.py(19)<module>()
53 -> def main():
54 (Pdb) s
55 > f:\pythoncode\ch04\max3.py(31)<module>()
56 -> if __name__=='__main__':
57 (Pdb) s
58 > f:\pythoncode\ch04\max3.py(32)<module>()
59 -> main()
60 (Pdb) s
61 --Call--
62 > f:\pythoncode\ch04\max3.py(19)main()
63 -> def main():
64 (Pdb) s
65 > f:\pythoncode\ch04\max3.py(25)main()
66 -> n1 = int(input('Enter first number: '))
67 (Pdb) s
68 --Call--
69 >c:\users\st\appdata\local\programs\python\python36-
70 2\lib\idlelib\run.py(340)write()
71 -> def write(self, s):
72 (Pdb) u
73 > f:\pythoncode\ch04\max3.py(25)main()
74 -> n1 = int(input('Enter first number: '))
75 (Pdb) n
76 Enter first number: 20
77 > f:\pythoncode\ch04\max3.py(26)main()
78 -> n2 = int(input('Enter second number: '))
79 (Pdb) n
80 Enter second number: 10
```

```
81 > f:\pythoncode\ch04\max3.py(27)main()
82 -> n3 = int(input('Enter third number: '))
83 (Pdb) n
84 Enter third number: 30
85 > f:\pythoncode\ch04\max3.py(28)main()
86 -> maximum = max3(n1, n2, n3)
87 (Pdb) p (n1, n2, n3)
88 (20, 10, 30)
```

Fig. 4.2(a) Execution of program `max3.py`

In Fig. 4.2 (a), line 2 (beginning with greater than sign) shows the script being executed. Line 3 begins with an arrow and marks the next line to be executed. Now, Python enters into debugging mode (line 4), and we see the `Pdb` prompt. Command `h` displays all the available commands (lines 4–20). Command `h s` displays the description of the `step` command (lines 24–28). Command `w` has been used to inspect the current position in the stack frame and it displays the stack trace with the statement to be executed next at the bottom (lines 29–38). Command `l` (line 39) lists 11 lines in the vicinity of the current statement (lines 40–50). Next, to execute lines of code, we use `s` command. Execution of `s` command in line 51 yields the definition of function `max3` for future use. Similarly, execution of `s` command in line 54 yields the definition of function `main` for future use. Now, line 56 shows that the step to be executed next is the `if` statement in the script. On execution of `s` command (line 57), the conditional expression yields `True`, and the control moves to the next statement that invokes the function `main` (line 59). Execution of `s` command (line 60) moves the control to the beginning of the function `main` (line 63):

`h`: help on a debugging command

`w`: determine current position in the stack frame

`l`: display 11 lines in the vicinity of the current line of code

`s`: execute the current statement and move the control to the next statement (possibly in the function being invoked)

```
def main():
```

Now execution of `s` command moves the control to the next statement (line 66).

```
n1 = int(input('Enter first number: '))
```

At this stage, when `s` command is executed (line 67), the control steps into the function `input`. But we are not interested in the detailed execution of this function.

Instead, we would like to execute the function `input` as a single unit. So, we need to move one level up in the stack frame. For this purpose, we execute the command `u` (line 72). Subsequently, we execute command `n` to execute the current line as a single unit. Thus, user inputs are entered on the execution of `n` commands (lines 75–86). At this stage, we display `(n1, n2, n3)` (lines 87–88). As we move to the execution of the function `max3`, we show the details in a separate figure (Fig. 4.2 (b1)).

`u`: move one level up in the stack frame

`n`: execute the current statement/function completely and move the control to the next statement

```
01 (Pdb) l
```

```
** (***)
02 23      Return Value: None
03 24      ''
04 25      n1 = int(input('Enter first number: '))
05 26      n2 = int(input('Enter second number: '))
06 27      n3 = int(input('Enter third number: '))
07 28 ->  maximum = max3(n1, n2, n3)
08 29      print('Maximum number is', maximum)
09 30
10 31  if __name__=='__main__':
11 32      main()
12 [EOF]
13 (Pdb) n
14 > f:\pythoncode\ch04\max3.py(29)main()
15 -> print('Maximum number is', maximum)
16 (Pdb) s
17 --Call--
18 >c:/users/st/appdata/local/programs/python/python36-
19 32\lib\idlelib\run.py(340)write()
20 -> def write(self, s):
21 (Pdb) u
22 > f:\pythoncode\ch04\max3.py(29)main()
23 -> print('Maximum number is', maximum)
24 (Pdb) n
25 Maximum number is 0
26 --Return--
27 > f:\pythoncode\ch04\max3.py(29)main()->None
28 -> print('Maximum number is', maximum)
29 (Pdb) s
30 --Return--
31 > f:\pythoncode\ch04\max3.py(32)<module>()->None
32 -> main()
33 (Pdb) q
34
35 Traceback (most recent call last):
36 File "F:/PythonCode/Ch04/max3.py", 37 line 32, in <module>
37 main()
38 bdb.BdbQuit
```

Fig. 4.6 On-the-fly execution of program max3.py

Fig. 4.2 (b1) Execution of program max3.py

In Fig. 4.2 (b1), using `l` command in line 1, we inspect the next statement to be executed. The arrow in line 7 marks the next line to be executed in the code, i.e. line 28. To completely execute the function `max3` with the input numbers, we use the command `n`. Now, the control moves to line 29 in the script. On executing the command `n` once more, the value of `maximum` is displayed which being 0 (line 25) is incorrect. Note that `main` function is now completely executed and the control returns to the point following line 32 in the script which invoked the function `main`. Now we have reached the end of the script as indicated by the response of the debugger:

```
f:\pythoncode\ch04\max3.py (32)<module>() ->None
```

Since there are no further statements to be executed, we use `q` command to quit from the debugger.

`q`: quit the debugger

Although we know that a bug exists in the function `max3`, we have still not found it. To find the bug, we need to step into the function `max3` instead of executing it completely in one go using command `n`. This has been shown in Fig. 4.2 (b2). At line 13, on the execution of the command `s`, the control transfers to the statement:

```
def max3(n1, n2, n3):
```

On executing command `s` once again, the arguments are passed to the function parameters and the control moves to the statement `maxNumber = 0` (line 19). We verify the argument values passed from `main` function using the command `a` (line 20), which displays the following output:

```
n1 = 20
```

```
n2 = 10
```

```
n3 = 30
```

```
a: print arguments and their values for current function
```

We then use command `s` to execute the current statement that assigns value 0 to variable `maxNumber` and the control steps to the next line (conditional statement) in execution. By default, if no command is specified, and *enter* key is pressed, `pdb` repeats execution of the most recent `pdb` command i.e. command `s` in this case (line 27). Since condition `n1 > n2` yields `True`, control is transferred to statement `if n1 > n3:`. However, since the condition `n1 > n3` turned out to be `False`, the control is transferred to `return` statement. This reminds us that we need to include an `else` clause associated with the statement `if n1 > n3:` which assigns `n3` to variable `maxNumber` if the condition `n1 > n3` yields `False`. Anyway, let us complete the execution of the code as it would illustrate some more points. Now, the execution of `s` command brings the control back to the main function. Finally, the value of `maxNumber` is printed (lines 43) on using next command `n`. We exit out of the debugging mode by using command `quit` (line 51).


```
01 (Pdb) l
02 23      Return Value: None
03 24      ''
04 25      n1 = int(input('Enter first number: '))
05 26      n2 = int(input('Enter second number: '))
06 27      n3 = int(input('Enter third number: '))
07 28 ->  maximum = max3(n1, n2, n3)
08 29      print('Maximum number is', maximum)
09 30
10 31  if __name__=='__main__':
11 32      main()
12 [EOF]
13 (Pdb) s
14 --Call--
15 > f:\pythoncode\ch04\max3.py(3)max3()
16 -> def max3(n1, n2, n3):
17 (Pdb) s
18 > f:\pythoncode\ch04\max3.py(9)max3()
19 -> maxNumber = 0
20 (Pdb) a
21 n1 = 20
22 n2 = 10
23 n3 = 30
24 (Pdb) s
25 > f:\pythoncode\ch04\max3.py(10)max3()
26 -> if n1 > n2:
27 (Pdb)
28 > f:\pythoncode\ch04\max3.py(11)max3()
29 -> if n1 > n3:
30 (Pdb)
31 > f:\pythoncode\ch04\max3.py(17)max3()
32 -> return maxNumber
33 (Pdb) p maxNumber
34 0
35 (Pdb) s
36 --Return--
37 > f:\pythoncode\ch04\max3.py(17)max3()->0
38 -> return maxNumber
39 (Pdb) s
40 > f:\pythoncode\ch04\max3.py(29)main()
41 -> print('Maximum number is', maximum)
42 (Pdb) n
43 Maximum number is 0
44 --Return--
45 > f:\pythoncode\ch04\max3.py(29)main()->None
46 -> print('Maximum number is', maximum)
```

```
47 (Pdb) n
48 --Return--
49 > f:\pythoncode\ch04\max3.py(32)<module>()->None
50 -> main()
51 (Pdb) quit
52
53 Traceback (most recent call last):
54   File "F:/PythonCode/Ch04/max3.py", line 32, in <module>
55     main()
56 bdb.BdbQuit
```

Fig. 4.2 (b2) Execution of program max3.py

Often it is convenient to set some breakpoints and execute the code as usual up to a breakpoint. In the example at hand, we set the breakpoint using the command `b max3`. Using the command `c`, execution of the code continues until the breakpoint (function `max`) is reached (Fig. 4.3). Now the debugging process can be continued as before.

`b`: set the breakpoint

`c`: continue the execution till the breakpoint

```
1 >>>
2 > f:\pythoncode\ch04\max3.py(3)<module>()
3 -> def max3(n1, n2, n3):
4     (Pdb) b max3
5 Breakpoint 1 at f:\pythoncode\ch04\max3.py:3
6 (Pdb) c
7 Enter first number: 20
8 Enter second number: 10
9 Enter third number: 30
10 > f:\pythoncode\ch04\max3.py(9)max3()
11 -> maxNumber = 0
```

Fig. 4.3 Execution of program maximum.py

SUMMARY

1. Debugging is the process of examining the source code from the point of determining bugs and eliminating them to make the program bug-free.
2. Python debugger pdb is an interactive debugger for debugging Python programs. The module pdb contains the class pdb. The debugger helps in debugging the code step by step, allows setting of breakpoints, printing some lines in the vicinity of the current line of the code, supports evaluating and printing the result of evaluating an expression in the current context, and examining the stack frame
3. Function `set_trace` of module pdb is used to tell the starting point of debugging when the program is normally run. By mentioning this statement in the very beginning, we intend to specify that the program should start running in debugging mode from the starting point of its execution.
4. Python debugger pdb supports the following commands:

Command	Explanation
h or help	In the absence of any argument, it lists all the available commands. With an argument, it prints the description about that command.
w or where	Prints the stack trace (sequence of function calls currently in execution, most recent function call being at the beginning). Also shows the statement to be executed next.
u or up	Moves to the stack frame one level up in the stack trace.
d or down	Moves to the stack frame one level down in the stack trace.
s or step	Executes the current statement and moves the control to either next statement or the function that is being invoked in current statement.
n or next	Executes the current statement and moves the control to the next statement. Unlike step command, if a function is being invoked in the current statement, the invoked function gets executed completely.
r or return	Continue execution until the current function returns.
j(ump) linen o	Jumps to the given line number for the next statement to be executed.
l or list	List 11 lines in the vicinity of the current statement.
b or break [[file:]line func[,cond]]	Sets the breakpoint at the line specified (name of file optional). If the argument func specifying function name is provided, breakpoint is set at the first executable statement of the function. The second argument may be used to denote a condition which must evaluate to True for setting the breakpoint.
tbreak [[file:]line func[,cond]]	Similar to break command. However, breakpoints being set are automatically removed once they are reached.
cl or clear [[file:]line func]	Clears the specified breakpoint. In the absence of any argument, it clears all the breakpoints.

nc[,cond]	
c or conti nue	Continue execution until the breakpoint is reached.
a or args	Prints the argument list of the current function along with their values.
p or print (expr essio n)	Prints the value of the specified expression in the current context.
q or quit	Quits from the Python debugger.

EXERCISES

1. Consider the following Python code intended to compute the sum of n natural numbers. During testing, it was found that sum printed by program always excludes the last number. Debug the script (Fig. 4.4) using the debugger discussed in the chapter.

```

01 def summation(n):
02     """
03     Objective: To find sum of first n positive integers.
04     Input Parameter: n - numeric value
05     Return Value: total - numeric value
06     """
07     total = 0
08     for count in range(1, n):
09         total += count
10     return total
11
12 def main():
13     """
14     Objective: To find sum of first n positive integers based on user
15     input
16     Input Parameter: None
17     Return Value: None
18     """
19     n = int(input('Enter number of terms: '))
20     total = summation(n)
21     print('Sum of first', n, 'positive integers: ', total )
22
23 if __name__=='__main__':
24     main()

```

Fig. 4.4 Program to compute the sum of n natural numbers

2. Consider the following Python code (Fig. 4.5) intended to print inverse right triangle for given number of rows `nRows`. For example, for `nRows = 5`, the following inverted triangle should be printed:

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

During testing, it was found that program does not produce even the single line of output. Debug the following script (Fig. 4.5) using the debugger discussed in the chapter.

3. Consider the Python script given in Fig. 4.6 intended to compute the percentage. During testing, it was found that percentage computed was not accurate rather rounded to lower bound integer value. Debug the script (Fig. 4.6) using the debugger discussed in the chapter.
4. Consider the Python script in Fig. 4.7, intended to determine whether the given year is a leap year. During testing, it was found that an year such as 1800 or 2100, despite being non-leap year, was also displayed as a leap-year. Debug the function `isLeapYear` (Fig. 4.7) using the debugger discussed in the chapter.

```
01 def invertedRightTriangle(nRows):
02     """
03     Objective: To print right triangle
04     Input Parameter: nRows - integer value
05     Return Value: None
06     """
07     for i in range(nRows, 0):
08         print('*' * i)
09
10 def main():
11     """
12     Objective: To print right triangle
13     Input Parameter: None
14     Return Value: None
15     """
16     nRows = int(input('Enter no. of rows: '))
17     invertedRightTriangle(nRows)
18
19 if __name__=='__main__':
20     main()
```

Fig. 4.5 Program to print inverse right triangle

```

01 def main():
02     """
03         Objective: To display percentage of marks scored by the student
04         Input Parameter: None
05         Return Value: None
06     """
07     totalMarks = 0
08     i = 0
09     while True:
10         marks = input('Marks for subject ' + str(i + 1) + ': ')
11         if marks == '': # End of input
12             break
13         marks = int(marks)
14         if marks < 0 or marks > 100:
15             print('INVALID MARKS !! ')
16             continue # Marks to be entered again
17         i = i + 1
18         totalMarks += marks
19     percentage = totalMarks // i
20     print('Total marks', int(totalMarks))
21     print('Percentage', round(percentage,2))
22
23 if __name__=='__main__':
24     main()

```

Fig. 4.6 Program to compute percentage

```

01 def isLeapYear(year):
02     """
03         Objective: To determine whether a given year is a leap year
04             or not
05         Input Parameter: year - numeric value
06         Return Value: True if year is a leap year, False otherwise
07     """
08     # Approach: if century year, it should be divisible by 400,
09     #             else by 4.
10     return year%400==0 or year%100==0 and year%4==0

```

Fig. 4.7 Program to determine whether the given year is a leap year

5. Consider the Python script (Fig. 4.8), intended to find HCF. During testing, it was found that program yields an error for numbers having no common factor other than 1. Debug the script (Fig. 4.8) using the debugger

discussed in the chapter.

```
01 def findHCF(num1, num2):
02     """
03         Objective: To find HCF of two numbers, num1 and num2.
04         Input Parameters: num1, num2 - numeric values
05         Return Value: HCF - numeric value
06     """
07     if num1 < num2:
08         minNum = num1
09     else:
10         minNum = num2
11     for i in range(minNum, 1, -1):
12         if (num1 % i == 0) and (num2 % i == 0):
13             HCF = i
14     return HCF
15
16
17 def main():
18     """
19         Objective: To take two numbers as an input and find their HCF
20         Input Parameter: None
21         Return Value: None
22     """
23     num1 = int(input('Enter first number:: '))
24     num2 = int(input('Enter second number:: '))
25     print(findHCF(num1, num2))
26
27 if __name__=='__main__':
28     main()
```

Fig. 4.8 Program to find HCF

CHAPTER 5

SCOPE

CHAPTER OUTLINE

[5.1 Objects and Object IDs](#)

[5.2 Scope of Objects and Names](#)

In this chapter, we will review the objects and their mapping to names, the scope of names and parameter passing mechanisms in Python. Recall that in Python, the terms name and variable are used as synonyms.

5.1 OBJECTS AND OBJECT IDS

Each object in Python is assigned a unique identifier that can be accessed using the function `id`.

each Python object has a unique identifier

```

01 def main():
02     # To understand objects and their ids
03     a = 5
04     print('a = ', a, 'id(a): ', id(a))
05     b = 3 + 2
06     print ('b = ', b, 'id(b): ', id(b))
07     a = 7
08     print('a = ', a, 'id(a): ', id(a))
09     print('b = ', b, 'id(b): ', id(b))
10
11 if __name__=='__main__':
12     main()

```

Fig. 5.1 Program to illustrate objects and their ids
(objectId.py)

On an execution of the above script (Fig. 5.1), it produced the following output:

variables a and b refer to the same object 5 and thus have same id

a = 5 id(a): 10538176

b = 5 id(b): 10538176

a = 7 id(a): 10538240

b = 5 id(b): 10538176

Recall that when the script objectId is executed, Python makes a note of the definition of the function main in the global frame. Next, on encountering the if statement, Python checks whether the name of the current module is __main__. This being true, the

expression `__ name__ == '__main__'` evaluates as True, and the function `main` gets invoked. Next, the statements in the `main` function are executed in a sequence. Line 2 being a comment is ignored by Python. Execution of line 3 creates an `int` object 5 and assigns it the name `a`. This object has a unique object id but can have multiple names as the execution of the script proceeds. For example, execution of the statement

```
b = 3 + 2
```

in line 5 does not generate a new object, it only associates the name `b` to the `int` object 5 created earlier. Now, `a` and `b` have the same object id. However, execution of line 7 creates an `int` object 7 and associates it with the name `a`. The name `b` continues to be associated with `int` object 5 created earlier. It is important to emphasize that different object ids may be generated when a script is executed again.

Python Tutor: tool for visualizing execution of Python code

It comes as good news that there are some open source tools available online that can be used to visualize the execution of Python code. For example, the following link

<http://www.pythontutor.com/visualize.html#mode=display>

opens a user interface for visualizing Python code. Figure 5.2 shows this interface. We select the language Python 3.6 and copy the code in Fig. 5.1 in the box.

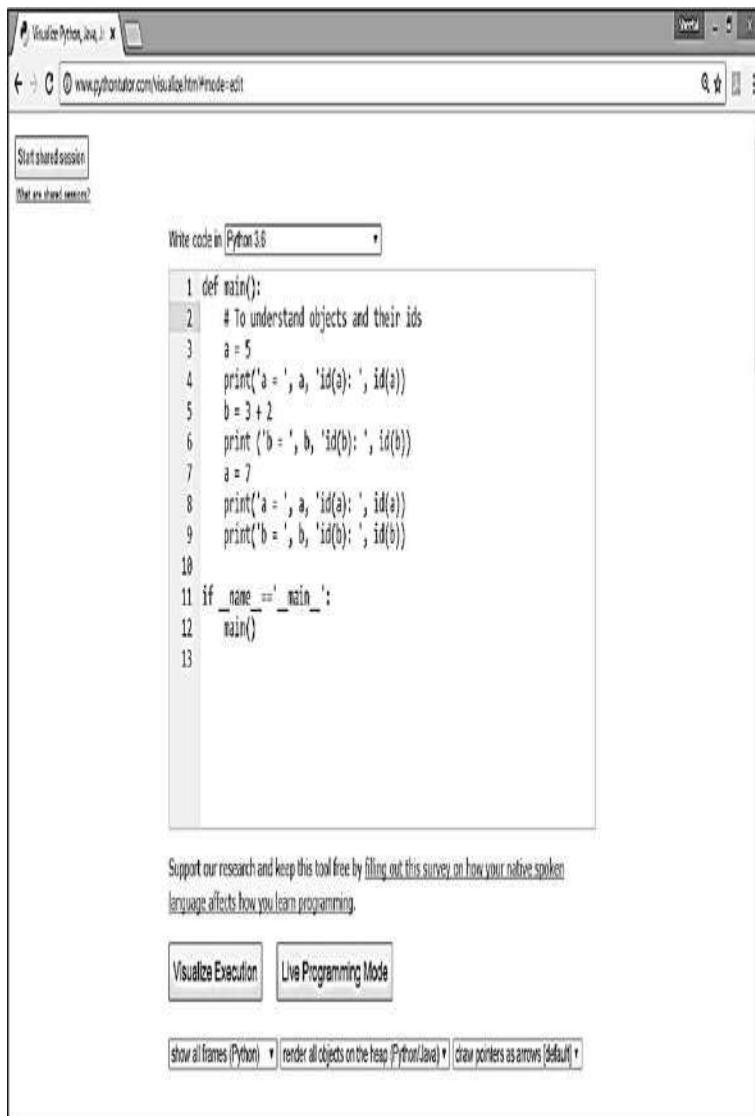


Fig. 5.2 Python tutor interface

We can see the line numbers appearing on the left of each line of code. If we want to discuss our code with a friend, we can start a shared session by clicking on the icon *Start shared session*. This would open a chat box and generate a link that we can send to our friend. Once he/she copies it in the browser, we are ready for an interactive session, and each of us can see actions performed by the other person. For our discussion in this section, we have chosen the following options:

shared session

show all frames (Python) ▾ **render all objects on the heap (Python/Java)** ▾ **draw pointers as arrows [default]** ▾

Once we have learned to use the visualizer, we can play with other options. Finally, to visualize the execution of code, we click the icon *Visualize Execution*, and a bar showing the progress of program execution appears. Clicking somewhere on this bar would execute a fraction of the code. Alternatively, we can use the forward and back buttons. To begin with, we prefer the latter option. Anytime, we want to modify the code, we can click on **Edit code**.

visualize execution

To start visualization, we click **<Forward>**. On encountering the `main` function definition, the global frame lists the identifier `main` as shown in Fig. 5.3. On the screen, we notice two arrows, a red arrow (shown as dark grey in the figure) and a green arrow (shown as light grey in the figure). The red arrow marks the next line to be executed, and the green arrow marks the line just executed. Now clicking **<Forward>**, executes the `if` statement. Clicking **<Forward>** again executes the call to the function `main` and the visualizer shows the frame for the `main` function. Clicking **<Forward>** next, moves the next line pointer to line 3, and its execution shows the creation of `int` object 5 having name `a`. Next, clicking **<Forward>** shows the output of the execution of line 4 in **<Print output>** box (Fig. 5.4). Next click executes line 5. Now, the name `b` is mapped to the `int` object 5 created earlier. Further click executes line 6 and the output appears in the **<Print output>** box. This is shown in Fig. 5.5.

<Forward> button: Move a step forward during program execution

<**Back**> button: Move a step backward during program execution

The screenshot shows a Python tutor visualization window. On the left, a code editor displays a Python script named `VisualizePython.java` with the following content:

```
1 def main():
2     # To understand objects and their ids
3     a = 5
4     print('a = ', a, 'id(a): ', id(a))
5     b = 3 + 2
6     print('b = ', b, 'id(b): ', id(b))
7     a = 7
8     print('a = ', a, 'id(a): ', id(a))
9     print('b = ', b, 'id(b): ', id(b))
10
11 if __name__ == '__main__':
12     main()
```

Below the code editor are two status messages: "line that has just executed" and "next line to execute". A note at the bottom says, "Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there." At the bottom of the code editor are navigation buttons: "First", "Back", "Step 2 of 12", "Forward", and "Last".

On the right, there is a "Print output" panel with a resize handle in the bottom-right corner. Below it is a "Frames" tab and a "Objects" tab. The "Global frame" tab is selected, showing a tree structure with a node labeled "main" pointing to a "function main()".

Fig. 5.3 Visualization in Python tutor

This screenshot shows the same Python tutor interface after the code has been executed. The "Print output" panel now contains the following text:

```
a = 5 id(a): 10538176
```

```
⇒ 5 b = 3 + 2
  6 print('b = ', b, 'id(b): ', id(b))
  7 a = 7
  8 print('a = ', a, 'id(a): ', id(a))
  9 print('b = ', b, 'id(b): ', id(b))
10
11 if __name__ == '__main__':
12     main()

Edit code | Live programming

⇒ line that has just executed
⇒ next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<<First <Back Step 7 of 12 Forward> Last>>
```

The screenshot shows a Python code editor with a visualization pane on the right. The code is as follows:

```
⇒ 5 b = 3 + 2
  6 print('b = ', b, 'id(b): ', id(b))
  7 a = 7
  8 print('a = ', a, 'id(a): ', id(a))
  9 print('b = ', b, 'id(b): ', id(b))
10
11 if __name__ == '__main__':
12     main()
```

The visualization pane is titled "Frames" and "Objects". It shows the "Global frame" containing a "function main()". Inside the "main" frame, there is a variable "a" which is an "int" object with the value "5".

Fig. 5.4 Visualization in Python tutor

Visualize Python dev | X

Start shared session

What are shared sessions?

Python 3.6

```

1 def main():
2     # To understand objects and their ids
3     a = 5
4     print('a = ', a, 'id(a): ', id(a))
5     b = 3 + 2
6     print('b = ', b, 'id(b): ', id(b))
7     a = 7
8     print('a = ', a, 'id(a): ', id(a))
9     print('b = ', b, 'id(b): ', id(b))
10
11 if __name__ == '__main__':
12     main()

```

Edit code | Live programming

← line that has just executed

→ next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<<First < Back Step 9 of 12 Forward> Last>>

Fig. 5.5 Visualization in Python tutor

The screenshot shows a Python Tutor session titled "Visualize Python, live...". The code in the editor is:

```

1 def main():
2     # To understand objects and their ids
3     a = 5
4     print('a = ', a, 'id(a): ', id(a))
5     b = 3 + 2
6     print ('b = ', b, 'id(b): ', id(b))
7     a = 7
8     print('a = ', a, 'id(a): ', id(a))
9     print('b = ', b, 'id(b): ', id(b))
10
11 if __name__ == '__main__':
12     main()

```

The "Print output" box contains the following text:

```

a = 5 id(a): 10538176
b = 5 id(b): 10538176
a = 7 id(a): 10538140
b = 5 id(b): 10538176

```

The "Frames" section shows the Global frame with the function main(). Inside, variable a points to an int object with value 5, and variable b points to an int object with value 7. The "Return value" is shown as None.

Fig. 5.6 Visualization in Python tutor

Next, clicking <**Forward**> executes line 7, resulting in creation of a new `int` object 7 and its mapping to `a`. Further clicks execute lines 8 and 9 and the output appears in the <**Print output**> box (Fig. 5.6). Next click shows return value `None` associated with the function `main` as it does not return any value. A further click brings us to the end of the program. The final response of the visualizer is shown in Fig. 5.7. If we choose the option <**Use text labels for pointers**>, the visualizer will show the object `ids` as `id1`, `id2`, etc.

The screenshot shows a Python tutor session titled "Visualize Python Java, x". The code in the editor is:

```

1 def main():
2     # To understand objects and their ids
3     a = 5
4     print('a = ', a, 'id(a): ', id(a))
5     b = 3 + 2
6     print ('b = ', b, 'id(b): ', id(b))
7     a = 7
8     print('a = ', a, 'id(a): ', id(a))
9     print('b = ', b, 'id(b): ', id(b))
10
11 if __name__ == '__main__':
12     main()

```

The output window shows:

```

a = 5 id(a): 10538176
b = 5 id(b): 10538176
a = 7 id(a): 10538240
b = 5 id(b): 10538176

```

The visualization pane shows the call stack and variable states:

- Global frame:** Function `main()`
- main:** Variable `a` is an `int` object with value `5`. Variable `b` is an `int` object with value `7`.
- Return value:** `None`

Fig. 5.7 Visualization in Python tutor

We noted above that Python created an `int` object `5` on execution of line 3. However, only a reference to that object was created on execution of line 5. The general principle is expressed by saying that Python caches or interns small integer objects (typically, up to 100) for future use. But, the same may not hold for other forms of data. For example, examine the following Python shell output:

```
>>> print(id(2.4))
```

```
46078432
```

```
>>> print(id(2.4))
```

```
47619216
```

```
>>> print(id(2.4))
```

```
47619024
```

```
>>> print(id(2.4))
```

```
46078432
```

```
>>> print(id(2.4))
```

```
47619024
```

```
>>> print(id(2.4))
```

```
47619216
```

different identifiers for the same value

Note that the first three instructions create new objects. However, subsequent instructions sometimes used the objects created earlier. The `del` operator makes a name (i.e. the association between the name and the object) undefined, for example, examine, the following sequence of statements:

```
>>> a = 5
```

```
>>> b = 5
```

```
>>> print(id(a), id(b))
```

```
10538176 10538176
```

```
>>> del a
```

```
>>> print(a)
```

```
File "<pyshell#26>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined

>>> print(b)

5

>>> del b

>>> print(b)

Traceback (most recent call last):
File "<pyshell#29>", line 1, in <module>
    print(b)
NameError: name 'b' is not defined
```

accessing an undefined name leads to error

variable b continues to refer to object 5

The first statement creates an int object 5 and binds it to name a. The second statement does not create a new object. Instead, it binds the same object to name b and thus creates another reference to int object a. Figure 5.8 illustrates this.

The screenshot shows a Python Tutor interface. On the left, a code editor displays the following Python script:

```
1 a = 5
⇒ 2 b = 5
→ 3 print(id(a), id(b))
4 print(a)
5 del a
6 print(b)
7 del b
8 print(b)
```

Below the code editor are two status messages:

- ⇒ line that has just executed
- next line to execute

A note below the code editor says: "Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there."

On the right, a visualization panel titled "Print output (drag lower right corner to resize)" shows the current state of memory. It includes tabs for "Frames" and "Objects". Under "Frames", a "Global frame" is shown with variables `a` and `b`. Variable `a` is associated with the value `5` (labeled "int"). Variable `b` also points to the same `5` value. A "Frames" tab is highlighted.

At the bottom of the visualization panel are navigation buttons: "Step 3 of 8" (highlighted in blue), "Forward >", and "Last >>".

Fig. 5.8 Visualization in Python tutor

Python keeps a count of the number of references to an object. When the statement

reference count: the number of references (names) associated with an object

`del a`

is executed, it reduces the reference count of `int` object 5 from 2 to 1 and removes the binding of name `a` to `int` object 5 as shown in Fig. 5.9. Therefore, an attempt to access `a` now yields an error. However, the name `b` continues to refer to `int` object 5 created on execution of the statement

variable `a` no more refers object 5 . reference count of object 5 decreases by 1

`b = 5`

When the statement

`del b`

The screenshot shows a Python Tutor session window. On the left, a code editor displays the following Python script:

```
Python 3.6
1 a = 5
2 b = 5
3 print(id(a), id(b))
4 print(a)
⇒ 5 del a
⇒ 6 print(b)
7 del b
8 print(b)
```

On the right, a visualization pane shows the state of variables. It contains a "Print output" box with the text "12538176 12538176" and "5". Below it is a "Frames" tab showing a "Global frame" with a variable "b" pointing to the value "5". There is also an "Objects" tab.

At the bottom, there is a toolbar with buttons: << First, < Back, Step 6 of 8, Forward >, Last >>.

Fig. 5.9 Visualization in Python tutor

This screenshot shows a later step in the same Python Tutor session. The code editor now only contains the first line of code:

```
Python 3.6
1 a = 5
```

The visualization pane remains the same, showing the "Print output" box with "12538176 12538176" and "5", and the "Global frame" showing "b" pointing to "5".

The screenshot shows a Python script being run in a tutor. The code is:

```
1 b = 5
2 print(id(a), id(b))
3 print(a)
4 del a
5 print(b)
6 print(b)
7 del b
8 print(b)
```

The output window shows the value '5'.

Below the code, there are two buttons: 'Frames' and 'Objects'. A legend indicates:

- Line that has just executed (grey arrow)
- Next line to execute (grey arrow)

A note says: "Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there." Below this is a progress bar with a slider at the end. At the bottom are navigation buttons: '<< First', '< Back', 'Step 8 of 8', 'Forward >', and 'Last >>'.

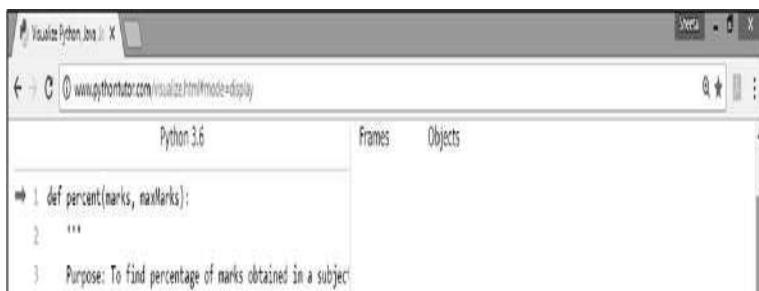
Fig. 5.10 Visualization in Python tutor

is executed, it reduces the reference count of `int` object 5 from 1 to 0 (Fig. 5.10), and removes the binding of name `b` to `int` object 5. Therefore, an attempt to access `b` now yields an error. We conclude this section by giving another example of a Python program (Fig. 5.11). We execute the script as shown in Fig. 5.12. In this figure, we see a progress bar indicating what fraction of the script we have executed so far. Below the progress bar we see the following line:

progress bar marks part of the script executed so far


```
01 def percent(marks, maxMarks):  
02     '''  
03     Purpose: To find percentage of marks obtained in a subject  
04     Input Parameters: marks, maxMarks - numeric value  
05     Return Value: percentage - numeric value  
06     '''  
07     percentage = (marks / maxMarks) * 100  
08     return percentage  
09  
10  
11 def main():  
12     # To compute percentage  
13     maxMarks = float(input('Enter total marks:: '))  
14     marks = float(input('Enter marks obtained:: '))  
15     percentage = percent(marks, maxMarks)  
16     print('Percentage is :: ', percentage)  
17  
18 if __name__=='__main__':  
19     main()
```

Fig. 5.11 Program to compute percentage (percentage.py)



```

4 Input Parameters: marks, maxMarks - numeric value
5 Return Value: percentage - numeric value
6 """
7 percentage = (marks / maxMarks) * 100
8 return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input('Enter total marks:: '))
14     marks = float(input('Enter marks obtained:: '))
15     percentage = percent(marks, maxMarks)
16     print('Percentage is :: ', percentage)
17
18 if __name__ == '__main__':
19     main()

```

[Edit code](#) | [Live programming](#)

Line that has just executed

next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<<End <<Back Step 1 of 5 Forward>> Last>>

Fig. 5.12 Visualization in Python tutor

Step 1 of 5

It means that we are about to execute the first of the five steps in the script. These five steps are as follows:

1. Definition of function `percent(marks, maxMarks)`
2. Definition of function `main()`
3. `if` statement
4. Invoking the function `main()`
5. Execution of function `main()`

On clicking **<Forward>**, step 1 is executed and we see the function `percent` in the global frame. On execution of next step, the function `main` is also shown in the global frame. When step 3 is executed, the condition `if __name__ == '__main__'` evaluates as True. Therefore, in step 4 the function `main` is invoked (Fig. 5.13). Next click executes the call to function `main` and the

visualizer shows the function `main` among frames. Next click moves the next line pointer to line 13, and its execution (shown on the right hand side) yields a message that prompts the user to enter total marks in **<Input Box>** and hit the **Submit** button (Fig. 5.14). This input message along with the values entered as input are also shown in **<Print output>**. At this stage, we enter 500 as total marks, and click **Submit**. Now execution of line 13 is complete, resulting in creation of an instance of `float` object `500.0`. This object is named as `maxMarks`. Next click executes line 14, again prompting for marks obtained (say, 450). As before, an instance of `float` object `450.0` is created, and named as `marks`. The control moves the next line pointer to line 15 (Fig. 5.15).

<Input Box>

<Print Output> box

The screenshot shows a Python Tutor interface with the following details:

- Title Bar:** VisualizePython.java X
- Address Bar:** www.pythontutor.com/visualizetut/mode=display
- Code Editor:** Python 3.6

```
1 def percent(marks, maxMarks):
2     """
3         Purpose: To find percentage of marks obtained in a subject
4         Input Parameters: marks, maxMarks - numeric value
5         Return Value: percentage - numeric value
6     """
7     percentage = (marks / maxMarks) * 100
8     return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input('Enter total marks: '))
14     marks = float(input('Enter marks obtained: '))
15     percentage = percent(marks, maxMarks)
16     print('Percentage is :: ', percentage)
17
18 if __name__ == '__main__':
19     main()
```

- Frames Tab:** Shows the Global frame containing the percent function and the main function.
- Objects Tab:** Shows the main object which contains the percent function.
- Annotations:**
 - Line 11 is highlighted with a red arrow pointing to it.
 - A red arrow points from the line "if __name__ == '__main__': main()" to the main() function in the Global frame.
- Bottom Buttons:** Edit code | Live programming, Line that has just executed, next line to execute, Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there., <<First, <Back, Step 5 of 5, Forward>, Last>

Fig. 5.13 Visualization in Python tutor

This screenshot is identical to Fig. 5.13, showing the same Python code, visualization, and interface elements for the 'VisualizePython.java' script.

```
6 """
7 percentage = (marks / maxMarks) * 100
8 return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input("Enter total marks: "))
14     marks = float(input("Enter marks obtained: "))
15     percentage = percent(maxMarks, marks)
16     print("Percentage is :: ", percentage)
17
18 if __name__ == '__main__':
19     main()
```

Edit code | Live programming

Line that has just executed
Next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<<First < Back Enter user input below: Forward> Last>

Enter total marks: 500

Fig. 5.14 Visualization in Python tutor

Visualize Python.lis | X

www.pythontutor.com/visualize.html#mode=display

Python 3.6

```
1 def percent(marks, maxMarks):
2     """
3         Purpose: To find percentage of marks obtained in a subject
4         Input Parameters: marks, maxMarks - numeric value
5         Return Value: percentage - numeric value
6     """
7     percentage = (marks / maxMarks) * 100
8     return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input('Enter total marks: '))
14     marks = float(input('Enter marks obtained: '))
15     percentage = percent(marks, maxMarks)
16     print('Percentage is :: ', percentage)
17
18 if __name__ == '__main__':
19     main()
```

Print output (drag lower right corner to resize)

Enter total marks:: 500
Enter marks obtained:: 450

Frames Objects

Global frame

percent → function percent(marks, maxMarks)

main → function main()

main

maxMarks → float 500.0

marks → float 450.0

Edit code | Live programming

line that has just executed

next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 8 of 14 Forward > Last >

The screenshot shows a Python script being run in the Python Tutor environment. The script calculates a percentage based on user input for total marks and marks obtained. The Python Tutor interface provides a visual representation of the program's state, including a call graph and variable scopes. The call graph shows the 'percent' function being called from the 'main' function, which in turn is called from the global frame. Variables are shown in the 'Objects' section, with arrows indicating their values: 'maxMarks' is 500.0 and 'marks' is 450.0.

Fig. 5.15 Visualization in Python tutor

The screenshot shows the Python tutor visualization interface. On the left, the code is displayed:

```

1 def percent(marks, maxMarks):
2     """
3         Purpose: To find percentage of marks obtained in a subject
4         Input Parameters: marks, maxMarks - numeric value
5         Return Value: percentage - numeric value
6     """
7     percentage = (marks / maxMarks) * 100
8     return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input('Enter total marks:: '))
14     marks = float(input('Enter marks obtained:: '))
15     percentage = percent(marks, maxMarks)
16     print('Percentage is :: ', percentage)
17
18 if __name__ == '__main__':
19     main()

```

On the right, the visualizer shows the state of variables:

- Frames:** Global frame, percent, main.
- Objects:**
 - Global frame: percent (function), main (function).
 - Local frame (under percent): marks (float 450.0), maxMarks (float 500.0).
 - Local frame (under main): maxMarks (float 500.0), marks (float 450.0).

Below the code editor, there are status messages and navigation controls:

- Edit code | Live programming
- Line that has just executed
- next line to execute
- Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.
- <<First <Back Step 9 of 14 >Forward> Last>

Fig. 5.16 Visualization in Python tutor

Next, clicking **<Forward>** executes the call to the function `percent` and the visualizer shows the function `percent` among frames (Fig. 5.16). Note that the parameters `marks` and `maxMarks` are mapped to `float` objects created earlier. Next, clicking **<Forward>**, moves the line pointer to line 7. Another click executes it and creates the `float` object `percentage` having value `90.0`. In the visualizer, execution of return statement at

line 8 requires two clicks. On first click, Python creates a return value i.e. the float object (to be returned to the main function) 90.0 which was created earlier as shown in Fig. 5.17. The second click returns the value 90.0 from the function percent by associating it with the variable percentage of function main (line 15). This completes the execution of line 15 (Fig. 5.18). Now, the control moves to line 16. Note that the variable percentage of the function main now maps to the float object 90.0 (Return value returned by function percent). Further, since we had opted to show exited frames, visualizer is still showing the lightly highlighted frame for the function percent. Next click shows the output on execution of line 16 in <Print output> box (Fig. 5.19) and moves the line pointer to line 17. It also shows return value None associated with the function main() as it does not return any value. A further click brings us to the end of the program. The final response of the visualizer is shown in Fig. 5.20.

The screenshot shows a Python Tutor interface with the following details:

- Code Area:**

```

1 def percent(marks, maxMarks):
2     """
3         Purpose: To find percentage of marks obtained in a subject
4         Input Parameters: marks, maxMarks - numeric value
5         Return Value: percentage - numeric value
6     """
7     percentage = (marks / maxMarks) * 100
8     return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input('Enter total marks:: '))
14     marks = float(input('Enter marks obtained:: '))
15     percentage = percent(marks, maxMarks)
16     print('Percentage is :: ', percentage)
17
18 if __name__ == '__main__':
19     main()

```
- Console Area:**

```

Enter total marks:: 500
Enter marks obtained:: 450

```
- Variables Panel:**

Variable	Type	Value
Global frame		
percent	function	percent(marks, maxMarks)
main	function	main()
main	frame	
maxMarks	float	500.0
marks	float	450.0
percent	float	90.0
marks	float	450.0
maxMarks	float	500.0
percentage	float	90.0
Return value		
- Execution Status:**
 - Line 11 is highlighted.
 - Line 12 is the next line to execute.
 - A tooltip says: "Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there."
- Page Navigation:**

<<First <<Back >>Scan 1 of 14 <<Forward>> Last>>

Fig. 5.17 Visualization in Python tutor

This screenshot shows the same Python Tutor interface as the previous one, but with a different state:

- Code Area:**

```

1 def percent(marks, maxMarks):
2     """
3         Purpose: To find percentage of marks obtained in a subject
4         Input Parameters: marks, maxMarks - numeric value
5         Return Value: percentage - numeric value

```
- Console Area:**

```

Enter total marks:: 500
Enter marks obtained:: 450

```
- Variables Panel:**

Variable	Type	Value
Global frame		
percent	function	percent(marks, maxMarks)
main	function	main()
main	frame	
maxMarks	float	500.0
marks	float	450.0
percent	float	90.0
marks	float	450.0
maxMarks	float	500.0
percentage	float	90.0
Return value		
- Execution Status:**
 - Line 1 is highlighted.
 - Line 2 is the next line to execute.
- Page Navigation:**

<<First <<Back >>Scan 1 of 14 <<Forward>> Last>>

```

6
7 percentage = (marks / maxMarks) * 100
8 return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input('Enter total marks:: '))
14     marks = float(input('Enter marks obtained:: '))
15     percentage = percent(marks, maxMarks)
16     print('Percentage is :: ', percentage)
17
18 if __name__ == '__main__':
19     main()

```

[Edit code](#) | [Live programming](#)

☞ line that has just executed
➡ next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

The Python tutor visualization shows the call stack and variable values. At the top, the Global frame contains a reference to the function `percent(marks, maxMarks)`. Below it, the `main` frame contains references to `maxMarks`, `marks`, and `percentage`. The `percent` frame contains a reference to the function `main()`. The `main` frame is expanded to show its local variables: `maxMarks` (float, 500.0), `marks` (float, 450.0), and `percentage` (float, 90.0). The `percent` frame is expanded to show its local variable: `Return value`.

Fig. 5.18 Visualization in Python tutor

Visualize Python.html X

www.pythontutor.com/visualize.html mode=display

Python 2.7

Print output (drag lower right corner to resize)

```
1 def percent(marks, maxMarks):
2     """
3         Purpose: To find percentage of marks obtained in a subject
4         Input Parameters: marks, maxMarks - numeric value
5         Return Value: percentage - numeric value
6     """
7     percentage = (marks / maxMarks) * 100
8     return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(raw_input("Enter total marks:: "))
14     marks = float(raw_input("Enter marks obtained:: "))
15     percentage = percent(marks, maxMarks)
16     print "Percentage Is :: ", percentage
17
18 if __name__ == '__main__':
19     main()
```

Frames Objects

Global frame

percent

main

main

maxMarks

marks

percentage

Return value

percent

marks

maxMarks

percentage

Return value

None

Edit code | Live programming

Line that has just executed

Next line to execute.

NEW! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to jump there.

<<First < Back Step 14 of 14 Forward > Last >>

The screenshot shows the Python Tutor visualization interface. On the left, the Python code is displayed with line numbers. Lines 1-10 define a 'percent' function that takes 'marks' and 'maxMarks' as parameters and returns their percentage. Lines 11-18 define a 'main' function that prompts the user for 'total marks' and 'marks obtained', then calls the 'percent' function with these values and prints the result. The right side of the interface shows the execution environment. At the top, there's a text input field with placeholder text 'Print output (drag lower right corner to resize)'. Below it, a message box displays 'Enter total marks:: 500' and 'Enter marks obtained:: 450'. A 'Percentage Is :: 90.0' message follows. Below this is a 'Frames' tab and an 'Objects' tab. The 'Objects' tab shows a call stack: 'Global frame' at the top, followed by 'percent' and 'main'. Arrows point from 'percent' to 'maxMarks' and 'marks' in the 'main' frame, and from 'main' to 'percentage'. The 'main' frame also contains 'Return value'. The bottom part of the visualization shows variable values: 'maxMarks' is 500.0, 'marks' is 450.0, and 'percentage' is 90.0. A 'None' value is shown for 'percent'. A legend at the bottom explains symbols: a grey bar for 'Edit code | Live programming', a red arrow for 'Line that has just executed', and a green arrow for 'Next line to execute.'. A note about breakpoints says: 'NEW! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to jump there.' At the very bottom are navigation buttons: '<<First', '< Back', 'Step 14 of 14', 'Forward >', and 'Last >>'.

Fig. 5.19 Visualization in Python tutor

The screenshot shows a Python script named `Visualize_Python.py` running in Python 3.6. The code defines a function `percent` and calls it from `main`. The output window shows the input of total marks (500) and obtained marks (450), resulting in a percentage of 90.0.

```

1 def percent(marks, maxMarks):
2     ...
3     Purpose: To find percentage of marks obtained in a subject
4     Input Parameters: marks, maxMarks - numeric value
5     Return Value: percentage - numeric value
6     ...
7     percentage = (marks / maxMarks) * 100
8     return percentage
9
10
11 def main():
12     # To compute percentage
13     maxMarks = float(input('Enter total marks:: '))
14     marks = float(input('Enter marks obtained:: '))
15     percentage = percent(marks, maxMarks)
16     print('Percentage is :: ', percentage)
17
18 if __name__ == '__main__':
19     main()

```

The right side of the interface displays a visualization of the variable scope. It shows the `Global frame` containing the `percent` function and the `main` function. Inside the `main` function, variables `maxMarks`, `marks`, and `percentage` are defined with their respective values (500.0, 450.0, 90.0). The `Return value` is also shown as 90.0. Arrows indicate the flow of data between these variables and the function call.

Fig. 5.20 Visualization in Python tutor

5.2 SCOPE OF OBJECTS AND NAMES

5.2.1 Namespaces

As the term suggests, a namespace is a space that holds some names. A namespace defines a mapping of names to the associated objects. In Python, a module, class, or function defines a namespace. Names that appear in

global frame (usually outside of the definition of classes, functions, and objects) are called global names and collectively they define the namespace called global namespace. The names introduced in a class or function are said to be local to it. The region in a script in which a name is accessible is called its scope. Thus, the scope of a name is resolved in the context of the namespace in which it is defined. For example, each of the functions `f1` and `f2` defined in a script may have the name `x`. Indeed, the variable `x` defined in function `f1` may refer to an object of type different from that of the object associated with variable `x` in the function `f2`. A namespace maps names to objects. Being an object-oriented language, definitions of functions and classes are also examples of objects.

namespace holds names and their object bindings

global names: usually appear outside of the definition of all classes, functions, and objects

scope of a name: the region of the code in which it is accessible

5.2.2 Scope

LEGB Rule

The scope rules for names in Python are often summarized as LEGB rule. LEGB stands for local, enclosing, global, and built in. All names defined within the body of a function are local to it. Function parameters are also considered local. If a name is not locally found, Python recursively searches for its definition in an enclosing scope. Names defined in the Python script but usually outside of any function or class definition are called global. Python defines some built-in names such as `len` and `abs`, which can be accessed from anywhere in a program.

LEGB rule for scope: Local, Enclosing, Global, Built-in

In light of the above discussion, the scope of a name in the context of a function, say f , may be described as follows:

All names introduced locally in function f can be accessed within it.

If a name is not defined locally in function f , but is defined in a function g , that encloses it, then it is accessible in the function f . If the name being accessed is not defined in the function g that immediately encloses f , then Python looks for it in a function that encloses g , and so on.

If a name is not defined in any enclosing function, Python looks for it outside of all functions and if found, it is accessible in function f .

We will illustrate the notion of scope and namespaces with the help of several examples.

Example 5.1

On executing the script `scope1` (Fig. 5.21), Python responds with the following output:

```
access to global variable a
```

```
global a: 4
```

Note that as the variable a has global scope, it is accessible in function f .

```
1 a = 4
2 def f():
3     print('global a: ', a)
4 f()
```

Fig. 5.21 Program to illustrate scope of variables (scope1.py)

Example 5.2

On executing the script scope2 (Fig. 5.22), Python responds with the following output:

```
a (line 4): Found locally in function f
a (line 6): global definition used
```

```
local a: 5
```

```
global a: 4.2
```

Note that the name `a` introduced in line 3 in the function `f` is local to it and has associated value 5. Thus defining the value of name `a` in the function `f`, does not affect the value of the global name `a`.

```
1 a = 4.2
2 def f():
3     a = 5
4     print('local a: ', a)
5 f()
6 print('global a: ', a)
```

Fig. 5.22 Program to illustrate scope of variables (`scope2.py`)

Example 5.3

On executing the script `scope3` (Fig. 5.23), Python responds with the following output:

```
inside function g, b: 5
```

In this example, during execution of function `g`, when the variable `a` is to be accessed, Python looks for it in the local scope of function `g`, as it is not defined in the body of function `g`, it is searched in the next enclosing scope, i.e., the scope of function `f`, where the variable `a` is indeed defined, and therefore the value 5 of the variable `a` in function `f` gets bound to the occurrence of variable `a` in the function `g`.

```
a: searched locally in function g, failed
: searched locally in function f and found
```

```
1 a = 6
2 def f():
3     a = 5
4     def g():
5         b = a
6         print('inside function g, b: ', b)
7     g()
8 f()
```

Fig. 5.23 Program to illustrate scope of variables (scope3.py)

Example 5.4

On executing the script scope4 (Fig. 5.24), Python responds with the following output:

a local variable defined in a nested scope is not visible outside it

```
in outer function g, a =
Traceback (most recent call last):
  File "F:/PythonCode/Ch05/scope4.py",
line 6, in <module>
    f()
  File "F:/PythonCode/Ch05/scope4.py",
line 5,
    in f
    print('in outer function g, a =', a)
```

```
NameError: global name 'a' is not defined
```

In this example, the variable `a` is defined in the body of inner function `g`. When we attempt to access the name `a` in line 5 of the outer function `f`, Python looks for its definition first inside the body of function `f`, as there is no definition of `a` in the function `f`, it looks for definition of `a` in the next available enclosing scope, which is the global scope. Again, there is no definition of the name `a` in global name space, and hence the error message is printed.

```
1 def f():
2     def g():
3         a = 5
4     g()
5     print('in outer function g, a =', a)
6 f()
```

Fig. 5.24 Program to illustrate scope of variables (`scope4.py`)

Example 5.5

On executing the script `scope5` (Fig. 5.25), Python responds with the following output:

```
local variable should not be referenced before assignment
```

```
Traceback (most recent call last):
```

```
  File "F:/PythonCode/Ch05/scope5.py",
line 9,
    in <module>
```

```
f()

File "F:/PythonCode/Ch05/scope5.py",
line 8,

    in f

        g()

    File "F:/PythonCode/Ch05/scope5.py",
line 5,

        in g

        b = a

UnboundLocalError: local variable 'a'
referenced before assignment
```

In this example, while executing line 5 in function `g`, when the variable `a` is accessed, Python looks for local definition of `a` in the function `g`. Indeed, it does find a definition of `a` in `g` at line 7, but that definition occurs only after line 5, where it is used. Hence, the system generates the error message: `UnboundLocalError: local variable 'a' referenced before assignment`. Here, we would like to emphasize that while accessing a global name in a function nested at any level is fine, assignment to a global name within a function yields a new local variable.

```
1 a = 4
2 def f():
3     a = 5
4     def g():
5         b = a
6         print('inside function g', 'a = ', a, 'b = ', b)
7     a = 5
8     g()
9 f()
```

Fig. 5.25 Program to illustrate scope of variables (scope5.py)

Example 5.6

On executing the script scope6 (Fig. 5.26), Python responds with the following output:

```
inside f, before calling g, a = 5
inside g, before modifying a, a = 5
inside g, after modifying a, a = 10
inside f, after calling g, a = 10
after calling f, a = 4
```

Python allows us to access a variable in an enclosing scope. Thus in the script scope3, we could access the value the local variable `a` defined in the enclosing function `f` in the nested function `g`. Similarly, in the script scope6, local definition of variable `a` in function `f` (line 3) is visible inside the nested function `g`. However, in order to modify the value of a local variable in the enclosing scope, Python provides the keyword `nonlocal`. In line 5, using the keyword `nonlocal`, we

tell Python that the name `a` being used in function `g` refers to the same object as the one associated with name `a` defined in function `f` because the function `f` is enclosing the function `g`. Thus, the assignment statement in line 7 modifies the variable `a` defined in function `f`, the new value being 10.

modifying the variable in enclosing scope using the keyword `nonlocal`

```
01 a = 4
02 def f():
03     a = 5
04     def g():
05         nonlocal a
06         print('inside g, before modifying a, ', 'a = ', a)
07         a = 10
08         print('inside g, after modifying a, ', 'a = ', a)
09     print('inside f, before calling g, ', 'a = ', a)
10     g()
11     print('inside f, after calling g, ', 'a = ', a)
12 f()
13 print('after calling f, ', 'a = ', a)
```

Fig. 5.26 Program to illustrate scope of variables (`scope6.py`)

Example 5.7

On executing the script `scope7` (Fig. 5.27), Python responds with the following output:

modifying a global variable

```
inside function g, global a = 4  
inside function f, local a = 5  
outside of all function definitions a = 4
```

In this example, in line 4, we tell Python that the variable `a` used in the scope of function `g` is from the global scope by using the keyword `global`. Thus, the assignment statement in line 5 modifies the global variable `a`, the new value being 4. However, in line 8, as the variable `a` is locally available, no attempt is made to search it in the global scope. The value 5 is assigned to local variable `a`. Of course, this assignment of value 5 to local variable in function `f` does not affect the value of global variable which remains unaltered i.e. 4.

```
01 a = 3
02 def f():
03     def g():
04         global a
05         a = 4
06         print('inside function g, global a = ', a)
07     g()
08     a = 5
09     print('inside function f, local a = ', a)
10 f()
11 print('outside of all function definitions a = ', a)
```

Fig. 5.27 Program to illustrate scope of variables (`scope7.py`)

Example 5.8

On executing the script `scope8` (Fig. 5.28), Python responds with the following output:

```
>>>

in global namespace: id(f): 40585712

global f

before re-definition of f, id(f): 40585712

after re-definition of f: , id(f):
40686704

inner f

in global namespace: id(f): 40686704

global f
```

```
inside h: id(f): 40686704

inner f

Traceback (most recent call last):

  File "F:/PythonCode/Ch05/scope8.py",
line 27,

    in <module>

  main()

  File "F:/PythonCode/Ch05/scope8.py",
line 24,

    in main

    h()

  File "F:/PythonCode/Ch05/scope8.py",
line 15,

    in h

    fprime()

NameError: global name 'fprime' is not
defined
```

In this example, we demonstrate that the same rules apply to resolve the names of function objects and numeric objects. On execution of line 18, `id(f)` is printed for the global function `f`. Next, the function `f` is assigned to the name `fprime`. On invoking the function `f`, line 2 is executed. During execution of function `g`, the execution of line 6 prints the `id(f)` for the global function `f`. Subsequently, the global function `f` is redefined in the function `g`, and the execution of line 9 prints the `id` of this modified function `f`. However, it does not affect the definition of function `fprime`. Thus, execution of line 23 invokes the function `fprime`, thereby, line 2 is executed

illustration of scope rules for function objects

```
01 def f():
02     print('global f')
03
04 def g():
05     global f
06     print('before re-definition of f, id(f): ', id(f))
07     def f():
08         print('inner f')
09     print('after re-definition of f: , id(f): ', id(f))
10     f()
11
12 def h():
13     print('inside h: id(f): ', id(f))
14     f()
15     fprime()
16
17 def main():
18     print('in global namespace: id(f): ', id(f))
19     fprime = f
20     f()
21     g()
22     print('in global namespace: id(f): ', id(f))
23     fprime()
24     h()
```

```

41     """
25
26 if __name__ == '__main__':
27     main()

```

Fig. 5.28 Program to illustrate scope of variables (`scope8.py`)

Subsequently, the function `h` is invoked, that further invokes functions `f` and `fprime`. As the definition of global function `f` has been modified, the execution of line 8 prints `inner f`. However, when we attempt to invoke function `fprime` (line 15), Python first looks for its definition within function `h`, and when it does not find the definition of function `fprime` locally, it looks for the definition in the next outer scope, i.e., the global namespace. But no definition of `fprime` is found even in the global namespace. Hence the system flags the error message `NameError: global name 'fprime' is not defined`.

SUMMARY

1. Python visualizer is an online tool for visualizing the execution of Python code.
2. A namespace defines a mapping of names to the associated objects.
3. Names that appear in global frame outside of the definition of classes, functions, and objects are called global names, and collectively they define the namespace called global namespace.
4. The names introduced in a class, or function are said to be local to it.
5. The region in a script in which a name is accessible is called its scope.
6. The scope rules for names in Python are often summarized as LEGB rule. LEGB stands for local, enclosing, global and built in.
7. If a name is not locally defined, Python recursively searches for its definition in an enclosing scope.
8. Python defines some built-in names such as `len` and `abs` which can be accessed from anywhere in a program.

EXERCISES

1. Study the program segments given below. Give the output produced, if any.

```

1. globalVar = 10
def test():
    localVar = 20

```

```

        print('Inside function test :
globalVar =', globalVar)
        print('Inside function test : localVar
=', localVar)
    test()
    print('Outside function test : globalVar
=', globalVar)
    print('Outside function test : localVar
=', localVar)
2. globalVar = 10
def test():
    localVar = 20
    globalVar = 30
    print('Inside function test :
globalVar =', globalVar)
    print('Inside function test : localVar
=', localVar)
    test()
    print('Outside function test : globalVar
=', globalVar)
3. globalVar = 10
def test():
    global globalVar
    localVar = 20
    globalVar = 30
    print('Inside function test :
globalVar =', globalVar)
    print('Inside function test : localVar
=', localVar)
    test()
    print('Outside function test : globalVar
=', globalVar)
4. def test1():
    test1.a = 10
    def test2():
        test1.a = 8
        print('Inside function test2: ',
test1.a)
        test2()
        print('Outside function test2 ',
test1.a)
    test1()
5. a = 4
def f():
    a = 5
    def g():
        nonlocal a
        a = 10
        print('inside function g,', 'a = ',
a)
    def h():
        nonlocal a
        a = 20
        print('inside function h,', 'a = ',
a)
    h()
    g()
    print('inside function f,', 'a = ', a)
f()
6. x = 2
def test():

```

```
x = x + 1
print(x)
print(x)
7. x = 2
def test():
    global x
    x = x + 1
    print(x)
print(x)
```

CHAPTER 6

STRING

CHAPTER OUTLINE

6.1 Strings

6.2 String Processing Examples

6.3 Pattern Matching

Elementary forms of data such as numeric and Boolean are called scalar data types. Several applications require more complex forms of data. For example, the name of a person, or an SMS may be expressed in the form of a string. We have already been dealing with strings. In this chapter, we shall discuss more about strings

6.1 STRINGS

As mentioned earlier, a string is a sequence of characters. A string may be specified by placing the member characters of the sequence within quotes (single, double or triple). Triple quotes are typically used for strings that span multiple lines. The following assignment statement assigns the string 'Hello Gita' to the variable message. We may also say that the string 'Hello Gita' has been bound to the name message.

a string is a sequence of characters enclosed within quotes

```
>>> message = 'Hello Gita'
```

Python function `len` finds the length of a string. Thus,

```
>>> len(message)
```

Individual characters within a string are accessed using a technique known as indexing. In Fig. 6.1, we illustrate the notion of indexing with the help of the string 'Hello Gita'.

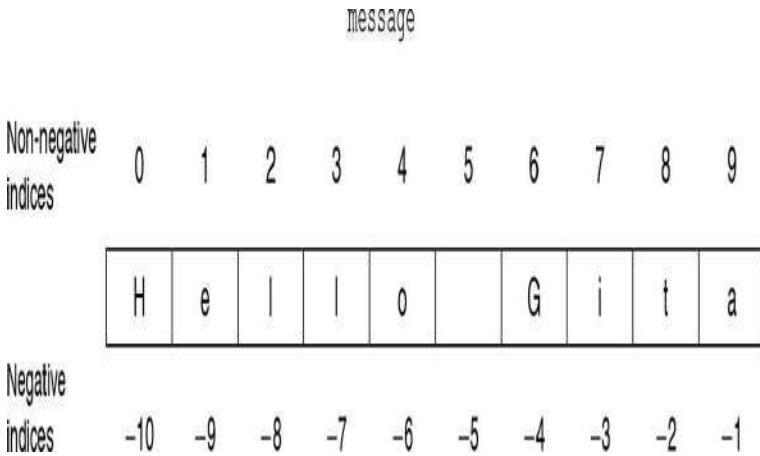


Fig. 6.1 Indexing for variable message

Note that the indices start with 0 (index for accessing the first character from the left) and end with one less than the length of the string (index for accessing the last character). The following instructions illustrate how to access individual characters in a string via indices. An index is specified within the square brackets, for example:

indexing for accessing individual characters in a string

Python allows negative as well as nonnegative indexes

```
>>> message[0]  
'H'  
  
>>> message[6]  
'G'  
  
>>> index = len(message)-1
```

```
>>> message[index]
```

```
'a'
```

The expression `message[0]` yields character 'H' since 'H' is stored at index 0. Similarly, the value of `len(message)` being 10, when `index` is set equal to `len(message)-1`, `message[index]` yields the last character of the string `message` at index 9, i.e. 'a'. Sometimes it is more convenient to access the elements of a string from the right end. For this purpose, Python provides negative indices. The negative indices range from $-(\text{length of the string})$ to -1. For the string `message`, the negative indices would range from -10 to -1. Thus, the entire range of valid indices would be $\{-10, -9, \dots, -1, 0, 1, \dots, 9\}$. Consequently, `message[-1]` would yield 'a' and `message[-index]` would yield 'e':

negative indexes: used to access the string from the right end

```
>>> message[-1]
```

```
'a'
```

```
>>> message[-index]
```

```
'e'
```

If we try to access an index which is not in the valid index range of a string, *IndexError* will be generated:

attempt to access an invalid index yields error

```
>>> message[15]
```

```
Traceback (most recent call last):
```

```
File "<pyshell#1>", line 1, in <module>
```

```
    message[15]
```

```
IndexError: string index out of range
```

Strings in Python are immutable, i.e., a component of a string cannot be altered, and an attempt to do so would lead to an error:

```
Python strings are immutable

>>> message[6] = 'S'

Traceback (most recent call last):

  File "<pyshell#2>", line 1, in <module>

    message[6] = 'S'

TypeError: 'str' object does not support item assignment
```

As mentioned earlier, strings can be concatenated using concatenation operator `+` and can be repeated a multiple number of times using the operator `*`.

```
string concatenation and repetition

>>> 'Computer' + ' Science'

'Computer Science'

>>> 'Hi' * 3

'HiHiHi'
```

Strings may also be compared using relational operators mentioned in [chapter 1](#). Also, recall that the functions `max` and `min` may be used to find maximum and minimum respectively of several values, for example:

```
max(): to find the largest value
```

```
min(): to find the smallest value

>>> max('AZ', 'C', 'BD', 'BT')

'C'

>>> min('BD', 'AZ', 'C')

'AZ'

>>> max('hello', 'How', 'Are', 'You',
'sir')

'sir'
```

6.1.1 Slicing

Sometimes, we need to retrieve a substring, also called a slice, from a string. This can be done by specifying an index range. For example, to extract the substring comprising the character sequence having indices from start to end-1, we specify the range in the form start:end as illustrated below:

```
slicing for retrieving a substring

>>> message = 'Hello Sita'

>>> message[0:5]

'Hello'

>>> message[-10:-5]

'Hello'
```

If the value of start or end is not specified, Python assumes 0 as the default value of start, and length of the string as the default value of end, for example:

```
for a slice of string s, default start index is:  
0 or -(len(s)) default end index is:  
len(s)-1 or -1
```

```
>>> message[:5]  
  
'Hello'  
  
>>> message[5:]  
  
' Sita'  
  
>>> message[:]  
  
'Hello Sita'  
  
>>> message[:5] + message[5:]  
  
'Hello Sita'  
  
>>> message[:15]  
  
'Hello Sita'  
  
>>> message[15:]  
  
''  
  
>>> message[:15] + message[15:]  
  
'Hello Sita'
```

Note that `message[:n] + message[n:]` always yields `message`, irrespective of the value of `n` being negative, positive, or even out of range. Indeed, values used for defining a slice may be arbitrary integers or even `None`, for example:

```
>>> message[8:20]  
  
'ta'  
  
>>> message[6:None]  
  
'Sita'
```

Apart from extracting a consecutive subsequence of characters from a string, Python also allows us to extract a subsequence of the form `start:end:inc`. This subsequence will include every `inc`th element of the sequence in the range `start` to `end-1`, for example:

```
>>> message[0:len(message):2]
```

```
'HloSt'
```

```
>>> message[0:len(message):3]
```

```
'HlSa'
```

We have already mentioned that a string is a sequence of characters. The slicing operation discussed above can also be applied to other sequences such as lists and tuples, to be discussed later.

6.1.2 Membership

Python also allows us to check for membership of the individual characters or substrings in strings using `in` operator. Thus, the expression `s in str1` yields `True` or `False` depending on whether `s` is a substring of `str1`, for example:

determining whether a string is a substring of another string

```
>>> 'h' in 'hello'
```

```
True
```

```
>>> 'ell' in 'hello'
```

```
True
```

```
>>> 'h' in 'Hello'
```

```
False
```

In Python, we use a `for` loop to iterate over each element in a sequence. In the following example, we

construct the string 'h e l l o' from the string 'hello' by iterating over each element of the string:

```
iterating over a string

>>> helloSpaced = ''

>>> for ch in 'hello':  
    helloSpaced = helloSpaced + ch + ' '  
  
>>> helloSpaced  
  
'h e l l o'
```

6.1.3 Built-in Functions on Strings

Next, we examine some built-in functions that can be applied on strings.

Function count

Suppose, we wish to find the number of occurrences of character 'c' in the string 'Encyclopedia'. To achieve this, we apply the function `count` with argument 'c' to the string 'Encyclopedia':

```
count() : to count occurrences of a string in another string
```

```
>>> 'Encyclopedia'.count('c')
```

```
2
```

As an application of the function `count`, examine the following code intended to find the count of all the vowels in the string 'Encyclopedia':

```
>>> vowels = 'aeiou'  
  
>>> vowelCount = 0  
  
>>> for ch in vowels:  
    if ch in 'aeiou':  
        vowelCount += 1
```

```
vowelCount += 'Encyclopedia'.count(ch)

>>> vowelCount
```

4

The system responds with 4 as the value of the `vowelCount`, even though the number of vowels in the search string '`Encyclopedia`' is 5. As the character '`E`' was not included in the string `vowels`, it was not included in counting too. To include the count of uppercase vowels also, we just need to include vowels in uppercase also in the string `vowels`:

```
vowels = 'AEIOUaeiou'
```

Rest of the code remains the same.

Functions `find` and `rfind`

Examine the string `colors`. Suppose we wish to find out whether '`red`' is present as a substring in the string `colors`. We can do so by using the function `find` that returns the index of the first occurrence of string argument '`red`' in `colors`:

```
finding index of first occurrence of a string in another string

>>> colors = 'green, red, blue, red, red,
green'

>>> colors.find('red')
```

7

To find the last occurrence of a string, we use the function `rfind` that scans the string in reversed order from the right end of the string to the beginning:

```
finding index of the last occurrence of a string in another
string
```

```
>>> colors.rfind('red')
```

23

If the function `find` fails to find the desired substring, it returns `-1`:

```
>>> colors.find('orange')
```

-1

*Functions capitalize, title, lower, upper,
and swapcase*

Python provides several functions that enable us to manipulate the case of strings. For example, the function `capitalize` can be used for converting the first letter of a string to uppercase character and converting the remaining letters in the string to lowercase (if not already so).

transforming a string to sentence case

```
>>> 'python IS a Language'.capitalize()  
  
'Python is a language'
```

Python function `title` can be used to capitalize the first letter of each word in a string and change the remaining letters to lowercase (if not already so):

```
>>> 'python IS a PROGRAMMING  
Language'.title()  
  
'Python Is A Programming Language'
```

Python functions `lower` and `upper` are used to convert all letters in a string to lowercase and uppercase, respectively. Suppose we want to check for the equivalence of a pair of email ids. Since email ids are not casesensitive, we convert both email ids to either uppercase or lowercase before testing for equality:

```
>>> emailId1 = 'geek@gmail.com'

>>> emailId2 = 'Geek@gmail.com'

>>> emailId1 == emailId2

False

>>> emailId1.lower() == emailId2.lower()

True

>>> emailId1.upper() == emailId2.upper()
```

Python function `swapcase` may be used to convert lowercase letters in a string to uppercase letters and vice versa, for example:

```
>>> 'pYTHON IS PROGRAMMING
LANGUAGE'.swapcase()

'Python is programming language'
```

Functions `islower`, `isupper`, and `istitle`

The functions `islower` and `isupper` can be used to check if all letters in a string are in lowercase or uppercase, respectively, for example:

checking case (lower/upper) of a string

```
>>> 'python'.islower()

True

>>> 'Python'.isupper()

False
```

The function `istitle` returns `True` if a string `s` (comprising atleast one alphabet) is in title case, for

example:

```
checking whether the string is in title case

>>> 'Basic Python Programming'.istitle()

True

>>> 'Basic PYTHON Programming'.istitle()

False

>>> '123'.istitle()

False

>>> 'Book 123'.istitle()

True
```

Function replace

The function `replace` allows to replace part of a string by another string. It takes two arguments as inputs. The first argument is used to specify the substring that is to be replaced. The second argument is used to specify the string that replaces the first string. For example:

```
replacing a substring with another string

>>> message = 'Amey my friend, Amey my
guide'

>>> message.replace('Amey', 'Vihan')

'Vihan my friend, Vihan my guide'
```

Functions strip, lstrip, and rstrip

The functions `lstrip` and `rstrip` remove whitespaces from the beginning and end, respectively. The function `strip` removes whitespaces from the beginning as well as the end of a string. We may choose to remove any other character(s) from the beginning or end by explicitly passing the character(s) as an argument to the function. The following examples illustrate the use of the functions `lstrip`, `rstrip`, and `strip`:

```
removing whitespace from the beginning/ end of a string

>>> ' Hello How are you! '.lstrip()

'Hello How are you! '

>>> ' Hello How are you! '.rstrip()

' Hello How are you! '

>>> ' Hello How are you! '.strip()

'Hello How are you! '
```

Functions split and partition

The function `split` enables us to split a string into a list of strings based on a delimiter. For example:

```
splitting a string into substrings

>>> colors = 'Red, Green, Blue, Orange,
Yellow, Cyan'

>>> colors.split(',')

['Red', ' Green', ' Blue', ' Orange', '
Yellow', ' Cyan']
```

Note that the function `split` outputs a sequence of strings enclosed in square brackets. A sequence of objects enclosed in square brackets defines a list. We shall discuss more about lists in the next chapter.

If no delimiter is specified, Python uses whitespace as the default delimiter:

```
>>> colors.split()  
  
['Red,', 'Green,', 'Blue,', 'Orange,',  
'Yellow,', 'Cyan']
```

The function `partition` divides a string `S` into two parts based on a delimiter and returns a tuple (discussed in the next chapter) comprising string before the delimiter, the delimiter itself, and the string after the delimiter, for example:

```
partition(): partitioning a string into two parts  
  
>>> 'Hello. How are you?'.partition('.')
```

('Hello', '.', ' How are you?')

Function join

Python function `join` returns a string comprising elements of a *sequence* separated by the specified delimiter. For example,

```
joining a sequence of strings  
  
>>> ' > '.join(['I', 'am', 'ok'])  
  
'I > am > ok'  
  
>>> ' '.join(['I', 'am', 'ok'])  
  
'I am ok'  
  
>>> ' > '.join("I", "am", "ok")  
  
" " > I > ' > , > > ' > a > m > ' > , > > '  
> o > k > ""
```

In the first example, the sequence comprises three elements, namely, 'I', 'am', and 'ok', which are combined to form the string 'I > am > ok'. In the second example, we use space as a delimiter instead of >. In the third example, each character in the string "'I > am > ok'" is an element of the sequence of characters in "'I > am > ok'".

Functions isspace, isalpha, isdigit, and isalnum

The functions `isspace`, `isalpha`, `isdigit`, and `isalnum` enable us to check whether a value is of the desire type. For example, we can check whether the name entered by a user contains only alphabets as follows:

Does a string comprises alphabets, digits, or whitespaces only?

```
>>> name = input('Enter your name : ')
```

```
Enter your name : Nikhil
```

```
>>> name.isalpha()
```

```
True
```

```
>>> name = input('Enter your name : ')
```

```
Enter your name : Nikhil Kumar
```

```
>>> name.isalpha()
```

```
False
```

Note that the blank character is not an alphabet. Similarly, to check the validity of a mobile number, we may want it to be a string of length 10 comprising of digits only. This can be achieved using functions `isdigit` and `len`:

```
>>> mobileN = input('Enter mobile no : ')
```

```
Enter mobile no : 1234567890  
>>> mobileN.isdigit() and len(mobileN)  
== 10
```

True

Python function `isspace` is used to check if the string comprises of all whitespaces:

```
check for a white space only string  
>>> '\n\t'.isspace()  
True
```

The function `isalnum` checks whether a string comprises of alphabets and digits only. For example, if the password is allowed to comprise of only alphabets and digits, we may use the function `isalnum` to ensure this constraint for a user specified password:

```
check for an alphanumeric string  
>>> password = input('Enter password : ')  
Enter password : Kailash107Ganga  
>>> password.isalnum()  
True  
>>> password = input('Enter password : ')  
Enter password : Kailash 107 Ganga  
>>> password.isalnum()  
False
```

Function startswith and endswith

Suppose we want to check if the last name of a person is Talwar. We can do this using Python function `endswith` as follows:

checking whether a string starts or ends with a particular string

```
>>> name = 'Ankita Narain Talwar'  
  
>>> name.endswith('Talwar')  
  
True
```

Similarly, to find whether a person's name begins with Dr. we can use the function `startswith` as follows:

```
>>> name = 'Dr. Vihan Garg'  
  
>>> name.startswith('Dr. ')  
  
True  
  
>>> name = ' Dr. Amey Gupta'  
  
>>> name.startswith('Dr. ')  
  
False
```

Functions encode and decode

Sometimes, we need to transform data from one format to another for the sake of compatibility. Thus, we use, Python function `encode` that returns the encoded version of a string, based on the given encoding scheme. Another function `decode` (reverse of function `encode`) returns the decoded string, for example:

encoding and decoding a string

```
>>> str1 = 'message'.encode('utf32')
```

```
>>> str1  
  
b'\xff\xfe\x00\x00m\x00\x00\x00e\x00\x00\x00s\x00\x00\x00s\x00\x00\x00a\x00\x00\x00g  
\x00\x00\x00e\x00\x00\x00'  
  
>>> str1.decode('utf32')  
  
'message'
```

One may also use alternative coding schemes such as `utf8` and `utf16`.

List of Functions

The functions described above are listed in [Table 6.1](#). Note that the original string `S` remains unchanged in each case.

Table 6.1 String functions

Functions	Explanation
S.count(str1)	Counts number of times string str1 occurs in the string S.
S.find(str1)	Returns index of the first occurrence of the string str1 in string S, and returns -1 if str1 is not present in string S.
S.rfind(str1)	Returns index of the last occurrence of string str1 in string S, and returns -1 if str1 is not present in string S.
S.capitalize()	Returns a string that has first letter of the string S in uppercase and rest of the letters in lowercase.
S.title()	Returns a string that has first letter of every word in the string S in uppercase and rest of the letters in lowercase.
S.lower()	Returns a string that has all uppercase letters in string S converted into corresponding lowercase letters.
S.upper()	Returns a string that has all lowercase letters in string S converted into corresponding uppercase letters.
S.swapcase()	Returns a string that has all lowercase letters in string S converted into uppercase letters and vice versa.
S.islower()	Returns True if all alphabets in string S (comprising atleast one alphabet) are in lowercase, else returns False.
S.isupper()	Returns True if all alphabets in string S (comprising atleast one alphabet) are in uppercase, else returns False.

Functions	Explanation
S.istitle()	Returns True if string S is in title case, i.e. only first letter of each word is capitalized and the string S contains at least one alphabet, and returns False otherwise..
S.replace(str1,str2)	Returns a string that has every occurrence of string str1 in S replaced by string str2.
S.strip()	Returns a string that has whitespaces in S removed from the beginning and the end.
S.lstrip()	Returns a string that has whitespaces in S removed from the beginning.
S.rstrip()	Returns a string that has whitespaces in S removed from the end.
S.split(delimiter)	Returns a list formed by splitting the string S into substrings. The delimiter is used to mark the split points.
S.partition(delimiter)	Partitions the string S into three parts based on delimiter and returns a tuple comprising the string before delimiter, delimiter itself and the string after delimiter.
S.join(sequence)	Returns a string comprising elements of the sequence separated by delimiter S.
S.isspace()	Returns True if all characters in string S comprise whitespace characters only, i.e. ' ', '\n', and '\t' else False.
S.isalpha()	Returns True if all characters in string S comprise alphabets only, and False otherwise.
S.isdigit()	Returns True if all characters in string S comprise digits only, and False otherwise.
S.isalnum()	Returns True if all characters in string S comprise alphabets and digits only, and False otherwise.
S.startswith(str1)	Returns True if string S starts with string str1, and False otherwise.
S.endswith(str1)	Returns True if string S ends with string str1, and False otherwise.
S.encode(encoding)	Returns S in an encoded form, based on the given encoding scheme.
S.decode(encoding)	Returns the decoded string S, based on the given encoding scheme.

6.2 STRING PROCESSING EXAMPLES

In this section, we will study several examples that illustrate the use of string processing functions described in the previous section.

6.2.1 Counting the Number of Matching Characters in a Pair of Strings

Given two strings `str1` and `str2` of alphabets, we wish to find the count of characters in `str1` that match a character in `str2`, ignoring any difference in case (lowercase or uppercase). If a character, say, `ch1`, in `str1` appears more often than once in `str2`, every occurrence of `ch1` in `str2` should be counted. For this purpose, we develop the function `nMatchedChar` (Fig. 6.2).

```

01 def nMatchedChar(str1, str2):
02     """
03     Objective: To count number of occurrences of characters in str1
04             that are also in str2
05
06     Input Parameters: str1, str2 - string
07     Return Value: count - numeric
08     """
09
10     temp1 = str1.lower()
11     temp2 = str2.lower()
12
13     count = 0
14     for ch1 in temp1:
15         # search for ch1 in temp2
16         for ch2 in temp2:
17             if ch1 == ch2:
18                 count += 1
19
20     return count

```

Fig. 6.2 Program to find number of matched characters in two strings (matchChar.py)

As we do not wish to distinguish among alphabets based on the case (lower or upper), we convert the input arguments to lowercase. To keep a count of the matched characters, we initialize the variable `count` to 0. The outer loop works by picking up a character `ch1` from `temp1` and comparing it against every character `ch2` in `temp2` in the nested loop. For each matched pair, the

variable `count` is incremented by one. At the end of the function, the value of `count` is returned. Let us examine an example:

```
> >> name1 = 'Ram Rahim'  
  
>>> name2 = 'SAMARTH RAHI'  
  
>>> nMatchedChar(name1, name2)
```

16

Note that the first 'R' in string 'Ram Rahim' matches 'R' in 'SAMARTH RAHI' at indices 4 and 8, first 'a' in string 'Ram Rahim' matches 'A' in 'SAMARTH RAHI' at indices 1, 3, and 9, first 'm' in string 'Ram Rahim' matches 'M' in 'SAMARTH RAHI' at index 2, first space character ' ' in string 'Ram Rahim' matches ' ' in 'SAMARTH RAHI' in at index 7, the second 'R' in string 'Ram Rahim' again matches 'R' in 'SAMARTH RAHI' at indices 4 and 8, and so on.

6.2.2 Counting the Number of Common Characters in a Pair of Strings

To count the number of common characters in two strings, say, `str1` and `str2`, we execute the inner loop, only if that character has not been encountered earlier. Also, we terminate the inner loop using `break` statement as soon as a character in `str1` matches a character in `str2` (Fig. 6.3).

```

01 def nCommonChars(str1, str2):
02     """
03     Objective: To count number of characters common in two strings
04     Input Parameters: str1, str2 - string
05     Return Value: count - numeric
06     """
07     temp1 = str1.lower()
08     temp2 = str2.lower()
09
10     count = 0
11     for i in range(len(temp1)):
12         ch1 = temp1[i]
13         if not (ch1 in temp1[:i]):
14             #If the character has not been encountered earlier
15             for ch2 in temp2:
16                 if ch1 == ch2:
17                     count += 1
18                     break
19     return count

```

Fig. 6.3 Function to find the number of common characters in two strings (matchChar.py)

6.2.3 Reversing a String

Next, we wish to find the reverse of a given string. For example, on reversing the string 'abccdb' we obtain the string 'bddcba'. To find reversed string of a given string str1, we begin with an empty string reverseStr. In case, the input string is a null string,

the reversed string `reverseStr` will also be a null string. In the case of a non-null string, for each character in the given string, we concatenate by appending it with the `reverseStr` built so far. The complete function `reverse` is given in Fig. 6.4.

Next, we present a recursive function for reversing a string. Note that it is comparatively easier to visualize the recursive solution. In the recursive approach (Fig. 6.5), we concatenate the reverse of the string left after removing first character i.e. `str1[1:]`, with the first character of the string `str1`, recursively, until empty string is left. In Chapters 8 and 16, we will discuss recursion in detail and give several examples of the use of recursion.

```
01 def reverse(str1):  
02     '''  
03     Objective: To reverse a string  
04     Input Parameter: str1 - string  
05     Return Value: reverse of str1 - string  
06     '''  
07     reverseStr = ''  
08     for i in range(len(str1)):  
09         reverseStr = str1[i] + reverseStr  
10    return reverseStr
```

Fig. 6.4 Function to find reverse of a string (`reverseStr.py`)

```
01 def reverse(str1):  
02     '''  
03     Objective: To reverse a string  
04     Input Parameter: str1 - string  
05     Return Value: reverse of str1 - string  
06     '''  
07     if str1 == '':  
08         return str1  
09     else:  
10        return reverse(str1[1:]) + str1[0]
```

Fig. 6.5 Program to find reverse of a string (`reverse2.py`)

Sometimes we are interested in strings that conform to a pattern, for example, we may be interested in words that terminate in `ing`. Patterns are defined using regular expressions, described below:

6.3.1 Some Important Definitions

Alphabet: An alphabet is a non-empty set of symbols, denoted by Σ .

String: A string is a finite sequence of symbols chosen from the alphabet Σ . An empty string '' containing no symbols is called null string and is often denoted by λ or ϵ . The length of a string is defined as the number of symbols in the string. The length of the null string is defined to be zero.

Language: It is the set of strings (words) defined over the alphabet Σ that conforms to some predefined pattern, or rule(s). In this section, we discuss a class of languages called regular languages. A regular language is described by a regular expression, described below. We shall use the terms regular expression and pattern interchangeably.

regular expressions: used to define regular languages

Each symbol of the alphabet defines a regular language, comprising the symbol itself. Thus, for the alphabet $\Sigma = \{0, 1\}$, 0 is a regular expression denoting the language $\{0\}$. Similarly, 1 is a regular expression denoting the language $\{1\}$. In general, if r is regular expression, $L(r)$ denotes the language described by r .

λ or ϵ is a regular expression that denotes the language comprising the null string only. Thus, $L(\epsilon) = \{\lambda\}$. The language containing no word, not even λ is called null or empty language $\{\}$, and is denoted by the regular expression φ .

empty language

If r and s are regular expressions, $r \mid s$ is also a regular expression and denotes the language $L(r \mid s) = L(r) \cup L(s)$. The operator \mid is called the union operator. Thus, the regular expression $0 \mid 1$ denotes the language $L(0 \mid 1) = L(0) \cup L(1) = \{0\} \cup \{1\} = \{0, 1\}$.

If r is a regular expression, so is (r) , denoting the same language, i.e. $L(r) = L((r))$. Parentheses are used to enforce precedence of operators in the regular expressions.

The regular expression rs denotes the language $L(rs) = L(r)L(s)$, i.e., the language formed by concatenating every string of the language $L(r)$ by every string of language $L(s)$. The regular expression $(0 \mid 1)1$ denotes the language $L((0 \mid 1)1) = L((0 \mid 1))L(1) = \{0, 1\} \{1\} = \{01, 11\}$. It is easy to see that the regular expressions $1(0 \mid 1)$ and $(0 \mid 1)1$ denote different languages. Thus, concatenation is not commutative. Concatenation has higher precedence than the union operator \mid . Thus, the regular expression $0 \mid 11$ would denote the language $L(0 \mid 11) = L(0) \cup L(11) = \{0\} \cup \{11\} = \{0, 11\}$.

If r is a regular expression, r^* is also a regular expression, that defines the language comprising the concatenation of an arbitrary number of strings that match the pattern r , i.e. belong to the language $L(r)$. Thus, $*$ denotes zero or more occurrences of the preceding pattern r . By definition, the null string also belongs to $L(r^*)$. As an example, 0^* defined over alphabet $\Sigma = \{0, 1\}$ defines the language, $L = \{\lambda, 0, 00, 000, 0000, \dots\}$. Similarly, $L((0 \mid 1)1)^* = \{01, 11\}^* = \{\lambda, 01, 11, 0101, 0111, 1101, 1111, 010101, 011101, 111101, 110101, \dots\}$.

Given a pattern r , r^+ denotes the language comprising one or more occurrences of strings that match the pattern r . Formally, $L(r^+) = \{w \in \Sigma^* : w \text{ has one or more occurrences of strings that match the regular expression } r\}$. Now $L(((0|1)1)^+) = \{01, 11, 0101, 0111, 1101, 1111, 010101, 011101, 111101, 110101, \dots\}$.

In Table 6.2, we describe the use of some important symbols used in regular expressions. In Table 6.3, we give examples of several regular expressions and the languages they denote.

For dealing with a regular expression in Python, we need to import the module `re`, which contains functions for handling the regular expressions. The function `search` of this module is used for matching a regular expression in the given string. It looks for the first location of a match in the given string. If the search is successful, it returns the `matchObject` instance matching the regular expression pattern, otherwise it returns `None`. The function `group` of `matchObject` instance returns the substring that matches the regular expression. In Table 6.4, we give some examples that illustrate the use of `search` function.

module `re` deals with pattern matching `search()`: used for pattern matching

Table 6.2 Symbols used in regular expressions

Symbols used in regular expressions	Meaning
*	Zero or more occurrences of the preceding pattern
.	Exactly one arbitrary character excluding newline
?	Zero or one occurrence of the preceding pattern
+	One or more occurrences of preceding pattern
{m}	Exactly m occurrences of the preceding pattern
{m, n}	At least m and at most n occurrences of preceding pattern. In the absence of n, there is no upper bound and in the absence of m, the lower bound is assumed to be zero.
[list-of-char]	A single character from list of characters enclosed between [], for example, [aeiou]
[.]	Matches dot (not an arbitrary character)
[a-z]	A single alphabet in the range a to z
[A-Z]	A single alphabet in the range A to Z
[0-9]	A single digit in the range 0 to 9
[^...]	When ^ occurs at the beginning of a list of symbols enclosed between [], it denotes a single character not in the list
^	Matches beginning of the string
\$	Matches end of the string or just before the newline character(if any) at the end of the string.
r ₁ r ₂	Regular expression r ₁ or r ₂
()	Groups pattern elements
\d	Any digit
\D	Any non-digit character
\s	Whitespace character
\S	Non-whitespace character
\w	Any alphanumeric character including _ (underscore character)

\W	Any non-alphanumeric character excluding _
----	--

Table 6.3 Examples of regular expressions and the corresponding languages

Regular expression	Set of matched patterns
RE1: python	{python}
RE2: (p P) ython	{python, Python}
RE3: a*	{ λ , a, aa, aaa, aaaa, aaaaa, ...}

Regular expression	Set of matched patterns
RE4: a ⁺	{a, aa, aaa, aaaa, aaaaa, aaaaaa, ...}
RE5: a?	{ λ , a}
RE6: [aeiou]	{a, e, i, o, u}
RE6: [ab]?	{ λ , a, b}
RE7: [ab] [*]	{ λ , a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, ...}
RE8: \d	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
RE9: \d{2}	{00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, ..., 99}
RE10: \d{2, 3}	{00, 01, 02, ..., 99, 000, 001, 002, ..., 999}
RE11: \D	{a, b, ..., z, A, B, ..., Z, _, *, \$, !, ...}
RE12: \w	{a, b, ..., z, A, B, ..., Z, 0, 1, ..., 9, _}
RE13: \s	{space, tab, newline, carriage return}

RE14: [^a-b]	The set comprising characters other than '0' and '1'.
RE15: (a b) (c)\$	{ac, bc} There should be no character after c in the string that matches the pattern
RE16: ^ (a) (0 1)*	{a, a0, a1, a00, a01, a10, a11, a000, ...} a should be first in the string which pattern matches
RE17: a*b	{a*b} When the backslash character (\) precedes a character with a special meaning, the special meaning of the character is ignored. For example, although the regular expression a*b defines a pattern comprising zero or more occurrences of a, followed by b, the pattern a\b, defines the string 'a*b'.

Table 6.4 Python examples of regular expressions

Regular expression	Example
Python	<pre>>>> string1 = 'Welcome to python shell' >>> match = re.search('python', string1) >>> match.group() 'python'</pre>
(p P)ython	<pre>>>> string1 = 'Welcome to Python shell' >>> match = re.search('(p P)ython', string1) >>> match.group() 'Python'</pre>

Regular expression	Example
Shel*	<pre>>>> string1 = 'Python Shell' >>> match = re.search('Shel*', string1) >>> match.group() 'Shell'</pre>

Shell?	>>> string1 = 'Python Shell' >>> match = re.search('Shell?', string1) >>> match.group() 'Shell'
Shell{1,2}	>>> string1 = 'Python Shelllll' >>> match = re.search('Shell{1,2}', string1) >>> match.group() 'Shell'
.....	>>> string1 = 'Python Shell' >>> if re.search('.....', string1): print('String length is greater than or equal to 5') String length is greater than or equal to 5.
^Python	>>> string1 = 'Python is a powerful language' >>> if re.search('^Python', string1): print('String starts with "Python"') String starts with "Python"
^power	>>> string1 = 'Python is a powerful language' >>> if re.search('^power', string1): print('String starts with "power"') else: print('String does not start with "power"') String does not start with "power"
powerful\$	>>> string1 = 'Python is a powerful language' >>> if re.search('powerful\$', string1): print('String ends with "powerful"') else: print('String does not ends with "powerful"') String does not ends with "powerful"
language\$	>>> string1 = 'Python is a powerful language' >>> if re.search('language\$', string1): print('String ends with "language"') else: print('String does not ends with "language"') String ends with "language"
\d\d\d\d\d	>>> string1 = 'Roll number is 23456' >>> match = re.search('\d\d\d\d\d', string1) >>> match.group() '23456'
\d{5}	>>> string1 = 'Roll number is 23456' >>> match = re.search('\d{5}', string1) >>> match.group() '23456'

Regular expression	Example
-[0-9]+.[0-9]+	>>> string1 = 'Decrease in price is -45.89'

	<pre>>>> match = re.search('-[0-9]+\.[0-9]+', string1) >>> match.group() '-45.89</pre>
\w*	<pre>>>> string1 = 'Python Shell' >>> match = re.search('\w*', string1) >>> match.group() 'Python'</pre>
\w*\s\w*	<pre>>>> string1 = 'We used Python Shell.' >>> match = re.search('\w*\s\w*', string1) >>> match.group() 'We used'</pre>
.*	<pre>>>> string1 = 'I use **Python**, do you?' >>> match = re.search('.*', string1) >>> match.group() 'I use **Python**, do you?'</pre>
(a b c)*	<pre>>>> string = 'abac12ccaab' >>> match = re.search('(a b c)*', string) >>> match.group() 'abac'</pre>
(a b c)*\d{1,2}c*	<pre>>>> string = 'abac12ccaab' >>> match = re.search('(a b c)*\d{1,2}c*', string) >>> match.group() 'abac12cc'</pre>
\w*\d\d.*b\$	<pre>>>> string = 'abac12ccaab' >>> match = re.search('\w*\d\d.*b\$', string) >>> match.group() 'abac12ccaab'</pre>
(a b c)*	<pre>>>> string = 'abac12ccaab' >>> match = re.search('(a b c)*', string) >>> match.group() 'abac'</pre>
(a b c)+	<pre>>>> string = 'abac12ccaab' >>> match = re.search('(a b c)+', string) >>> match.group() 'abac'</pre>

Next, we wish to find email ids from a string. A simple email id such as pranav.gupta@cs.iitd.ac.in can be expressed as a sequence of patterns:

a sequence of alphanumeric characters	denoted by [a-z0-9] +
a repeating (0 or more times) sequence of dots followed by alphanumeric characters	denoted by (\.[a-z0-9] +)*
@	denoted by @
sequence of alphabetic characters	denoted by [a-z] +
repeating (1 or more times) sequence of dot followed by alphabetic characters	denoted by (\.[a-z] +) +

Thus, an email id may be represented using the regular expression [a-z0-9] + (\.[a-z0-9] +)* @ [a-z] + (\.[a-z] +) +. Generally, a regular expression is preceded with r to denote a raw string. Use of a raw string as a regular expression avoids any confusion with some characters that have special meaning in regular expressions. Examine the following instructions:

regular expression for email-id

```
>>> match = re.search(r'[a-z0-9]+(\.[a-z0-9] +)* @ [a-z] + (\.[a-z] +) +', 'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com, raman@gmail.com')

>>> match.group()

'ram@gmail'
```

When several substrings of a given string match the regular expression, the function `group` returns the first substring matching the regular expression. However, if we wish to retrieve list of all substrings matching the regular expression, we may use the function `finditer`:

```
retrieving all substrings matching a regular expression

>>> for i in re.finditer(r'[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*@[a-zA-Z]+\([a-zA-Z]+\)+',
    'ram@gmail.com,
pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,
raman@gmail.com'):

    print(i.group())
```

ram@gmail.com

pranav.gupta@cs.iitd.ac.in

nik@yahoo.com

raman@gmail.com

Note that the null string matches the pattern
`r'(a(b|c))*'`. Next, let us find all words ending with
'ing' in a string. Again, we can use the function
`finditer` for this purpose.

```
searching for words ending with ing in a string

>>> for i in re.finditer(r'[A-Za-z]+ing',
    'Walking down the road, he was thinking
about the coming years.'):
    print(i.group())
```

Walking

thinking

coming

Another function `.findall` enables us to retrieve a list of
all the substrings matching a regular expression, for
example:

```
retrieving all matching patterns

>>> re.findall(r'[A-Za-z]+ing', 'Walking
down the road, he was thinking about the
coming years.')

['Walking', 'thinking', 'coming']
```

Next, we use the function `findall` to determine the number of words in a string as follows:

```
>>> message = 'Python:Python is an
interactive language. It is developed by
Guido Van Rossum'

>>> words = re.findall('\w+', message)

>>> len(words)
```

13

Next, we use the function `findall` to match the email ids.

```
>>> re.findall(r'([a-zA-Z0-9]+(\.[a-zA-
Z0-9]+)*@[a-zA-Z]+\(\.[a-zA-Z]+\)+)', 'ram@gmail.com,
pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,
raman@gmail.com')

[('ram@gmail.com', '', '.com'),
('pranav.gupta@cs.iitd.ac.in', '.gupta',
'.in'), ('nik@yahoo.com', '', '.com'),
('raman@gmail.com', '', '.com')]
```

Note that `findall` returns a list of matched patterns within the parenthesis only if the entire regular expression matches.

Next, given a snippet of Python code, we wish to extract all the single line comments contained in it. As a single line comment begins with `#` and terminates with

an end of line character, it can be represented using the regular expression `#.*`.

```
extracting comments

>>> pythonCode = '''

Python code to add two numbers.

'''

a = 5 #number1

b = 5 #number2

#Compute addition of two numbers

c = a + b

'''

>>> for i in re.finditer(r'(#.*)',
pythonCode):

    print(i.group())

#number1

#number2

#Compute addition of two numbers
```

Next, we wish to extract multi-line comments, enclosed in triple quotes (""""). Such a comment may be represented using the regular expression: """.*?""".

Recall that a dot in a regular expression includes any character except end of line. However, if we include `re.DOTALL` as the third argument in the function `finditer`, dot matches newline character also. For example:

```
>>> for i in re.findall(r'("".*?"")',  
pythonCode, re.DOTALL):  
  
    print(i.group())
```

Python code to add two numbers.

```
"""
```

As our final example of regular expressions, we wish to extract names of students from a string comprising comma separated names, using the `split` function:

```
>>> re.split(r',', 'Mira,Ronit,Vivek')  
  
['Mira', 'Ronit', 'Vivek']
```

However, if the string to be searched for names is a multi-line string, either of the two symbols, comma and end of line character, would serve as a separator:

```
re.split(): returns a list of the substrings delimited  
by the regular expression provided  
  
>>> re.split(r',|\n', '''Mira,Rohit,Vivek  
Aiysha,Renuka,Robin  
Sneha,Ravi''')  
  
['Mira', 'Rohit', 'Vivek', 'Aiysha',  
'Renuka', 'Robin', 'Sneha', 'Ravi']
```

SUMMARY

1. A string is a sequence of characters enclosed in quotes (single, double, or triple).
2. Strings in Python are immutable, i.e. a component of a string cannot be altered.
3. Indexing is used for accessing individual characters within the string.
4. The non-negative indices start with 0 (index for accessing the first character) and end with one less than the length of the string (index for accessing the

last character). The negative indices range from $- (length \text{ of the string})$ to -1 . An attempt to access a component of a string outside its valid index range will generate index *out of range* error.

5. Retrieving a substring from a string is called slicing.
This can be done by specifying an index range.
6. Membership of individual characters or substrings in a string can be checked using `in` operator.
7. Some of the important functions applied on strings are listed in the following table:

Functions	Explanation
S.count(str)	Counts number of times string str occurs in the string S.
S.find(str)	Returns index of the first occurrence of the string str in string S, and returns -1 if str is not present in string S.
S.rfind(str)	Returns index of the last occurrence of string str in string S, and returns -1 if str is not present in string S.
S.capitalize()	Returns a string that has first letter of the string S in uppercase and rest of the letters in lowercase.
S.title()	Returns a string that has first letter of every word in the string S in uppercase and rest of the letters in lowercase.
S.lower()	Returns a string that has all uppercase letters in string S converted into corresponding lowercase letters.
S.upper()	Returns a string that has all lowercase letters in string S converted into corresponding uppercase letters.
S.swapcase()	Returns a string that has all lowercase letters in string S converted into uppercase letters and vice versa.
S.islower()	Returns True if all alphabets in string S are in lowercase, else False.
S.isupper()	Returns True if all alphabets in string S are in uppercase, else False.
S.istitle()	Returns True if string S is in title case, i.e. first letter of each word is capitalized and the string S contains at least one alphabet.
S.replace(str1, str2)	Returns a string that has every occurrence of string str1 in S replaced with an occurrence of string str2.
S.strip()	Returns a string that has whitespaces in S removed from the beginning and the end.
S.lstrip()	Returns a string that has whitespaces in S removed from the beginning.
S.rstrip()	Returns a string that has whitespaces in S removed from the end.
S.split()	Returns a list formed by splitting the string S into

<code>s. it(de limite r)</code>	substrings. The delimiter is used to mark the split points.
<code>S.par titio n(del imiter)</code>	Partitions the string S into three parts based on delimiter and returns a tuple comprising the string before delimiter, delimiter itself and the string after delimiter.
<code>S.joi n(seq uence)</code>	Returns a string comprising elements of the sequence separated by delimiter S.
<code>S.iss pace()</code>	Returns True if all characters in string S comprise whitespace characters only, i.e. ' ', '\n', and '\t' else False.
<code>S.isa lpha()</code>	Returns True if all characters in string s comprise alphabets only, and False otherwise.
<code>S.isd igit()</code>	Returns True if all characters in string S comprise digits only, and False otherwise.
<code>S.isa lnum()</code>	Returns True if all characters in string s comprise alphabets and digits only, and False otherwise.
<code>S.sta rtswi th(st r)</code>	Returns True if string s starts with string str, and False otherwise.
<code>S.end swith (str)</code>	Returns True if string S ends with string str, and False otherwise.
<code>S.en code(e ncodi ng)</code>	Returns S in an encoded format, based on the given encoding scheme.
<code>S.de code(e ncodi ng)</code>	Returns the decoded string S, based on the given encoding scheme.

8. A regular expression defines a pattern. It is used for extracting sequences of characters in a string that match the pattern. A regular expression involves symbols such as symbols from the alphabet of the language, null string, parenthesis, * (star), and + (plus).
9. We need to import the `re` module for working with regular expressions.
10. The function `search` of the Python module `re` is used for matching a regular expression in a given string. It searches for the first location of a match and returns

the `matchObject` instance matching the regular expression, and `None` otherwise.

EXERCISES

1. Write a function that takes a string as a parameter and returns a string with every successive repetitive character replaced with a star(*). For example, '`'balloon'`' is returned as '`'bal*o*n'`'.
2. Write a function that takes two strings and returns `True` if they are anagrams and `False` otherwise. A pair of strings is anagrams if the letters in one word can be arranged to form the second one.
3. Write a function that takes a sentence as an input parameter and displays the number of words in the sentence.
4. Write a function that takes a sentence as an input parameter and replaces the first letter of every word with the corresponding uppercase letter and rest of the letters in the word by corresponding letters in lowercase without using built-in function.
5. Write a function that takes a string as an input and determines the count of the number of words without using regular expression.
6. What will be the output on executing each of the statements, following the assignment statement:
`address = 'B-6, Lodhi road, Delhi'`
 1. `len(address)`
 2. `address[17:-1]`
 3. `address[-len(address) : len(address)]`
 4. `address[:-12] + address[-12:]`
 5. `address.find('delhi')`
 6. `address.swapcase()`
 7. `address.split(',')`
 8. `address.isalpha()`
7. Examine the following string:
`greeting = 'Good Morning. Have a Good Day!!'`
What will be the output for the following function calls:
 1. `greeting.count('Good')`
 2. `greeting.find('a')`
 3. `greeting.rfind('a')`
 4. `greeting.capitalize()`
 5. `greeting.lower()`
 6. `greeting.upper()`
 7. `greeting.swapcase()`
 8. `greeting.istitle()`
 9. `greeting.replace('Good', 'Sweet')`
 10. `greeting.strip()`
 11. `greeting.split()`
 12. `greeting.partition('.')`
 13. `greeting.startswith('good')`
 14. `greeting.endswith('!!!')`
8. Determine the patterns extracted by the following regular expressions.
 1. `string1 = 'Python Programming Language'`
`1. match1 = re.search('.....m?', string1)`
`print(match1.group())`
 2. `match2 = re.search('.....m{1,2}', string1)`

```

        print(match2.group())
3.match3 = re.search('.*Language$', string1)
        print(match3.group())
4.match4 = re.search('\w*\s\w*', string1)
        print(match4.group())
5.match5 = re.search('.*', string1)
        print(match5.group())
2.string2 = 'Car Number DL5645'
1.match1 = re.search('\w\w?\d{1,4}', string2)
        print(match1.group())
2.match2 = re.search('.*5', string2)
        print(match2.group())
3.match3 = re.search('.*5?', string2)
        print(match3.group())
4.match4 = re.search('\d{3}', string2)
        print(match4.group())
5.match5 = re.search('^C.*5$', string2)
        print(match5.group())
3.string3 = 'cdcccdcccc343344aabb'
1.match1 = re.search('(c|d)*\d*(a|b)*', string3)
        print(match1.group())
2.match2 = re.search('(cd)*d', string3)
        print(match2.group())
3.match3 = re.search('(cc|cd)*(3|4)*(aa|bb)', string3)
        print(match3.group())
4.match4 = re.search('(cc|cd|dd)*(3|4)*(aa|bb)', string3)
        print(match4.group())
5.match5 = re.search('(cc|cd|dd)*(3|4)*(aa|bb)*', string3)
        print(match5.group())

```

CHAPTER 7

MUTABLE AND IMMUTABLE OBJECTS

CHAPTER OUTLINE

- [7.1 Lists](#)
- [7.2 Sets](#)
- [7.3 Tuples](#)
- [7.4 Dictionary](#)

Elementary forms of data such as numeric and Boolean are called scalar data types. Several applications require more complex forms of data, for example, the name of a person, coordinates of a point, a set of objects, or a list of personal records of individuals. In the previous chapter, we discussed about strings (type `str` – an immutable structure). In this chapter, we will discuss some other mutable and immutable objects, namely, lists, tuples, sets, and dictionaries.

7.1 LISTS

A list is an ordered sequence of values. It is a non-scalar type. Values stored in a list can be of any type such as string, integer, float, or list, for example, a list may be used to store the names of subjects:

list: a comma separated ordered sequence of values, enclosed in square brackets

```
>>> subjects=['Hindi', 'English', 'Maths',  
'History']
```

Note that the elements of a list are enclosed in square brackets, separated by commas. Elements of a list are arranged in a sequence beginning index 0, just like characters in a string. In Fig. 7.1, we show the representation of the list `subjects` as in PythonTutor.

indexing in a list

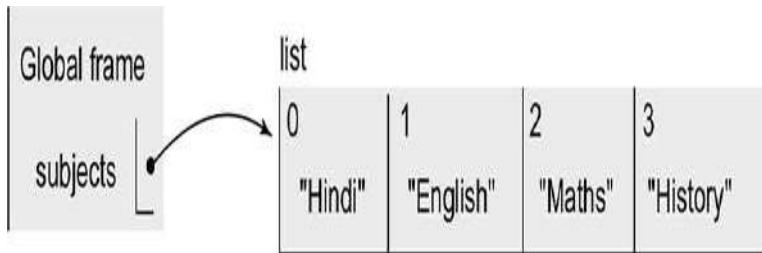


Fig. 7.1 Representation of list `subjects`

To understand the lists better, let us invoke the function `id` that returns object identifier for the list object `subjects`:

```
>>> id(subjects)
```

```
57135752
```

Next, examine the following statements:

different variables may refer to the same list object

```
>>> temporary = subjects
```

```
>>> id(temporary)
```

```
57135752
```

Note that each of the names `subjects` and `temporary` is associated with the same list object having object id 57135752. PythonTutor representation of the lists `subjects` and `temporary`, at this point, is shown in **Fig. 7.2.**

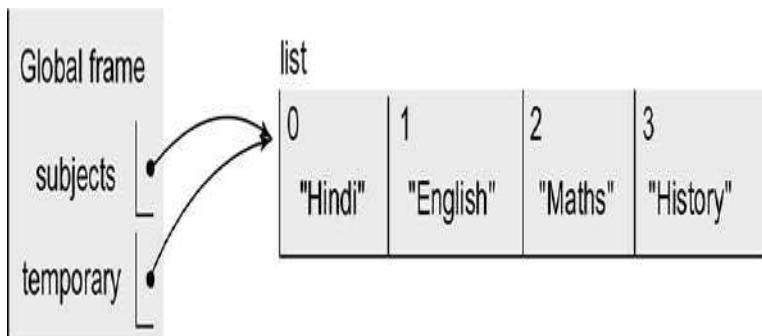


Fig. 7.2 Representation of the lists `subjects` and `temporary`

This method of accessing an object by different names is known as aliasing. Unlike strings, lists are mutable, and therefore, one may modify individual elements of a list. However, modifying an element of a list does not alter its object id, for example:

aliasing: to access objects using different names

lists are mutable

```
>>> temporary[0] = 'Sanskrit'

>>> print(temporary)

['Sanskrit', 'English', 'Math', 'History']

>>> print(subjects)

['Sanskrit', 'English', 'Math', 'History']

>>> print(id(subjects), id(temporary))

57135752 57135752
```

As each of the two names `temporary` and `subjects` refers to the same list, on modifying the component `temporary[0]` of the list `temporary`, the change is reflected in `subjects[0]` as shown as shown in Fig. 7.3.

Next, we create a list of subjects and their corresponding subject codes. For this purpose, we represent each pair of *subject* and *subject code* as a list, and form a list of such lists:

two-dimensional list: list of lists

```
>>> subjectCodes = [ ['Sanskrit', 43],  
['English', 85] , ['Maths', 65],  
['History', 36]]
```

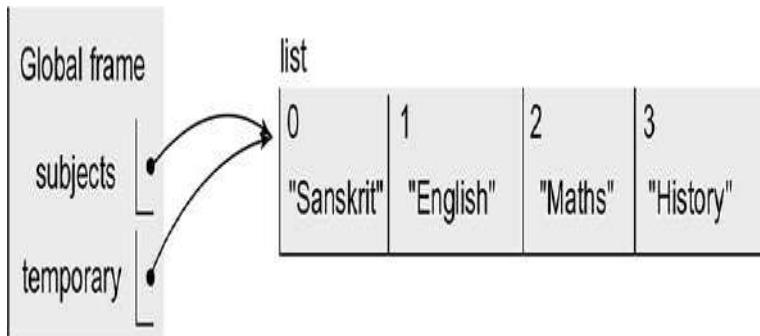


Fig. 7.3 Representation of lists `subjects` and `temporary`

A list of lists such as `subjectCodes`, each of whose elements itself is a list, is called a two-dimensional list. Thus, `subjectCodes[1]` being a list, its components may be accessed as `subjectCodes[1][0]` and `subjectCodes[1][1]`:

```
>>> subjectCodes[1]
```

```
['English', 85]
```

```
>>> print(subjectCodes[1]  
[0],subjectCodes[1][1])
```

```
English 85
```

As expected, `subjectCodes[1]` yields the list `['English', 85]`, i.e. element at index 1 of the list `subjectCodes`. As `subjectCodes[1]` is itself a list, `subjectCodes[1][0]` and `subjectCodes[1][1]` yield elements of the list `subjectCodes[1]` at indices 0

and 1, i.e. 'English' and 85, respectively. In general, when we write `subjectCodes[i][j]`, the first index `i` yields the list `subjectCodes[i]` in `subjectCodes`, and the second index `j` yields the element `subjectCodes[i][j]` within the list `subjectCodes[i]`.

As seen above, values contained in a list may be of different types. As another example, we may like to store name, address, and mobile number of a person in the list `details`:

heterogeneous list: list with values of different types

```
>>> details = ['Megha Verma', 'C-55, Raj  
Nagar, Pitam Pura, Delhi - 110034',  
9876543210]
```

Often times, we need to take a list as an input from the user. For this purpose, we use the function `input` for taking the input from the user, and subsequently apply the function `eval` for transforming the raw string to a list:

use of `eval` for converting raw string to a list

```
>>> details = eval(input('Enter details of  
Megha: '))  
  
Enter details of Megha: ['Megha Verma',  
'C-55, Raj Nagar, Pitam Pura, Delhi -  
110034', 9876543210]  
  
>>> details  
  
['Megha Verma', 'C-55, Raj Nagar, Pitam  
Pura, Delhi - 110034', 9876543210]
```

7.1.1 Summary of List Operations

In Table 7.1, we describe some functions and operations on lists with the help of the following lists:

```
>>> list1 = ['Red', 'Green']
```

```
>>> list2 = [10, 20, 30]
```

Table 7.1 Summary of operations that can be applied on lists

Operation	Example
Multiplication operator *	>>> list2 * 2 [10, 20, 30, 10, 20, 30]
Concatenation operator +	>>> list1 = list1 + ['Blue'] >>> list1 ['Red', 'Green', 'Blue']
Length operator len	>>> len(list1) 3
Indexing	>>> list2[-1] 30
Slicing Syntax: start:end:inc	>>> list2[0:2] [10, 20] >>> list2[0:3:2] [10, 30]
Function min	>>> min(list2) 10
Function max	>>> max(list1) 'Red'
Function sum (Not defined on strings)	>>> sum(list2) 60
Membership operator in	>>> 40 in list2 False

The membership operator `in` may be used in a `for` loop for sequentially iterating over each element in the list,

for example:

```
iterating over the elements of a list

>>> students = ['Ram', 'Shyam', 'Gita',
'Sita']

>>> for name in students:

    print(name)

Ram

Shyam

Gita

Sita
```

As another example, given a list of marks in a subject, let us find the number of students who have obtained marks ≥ 75 .

```
>>> marksList = [78, 32, 89, 99, 56]

>>> count = 0

>>> for marks in marksList:

    if marks >= 75:

        count += 1

>>> count
```

3

7.1.2 Function list

The function `list` takes a sequence as an argument and returns a list. For example, given a string of vowels '`'aeiou'`', we can convert it to the list of vowels for further use as follows:

converting a sequence to a list

```
>>> vowels = 'aeiou'  
  
>>> list(vowels)  
  
['a', 'e', 'i', 'o', 'u']
```

7.1.3 Functions append, extend, count, remove, index, pop, and insert

append: The function `append` insert the object passed to it at the end of the list, for example:

inserting an object at the end of a list

```
>>> a = [10, 20, 30, 40]  
  
>>> a.append(35)  
  
>>> a  
  
[10, 20, 30, 40, 35]
```

extend: The function `extend` accepts a sequence as an argument and puts the elements of the sequence at the end of the list, for example:

inserting a sequence of objects at the end of a list

```
>>> a = [10, 20, 30]  
  
>>> a.extend([35, 40])  
  
>>> a  
  
[10, 20, 30, 35, 40]  
  
>>> a.extend('abc')
```

```
>>> a  
[10, 20, 30, 35, 40, 'a', 'b', 'c']
```

count: The function `count` returns the count of the number of times the object passed as an argument appears in the list, for example:

counting the number of occurrences of an object in a list

```
>>> a = [10, 20, 30, 10, 50, 20, 60, 20,  
30, 55]
```

```
>>> a.count(20)
```

```
3
```

pop: The function `pop` returns the element from the list whose index is passed as an argument, while removing it from the list, for example:

removing and returning the element with the given index

```
>>> a = [10, 20, 30, 10, 50, 20, 60, 20,  
30, 55]
```

```
>>> a.pop(3)
```

```
10
```

```
>>> a.pop(3)
```

```
50
```

```
>>> a
```

```
[10, 20, 30, 20, 60, 20, 30, 55]
```

remove: The function `remove` takes the value to be removed from the list as an argument, and removes the first occurrence of that value from the list, for example:

removing the given element from the list

```
>>> a.remove(20)

>>> a

[10, 30, 20, 60, 20, 30, 55]
```

Next, suppose we have two lists, `names` – the list of the names of students and `rollNums` – the list of corresponding roll numbers:

```
>>> names = ['Ram', 'Shyam', 'Sita',
   'Gita', 'Sita']

>>> rollNums = [1, 2, 3, 4, 5]
```

To remove a student, 'Shyam' from the lists, we cannot apply the function `remove` as we do not know the *roll number* of 'Shyam'. However, since there is a correspondence between student names and their roll numbers, knowing the index of a student in the list `names` will solve the problem. Python function `index` can be used for finding the index of a given value in the list.

```
index() : to determine index of an element

>>> rollNums.pop(names.index('Shyam'))

2

>>> names.remove('Shyam')

>>> print(names, rollNums)

['Ram', 'Sita', 'Gita', 'Sita'] [1, 3, 4,
5]
```

`del`: The `del` operator is used to remove a subsequence of elements (`start:end:increment`) from a list, for

example:

```
deleting elements in the given index range
```

```
deleting an object
```

```
>>> a = [10, 20, 30, 20, 60, 20, 30, 55]
```

```
>>> del a[2:6:3]
```

```
>>> a
```

```
[10, 20, 20, 60, 30, 55]
```

```
>>> del a
```

```
>>> a
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#14>", line 1, in <module>
```

```
    a
```

```
NameError: name 'a' is not defined
```

Note that on execution of statement `del a`, name `a` no more refers to the list object. Informally, we may say that the object `a` has been deleted or `a` has been deleted.

```
attempt to access a deleted object leads to an error
```

7.1.4 Function `insert`

The `insert` function can be used to insert an object at a specified index. This function takes two arguments: the index where an object is to be inserted and the object itself, for example:

inserting an object at the specified index

```
>>> names = ['Ram', 'Sita', 'Gita',
'Sita']

>>> names.insert(2, 'Shyam')

>>> names

['Ram', 'Sita', 'Shyam', 'Gita', 'Sita']
```

7.1.5 Function `reverse`

The function `reverse` reverses the order of the elements in a list, for example:

reversing the order of elements in the list

```
>>> names = ['Ram', 'Sita', 'Sita',
'Anya']

>>> names.reverse()

>>> names

['Anya', 'Sita', 'Sita', 'Ram']
```

7.1.6 Functions `sort` and `reverse`

The function `sort` can be used to arrange the elements in a list in ascending order. For example, the names of students in the list `names` can be arranged in an ascending order using the function `sort`:

sorting the elements of the list

```
>>> names = ['Ram', 'Sita', 'Sita',
'Anya']

>>> names.sort()
```

```
>>> names  
  
['Anya', 'Ram', 'Sita', 'Sita']
```

If we wish to sort the list of names in descending order, we may use the argument `reverse = True` while invoking the function `sort`:

```
use keyword argument reverse = True for sorting the  
elements of a list in descending order
```

```
>>> names = ['Ram', 'Sita', 'Sita', 'Anya']  
  
>>> names.sort(reverse = True)  
  
>>> names  
  
['Sita', 'Sita', 'Ram', 'Anya']
```

Sometimes, we need to sort objects in a list based on key values obtained by applying a function to the objects in the list. For example, we may like to sort the strings in a list, based on their length. To achieve this, we need to provide an argument of the form `key = function` as illustrated below:

```
sorting the list according to the specific criteria  
  
>>> names = ['Ram', 'Sita', 'Purushottam',  
'Shyam']  
  
>>> names.sort(key = len)  
  
>>> names  
  
['Ram', 'Sita', 'Shyam', 'Purushottam']
```

7.1.7 List Functions

In Table 7.2, we list some important built-in functions that can be applied to lists. Many of these functions such as `append`, `reverse`, `sort`, `extend`, `pop`, `remove`,

and `insert` modify the list. Functions such as `count` and `index` do not modify the list. The function `dir` can be used to output the list of all the attributes including the functions that can be applied to a particular type. For example, `dir(list)` outputs the list including all the functions that can be applied to objects of the type `list`. Similarly, `dir(str)` outputs the list including all the functions that can be applied to objects of the type `str`.

`functions count and index do not modify the list`

Table 7.2 List functions

Function	Explanation
<code>L.append(e)</code>	Inserts the element <code>e</code> at the end of the list <code>L</code> .
<code>L.extend(L2)</code>	Inserts the items in the sequence <code>L2</code> at the end of the elements of the list <code>L</code> .
<code>L.remove(e)</code>	Removes the first occurrence of the element <code>e</code> from the list <code>L</code> .
<code>L.pop(i)</code>	Returns the element from the list <code>L</code> at index <code>i</code> , while removing it from the list.
<code>L.count(e)</code>	Returns count of occurrences of the object <code>e</code> in the list <code>L</code> .
<code>L.index(e)</code>	Returns index of an object <code>e</code> , if present in the list <code>L</code> .
<code>L.insert(i, e)</code>	Inserts element <code>e</code> at index <code>i</code> in the list <code>L</code> .
<code>L.sort()</code>	Sorts the elements of the list <code>L</code> .
<code>L.reverse()</code>	Reverses the order of elements in the list <code>L</code> .

7.1.8 List Comprehension

List comprehension provides a shorthand notation for creating lists. Suppose we want to create a list that contains cubes of numbers ranging from 1 to 10. To do this, we may create an empty list, and use a `for`-loop to append the elements to this list:

```
>>> cubes = []

>>> end = 10

>>> for i in range(1, end + 1):

    cubes.append(i ** 3)

>>> cubes

[1, 8, 27, 64, 125, 216, 343, 512, 729,
1000]
```

Alternatively, a simple one line statement can be used for achieving the same task.

shorthand notation for creating a list of cubes

```
>>> cubes = [x**3 for x in range(1, end +
1)]
```

The expression `[x**3 for x in range(1, end + 1)]` creates a list containing cube of each element in the range `(1, end + 1)`. Next, suppose we have a list comprising information about names and heights of the students. The name of each student is followed by the height of the corresponding student in centimetres. We wish to create a list comprising the students in the list `lst` whose height exceed or equal a threshold:

```
>>> lst = [['Rama', 160], ['Anya', 159],
['Mira', 158], ['Sona', 161]]

>>> threshold = 160
```

The desired list can be easily created using comprehensions:

```
>>> tall = [x for x in lst if x[1] >=
threshold]
```

```
>>> tall
```

```
[['Rama', 160], ['Sona', 161]]
```

Next, given a list `s1` of alphabets and another list `s2` of numbers, we wish to create a cross product of `s1` and `s2`. Each element of the list `crossProduct` is a two-element list of the type [*alphabet, number*]. This is easily achieved using comprehensions:

list comprehension for computing cross product of two lists

```
>>> s1 = ['a', 'b', 'c']
```

```
>>> s2 = [3, 5]
```

```
>>> crossProduct = [[x, y] for x in s1 for
y in s2]
```

```
>>> crossProduct
```

```
[['a', 3], ['a', 5], ['b', 3], ['b', 5],
['c', 3], ['c', 5]]
```

7.1.9 Lists as Arguments

Let us examine the script in Fig. 7.4. The function `listUpdate` updates the list `a` (line 10) by replacing the object `a[i]` by `value`. The `main` function invokes the function `listUpdate` with the arguments `lst`, `1`, and `15` corresponding to the formal parameters `a`, `i`, and `value`. As arguments are passed by reference, during execution of the function `listUpdate`, an access to the formal parameter `a` means access to the list `lst` created in the `main` function. Consequently, when we update the list `a` in the function `listUpdate`, it results in the corresponding update of the list `lst`, as visualized in PythonTutor (Fig. 7.5). Thus, the value at index `1` of the list `lst` gets updated to the value `15`.

arguments are passed by reference

```
01 def listUpdate (a, i, value):
02     """
03     Objective: To change value at a particular index in list
04     Input Parameters:
05         a - list
06         i - index of the object in the list to be updated
07         value - modified value at index i
08     Return Value: None
09     """
10    a[i] = value
11
12 def main():
13     """
14     Objective: To change value at a particular index in list
15     Input Parameter: None
16     Return Value: None
17     """
18    lst = [10, 20, 30, [40, 50]]
19    listUpdate(lst,1,15)
20    print(lst)
21
22 if __name__ == '__main__':
23     main()
```

Fig. 7.4 List passed as an argument

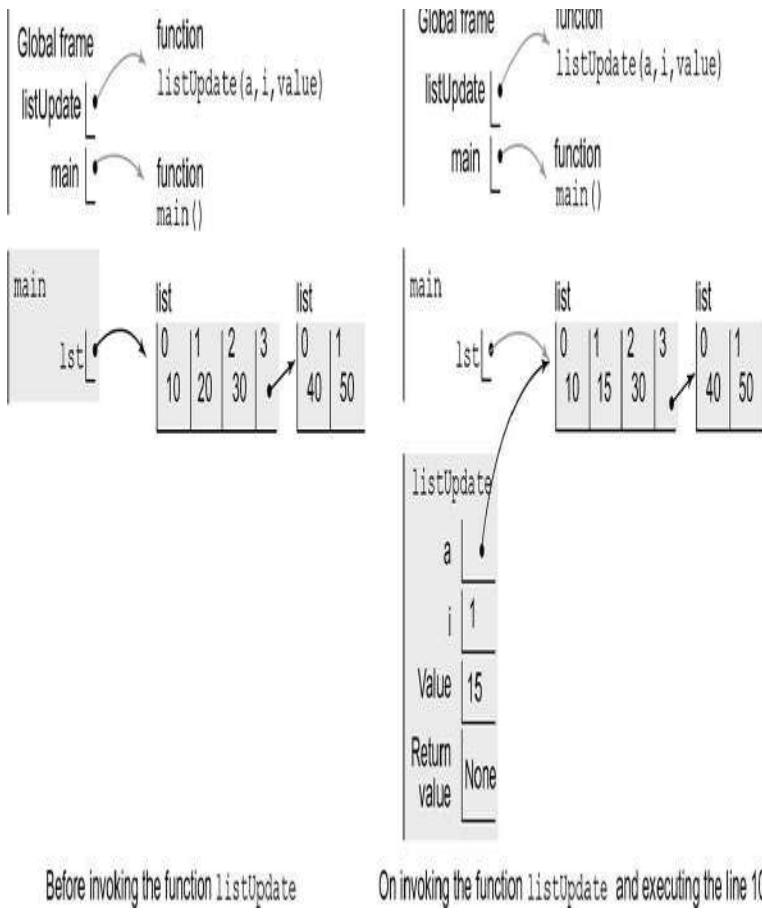


Fig. 7.5 List `lst` before and after the call to the function `listUpdate`

7.1.10 Copying List Objects

In the beginning of the chapter, we mentioned that a list is a sequence of values. To be more precise, a list comprises a sequence of references to values (objects). We have seen earlier that if the name `list1` refers to a list, the assignment of `list1` to another name, say, `list2` does not create a new list

assigning a list to another name does not create another copy of the list, instead it creates another reference

```
>>> list1 = [10, 20, [30, 40]]  
>>> list2 = list1
```

As the names `list1` and `list2` refer to the same list object (Fig. 7.6), any changes made in the list will relate to both the names `list1` and `list2`, for example (Fig. 7.7):

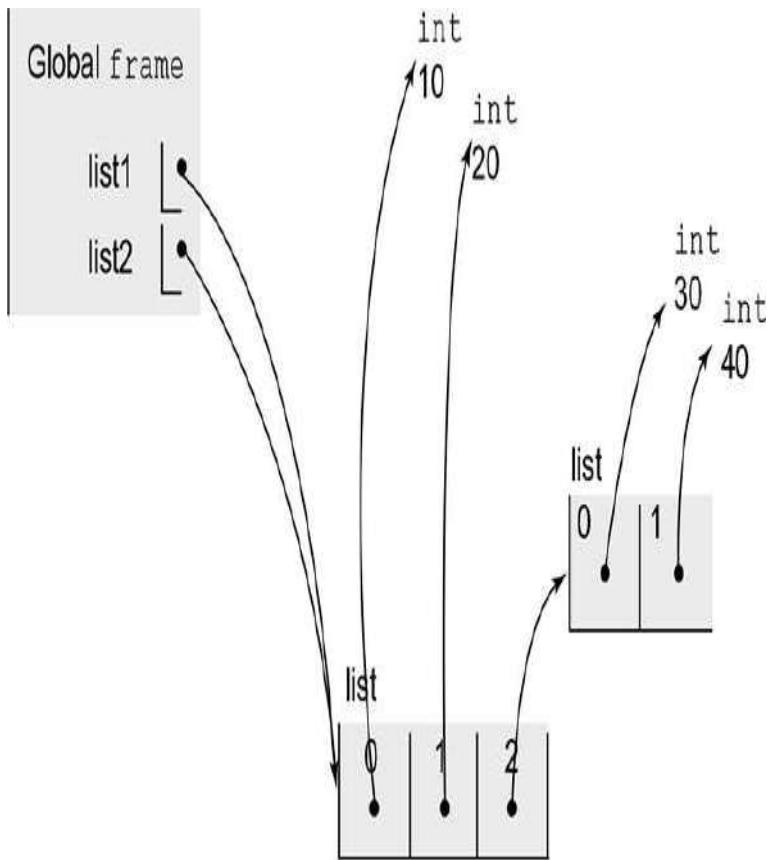
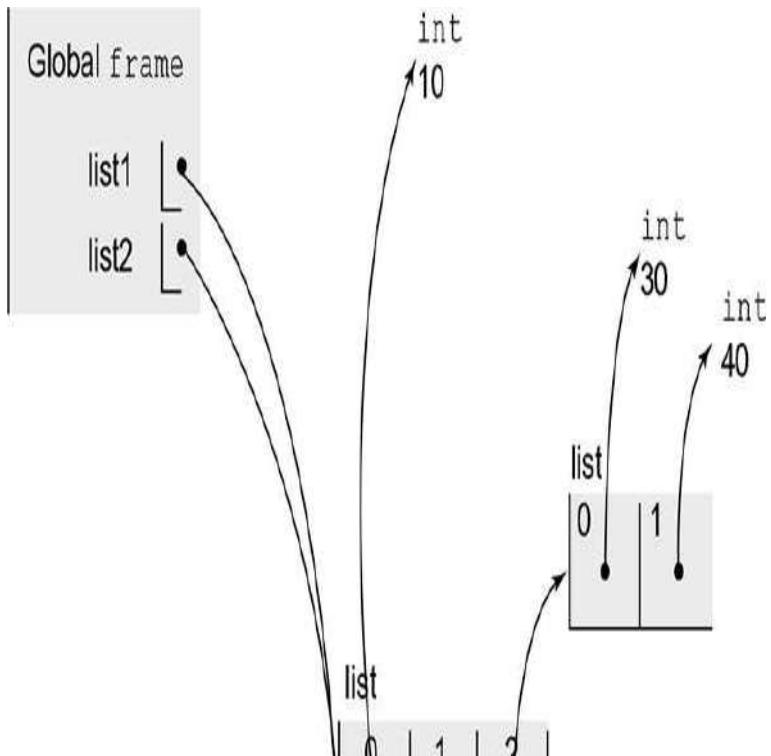


Fig. 7.6 list1 and list2 refer to the same list



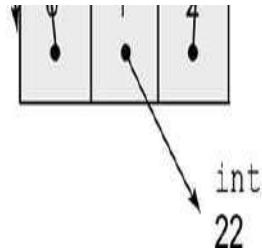


Fig. 7.7 `list1` and `list2` refer to the same list

```
>>> list1[1] = 22
```

```
>>> list1
```

```
[10, 22, [30, 40]]
```

```
>>> list2
```

```
[10, 22, [30, 40]]
```

However, sometimes we need to create a copy of the list as a distinct object. To create another instance of the list object having different storage, we need to import the `copy` module and invoke the function `copy.copy()`:

`copy()` : to create a copy of the list elements excluding the nested objects

```
>>> import copy
```

```
>>> list1 = [10, 20, [30, 40]]
```

```
>>> list3 = copy.copy(list1)
```

Note that the `copy` function creates a new copy `list3` of the `list1`. Consequently, on modifying `list1[1]`, `list3[1]` remains unaltered:

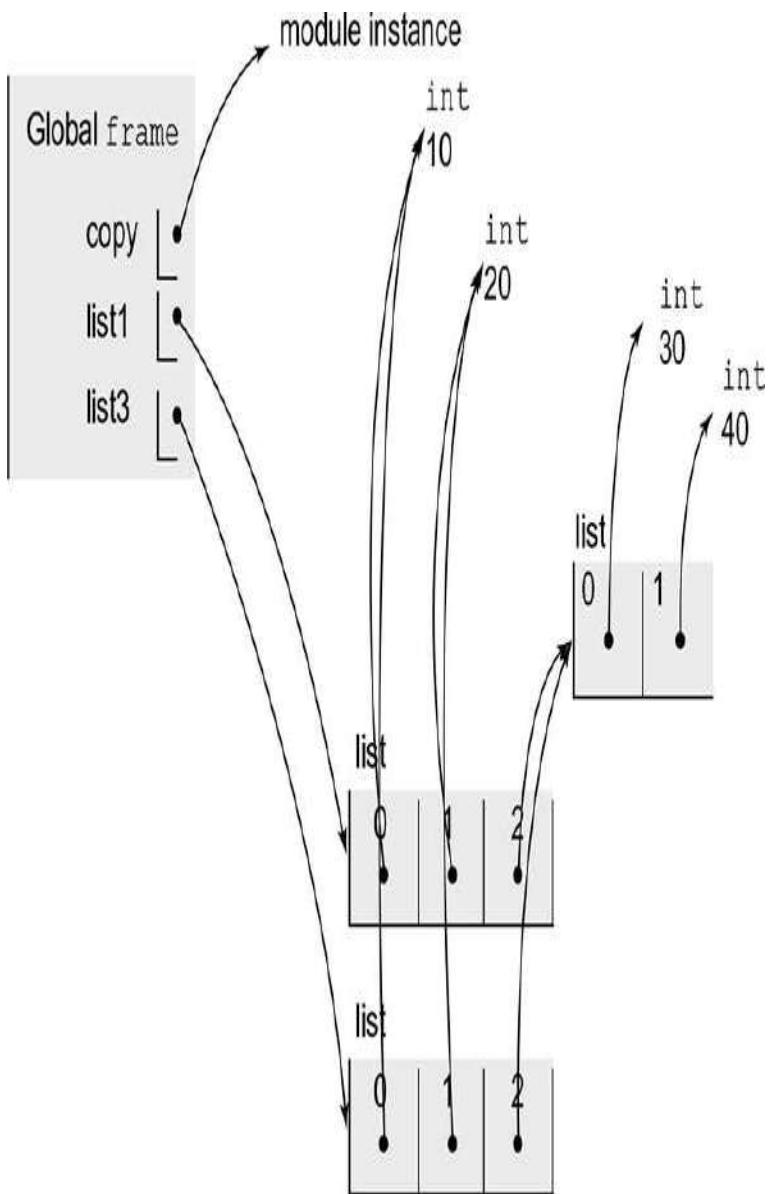


Fig. 7.8 `list1` and `list3` refer to the different lists

```
>>> list1[1] = 25
```

```
>>> list1
```

```
[10, 25, [30, 40]]
```

```
>>> list3
```

```
[10, 20, [30, 40]]
```

However, the `copy` function creates a shallow copy i.e. it does not create copies of the nested objects. Thus, the two lists share the same nested objects. Thus, `list1[2]` and `list3[2]` refer to the same nested list `[30, 40]`. Next, let us modify `list1[2][0]` in sub-list `list1[2]` of list `list1` to value `35`.

```
>>> list1[2][0] = 35
```

```
>>> list1
```

```
[10, 25, [35, 40]]
```

```
>>> list3
```

```
[10, 20, [35, 40]]
```

As discussed above, since `list1[2]` and `list3[2]` refer to the same sub-list at index 2 (Fig. 7.9), when we modify the list `list1[2]`, the modification is also visible in the list `list3[2]`.

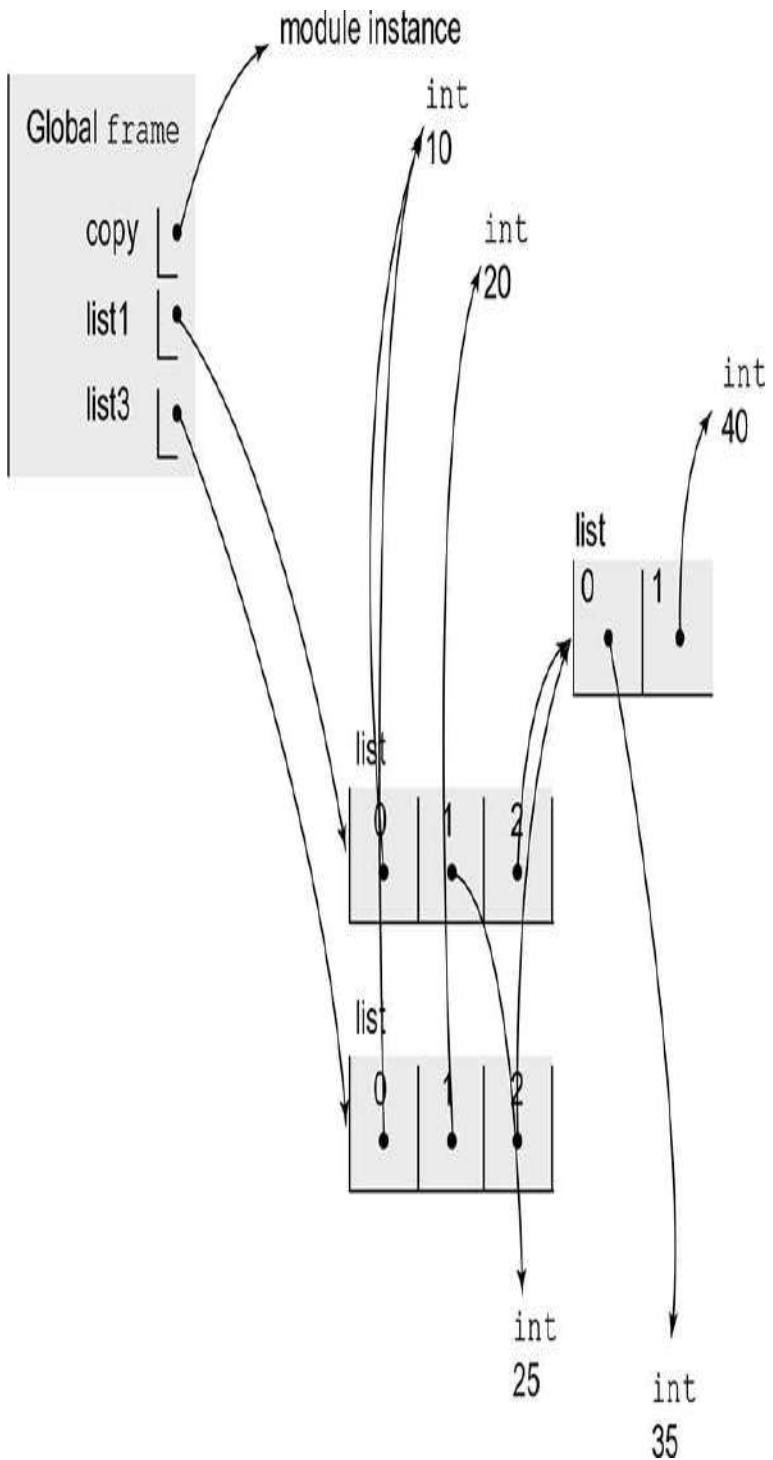


Fig. 7.9 `list1` and `list3` refer to the different lists

If we wish to create a copy of a list object so that the nested objects (at all levels) get copied to new objects, we use the function `deepcopy` of the `copy` module:

`deepcopy()` : to create a copy of a list including copies of the nested objects at all level

```
>>> import copy  
  
>>> list1 = [10, 20, [30, 40]]  
  
>>> list4 = copy.deepcopy(list1)
```

Note that `list1` and `list4` are distinct objects ([Fig. 7.10](#)), having the nested lists `list1[2]` and `list4[2]` as distinct objects. Thus, modifying the value of `list1[2][0]` to 35 does not affect the object `list4[2][0]` of `list4` which remains unaltered as shown in [Fig. 7.11](#).

```
>>> list1[2][0] = 35  
  
>>> list1  
  
[10, 20, [35, 40]]  
  
>>> list4  
  
[10, 20, [30, 40]]
```

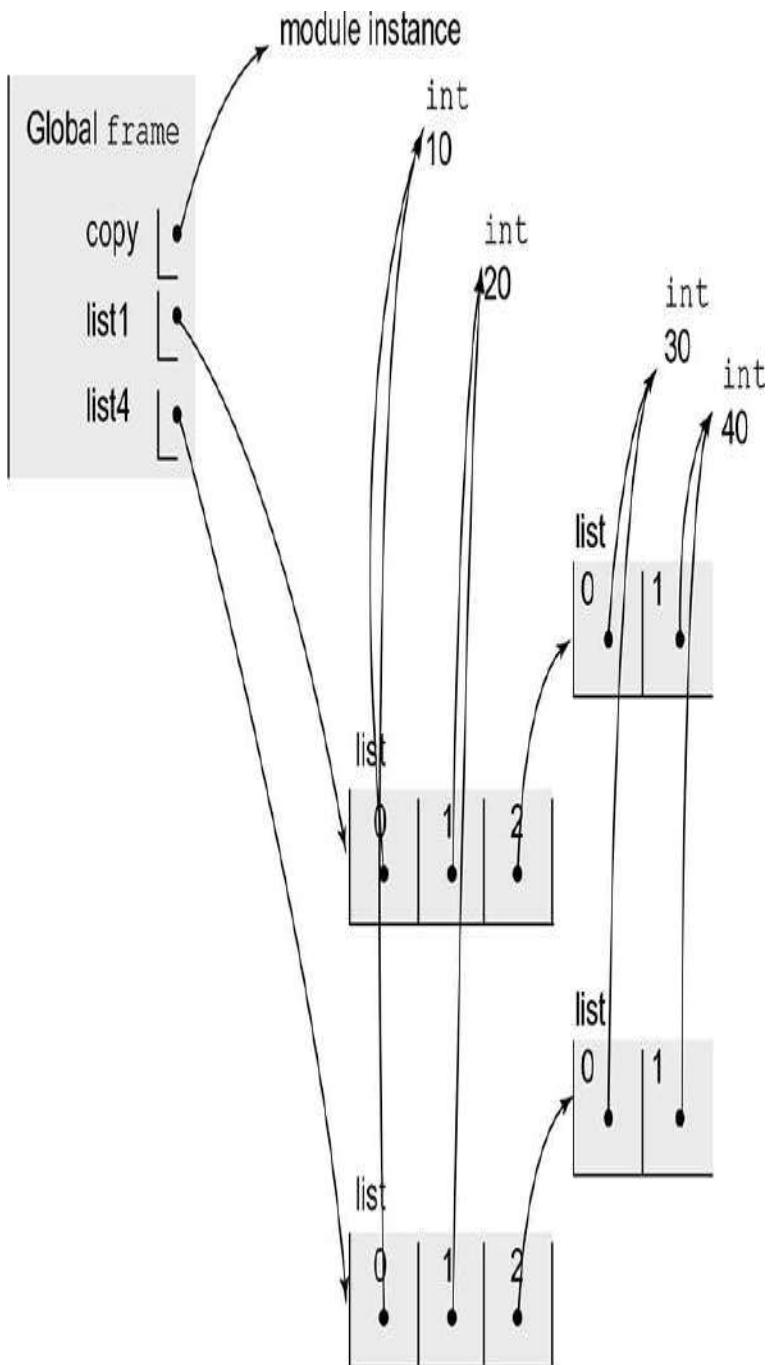
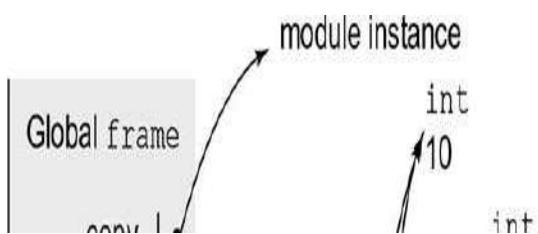



Fig. 7.10 `list1` and `list4` refer to the different lists at all levels of nesting



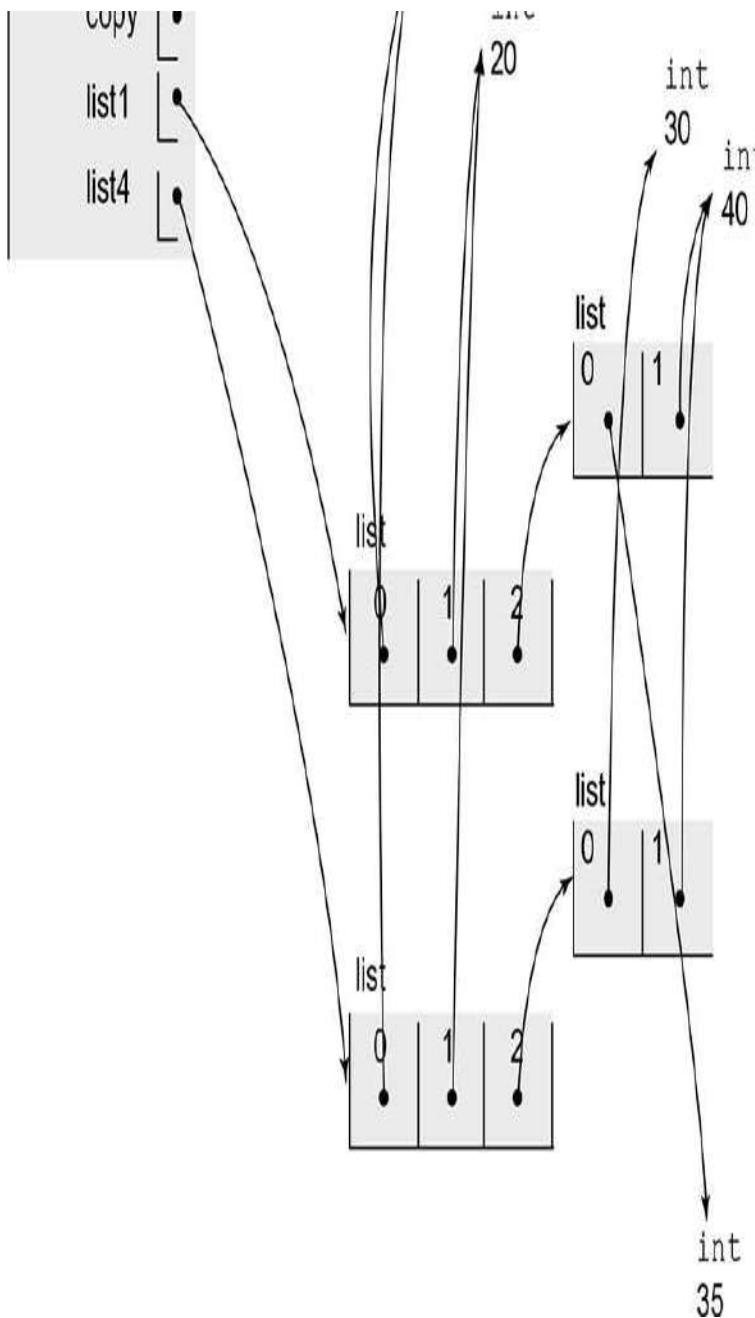


Fig. 7.11 `list1` and `list4` refer to the different lists at all levels of nesting

7.1.11 `map`, `reduce`, and `filter` Operations on a Sequence

Python provides several built-in functions based on expressions, which work faster than loop-based user defined code.

The function `map` is used for transforming every value in a given sequence by applying a function to it. It takes two input arguments: the iterable object (i.e. object which can be iterated upon) to be processed and the function to be applied, and returns the map object obtained by applying the function to the list as follows:

mapping each value in the sequence by applying the function

```
result = map(function, iterable object)
```

The function to be applied may have been defined already, or it may be defined using a lambda expression which returns a `function` object.. A lambda expression consists of the `lambda` keyword followed by a comma-separated list of arguments and the expression to be evaluated using the list of arguments in the following format:

`lambda` expression: used to define the expression to be evaluated using the arguments

`lambda arguments: expression`

For example, let us define a lambda function that takes a number `x` as an input parameter and returns the cube of it. The following statement defines a `lambda` function and assigns it the name `cube`. The function `cube` can now be used as usual:

`lambda` expression to compute the cube of a number

```
>>> cube = lambda x: x ** 3
```

```
>>> cube(3)
```

27

Similarly, the function `sum2Cubes` may be used to compute the sum of cubes of two numbers:

lambda expression to compute the sum of cubes of two numbers

```
>>> sum2Cubes = lambda x, y: x**3 + y**3  
>>> sum2Cubes(2, 3)
```

35

Next, suppose we wish to compute the cubes of all values in the list `lst`. The following statements use the `map` function to map every value in the list `lst` to its cube. Since the function `map` returns a `map` object, we have used the function `list` to convert it to list.

mapping every value in the sequence to its cube

```
>>> lst = [1, 2, 3, 4, 5]  
  
>>> list(map(lambda x: x ** 3, lst))  
  
[1, 8, 27, 64, 125]  
  
>>> list(map(cube, lst))  
  
[1, 8, 27, 64, 125]
```

We may also use any system defined or user-defined function as an argument of the `map` function, for example:

```
>>> list(map(abs, [-1, 2, -3, 4, 5]))  
  
[1, 2, 3, 4, 5]
```

Suppose we wish to compute the sum of cubes of all elements in a list. Note that adding elements of the list is a repetitive procedure of adding two elements at a time. The function `reduce`, available in `functools` module, may be used for this purpose. It takes two arguments: a function, and a iterable object, and applies the function on the iterable object to produce a single value:

```
computing sum of cubes of all elements in the list

>>> lstCubes = list(map(lambda x: x ** 3,
lst))

>>> import functools

>>> sumCubes = functools.reduce(lambda x,
y: x + y, lstCubes)

>>> sumCubes
```

225

Thus, in the above example, we are able to compute the sum of all the elements of the list `lstCubes`. In a parallel environment, the sum may be computed by adding two values at a time, and then summing over the sums obtained in the first step, and so on. Thus, the sum of the elements of the list [1, 2, 3, 4, 5, 6, 7, 8] may be computed as $((1+2) + (3+4)) + ((5+6) + (7+8))$.

Next, we compute the sum of cubes of only those elements in the list `lstCubes` that are even. Thus, we need to filter the even elements from the list `lstCubes` and before applying the function `reduce`. Python provides the function `filter` that takes a function and a iterable object as the input parameters and returns only those elements from the iterable object for which function returns `True`. In the following statement, the `filter` function returns an iterable object of even elements of the list `lstCubes`, which is assigned to variable `evenCubes`. Since the function `filter` returns a `filter` object, we have used the function `list` to convert it to list.

filtering elements from a list satisfying the given condition

```
>>> evenCubes = list(filter(lambda x: x%2
== 0, lstCubes))
```

```
>>> evenCubes  
  
[8, 64]  
  
>>> sumEvenCubes = functools.reduce(lambda  
x, y: x + y, evenCubes)  
  
>>> sumEvenCubes  
  
72
```

Alternatively, the sum of odd cubes may be computed as follows:

```
>>> sumEvenCubes = functools.reduce(lambda  
x, y: x + y, filter(lambda x: x%2 == 0,  
lstCubes))  
  
>>> sumEvenCubes  
  
72
```

The functions `map`, `reduce`, and `filter` are often used to exploit the parallelism of the system to distribute computation to several processors. However, details of the parallel programming are beyond the scope of the book.

7.2 SETS

A set in Mathematics refers to an unordered collection of objects without any duplicates. An object of type set may be created by enclosing the elements of the set within curly brackets. The set objects may also be created by applying the `set` function to lists, strings, and tuples. Next, we give some examples of sets:

`set`: a comma separated unordered sequence of values enclosed within curly braces

`set()`: to convert a sequence to a set

```
>>> {1, 2, 3}  
  
{1, 2, 3}  
  
>>> vowels = set('aeiou')  
  
>>> vowels  
  
{'e', 'a', 'o', 'u', 'i'}  
  
>>> vehicles = set(['Bike', 'Car',  
'Bicycle', 'Scooter'])  
  

```

The elements of a set are required to be immutable objects. Therefore, whereas objects of types such as `int`, `float`, `tuple`, and `str` can be members of a set, a list cannot be a member of a set. Unlike lists, we cannot access elements of a set through indexing or slicing. Also, we cannot apply `+` and `*` operators on sets. However, using the membership operator `in`, we can iterate over elements of a set:

elements of a set must be immutable

set type does not support indexing, slicing, + operator, and * operator

iterating over elements of the set

```
>>> for v in vowels:  
    print(v, end = ' ')  
  
e a o u i
```

The functions `min`, `max`, `sum`, and `len` work for sets in the same manner as defined for lists.

applying `min()`, `max()`, `sum()`, and `len()` functions to sets

```
>>> min(digits)  
0  
  
>>> max(digits)  
9  
  
>>> sum(digits)  
45  
  
>>> len(vehicles)
```

4

The membership operator `in` checks whether an object is in the set.

```
>>> 'Bike' in vehicles  
True
```

7.2.1 Set Functions `add`, `update`, `remove`, `pop`, and `clear`

Suppose we wish to add an element 'Bus' to the set `vehicles`. It can be done using the function `add`. To include new elements in a set, we use `update` function. The input argument to the `update` function may be an object such as list, set, range, or tuple, for example:

```
adding elements to a set

>>> vehicles.add('Bus')

>>> vehicles

{'Car', 'Bike', 'Bicycle', 'Bus',
'Scooter'}

>>> vehicles.update(['Truck', 'Rickshaw',
'Bike'])

>>> vehicles

{'Car', 'Bike', 'Bicycle', 'Rickshaw',
'Bus', 'Truck', 'Scooter'}
```

It is important to point out that an attempt to add a duplicate value ('Bike' in the above case) is just ignored by Python. Next, we define the set `heavyVehicles` as follows:

```
duplicate elimination

>>> heavyVehicles = {'Truck', 'Bus',
'Crane'}
```

An element may be removed from a set using the function `remove`, for example:

```
removing an element from the set

>>> heavyVehicles.remove('Crane')

>>> heavyVehicles

{'Bus', 'Truck'}
```



```
>>> heavyVehicles.remove('Car')
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#432>", line 1
```

```
    heavyVehicles.remove('Car')
```

```
KeyError: 'Car'
```

Note that an attempt to remove an element from a set, which is not a member of the set, yields an error.

Although the function `pop` is defined for objects of type `set`, the element removed by the function `pop` is unpredictable.

```
element removed using pop function is unpredictable
```

```
>>> heavyVehicles = {'Truck', 'Bus',  
'Crane'}
```

```
>>> heavyVehicles.pop()
```

```
'Bus'
```

To remove all the elements of a set, without deleting the set object, we use the `clear` function:

```
removing all elements of a set
```

```
>>> heavyVehicles.clear()
```

```
>>> heavyVehicles
```

```
set()
```

7.2.2 Set Functions `union`, `intersection`, `difference`, and `symmetric_difference`

Python provides functions `union`, `intersection`, and `symmetric_difference` to carry out these mathematical operations. We define two sets `fruits` and `vegetables`:

```
>>> fruits = {'Apple', 'Orange', 'Tomato',
'Cucumber', 'Watermelon'}
```

```
>>> vegetables = {'Cucumber', 'Potato',
'Tomato', 'Watermelon', 'Cauliflower'}
```

Now, to determine all eatables whether fruits or vegetables, we take union of the above two sets.

union of two sets

```
>>> eatables = fruits.union(vegetables)
```

```
>>> eatables
```

```
{'Orange', 'Tomato', 'Cucumber',
'Watermelon', 'Cauliflower', 'Apple',
'Potato'}
```

Alternatively, we may use the union operator | for achieving the same result:

```
>>> eatables = fruits | vegetables
```

```
>>> eatables
```

```
{'Orange', 'Tomato', 'Cucumber',
'Watermelon', 'Cauliflower', 'Apple',
'Potato'}
```

Next, to determine those eatables which are both fruits and vegetables, we take intersection of sets fruits and vegetables using either the function intersection or the intersection operator & as follows:

intersection of two sets

```
>>> fruitsAndVegetables =
fruits.intersection(vegetables)
```

```
>>> fruitsAndVegetables  
{'Tomato', 'Cucumber', 'Watermelon'}  
  
>>> fruitsAndVegetables = fruits &  
vegetables  
  
>>> fruitsAndVegetables  
  
{'Tomato', 'Cucumber', 'Watermelon'}
```

Next, we find the fruits which are not vegetables by taking difference of the two sets, either using the function `difference` or using the difference operator:

```
set difference  
  
>>> onlyFruits =  
fruits.difference(vegetables)  
  
>>> onlyFruits  
  
{'Orange', 'Apple'}  
  
>>> onlyFruits = fruits - vegetables  
  
>>> onlyFruits  
  
{'Orange', 'Apple'}
```

Similarly, we can determine vegetables which are not fruits as follows:

```
>>> onlyVegetables =  
vegetables.difference(fruits)  
  
>>> onlyVegetables  
  
{'Cauliflower', 'Potato'}  
  
>>> onlyVegetables = vegetables - fruits
```

```
>>> onlyVegetables  
  
{'Cauliflower', 'Potato'}
```

Next, to determine eatables which are either only fruits or only vegetables, we may take union of the sets `onlyFruits` and `onlyVegetables`:

```
>>> fruitsXORVegs = onlyFruits |  
onlyVegetables  
  
>>> fruitsXORVegs  
  
{'Orange', 'Apple', 'Cauliflower',  
'Potato'}
```

Alternatively, we may use the function `symmetric_difference`, which determines the elements that are in one of the either sets but not in both:

```
symmetric difference  
  
>>> fruitsXORVegs =  
fruits.symmetric_difference(vegetables)  
  
>>> fruitsXORVegs  
  
{'Orange', 'Apple', 'Cauliflower',  
'Potato'}
```

The functions `set.intersection`, `set.union`, and `set.difference` can be applied to two or more sets to carry out intersection, union, and difference operations on sets:

```
>>> digits1 = {0, 1, 2, 3}  
  
>>> digits2 = {2, 4, 5, 6}  
  
>>> digits3 = {0, 7, 8, 9, 2}  
  
>>> set.union(digits1, digits2, digits3)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

>>> set.intersection(digits1, digits2,
digits3)

{2}

>>> set.difference(digits1, digits2,
digits3)

{1, 3}
```

The function `set.difference` works by taking the union of all sets except the first one and subtracting the union from the first set.

7.2.3 Function `copy`

We make use of the function `copy` to create a separate copy of a set so that changes in one copy are not reflected in the other:

```
copying the elements of a set

>>> digits = {0, 1, 2, 3, 7, 8, 9}

>>> digits2 = digits.copy()

>>> digits.update({4, 5, 6})

>>> digits

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

>>> digits2

{0, 1, 2, 3, 7, 8, 9}
```

7.2.4 Subset and Superset Test

To check whether a given set is the subset or superset of another set, we use the operators `<=` and `>=`, respectively. For example, let the sets `multiples2` and

`multiples4` comprise multiples of 2 and 4, respectively. We can check whether `multiples2` is a subset or superset of `multiples4` using these operators:

```
subset and superset operations

>>> multiples2 = {2, 4, 6, 8, 10, 12, 14}

>>> multiples4 = {4, 8, 12}

>>> isSubset = multiples4 <= multiples2

>>> isSubset

True

>>> isSuperset = multiples2 >= multiples4

>>> isSuperset

True
```

The functions `issubset` and `issuperset` can also be used for the same purpose.

7.2.5 List of Functions

In Table 7.3, we list some important built-in functions that can be applied to sets.

Table 7.3 Set functions

Function	Description
S.add(e)	Adds the element e to the set S if not already present.
S1.update(L1)	Adds the items in object L1 to the set S1 if not already present.
S.remove(e)	Removes the element e from the set S.
S.pop()	Removes an element from the set S.
S.clear()	Removes all elements from the set S.
S.copy()	Creates a copy of the set S.
S1.union(S2)	Returns union of sets S1 and S2.
S1.intersection(S2)	Returns a set containing elements common in the sets S1 and S2.
S1.difference(S2)	Returns a set containing elements in the set S1 but not in the set S2.
S1.symmetric_difference(S2)	Returns a set containing elements that are in one of the two sets S1 and S2, but not in both.

7.2.6 Finding Common Factors

Suppose, given two numbers `n1` and `n2`, we wish to find their common factors. To check whether a number `i` is a factor of another number, say `num`, we only need to find the remainder `num % i`. If the remainder is zero, then `i` is a factor of `num`, otherwise, `i` is not a factor of `num`. We represent the collection of common factors as a set. We also note that a common factor cannot exceed smaller of the two numbers, say `n1` and `n2`. To build a set of common factors, we make use of comprehensions. For each number `i` in `range(1, min(n1, n2) + 1)`, we include it in the set if it is a factor of each of `n1` and `n2`. Thus, our code comprises just one line:

```
set comprehension: applied to find common factors

commonFactors = {i for i in range(1,
min(n1+1, n2+1)) if n1%i == 0 and n2%i ==
0}
```

7.2.7 Union and Intersection Operation on Lists

Given a list `lst1` of people who like basketball and another list `lst2` of individuals who like badminton; suppose, we wish to construct the following lists:

A list of individuals who like either basketball or badminton, i.e. union of lists `lst1 | lst2`.

A list of people who like both basketball and badminton, i.e., the intersection of lists `lst1 & lst2`.

A list of individuals who like basketball but not badminton, i.e. the difference `lst1 - lst2`.

Although Python does not support such operators on lists, we can convert lists into sets, apply the necessary operations on sets, and finally convert the sets back into lists. It is important to emphasize that a list should be converted to sets only when the order of elements in the list is not important. Also, transforming a list into the set will remove repeating elements. In Fig. 7.12, we define the functions `union`, `intersection`, and `difference` for lists.

```
01 def union(L1, L2):
02     """
03     Objective: To find union of two lists
04     Input Parameters: L1 and L2 - list
05     Return Value: list
06     """
07     return list(set(L1) | set(L2))
08
09 def intersection(L1, L2):
10     """
11     Objective: To find intersection of two lists
12     Input Parameters: L1 and L2 - list
13     Return Value: list
14     """
15     return list(set(L1) & set(L2))
16
17 def difference(L1, L2):
18     """
19     Objective: To find difference of two lists
20     Input Parameters: L1 and L2 - list
21     Return Value: list
22     """
23     return list(set(L1) - set(L2))
```

Fig. 7.12 Functions to find union, intersection, and difference of lists (*lists.py*)

7.3 TUPLES

A tuple is a non-scalar type defined in Python. Just like a list, a tuple is an ordered sequence of objects. However, unlike lists, tuples are immutable, i.e. elements of a tuple cannot be overwritten. A tuple may be specified by enclosing in the parentheses, the elements of the tuple (possibly of heterogeneous types), separated by commas, for example, the tuple `t1` comprises five objects:

tuple: a comma separated ordered sequence of values enclosed in parenthesis

```
>>> t1 = (4, 6, [2, 8], 'abc', {3, 4})  
  
>>> type(t1)  
  
<class 'tuple'>
```

If a tuple comprises a single element, the element should be followed by a comma to distinguish a tuple from a parenthesized expression, for example:

singleton tuple: comma required after the element

```
>>> (2)  
  
2  
  
>>> (2,)  
  
(2,)
```

A tuple having a single element is also known as singleton tuple. Another notation for tuples is just to list the elements of a tuple, separated by commas:

```
>>> 2, 4, 6  
  
(2, 4, 6)
```

Tuples being immutable, an attempt to modify an element of a tuple yields an error:

```
tuples are immutable

>>> t1[1] = 3

Traceback (most recent call last):

  File "<pyshell#61>", line 1, in <module>

    t1[1] = 3

TypeError: 'tuple' object does not support
item assignment
```

However, an element of a tuple may be an object that is mutable, for example:

```
elements of a tuple may be mutable

>>> t1 = (1, 2, [3, 4])

>>> t1[2][1] = 5

>>> t1

(1, 2, [3, 5])
```

7.3.1 Summary of Tuple Operations

In Table 7.4, we summarize the operations on tuples and use the following tuples `t1` and `t2` for the purpose of illustration:

```
>>> t1 = ('Monday', 'Tuesday')

>>> t2 = (10, 20, 30)
```

Table 7.4 Summary of operations that can be applied on tuples

Operation	Example
Multiplication operator *	>>> t1 * 2 ('Monday', 'Tuesday', 'Monday', 'Tuesday')
Concatenation operator +	>>> t3 = t1 + ('Wednesday',) >>> t3 ('Monday', 'Tuesday', 'Wednesday')

Operation	Example
Length operator <code>len</code>	<code>>>> len(t1)</code> 2
Indexing	<code>>>> t2[-2]</code> 20
Slicing	<code>>>> t1[1:2]</code>
Syntax: <code>start:end:inc</code>	<code>('Tuesday',)</code>
Function <code>min</code>	<code>>>> min(t2)</code> 10
Function <code>max</code>	<code>>>> max(t2)</code> 30
Function <code>sum</code> (not defined on strings)	<code>>>> sum(t2)</code> 60
Membership operator <code>in</code>	<code>>>> 'Friday' in t1</code> False

7.3.2 Functions `tuple` and `zip`

The function `tuple` can be used to convert a sequence to a tuple, for example:

converting a sequence to a tuple

```
>>> vowels = 'aeiou'

>>> tuple(vowels)

('a', 'e', 'i', 'o', 'u')
```

```
>>> tuple([4,10,20])
```

```
(4, 10, 20)
```

```
>>> tuple(range(5))
```

```
(0, 1, 2, 3, 4)
```

The function `zip` is used to produce a zip object (iterable object), whose *i*th element is a tuple containing *i*th element from each iterable object passed as argument to the `zip` function. We have applied `list` function to convert the `zip` object to a list of tuples. For example,

producing a list of tuples containing corresponding element from each iterable object

```
>>> colors = ('red', 'yellow', 'orange')
```

```
>>> fruits = ['cherry', 'banana',
'orange']
```

```
>>> quantity = ('1 kg', 12, '2 kg')
```

```
>>> fruitColor = list(zip(colors, fruits))
```

```
>>> fruitColor
```

```
[('red', 'cherry'), ('yellow', 'banana'),
('orange', 'orange')]
```

```
>>> fruitColorQuantity =
list(zip(fruitColor, quantity))
```

```
>>> fruitColorQuantity
```

```
[([('red', 'cherry'), '1 kg'), (('yellow',
'banana'), 12), (('orange', 'orange'), '2
kg')]
```

```
>>> list(zip(colors, fruits, quantity))
```

```
[('red', 'cherry', '1 kg'), ('yellow',
'banana', 12), ('orange', 'orange', '2
kg')]
```

7.3.3 Functions count and index

The function `count` is used to find the number of occurrences of a value in a tuple, for example:

```
counting number of occurrences of a value in a tuple

>>> age = (20, 18, 17, 19, 18, 18)

>>> age.count(18)

3
```

The function `index` is used to find the index of the first occurrence of a particular element in a tuple, for example:

```
finding index of the first occurrence of an element in the
tuple

>>> age.index(18)

1
```

These functions are summarized in [Table 7.5](#):

Table 7.5 Tuple functions

Function	Explanation
<code>T.count(e)</code>	Returns count of occurrences of element e in tuple T.
<code>T.index(e)</code>	Returns index of the first occurrence of element e in tuple T.

7.4 DICTIONARY

Unlike lists, tuples, and strings, a dictionary is an unordered sequence of *key-value* pairs. Indices in a dictionary can be of any immutable type and are called keys. Beginning with an empty dictionary, we create a dictionary of *month_number-month_name* pairs as follows:

```
dictionary: a comma separated unordered sequence of key-
value pairs enclosed in curly braces

>>> month = {}

>>> month[1] = 'Jan'

>>> month[2] = 'Feb'

>>> month[3] = 'Mar'

>>> month[4] = 'Apr'

>>> month

{1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr'}
```

```
>>> type(month)

<class 'dict'>
```

Key and value in a *key-value* pair in a dictionary are separated by a colon. Further, the *key:value* pairs in a dictionary are separated by commas and are enclosed between curly parentheses. Next, we define another dictionary comprising vegetable-price pairs:

determining value corresponding to a key

```
>>> price = {'tomato':40, 'cucumber':30,
'potato':20, 'cauliflower':70,
'cabbage':50, 'lettuce':40, 'raddish':30,
'carrot':20, 'peas':80}
```

```
>>> price['potato']

20

>>> price['carrot']

20

>>> price.keys()

dict_keys(['tomato', 'cucumber', 'potato',
'cauliflower', 'cabbage', 'lettuce',
'raddish', 'carrot', 'peas'])

>>> price.values()

dict_values([40, 30, 20, 70, 50, 40, 30,
20, 80])

>>> price.items()

dict_items([('tomato', 40), ('cucumber',
30), ('potato', 20), ('cauliflower', 70),
('cabbage', 50), ('lettuce', 40),
('raddish', 30), ('carrot', 20), ('peas',
80)])
```

Note that the search in a dictionary is based on the key. Therefore, in a dictionary, the keys are required to be unique. However, the same value may be associated with multiple keys. In particular, in the dictionary `price` of *vegetable-price* pairs, we have already noted that whereas the names of vegetables are unique throughout the dictionary, the prices may be duplicated (`price['potato'] == price['carrot']` and `price['tomato'] == price['lettuce']`). The functions `keys`, `values`, and `items` return objects comprising of keys, values, and key-value tuples in the dictionary respectively. If required, a function such as `list` or `tuple` may be applied to the output of the functions `keys`, `values`, and `items` to obtain a list or tuple of keys, values, or key-value tuples. As keys in a dictionary are immutable, lists cannot be used as keys.

However, values associated with keys can be mutable objects and thus, may be changed at will, for example:

keys: must be unique values: may be duplicated

```
keys() : return an object of keys  
values() : return an object of values  
items() : return an object of key-value tuples
```

keys are immutable

```
>>> price['tomato'] = 25
```

Keys in a dictionary may be of heterogeneous types, for example:

```
>>> counting = {1:'one', 'one':1, 2:'two',  
'two':2}
```

7.4.1 Dictionary Operations

Table 7.6 gives some operations that can be applied to a dictionary and illustrate these operations using a dictionary of digit-name pairs:

```
digits = {0:'Zero', 1:'One', 2:'Two',  
3:'Three', 4:'Four', 5:'Five', 6:'Six', 7:  
'Seven', 8:'Eight', 9:'Nine'}
```

Table 7.6 Summary of operations that can be applied on dictionaries

Operation	Examples
Length operator <code>len</code> (number of key-value pairs in a dictionary)	<pre>>>> len(digits)</pre> <p>10</p>
Indexing	<pre>>>> digits[1]</pre> <p>'One'</p>
Function <code>min</code>	<pre>>>> min(digits)</pre> <p>0</p>
Function <code>max</code>	<pre>>>> max(digits)</pre> <p>9</p>
Function <code>sum</code> (assuming keys are compatible for addition)	<pre>>>> sum(digits)</pre> <p>45</p>
Membership operator <code>in</code>	<pre>>>> 5 in digits</pre> <p>True</p> <pre>>>> 'Five' in digits</pre> <p>False</p>

Next, consider a dictionary of winter months:

```
>>> winter = {11:'November', 12:  
'December', 1:'January', 2:'February'}
```

Membership operation `in`, and functions `min`, `max` and `sum` apply only to the keys in a dictionary. Thus, applying these operations on the dictionary `winter` is equivalent to applying them on `winter.keys()` as shown below:

```
min, max, sum, and in apply to keys in a dictionary

>>> 2 in winter, min(winter), max(winter),
sum(winter)

(True, 1, 12, 26)

>>> 2 in winter.keys(),
min(winter.keys()), max(winter.keys()),
sum(winter.keys())

(True, 1, 12, 26)
```

7.4.2 Deletion

We may remove a *key-value* pair from a dictionary using `del` operator, for example:

```
removing a key-value pair

>>> del winter[11]

>>> winter

{12: 'December', 1: 'January', 2:
'February'}
```

To delete a dictionary, we use `del` operator:

```
deleting an entire dictionary

>>> del winter
```

To remove all the key–value pairs in a dictionary, we use the `clear` function. Note that while assigning an empty dictionary `{}` creates a new object, the function `clear` removes all *key-value* pairs from an existing dictionary:

```
clear(): to remove all key-value pairs
```

```
>>> winter = {11:'November ', 12:  
'December', 1:'January', 2:'February'}  
  
>>> months = winter  
  
>>> months.clear()  
  
>>> months, winter  
  
({}, {})
```

Note that as the names `winter` and `months` refer to the same dictionary object, on applying `clear` function to `months`, the dictionary `winter` also becomes empty.

7.4.3 Function `get`

The function `get` is used to extract the value corresponding to a given key, for example:

```
retrieving value corresponding to a key  
  
>>> passwords = {'Ram':'ak@607',  
'Shyam':'rou.589', 'Gita':'yam@694'}  
  
>>> passwords.get('Ram', -1)  
  
'ak@607'
```

The first parameter is used to specify the key and the second parameter is used to specify the value to be returned in case the key is not found in the dictionary. In case, the second parameter is not specified, the system returns `None`, for example:

`get()` returns second argument if the key is not present in the dictionary

```
>>> passwords.get('Raman', -1)  
  
-1
```

```
>>> print(passwords.get('Raman'))
```

```
None
```

7.4.4 Function update

The function `update` is used to insert in a dictionary, all the key–value pairs of another dictionary, for example:

inserting key-value pairs

```
>>> morePasswords = {'Raman': 'vi97@4',  
'Kishore': '23@0jsk'}
```

```
>>> passwords.update(morePasswords)
```

```
>>> passwords
```

```
{'Ram': 'ak@607', 'Shyam': 'rou.589',  
'Gita': 'yam@694', 'Raman': 'vi97@4',  
'Kishore': '23@0jsk'}
```

7.4.5 Function copy

To create a shallow copy of a dictionary object, we make use of the `copy` function, for example:

creating shallow copy of a dictionary

```
>>> newPasswords = morePasswords.copy()
```

```
>>> id(newPasswords), id(morePasswords)
```

```
(54782832, 54781104)
```

7.4.6 List of Functions

In Table 7.7, we list some of the built-in functions that can be applied to objects of `dict` type. Whereas many of these functions such as `update` and `clear` modify a dictionary, functions such as `items`, `keys`, `values`, and `get` do not alter a dictionary.

Table 7.7 Dictionary functions

Function	Explanation
D.items()	Return an object comprising of tuples of key-value pairs present in dictionary D.
D.keys()	Return an object comprising of all keys of dictionary D.
D.values()	Return an object comprising of all values of dictionary D.
D.clear()	Removes all key-value pairs from dictionary D.
D.get(key, default)	For the specified key, the function returns the associated value. Returns the default value in the case key is not present in the dictionary D.
D.copy()	Creates a shallow copy of dictionary D.
D1.update(D2)	Adds the key-value pairs of dictionary D2 to dictionary D1.

7.4.7 Inverted Dictionary

Suppose we are maintaining a dictionary of words and their meanings of the form *word:meaning*. For simplicity, we assume that each word has a single meaning. Even so, there might be a few words that may have shared meaning. Given this dictionary, we wish to find synonyms of a word, i.e. given a word, we wish to find the list of words that have the same meaning. For this purpose, we would like to build an inverted dictionary `invDict` of *meaning:list-of-words*. In the function `buildInvDict`, we begin with an empty dictionary `invDict` that uses *meaning* as the key. For every key-value pair of *word:meaning* in the dictionary `dict1`, if it is the first occurrence *meaning*, then *meaning:[word]* is entered in the inverted dictionary. If the key *meaning* already exists in the inverted dictionary, then *word* is appended to the list

`invDict[meaning]`. Finally, we use comprehensions to drop the words having only one meaning:

```
dictionary comprehension for finding synonyms in a
dictionary
```

```
{meaning: invDict[x] for x in invDict if
len(invDict[x])>1}
```

The complete program appears in [Fig. 7.13](#).

We conclude this section with a sample run of the program `invDict` ([Fig. 7.13](#)).

```
>>>
```

```
Enter word meaning dictionary:
```

```
{'dubious':'doubtful',
'hilarious':'amusing',
'suspicious':'doubtful',
'comical':'amusing', 'hello':'hi'}
```

```
Inverted Dictionary:
```

```
{'doubtful': ['dubious', 'suspicious'],
'amusing': ['hilarious', 'comical']}
```

```
01 def buildInvDict(dict1):
02     """
03     Objective: To construct inverted dictionary
04     Input Parameter: dict1 : dictionary
05     Return Value: invDict : dictionary
06     """
07     invDict = {}
08     for key,value in dict1.items():
09         if value in invDict:
10             invDict[value].append(key)
11         else:
12             invDict[value] = [key]
13     invDict = {x:invDict[x] for x in invDict if len(invDict[x])>1}
14     return invDict
15
16 def main():
17     """
18     Objective: To find inverted dictionary
19     Input Parameter: None
20     Return Value: None
21     """
22     wordMeaning = eval(input('Enter word meaning dictionary: '))
```

```
23     meaningWord = buildInvDict(wordMeaning)
24     print('Inverted Dictionary:\n', meaningWord)
```

```
25
26 # Statements to initiate the call to main function.
27 if __name__ == '__main__':
28     main()
```

Fig. 7.13 Program to construct inverted dictionary
(invDict.py)

SUMMARY

1. The list is a non-scalar type. It is an ordered sequence of values specified within square brackets. Values in a list may be of any type such as str, int, float, or list.
2. Indexing is used for accessing elements of a list.
3. Unlike strings, lists are mutable, i.e. one may modify individual elements of a list.
4. A list of lists is called a two-dimensional list.
5. The following table lists operations that can be applied on the list:

```
>>> list1 = ['Red', 'Green']
>>> list2 = [10, 20, 30]
```

Operation	Example
Multiplication operator *	>>> list2 * 2 [10, 20, 30, 10, 20, 30]
Concatenation operator +	>>> list1 = list1 + ['Blue'] >>> list1 ['Red', 'Green', 'Blue']
Length operator len	>>> len(list1) 3
Indexing	>>> list2[-1] 30
Slicing Syntax: <i>start:end:inc</i>	>>> list2[0:2] [10, 20] >>> list2[0:3:2] [10, 30]
Function min	>>> min(list2) 10
Function max	>>> max(list1) 'Red'
Function sum (not defined on strings)	>>> sum(list2) 60
Membership operator in	>>> 40 in list2 False

6. The following table lists some important built-in functions that can be applied to lists:

Function	Explanation
<code>L.append(e)</code>	Inserts the element <code>e</code> at the end of the list <code>L</code> .
<code>L.extend(L2)</code>	Inserts the items in the sequence <code>L2</code> at the end of the elements of the list <code>L</code> .
<code>L.remove(e)</code>	Removes the first occurrence of the element <code>e</code> from the list <code>L</code> .
<code>L.pop(i)</code>	Removes the element at index <code>i</code> from the list <code>L</code> .
<code>L.count(e)</code>	Returns count of occurrences of object <code>e</code> in the list <code>L</code> .
<code>L.index(e)</code>	Returns index of the first occurrence of the object <code>e</code> , if present in the list <code>L</code> .
<code>L.insert(i,e)</code>	Inserts element <code>e</code> at index <code>i</code> in the list <code>L</code> .
<code>L.sort()</code>	Sorts the elements of the list <code>L</code> .
<code>L.reverse()</code>	Reverses the order of elements in the list <code>L</code> .

7. Comprehension provides an easy shorthand notation for creating lists, sets, and dictionaries. For example, the expression `[x**3 for x in range(1, end)]` creates a list containing cube of each element in the range `(1, end)`.
8. Assignment of a list to another variable does not create a new list. Instead, the new variable references the existing list that appears on the right-hand side of the `=` operator.
9. To create another instance of an object of types such as `list`, `set`, and `dict`, we need to import the `copy` module and invoke the function `copy.copy()`. However, to create a copy of a list object, so that embedded objects (at all levels) get copied, the function `deepcopy` of the `copy` module should be used.
10. The function `map` is used for transforming every value in a given iterable object by applying a function to it.
11. The function `filter` that takes a function and a iterable object as the input parameters and returns only those elements from the iterable object for which the function returns `True`.
12. The function `reduce`, available in `functools` module takes two arguments: a function, and an iterable object, and applies the function on the iterable object to produce a single value.
13. A set is an unordered collection of objects without any duplicates. An object of type `set` may be created by enclosing within curly brackets the elements of the set. A set object may also be created by applying the `set` function to a list, string or tuple.

14. The following table lists some important built-in functions that can be applied to sets:

Function	Description
S.add(e)	Adds the element e to the set S.
S1.update(L1)	Adds the items in object L1 to the set S1.
S.remove(e)	Removes the element e from the set S.
S.pop()	Removes an element from the set S.
S.clear()	Removes all elements from the set S.
S.copy()	Creates a copy of the set S.
S1.union(S2)	Returns union of sets S1 and S2.
S1.intersection(S2)	Returns a set containing elements common in the sets S1 and S2.
S1.difference(S2)	Returns a set containing elements in the set S1 but not in the set S2.
S1.symmetric_difference(S2)	Returns a set containing elements which are in one of the two sets S1 and S2, but not in both.

15. The operators `<=`, `==`, and `>=` may be used to check whether a given set is a subset, equal, or superset of another set.
16. A tuple is a non-scalar type. It is an ordered sequence of elements. Unlike lists, tuples are immutable, i.e. elements of a tuple cannot be overwritten.
17. A tuple with a single element is also known as singleton tuple. Python allows values contained in a tuple to be of varied types.
18. If a component of a tuple contains an object that itself is modifiable, that object can be modified.
19. The following table lists operations that can be applied on tuples:

```
>>> t1 = ('Monday', 'Tuesday')
>>> t2 = (10, 20, 30)
```

Operation	Example
Multiplication operator *	>>> t1 * 2 ('Monday', 'Tuesday', 'Monday', 'Tuesday')
Concatenation operator +	>>> t3 = t1 + ('Wednesday',) >>> t3 ('Monday', 'Tuesday', 'Wednesday')
Length operator len	>>> len(t1) 2
Indexing	>>> t2[-2] 20
Slicing Syntax: start:end:inc	>>> t1[1:2] 'Tuesday'
Function min	>>> min(t2) 10
Function max	>>> max(t2) 30
Function sum (not defined on strings)	>>> sum(t2) 60
Membership operator in	>>> 'Friday' in t1 False

20. The function `tuple` is used to convert a sequence to a tuple.
21. The function `zip` is used to produce a list, whose *i*th element is a tuple containing *i*th element from all the sequences in the call to the `zip` function.
22. The following table lists two built-in functions that can be applied on tuples.

Function	Explanation
<code>T.count(e)</code>	Returns count of occurrences of element <code>e</code> in tuple <code>L</code> .
<code>T.index(e)</code>	Returns index of the element <code>e</code> from tuple <code>L</code> .

23. A dictionary is a non-scalar type. It is an unordered collection of *key-value* pairs. A dictionary may be specified by including the *key-value* pairs within curly braces. A key and its associated value are colon-separated. Each key uniquely identifies the value associated with it, and also, acts as an index. As keys in

- a dictionary are immutable, lists cannot be used as keys.
24. The following table lists the operations that can be applied on the dictionary of digit-name pairs:
- ```
digits = {0:'Zero', 1:'One', 2:'Two',
 3:'Three', 4:'Four', 5:'Five', 6:'Six',
 7:'Seven', 8:'Eight', 9:'Nine'}
```
25. A *key-value* pair can be removed from a dictionary using `del` operator.
26. The following table lists some of the built-in functions that can be applied on the type `dict`:

| Function                         | Description                                                                                                                            |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>D.items()</code>           | Return an object comprising of tuples of key-value pairs present in dictionary D.                                                      |
| <code>D.keys()</code>            | Return an object comprising of all keys of dictionary D.                                                                               |
| <code>D.values()</code>          | Return an object comprising of all values of dictionary D.                                                                             |
| <code>D.clear()</code>           | Removes all key-value pairs from dictionary D.                                                                                         |
| <code>D.get(key, default)</code> | For the specified key, the function returns the associated value. Returns the default value in case of absence of key in dictionary D. |
| <code>D.copy()</code>            | Creates a shallow copy of dictionary D.                                                                                                |
| <code>D1.update(D2)</code>       | Adds the key-value pairs of dictionary D2 in dictionary D1.                                                                            |

## EXERCISES

1. Write a function that takes a list of values as input parameter and returns another list without any duplicates.
2. Write a function that takes a list of numbers as input from the user and produces the corresponding cumulative list where each element in the list at index i is the sum of elements at index j <= i.
3. Write a program that takes a sentence as input from the user and computes the frequency of each letter. Use a variable of dictionary type to maintain the count.
4. Identify the output produced when the following functions are invoked.

```
1. def func():
 l1 = list()
 l2 = list()
 for i in range(0,5):
 l1.append(i)
```

```

 l2.append(i+3)
 print(l1)
 print(l2)
2. def func():
 l1 = list()
 l2 = list()
 for i in range(0,5):
 l1.append(i)
 l2.append(i+3)
 l1, l2 = l2, l1
 print(l1)
 print(l2)

```

5. Determine the output of the following code snippets:

```

1. c = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = 0
for i in range(0, 10):
 if (c[i]%2 == 0):
 result += c[i]
print(result)
2. c = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = 0
for i in range(0, 10):
 if (c[i]%2 != 0):
 result += c[i]
print(result)
3. subject = 'computer'
subject = list(subject)
ch = subject[0]
for i in range(0, len(subject)-1):
 subject[i] = subject[i+1]
subject[len(subject)-1]=ch
print(''.join(subject))
4. quantity = [15, 30, 12, 34, 56, 99]
total = 0
for i in range(0, len(quantity)):
 if (quantity[i] > 15):
 total += quantity[i]
print(total)
5. x = [1, 2, 4, 6, 9, 10, 14, 15, 17]
for i in range(0, len(x)):
 if (x[i]%2 == 0):
 x[i] = 4*i
 elif (x[i]%3 == 0):
 x[i] = 9*i
 else:
 x[i] *= 2
print(x)

```

6. Write a function that takes n as an input and creates a list of n lists such that ith list contains first five multiples of i.

7. Write a function that takes a number as an input parameter and returns the correspond text in words, for example, on input 452, the function should return 'Four Five Two'. Use a dictionary for mapping digits to their string representation.

8. Given the following inputs, indicate in each case (a) to (x), whether the statements will execute successfully. If so, give what will be the outcome of execution? Also give the output of print statements (where applicable):

```

address = 'B-6, Lodhi road, Delhi'
list1 = [1, 2, 3]

```

```

list2 = ['a', 1, 'z', 26, 'd', 4]
tuple1 = ('a', 'e', 'i', 'o', 'u')
tuple2 = ([2,4,6,8], [3,6,9], [4,8], 5)
dict1 = {'apple': 'red', 'mango': 'yellow',
'orange': 'orange'}
dict2 = {'X': ['eng', 'hindi', 'maths',
'science'], 'XII': ['english', 'physics',
'chemistry', 'maths']}
1.list1[3] = 4
2.print(list1 * 2)
3.print(min(list2))
4.print(max(list1))
5.print(list(address))
6.list2.extend(['e', 5])
print(list2)
7.list2.append(['e', 5])
print(list2)
8.names = ['rohan', 'mohan', 'gita']
names.sort(key= len)
print(names)
9.list3 = [(x * 2) for x in range(1, 11)]
print(list3)
10.del list3[1:]
print(list3)
11.list4 = [x+y for x in range(1,5) for y
in range(1,5)]
print(list4)
12.tuple2[3] = 6
13.tuple2.append(5)
14.t1 = tuple2 +(5)
15 ','.join(tuple1)
16.list(zip(['apple', 'orange'], ('red',
'orange')))
17.dict2['XII']
18.dict2['XII'].append('computer science'),
dict2
19.'red' in dict1
20.list(dict1.items())
21.list(dict2.keys())
22.dict2.get('XI', 'None')
23.dict1.update({'kiwi':'green'})
print(dict1)

```

9. Consider the following three sets, namely vehicles, heavyVehicles, and lightVehicles:

```

>>>vehicles = {'Bicycle', 'Scooter', 'Car',
'Bike', 'Truck', 'Bus', 'Rickshaw'}
>>> heavyVehicles = {'Truck', 'Bus'}
>>> lightVehicles = {'Rickshaw', 'Scooter',
'Bike', 'Bicycle'}

```

Determine the output on executing the following statements:

```

1.lytVehicles = vehicles - heavyVehicles
print(lytVehicles)
2.hvyVehicles = vehicles - lightVehicles
print(hvyVehicles)
3.averageWeightVehicles = lytVehicles &
hvyVehicles
print(averageWeightVehicles)
4.transport = lightVehicles |
heavyVehicles

```

```
 print(transport)
5. transport.add('Car')
 print(transport)
6. for i in vehicles:
 print(i)
7. len(vehicles)
8. min(vehicles)
9. set.union(vehicles, lightVehicles,
 heavyVehicles)
```

<sup>!</sup>  
This section may be skipped on first reading without loss of  
continuity

# CHAPTER 8

## RECURTION

### CHAPTER OUTLINE

[8.1 Recursive Solutions for Problems on Numeric Data](#)

[8.2 Recursive Solutions for Problems on Strings](#)

[8.3 Recursive Solutions for Problems on Lists](#)

[8.4 Problem of Tower of Hanoi](#)

In the previous chapters, we have used functions for various programming tasks. When computation in a function is described in terms of the function itself, it is called a recursive function. Python allows a function to call itself, possibly with new arguments.

Recursion aims to solve a problem by providing a solution in terms of the simpler version of the same problem. A recursive function definition comprises two parts. The first part is the base part, which is the solution of the simplest version of the problem, often termed stopping or termination criterion. The second part is the inductive part, which is the recursive part of the problem that reduces the complexity of the problem by redefining the problem in terms of simpler version(s) of the original problem itself.

A recursive solution comprises:  
base part and inductive part

### 8.1 RECURSIVE SOLUTIONS FOR PROBLEMS ON NUMERIC DATA

#### 8.1.1 Factorial

Suppose we wish to find the factorial of a non-negative number  $n$ . To do this, we have to multiply numbers 1 to  $n$ :

```
n! = n * (n-1) * (n-2) * (n-3) * ... * 2 * 1
```

Factorial of 0 is of course defined to be 1. We will first discuss the iterative approach, thus, providing an insight into how space is allocated on system stack when a function is invoked. This is followed by the recursive approach for computing factorial.

#### *Iterative Approach*

Factorial of a number  $n$  can be computed by using a loop that iterates over elements in the `range(1, n + 1)`. An iterative function for computing the factorial of a number is given in Fig. 8.1.

```
01 def factorial(n):
02 """
03 Objective: To compute factorial of a number
04 Input Parameter: n - numeric value
05 Return Value: fact - numeric value
06 """
07 fact = 1
08 assert n >= 0
09 for i in range(1, n + 1):
10 fact = fact * i
11 return fact
12
13 def main():
14 """
15 Objective: To compute factorial of a number provided
16 by user
17 Input Parameter: None
```

```

17 Return Value: None
18
19 n = int(input('Enter the number: '))
20 fact = factorial(n)
21 print('Factorial of', n, 'is', fact)
22
23 if __name__=='__main__':
24 main()

```

**Fig. 8.1** Program to compute factorial of a number  
(factorial1.py)

#### *Recursive Approach*

Next, we rewrite the function `factorial` to compute factorial of a number using recursive approach. Recall that  $n! = n * (n-1) * (n-2) * (n-3) * \dots * 2 * 1$ . Alternatively, we may write the definition of  $n!$  as

$$n! = \begin{cases} 1 & \text{if } n==0 \text{ or } 1 \\ n * (n-1)! & \text{if } n > 1 \end{cases}$$

or equivalently as

```

factorial(n) = 1 if n==0 or 1
 = n * factorial(n-1) if n > 1

```

recursive definition of factorial

Thus, the problem of computing `factorial(n)` – a problem of size  $n$  – has been expressed in terms of a problem of size  $(n - 1)$ , i.e. to compute

factorial(n), we multiply by n the result of computing factorial(n-1). Now factorial(n-1) can be computed using the same technique, i.e. to compute factorial(n-1), we multiply by n-1 the result of computing factorial(n-2). Thus, we solve a problem of size (n - 1) in terms of a problem of size (n - 2), and so on until we arrive at the problem of size 1, which is a trivial task. Thus, the factorial of a number, say 3, it may be computed as follows:

```
factorial(3) = 3 * factorial(2)
 = 3 * (2 * factorial(1))
 = 3 * (2 * 1))
 = 3 * 2
 = 6
```

Now, we are ready to present a recursive function to compute factorial(n) (Fig. 8.2):

```
01 def factorial(n):
02 """
03 Objective: To compute factorial of a positive number
04 Input Parameter: n - numeric value
05 Return Value: factorial of n - numeric value
06 """
07 assert n >= 0
08 if n == 0 or n == 1:
09 return 1
10 else:
11 return n * factorial(n-1)
12
```

```
13 def main():
14 """
15 Objective: To compute factorial of a number provided
16 by user
17 Input Parameter: None
18 Return Value: None
19 """
20 n = int(input('Enter the number: '))
21 result = factorial(n)
22 print('Factorial of', n, 'is', result)
23
24 if __name__=='__main__':
25 main()
```

**Fig. 8.2** Program to compute factorial of a number  
(factorialRecur.py)

Let us examine the recursive function `factorial` to compute the factorial of a number, say 3. The script `factorialRecur` (Fig. 8.2) invokes the function `main` in line 24 which further invokes the function `factorial` in line 20. Whenever a function is called in a program, the program must remember the point of call, so that the control can be returned appropriately when the function is completely executed. Also, it should save the current state, i.e. binding of variables. For every call to a function, an entry in a stack, called stack frame or activation record, is created. In this entry, temporary storage is allocated to parameters, local variables, and return address. Like any other stack, the objects in the activation stack leave in the reverse order of their arrival.

call to a function creates an entry in the stack frame/  
activation record

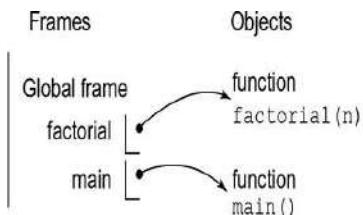
stack frame stores parameters, local variables, and return  
address

Figure 8.3 shows stack frames created and exited during the execution of the script in Fig. 8.2. Note that the exited frames (border lines as well as the content) are shown in light grey color. To begin with, when Python encounters the definitions of `factorial` and `main` functions in the program, an entry is created for `factorial` and `main` functions in the global frame (Fig. 8.3(a)). When the condition `if`

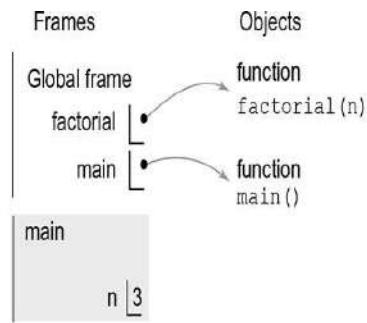
`__name__ == '__main__'` evaluates to `True`, the function `main` is invoked and a new frame is created. On entering 3 as the input value for `n` (line 19), the variable binding for it is created (Fig. 8.3(b)). On invoking the function `factorial` (line 20, Fig. 8.2) with the argument 3, a new frame is created containing a binding for the input parameter `n` (Fig. 8.3(c)). Further, since the value of `n` is greater than 1, during execution of line 11, a recursive call is made to the function `factorial` with 2 as the input argument. This creates another stack frame as shown in Fig. 8.3(d). Note that the parameter `n` is now bound to value 2. This recursive call further invokes function `factorial` with input argument 1 (Fig. 8.3(e)). In this frame, the parameter `n` is bound to value 1. As the value associated with variable `n` is now 1, the condition in line 8 becomes `True`. So, this instance of function call creates the returns value 1 (Fig. 8.3(f)). On completion of the execution of this instance of the function `factorial`, the frame for the function call that has just terminated is removed from the stack, and the control is returned to the previous frame (Fig. 8.3(g)) at the point of call (line 11), i.e. frame for function `factorial`, where parameter `n` is bound to value 2. This instance of function call now creates the returns value 2 as shown in Fig. 8.3(g). The function returns value 2 on execution of line 11. This value 2 is returned in line 11 (variable `n` is now bound to value 3, Fig. 8.3(h)). This instance of function call now creates the returns value 6 as shown in Fig. 8.3(h). Execution of line

`11` returns value `6` to the `main` function (Fig. 8.3(i)). In the `main` function, the just returned value `6` gets bound to the variable `result` (Fig. 8.3(i)). When the `main` function completes, it returns value `None` associated with it (Fig. 8.3(j)), and the control is returned to the global frame from the frame `main` function (Fig. 8.3(k)).

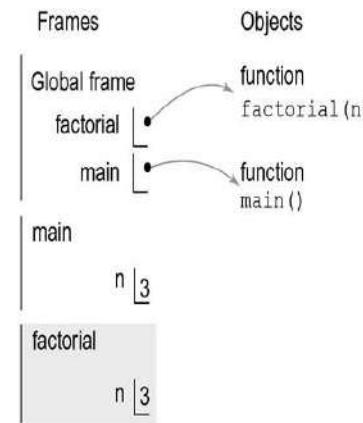
Note that the stack is growing in the downward direction. The value of any variable can be found in the current state of the program by looking at the most recent stack frame (highlighted frame in the visualizer).



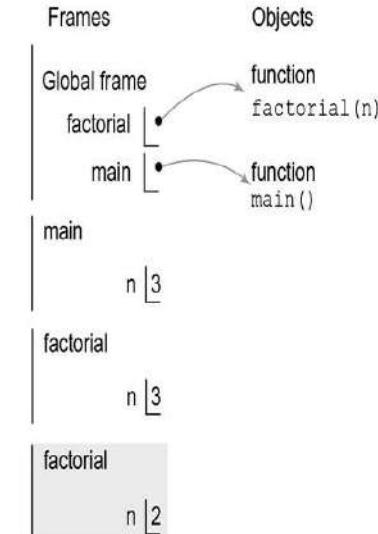
(a)



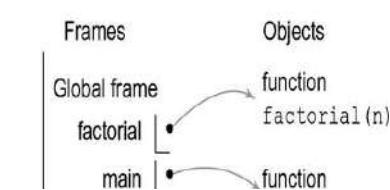
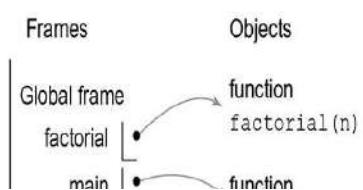
(b)

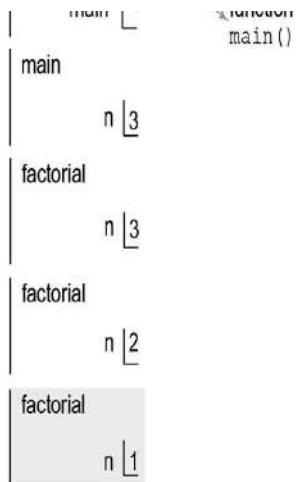


(c)

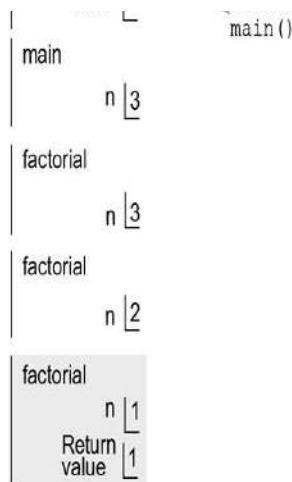


(d)

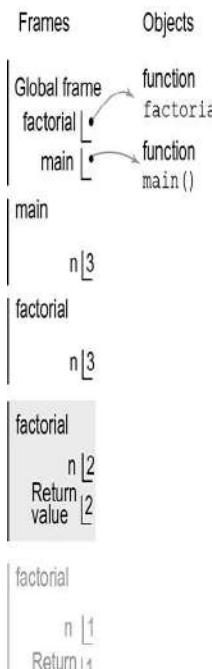




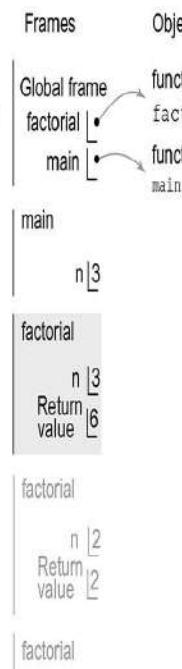
(e)



(f)



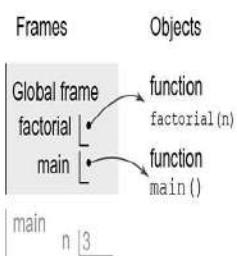
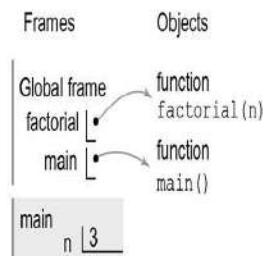
(g)

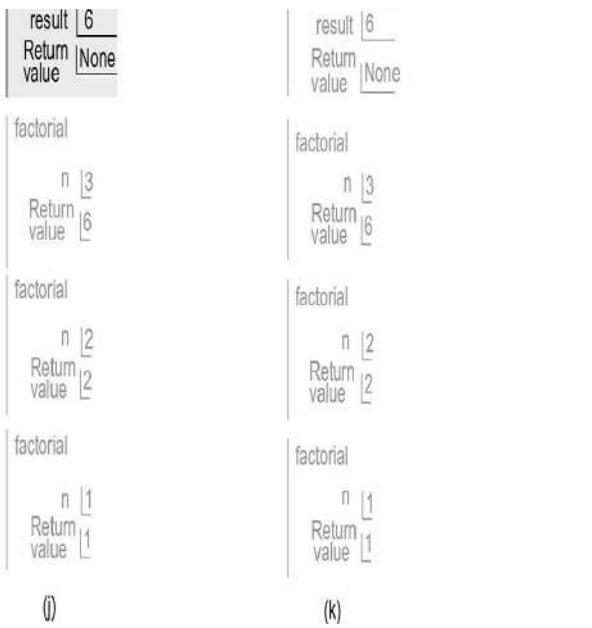


(h)



(i)





**Fig. 8.3** Recursive calls to function factorial

Next, we rewrite the function factorial with a slight change:

```

def factorial(n):
 """
 Objective: To compute factorial of a
 positive number

 Input Parameter: n - numeric value

 Return Value: factorial of n - numeric
 value

 """
 assert n >= 0

 if n == 0 or n == 1:
 return 1

 return n * factorial(n-1)

```

alternative definition for function factorial

If the base condition (`n == 0 or n == 1`) evaluates to `True`, control is automatically returned from the function with value 1. However, failure of the base condition leads to the execution of the line `return n * factorial(n-1)`. Note that in case `n` has value 0 or 1, this statement will not be executed as the control would have already returned to the caller function i.e. `main()`, in this example.

### 8.1.2 Fibonacci Sequence

Fibonacci sequence is a sequence of integers. The first and the second numbers in the sequence are 0 and 1. A subsequent term in the sequence is computed as the sum of immediately preceding two terms. Thus, we get the Fibonacci sequence as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Suppose, we want to find out the  $n$ th term in the Fibonacci sequence. It may be computed as follows:

$$\text{fibo}(n) = \begin{cases} 0 & \text{if } n==1 \\ 1 & \text{if } n==2 \\ \text{fibo}(n-1)+\text{fibo}(n-2) & \text{if } n>2 \end{cases}$$

#### *Iterative Approach*

In this section, we develop a function to compute the  $n$ th term of the Fibonacci sequence. If the value of `n` is 1 or 2, the function just needs to return 0 or 1, respectively. However, if  $n>2$ , the  $n$ th term is computed using a loop with iterator `i` that iterates over elements in the range `(3, n+1)`, and for each element in the `range(3, n+1)`, the new value is computed by taking the sum of the previous two terms that we call `secondLast` and `last`. Iterative function for finding the  $n$ th term of the Fibonacci sequence is given in Fig. 8.4.

```
01 def fibonacci(n):
02 """
03 Objective: To determine nth term in Fibonacci sequence
04 Input Parameter: n- numeric value
05 Return Value: result - numeric value
06 """
07 assert n > 0
08 secondLast = 0
09 last = 1
10 if n == 1:
11 return secondLast
12 elif n == 2:
13 return last
14 for i in range(3, n+1):
15 result = secondLast + last
16 secondLast = last
17 last = result
18 return result
19
20 def main():
21 """
22 Objective: To determine nth term in Fibonacci sequence based
23 on user input
24 Input Parameter: None
25 Return Value: None
26 """
27 n = int(input('Enter the value of n: '))
28 result = fibonacci(n)
29 print(n, 'th Fibonacci term', 'is', result)
30
31 if __name__=='__main__':
32 main()
```

---

**Fig. 8.4** Program to determine nth Fibonacci term (fib01.py)

### *Recursive Approach*

In this section, we compute  $n$ th Fibonacci term using recursive approach. As mentioned earlier,  $n$ th term in the Fibonacci sequence may be computed as follows:

$$\text{fib0}(n) = \begin{cases} 0 & \text{if } n==1 \\ 1 & \text{if } n==2 \\ \text{fib0}(n-1) + \text{fib0}(n-2) & \text{if } n>2 \end{cases}$$

recursive definition of  $n$ th fibonacci term

where  $\text{fib0}(n)$  denotes  $n$ th term of Fibonacci series.

We call the problem of computing  $\text{fib0}(n)$ , a problem of size  $n$ . In the above formulation, we have expressed the problem of computing  $\text{fib0}(n)$  – a problem of size  $n$ , in terms of two problems:

- a problem of computing  $\text{fib0}(n-1)$  – problem of size  $n-1$  and
- a problem of computing  $\text{fib0}(n-2)$  – problem of size  $n-2$

These problems of sizes smaller than that of the original problem can be further expressed in terms of problems of even smaller sizes, and so on until we reach a problem of size 1 or 2 – a trivial problem to solve. For example, 4th term of Fibonacci sequence may be computed as follows:

$$\begin{aligned} \text{fib0}(4) &= \text{fib0}(3) + \text{fib0}(2) \\ &= (\text{fib0}(2) + \text{fib0}(1)) + \text{fib0}(2) \\ &= (1 + \text{fib0}(1)) + \text{fib0}(2) \\ &= (1 + 0) + \text{fib0}(2) \end{aligned}$$

```
= 1 + fibo(2)
= 1 + 1
= 2
```

In light of the above discussion, we present a recursive function to compute  $n$ th term in the Fibonacci sequence (Fig. 8.5).

Note that during each call to function `Fibonacci`, a new stack frame will be allocated, and variable bindings will be created, allocating storage space for new values. Also, the program remembers the point of call to the function, so as to return control back to the point of call when the function completes. Suppose we wish to determine 4th Fibonacci term. Determining 4th Fibonacci number involves computing 3rd and 2nd Fibonacci terms. Figures 8.6(a) and (b) show the contents of the run-time stack as visualized in Python Tutor for computation of the 3rd and 2nd Fibonacci terms. Subsequently, 4th term is computed using these two values as shown in Fig. 8.6(c).

```
01 def fibo(n):
02 """
03 Objective: To determine nth term in Fibonacci sequence
04 Input Parameter: n: numeric value
05 Return Value: nth Fibonacci term - numeric
06 """
07 assert n > 0
08 if n == 1:
09 return 0
10 elif n == 2:
11 return 1
12 else:
13 return fibo(n-1) + fibo(n-2)
14
15 def main():
16 """
17 Objective: To determine nth term in Fibonacci sequence
18 based on user input
19 Input Parameter: None
20 Return Value: None
21 """
```

```
22 n = int(input('Enter the value of n: '))
23 result = fibo(n)
24 print(n, 'th Fibonacci term', 'is', result)
25
26 if __name__=='__main__':
27 main()
```

**Fig. 8.5** Program to determine  $n$ th term in the Fibonacci sequence (fiborecur.py)

Note that computation of 4th term in the Fibonacci sequence involves repetitive computations of second Fibonacci term. In general, computing  $n$ th term in the Fibonacci sequence in a recursive manner involves repetitive computations that add to space and time requirements of the problem. Thus, use of recursion should be best avoided whenever we can conceive of a simple iterative alternative solution. However, to learn recursion as a programming technique, we will begin with recursive solutions to the simple problems for which iterative solutions may be more efficient. Subsequently, we will give several examples of problems for which it is extremely hard, if not impossible, to conceive iterative solutions. The problems like tower of Hanoi, permutation generation, 8-queen's problem, Knight's tour problem, and the problem of creating patterns within pattern using fractals fall under this category.

a recursive solution may add to space and time requirements

prefer iterative solution over recursive if iterative solution is simpler to conceive

## 8.2 RECURSIVE SOLUTIONS FOR PROBLEMS ON STRINGS

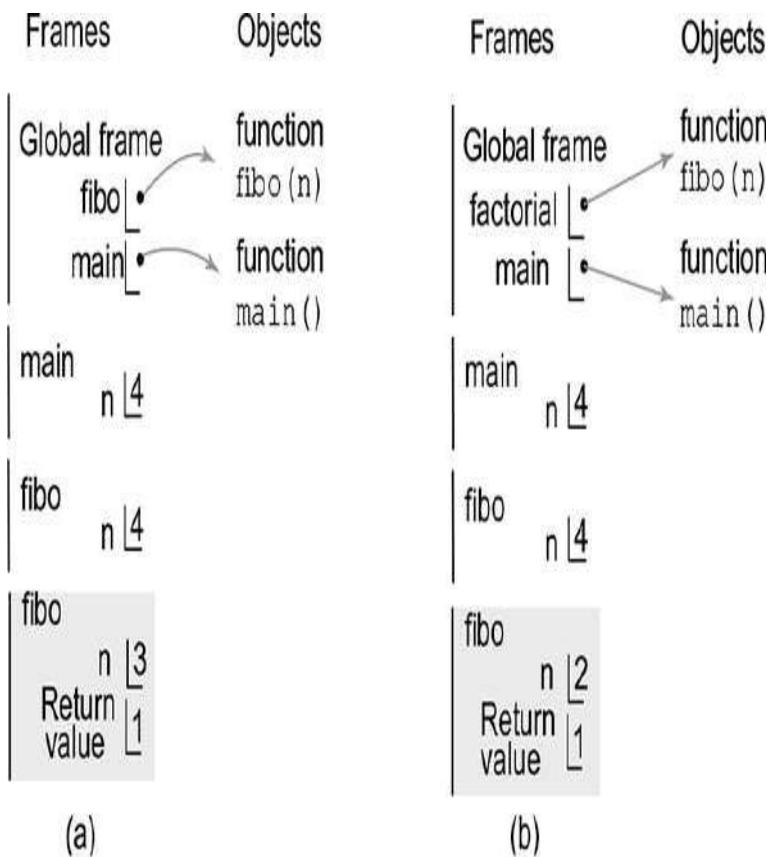
In this section, we will solve some problems related to strings using recursion.

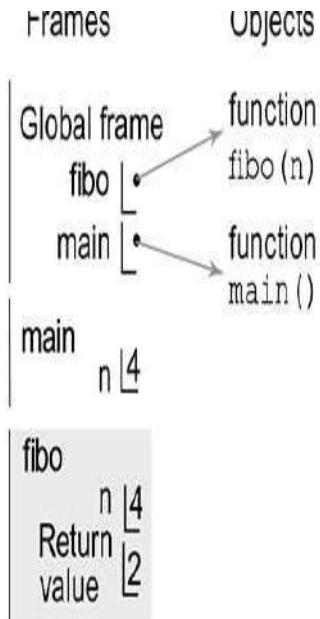
## 8.2.1 Length of a String

Let us begin with the problem of computing length of a string. Although Python provides a function `len` for computing the length of a string, we will develop our function `length` for this purpose using the concept of slicing. Let us first examine the base case when the string is an empty string. In this case, the length of the string is 0. However, if the string is non-empty, its length may be computed as 1 plus the length of the string excluding the first character. Thus, the length of the string can be expressed using recursion as follows:

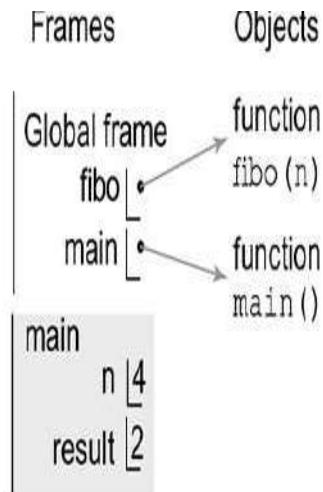
$$\text{length}(\text{str1}) = \begin{cases} 0 & \text{if } \text{str} == "" \\ 1 + \text{length}(\text{str}[1:]) & \text{otherwise} \end{cases}$$

recursive definition for length of a string





(c)



(d)

**Fig. 8.6** Recursive calls to function fibo

Thus, a problem of size  $n$  ( $n > 0$ ) has been expressed in terms of a problem of size  $(n - 1)$ . A recursive function to do this is given in Fig. 8.7. Suppose we need to compute the length of the string 'Zero'. We may do so using the function `length` as follows:

$$\begin{aligned}
 \text{length('Zero')} &= 1 + \text{length('ero')} \\
 &= 1 + (1 + \text{length('ro')}) \\
 &= 1 + (1 + (1 + \text{length('o')})) \\
 &= 1 + (1 + (1 + (1 + \text{length('')}))) \\
 &= 1 + (1 + (1 + (1 + (1 + 0)))) \\
 &= 1 + (1 + (1 + (1 + 1))) \\
 &= 1 + (1 + 2) \\
 &= 1 + 3 \\
 &= 4
 \end{aligned}$$

```
01 def length(str1):
02 """
03 Objective: To determine length of the input string
04 Input Parameter: str1- string
05 Return Value: numeric
06 """
07 if str1== '':
08 return 0
09 else:
10 return 1+length(str1[1:])
11
12 def main():
13 """
14 Objective: To determine length of string entered by user
15 Input Parameter: None
16 Return Value: None
17 """
18 str1 = input('Enter the string: ')
19 result = length(str1)
20 print('Length of string ', str1, ' is ', result)
21
22 if __name__=='__main__':
23 main()
```

**Fig. 8.7** Program to determine length of the string (`lenStr.py`)

Figure 8.8 gives stack frames created and exited on the execution of the script `lenStr` with 'Zero' as an input value for `str1`, as visualized in Python Tutor.

### 8.2.2 Reversing a String

A string is said to be a reversed string of another string if the first character in the reversed string is the last character of the given string, the second character in the reversed string is the second last character of the given string, and so on. To reverse a given string, the trivial base case is the case when the string is an empty string, in which case, the reverse of the given string is the empty string itself. However, if the string is non-empty, reversed string can be obtained as the last character of the string concatenated with the reverse of string obtained from the original string by dropping the last character:

$$\text{reverse}(\text{str1}) = \begin{cases} \text{if str} == "" \\ \text{str}[-1] + \text{reverse}(\text{str}[:-1]) \text{ otherwise} \end{cases}$$

recursive definition for reverse of a string

Thus, a problem of size  $n$  has been expressed as the problem of size  $(n - 1)$ . A recursive function to compute reverse string of a given string is given in Fig. 8.9. Suppose we need to reverse the string 'Zero', it may be done as follows:

```
reverse('Zero') = 'o' + reverse('Zer')
= 'o' + ('r' + reverse('Zer'))
= 'o' + ('r' + ('e' + reverse('Zer')))
= 'o' + ('r' + ('e' + ('Z' + reverse('')))))
= 'o' + ('r' + ('e' + ('Z' + '')))
= 'o' + ('r' + ('e' + 'Z'))
```

= 'o' + ('r' + 'eZ')

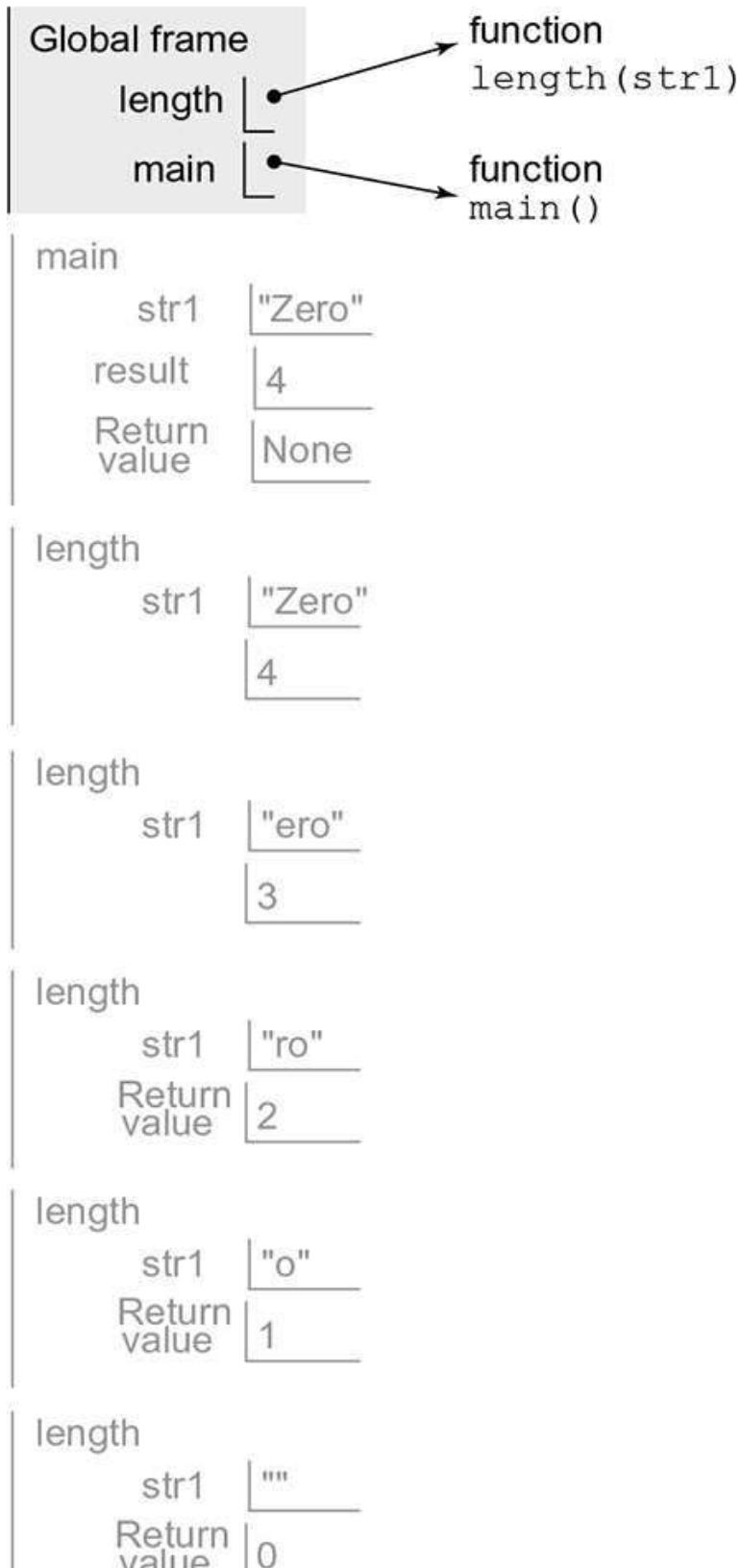
= 'o' + 'reZ'

= 'oreZ'



## Frames

## Objects



**Fig. 8.8** Recursive calls to function length

```
01 def reverse(str1):
02 """
03 Objective: To compute reverse of a string str1
04 Input Parameter: str1 - str
05 Return Value: str
06 """
07 if str1== '':
08 return str1
09 else:
10 return str1[-1] + reverse(str1[:-1])
11
12 def main():
13 """
14 Objective: To compute reverse of a string entered by user
15 Input Parameter: None
16 Return Value: None
17 """
18 str1 = input('Enter the string: ')
19 result = reverse(str1)
20 print('Reverse of string ', str1, ' is ', result)
21
22 if __name__=='__main__':
23 main()
```

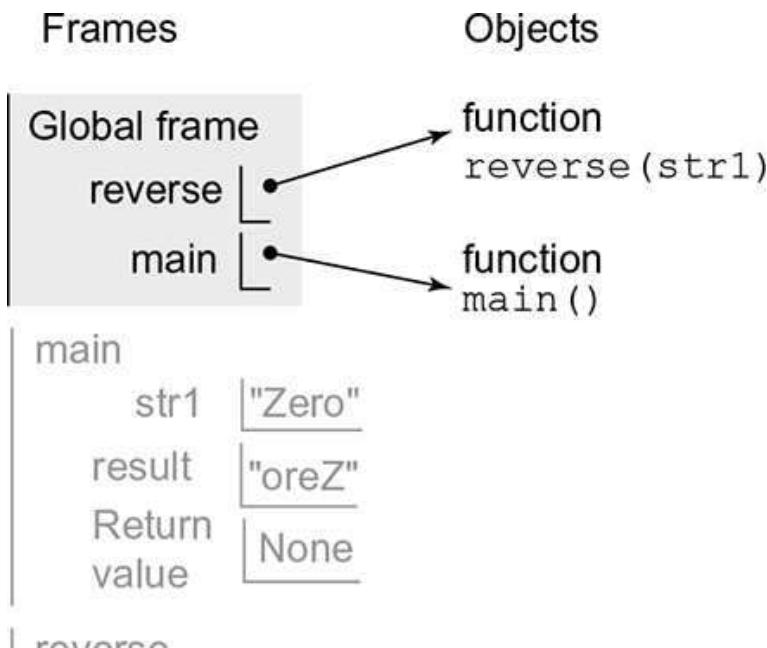
**Fig. 8.9** Program to reverse the string (`reverse.py`)

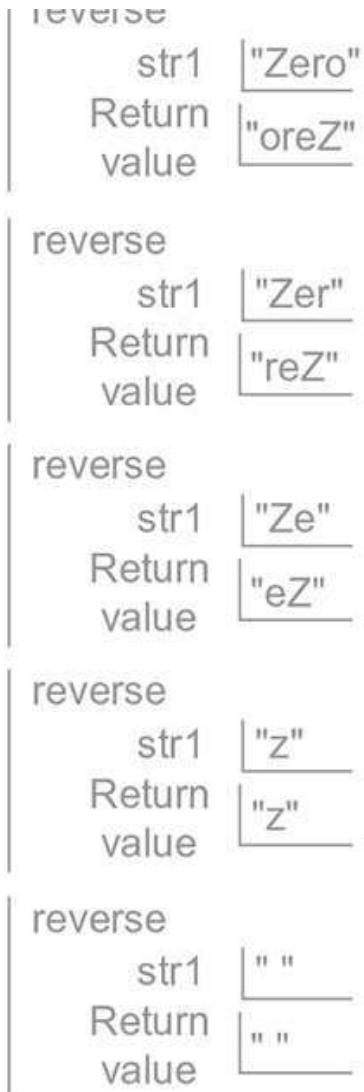
In Fig. 8.10, we present stack frames created and exited on the execution of script `reverse` in Python Tutor on the input string '`Zero`'.

### 8.2.3 Palindrome

A string is said to be a palindrome if the reverse of a string is equal to the string itself. Next, we examine how to check whether a given string is a palindrome. A string can be tested for being palindrome by comparing the characters in the string from the two extreme ends. Thus, we compare the character at index 0 with the character at index -1, the character at index 1 with the character at index -2, and so on, till we reach the middle of the string. In case, a mismatch is found on the way, given string is not a palindrome, and we return `False`; otherwise, it is a palindrome, and we return `True`. This approach can be easily expressed using recursion. The trivial base case is the case where the string is an empty string and as such a palindrome. However, if the string is non-empty, a string can be declared as palindrome if the following two conditions hold:

- First and last characters of the given string match.
- The remaining string, obtained by dropping the first and last characters of the original string is a palindrome.





**Fig. 8.10** Recursive calls to function `reverse`

Thus, the problem of checking whether a string is a palindrome can be expressed using the recursion as follows:

$$\text{isPalindrome}(\text{str1}) = \begin{cases} \text{True} & \text{if } \text{str1} == "" \\ \text{True} & \text{if } \text{str1}[0] == \text{str1}[-1] \\ & \text{and } \text{isPalindrome}(\text{str1}[1:-1]) \\ \text{False} & \text{otherwise} \end{cases}$$

is given string is a palindrome?

To begin with the base case, an empty string is trivially a palindrome. However, if the given string is not an empty string, we can express the problem of testing whether a string of size  $n$  is a palindrome in terms of a problem of size  $(n - 2)$ . Next, we apply the above method to test whether the string 'noon' is a palindrome as follows:

```

isPalindrome('noon') = str1[0]==str1[-1]
and isPalindrome('oo')

= True and isPalindrome('oo')

= True and (str1[0]==str1[-1] and
isPalindrome(''))

= True and (True and isPalindrome(''))

= True and (True and (True))

= True and (True)

= True

```

In light of the above discussion, we present a recursive function to determine whether a string is a palindrome ([Fig. 8.11](#)).

```

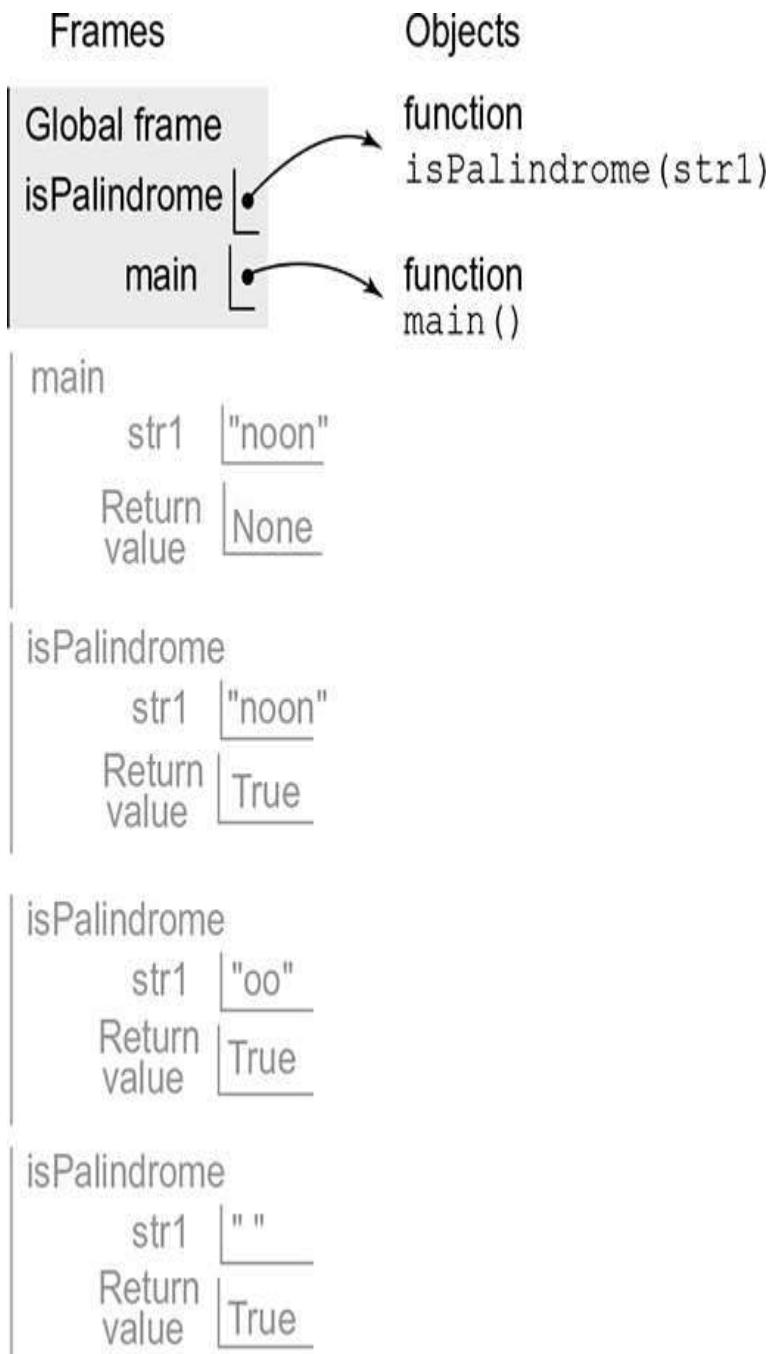
01 def isPalindrome(str1):
02 !!!

```

```
03 Objective: To check whether a string str1 is palindrome
04 Input Parameter: str1- string
05 Return Value: True or False
06 """
07 if str1 == '':
08 return True
09 else:
10 return (str1[0]==str1[-1] and isPalindrome(str1[1:-1]))
11
12 def main():
13 """
14 Objective: To check whether the string entered by user is
15 palindrome
16 Input Parameter: None
17 Return Value: None
18 """
19 str1 = input('Enter the string: ')
20 if (isPalindrome(str1)):
21 print('String is a palindrome')
22 else:
23 print('String is not a palindrome')
24
25 if __name__=='__main__':
26 main()
```

**Fig. 8.11** Program to determine whether a string is a palindrome  
(palindrome.py)

In Fig. 8.12, we present stack frames created and exited on the execution of script `palindrome` when 'noon' is entered as an input value for `str1`.



**Fig. 8.12** Recursive calls to function `isPalindrome`

### 8.3.1 Flatten a List

Let us examine the problem of flattening the list. Given a list of lists, the nesting of lists may occur up to any arbitrary level. By flattening a list, we mean to create a list of all data values in the given list. The data values in the flattened list appear in the left to right order of their appearance in the original list, ignoring the nested structure of the list. Thus, the list

```
[1, [2, [3, 4, [5, 6, [7, 8], [9, [10]]], 11],
12], 13], 14, [[15, 16]], 17]
```

will be flattened to obtain the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16, 17]
```

To create the flattened list, say `lst2`, for a given list `lst1`, we initialize list `lst2` as an empty list and append the data values in the list `lst1` to the list `lst2`, one by one. The overall approach to flatten a list can be summarized as follows:

```
for every element i in list lst1
 if i is not a list
 append it to list lst2
 otherwise,
 flatten list i
```

A recursive function to flatten a list is given in Fig. 8.13.

Let us examine the recursive program to flatten the list `[1, [2, 3]]`. As Python Tutor does not support the `eval` function, we will fix the value of the variable `lst1` in the program to be `[1, [2, 3]]`. This style of programming where values of the variables or parameters cannot be changed without modifying the program is called hard-coding and is strongly discouraged for all practical purposes. Anyway, while executing the program in Fig. 8.13 in the Python Tutor,

we replace line 20 in Fig. 8.13 by the assignment statement:

```
lst1 = [1, [2, 3]]
```

Python Tutor does not support eval function

hard-coding the values of variables

```
01 def flatten(lst1,lst2 = []):
02 """
03 Objective: To flatten a list lst1
04 Input Parameters: lst1, lst2 - list
05 Return Value: lst2- a list
06 """
07 for element in lst1:
08 if type(element)!= list:
09 lst2.append(element)
10 else:
11 flatten(element, lst2)
12 return lst2
13
14 def main():
15 """
16 Objective: To flatten the list entered by user
17 Input Parameter: None
18 Return Value: None
19 """
```

```

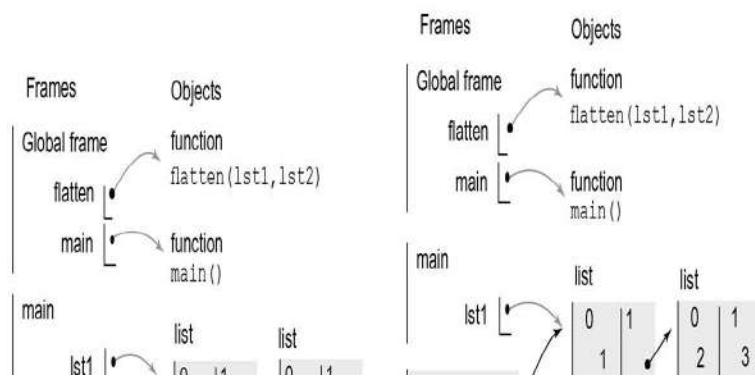
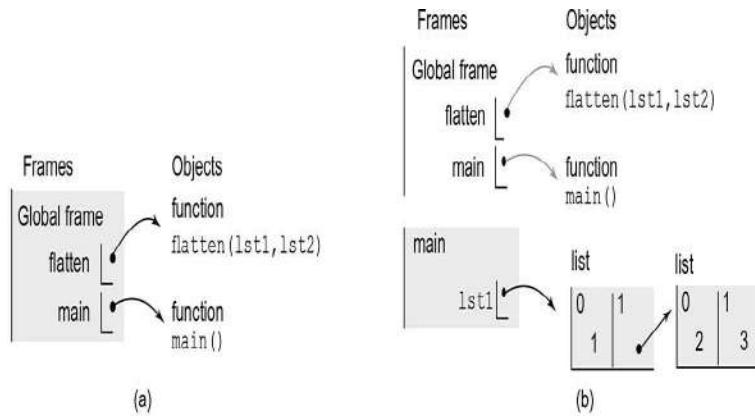
19 ...
20 lst1 = eval(input('Enter the list: '))
21 result = flatten(lst1)
22 print('Flattened List: ', result)
23
24 if __name__=='__main__':
25 main()

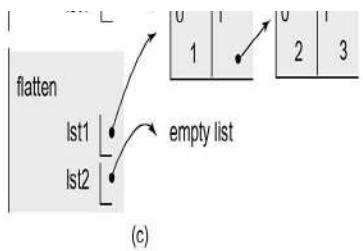
```

**Fig. 8.13** Program to flatten a list (`flatten.py`)

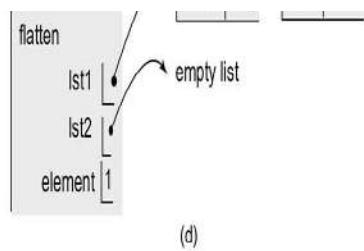
As discussed before, during each call to function `flatten`, a new stack frame will be allocated, and the variable bindings will be created allocating storage space for new values. In Fig. 8.14, we show stack frames created and exited during the execution of the script `flatten`. To begin with, when Python encounters the definitions of `flatten` and `main` functions in the program, an entry is created for `flatten` and `main` functions in the global frame (Fig. 8.14 (a)). When the condition `if __name__=='__main__'` evaluates to `True`, function `main` is invoked and a new frame is created. On assigning `[1, [2, 3]]` as the value associated with list `lst1`, variable binding for it is created as shown in Fig. 8.14(b). On execution of line 21 in Fig. 8.14, the function `flatten` gets invoked, and a new frame is created containing bindings for input parameters `lst1` and `lst2` (Fig. 8.14(c)). Now `lst1` refers to the list to be flattened and `lst2` refers to the flattened list being created (currently empty). Iterating over the list `lst1`, `element` is initialized to `lst1[0]`, i.e. `1` (Fig. 8.14(d)). Since the type of this value is not a list, there is no need to flatten it further. Thus, line 9 is executed, and `1` is appended to the list `lst2` (Fig. 8.14(e)). In the next iteration, `element` takes the value `[2, 3]` (Fig. 8.14(f)). Since the value of the `element` is a list, it needs to be flattened further. The execution of line 11 recursively invokes function `flatten` with the list `[2,3]` as the first argument (the list to be flattened) and the list `lst2` as the second argument (flattened list being

produced). This leads to the creation of new stack frame with respective input parameters, `lst1` pointing to `[2, 3]` and `lst2` pointing to list `[1]` as shown in Fig. 8.14(g). Iterating through this new list `lst1` yields 2 and 3 as values which need no flattening. Thus, resultant list `lst2` will contain `[1, 2, 3]` as shown in Fig. 8.14(k). Now the function returns `[1, 2, 3]` as the list `lst2` (Fig. 8.14(l)). As the current instance of function `flatten` has been completely executed, the control returns to the previous frame (Fig. 8.14(m)) at the point of call (line 11). Since the list `lst1` `[1, 2, 3]` is now exhausted, the function returns the flattened list `lst2` to the `main` function. The flattened list is received in the `main` function in the variable `result` (Fig. 8.14(n)). Thus, we see the new binding of the local variable `result` in the `main` frame with the list `[1, 2, 3]` returned by the function `flatten`. When the `main` function completes, it returns the value `None` associated with it, as it does not return any value (Fig. 8.14(n)), and the control is returned to the global frame from the frame `main` function (Fig. 8.14(o)).

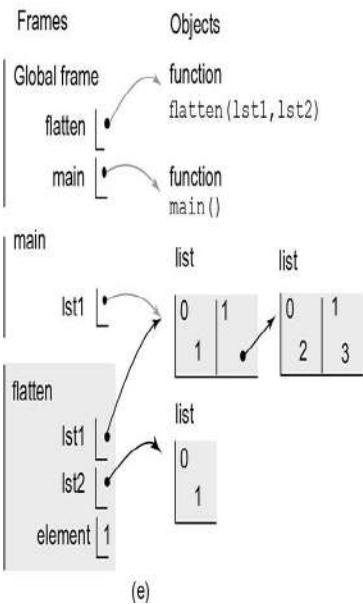




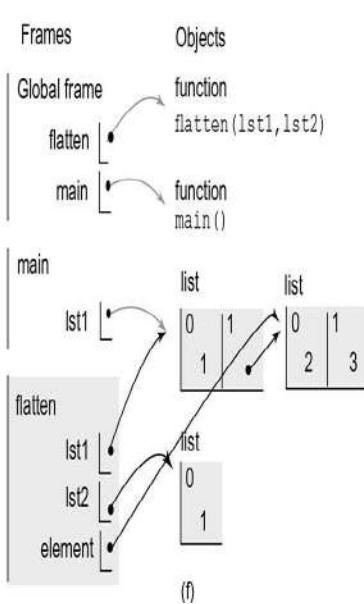
(c)



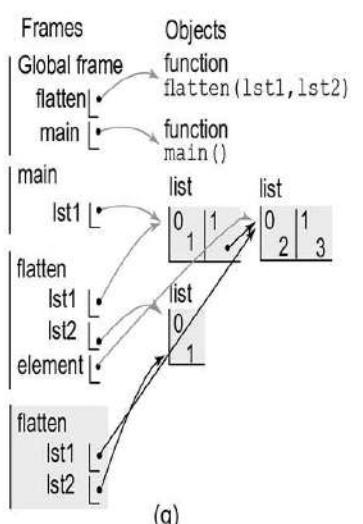
(d)



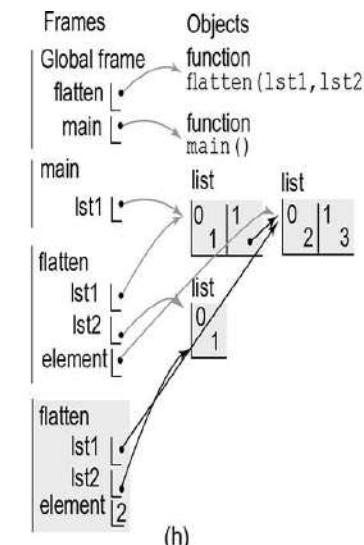
(e)



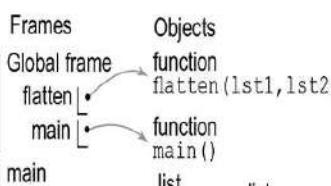
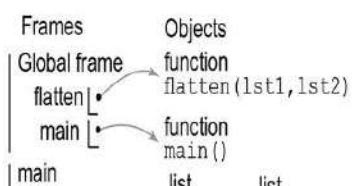
(f)

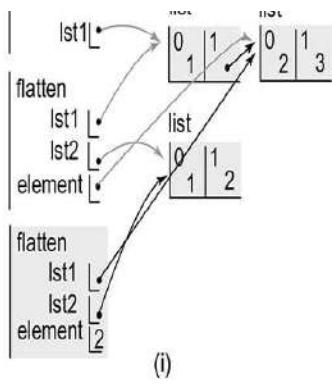


(g)

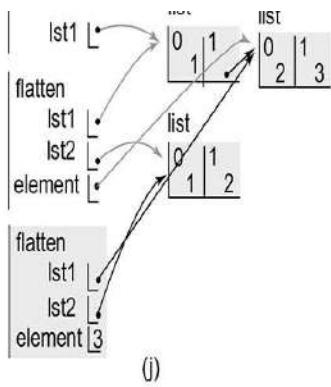


(h)

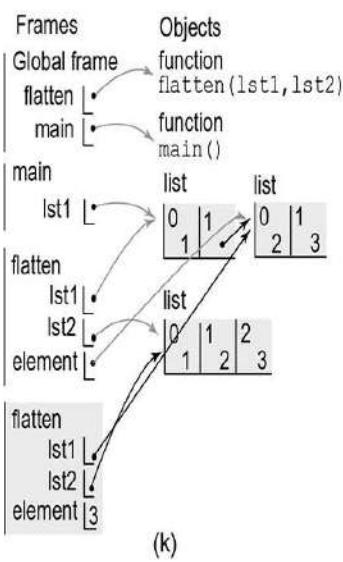




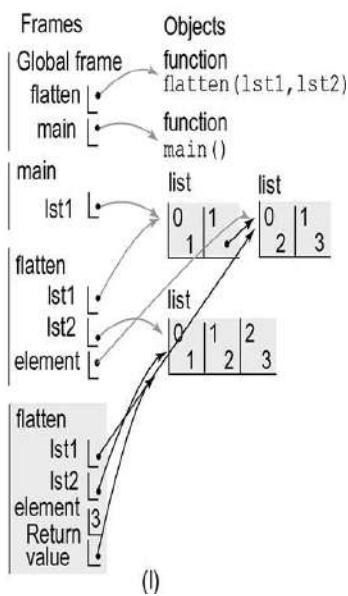
(i)



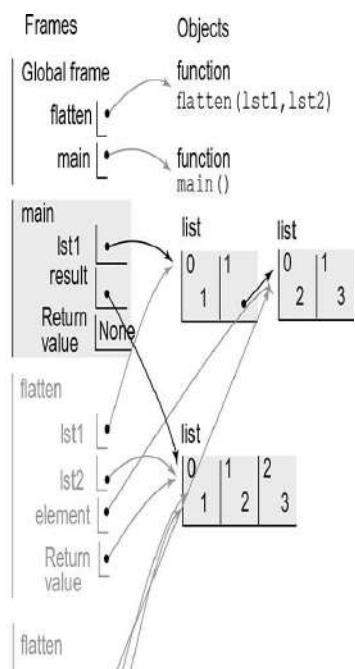
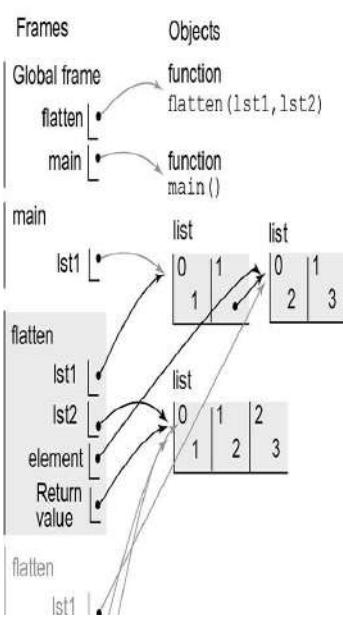
(j)

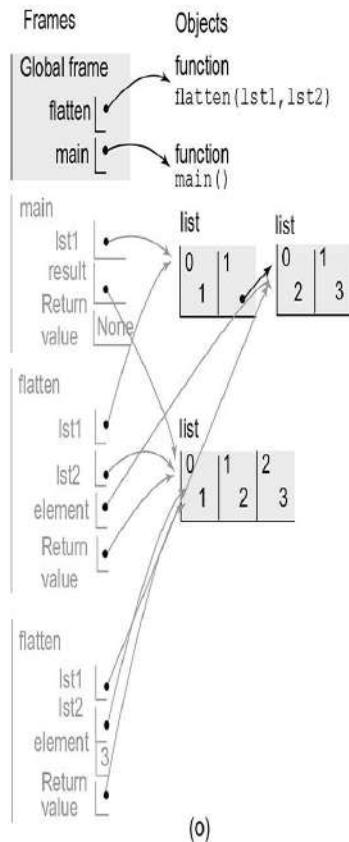
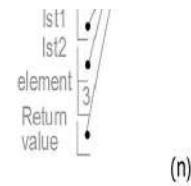
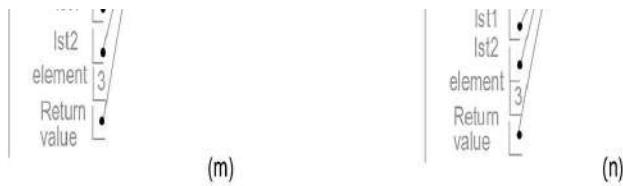


(k)



(l)





**Fig. 8.14** Recursive calls to function `flatten`

### 8.3.2 Copy

In this section, we develop a function for creating a copy of a list. As mentioned earlier, simply assigning a list object to another name does not create a copy of the list; instead, both the names refer to the same list object. So, to create a copy of a source list, say `lst1`, we begin with an empty destination list, say `lst2`, and append every element of list `lst1` to list `lst2` recursively. The trivial base case is the case when the list `lst1` is an empty list, implying that the destination list `lst2` will also be empty. However, if the list `lst1` is non-empty, a copy of a list can be created by appending the first element of list `lst1` to list `lst2`, and invoking the copy function for

the remaining list `lst1` (i.e., excluding the first element). This way of copying a sequence is also known as shallow copy. Thus, a problem of size  $n$  has been expressed in terms of a problem of size  $(n - 1)$ . Based on this discussion, we present the complete program that accepts a list from the user, creates a copy of the list and prints it (Fig. 8.15).

copying a list

```
01 def copy(lst1, lst2 = []):
02 """
03 Objective: To create copy of a list lst1
04 Input Parameters: lst1, lst2 - list
05 Return Value: lst2 - list
06 """
07 if lst1==[]:
08 return lst2
09 else:
10 lst2.append(lst1[0])
11 copy(lst1[1:], lst2)
12 return lst2
13
14 def main():
15 """
16 Objective: To create copy of the list entered by user
17 Input Parameter: None
18 Return Value: None
19 """
```

```

19
20 lst1 = eval(input('Enter the list: '))
21 lst2 = copy(lst1)
22 print('Copy of list lst1 is ', lst2)
23
24 if __name__=='__main__':
25 main()

```

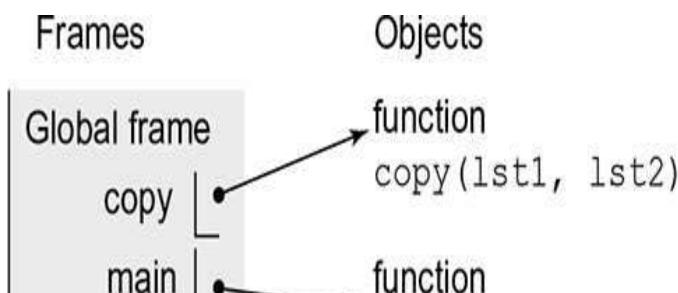
**Fig. 8.15** Program to create copy of a list (`copy.py`)

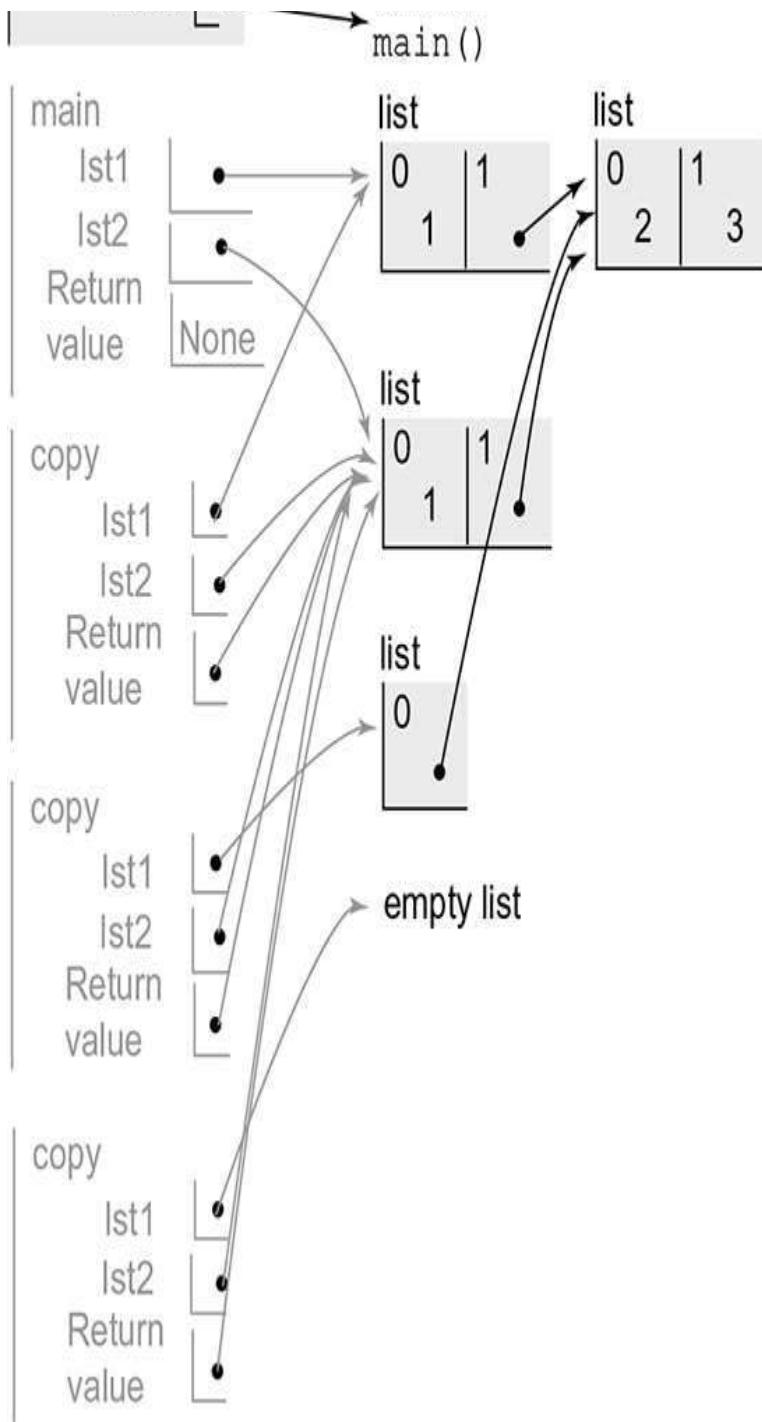
Let us examine the recursive function to create a copy of the list `[1, [2, 3]]`. As discussed before, a new stack frame will be allocated, and the variable bindings will be created, allocating storage space for new values. Also, the program remembers the point of call of the function, so as to return control back to the point of call when the function completes. Figure 8.16 shows the stack frames created and exited on execution of the script as visualized in Python Tutor with `[1, [2, 3]]` being the value of the list `lst1`.

### 8.3.3 Deep Copy

As mentioned earlier, in a shallow copy operation the embedded objects such as nested list are not copied, instead they are shared between the two lists. If we wish to copy the entire list, including embedded objects, such a copy is called deep copy.

deep copy : copying the entire list including embedded objects





**Fig. 8.16** Recursive calls to function `copy`

A deep copy of a list can be created in a manner similar to creating a shallow copy of the list. However, an element of list `lst1` is appended to list `lst2` only if it is not a list. In case the element itself is a list object, the `deepCopy` function is invoked for that element. The

trivial base case is the case when the list `lst1` is an empty list, in which case the resulting list `lst2` is also an empty list. However, if the list `lst1` is non-empty, a deep copy of the list is created. Based on this discussion, we present the complete program that accepts a list from the user, creates a deep copy of the list and prints it (Fig. 8.17). Figure 8.18 shows stack frames created and exited on execution of script in Fig. 8.17 as visualized in Python Tutor with `[1, [2, 3]]` as the value of the list `lst1`.

#### 8.3.4 Permutation

An arrangement of the elements of a list is called a permutation. For example, the list `[1, 2, 3]` will have the following six permutations:

```
[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3,
1], [3, 1, 2], [3, 2, 1]
```

```
01 def deepCopy(lst1, lst2 = []):
02 '''
03 Objective: To create deep copy of a list lst1
04 Input Parameters: lst1, lst2 - list
05 Return Value: lst2 - list
06 '''
07 if lst1==[]:
08 pass
09 else:
10 if type(lst1[0]) != list:
11 lst2.append(lst1[0])
12 else:
13 lst2.append([])
14 deepCopy(lst1[0], lst2[-1])
15 deepCopy(lst1[1:], lst2)
16 return lst2
17
```

```
18 def main():
19 """
20 Objective: To create deep copy of list entered by user
21 Input Parameter: None
22 Return Value: None
23 """
24 lst1 = eval(input('Enter the list: '))
25 lst2 = deepCopy(lst1)
26 print('Deep copy of list lst1 is ', lst2)
27
28 if __name__=='__main__':
29 main()
```

**Fig. 8.17** Program to create deep copy of a list (`deepCopy.py`)

In this section, we will develop a function to generate all permutations of a list. In general, a list with  $n$  elements will have  $n!$  permutations.

To understand the mechanism for generating the permutations, suppose we already have a permutation `perm` of two numbers: [1, 2]. To generate a permutation of three numbers {1, 2, 3} from this list, we apply the following procedure:

for each position `pos` in the list `perm`:

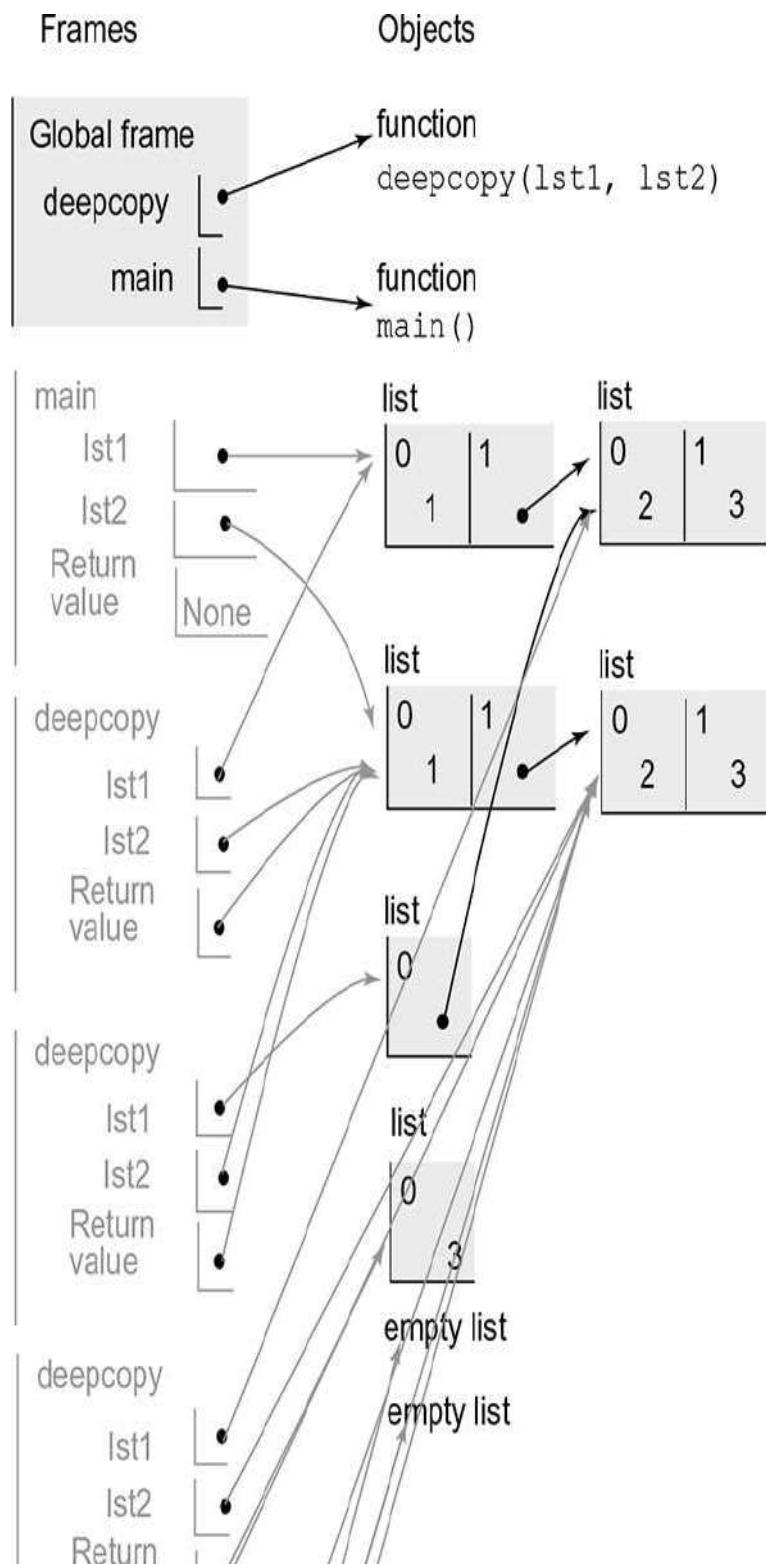
    insert 3 at `pos`

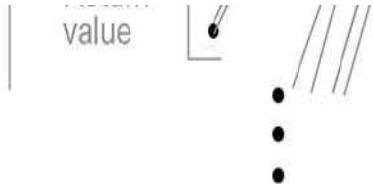
    print `perm`

    remove 3 from `perm` at `pos` (To get back the original permutation [1,2])

generating permutations of [1, 2, 3] from [1, 2] by inserting element 3 at different positions

Thus, on inserting 3 at position 0, 1, and 2, we get the permutations [3, 1, 2], [1, 3, 2], and [1, 2, 3], respectively.





**Fig. 8.18** Recursive calls to function `deepCopy`

A recursive function to generate permutations of a list is given in Fig. 8.19. The function is based on the assumption that the elements at index 0 through  $k-1$  of list `lst1` are already in the permutation list `lst2`. Thus, the problem at hand is to insert the element `lst1[k]` in the current permutation list, i.e. `lst2`, at every possible position one by one. On inserting `lst1[k]`, if the length of the permutation `lst2` becomes equal to `len(lst1)`, we print the permutation. Otherwise, we build up the permutation further by invoking the function `permute` once again, with the objective of inserting `lst1[k+1]` in the list `lst2`. Note that the element `lst1[k]` that was inserted at position `pos` in the permutation list `lst2` (line 10) is removed from there (line 19), so as to generate other possible permutations (line 23). Initially, the function `permute` is invoked with the list `lst1` as an argument. The other parameters are set to default values.

```

01 def permute(lst1, lst2 = [], k = 0, pos = 0):
02 """
03 Objective: To generate permutations of list lst1
04 Input Parameters: lst1, lst2 - list
05 k, pos - numeric value
06 Return Value: None
07 """
08
09 #To insert element lst1[k] at current position
10 lst2.insert(pos, lst1[k])
11
12 if len(lst2) == len(lst1):
13 print(lst2) #If a permutation is generated
14 else:

```

```

15 permute(lst1, lst2, k+1, 0) #To insert element lst1[k+1]
16
17 #To generate other permutations, remove
18 #element lst1[k] from current position
19 lst2.remove(lst1[k])
20
21 if pos < k:
22 #Put element lst1[k] at next available position
23 permute(lst1, lst2, k, pos+1)
24
25 def main():
26 """
27 Objective: To generate permutations of list entered by user
28 Input Parameter: None
29 Return Value: None
30 """
31
32 lst1 = eval(input('Enter the list: '))
33 permute(lst1)
34
35 if __name__=='__main__':
36 main()

```

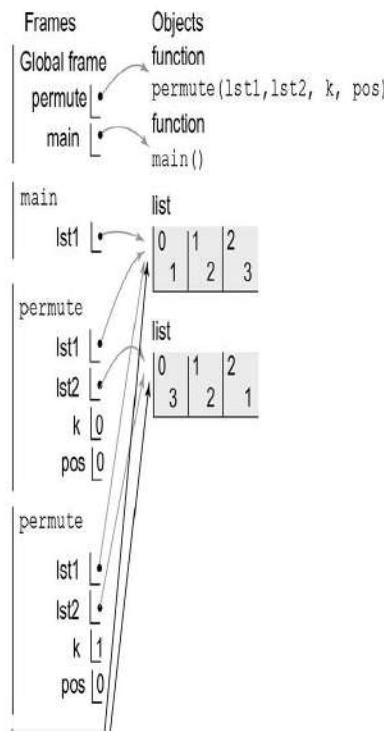
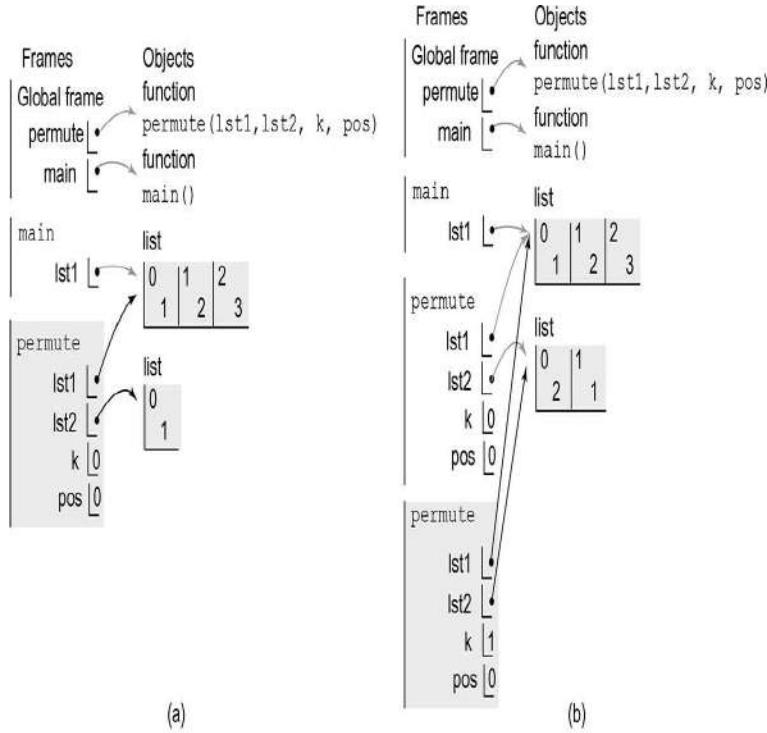
**Fig. 8.19** Program to create copy of a list (permute.py)

We have executed the script `permute` (Fig. 8.19) to generate permutations of the list [1, 2, 3]. In Fig. 8.20, we show the contents of the run-time stack as visualized in Python Tutor, for first three calls to the function `permute`.

#### 8.4 PROBLEM OF TOWER OF HANOI

In this section, we will discuss how to use recursion to solve the problem of Tower of Hanoi. First, let us describe the problem. There are three poles numbered 1, 2, and 3. In the first pole,  $n$  disks have been inserted. These disks are placed one above the other in a decreasing order of the diameter. The problem is to transfer all the disks from pole 1 to pole 3 using pole 2 as

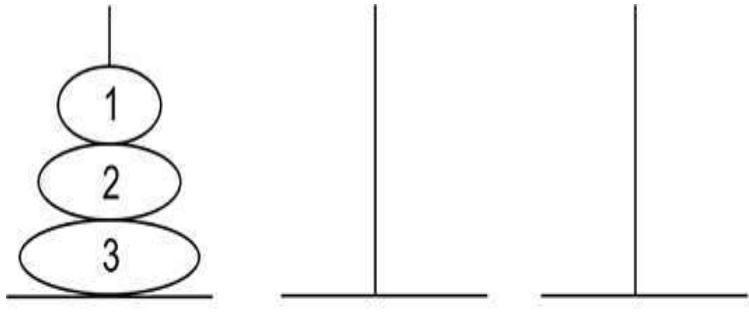
a spare pole, moving one disk at a time, and always keeping in mind that a disk with larger diameter is not to be placed over a disk with smaller diameter. See Fig. 8.21 for the case  $n = 3$ .



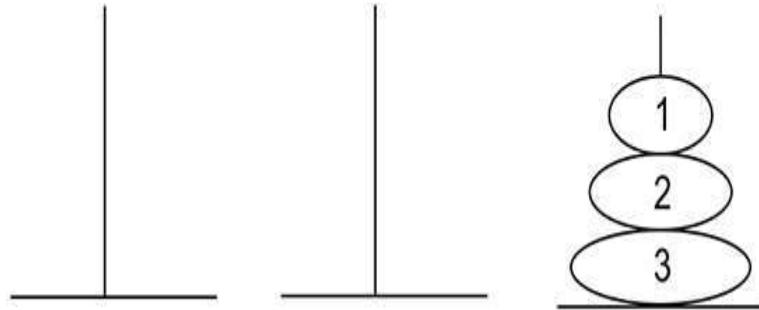
|         |    |
|---------|----|
| permute | // |
| lst1    | •  |
| lst2    | •  |
| k       | 2  |
| pos     | 0  |

(c)

**Fig. 8.20** Generating the first permutation using function  
permute



Initial configuration ( $n = 3$ )



Final configuration

**Fig. 8.21** Tower of Hanoi for  $n = 3$

Before solving the problem in general, let us consider some simple cases.

$$n = 1$$

The solution is trivial. Just move the disk from pole 1 to pole 3.

n = 2

Tower of Hanoi having one disk

Move the smaller disk at pole 1 to the spare pole 2. Now, the bigger disk at pole 1 can be moved to pole 3. Finally, move the smaller disk from the spare pole 2 to pole 3.  
This completes the task.

n = 3

Tower of Hanoi having two disks

When the largest disk is transferred to pole 3, it must be empty. Therefore, the first task we have to do is to transfer two disks to pole 2 using pole 3 as a spare pole. We have already seen above, how to transfer two disks from one pole to another using a spare pole. The only difference being that in the example above, pole 1, pole 2, and pole 3 were used as the source, spare, and destination poles, respectively, and currently pole 1, pole 3, and pole 2 are to be used as source, spare, and destination poles, respectively. The following steps will achieve this task:

Tower of Hanoi having three disks

Move a disk from pole 1 to pole 3

Move a disk from pole 1 to pole 2

Move a disk from pole 3 to pole 2

Now, the largest disk can be moved from pole 1 to pole 3.

Two disks in pole 2 are yet to be transferred to pole 3 using pole 1, which is currently empty as the spare pole. Since each disk in pole 2 is of a smaller diameter than the disk in pole 3, any of them can be shifted to pole 3. The following steps will achieve the transfer of two disks from pole 2 to pole 3.

Move a disk from pole 2 to pole 1

Move a disk from pole 2 to pole 3

Move a disk from pole 1 to pole 3

The task is now complete for  $n = 3$ .

In general, if there are  $n$  disks to be transferred from pole 1 to pole 3, using pole 2 as the spare pole, the task can be accomplished as follows:

Transfer  $(n - 1)$  disks from pole 1 to pole 2 using pole 3 as the spare pole.

Transfer one disk from pole 1 to pole 3

Transfer  $(n - 1)$  disks from pole 2 to pole 3 using pole 1 as the spare pole.

pseudo-code for solving Tower of Hanoi problem

Thus, a problem of size  $n$  has been expressed in terms of two problems of size  $(n - 1)$ , each of which can be solved using the same technique, i.e., by expressing each problem of size  $(n - 1)$  in terms of two problems of size  $(n - 2)$ , and so on until only one disk is to be transferred, which is a trivial task. These ideas have been coded in the form of function `hanoi` (Fig. 8.22).



```
01 def hanoi(n, source, spare, target):
02 """
03 Objective: To solve problem of Tower of Hanoi using n
04 discs and 3 poles
05 Input parameters: n, source, spare, target : numeric values
06 Return Value: None
07 """
08 if n==1:
09 print('Move a disk from', source, 'to', target)
10 elif n == 0:
11 return
12 else:
13 hanoi(n-1, source, target, spare)
14 print('Move a disk from', source, 'to', target)
15 hanoi(n-1, spare, source, target)
16
17 def main():
18 """
19 Objective: To solve Tower of Hanoi problem based on user input
20 Input Parameter: None
```

```
21 Return Value: None
22 """
23 n = int(input('Enter the number of discs: '))
24 source = int(input('Enter source pole: '))
```

```

25 spare = int(input('Enter spare pole: '))
26 target = int(input('Enter target pole: '))
27 hanoi(n, source, spare, target)
28
29 if __name__=='__main__':
30 main()

```

**Fig. 8.22** Program to solve problem of Hanoi(hanoi.py)

#### SUMMARY

1. Python allows a function to call itself, possibly with new parameter values. This technique is called recursion.
2. Recursion is a technique, which aims to solve a problem by providing a solution in terms of the smaller version(s) of the same problem. A recursive function definition comprises two parts. The first part is the base part which is the simplest version of the problem, often termed stopping or termination criterion. The second part is the inductive part which is the recursive part of the problem that reduces the complexity of the problem by defining a simpler version of the original problem itself.
3. Factorial of a number can be recursively expressed as follows:

$$\text{factorial}(n)=\begin{cases} 1 & \text{if } n == 0 \text{ or } 1 \\ n * \text{factorial}(n-1) & \text{if } n > 1 \end{cases}$$

4. nth term of the Fibonacci sequence can be recursively expressed as follows:

$$\text{fib}(n)=\begin{cases} 0 & \text{if } n==1 \\ 1 & \text{if } n==2 \\ \text{fib}(n-1)+\text{fib}(n-2) & \text{if } n>2 \end{cases}$$

5. Length of a string can be recursively expressed as follows:

$$\text{length}(\text{str1})=\begin{cases} 0 & \text{if } \text{str}==" \\ 1+\text{length}(\text{str}[1:]) & \text{otherwise} \end{cases}$$

6. Reverse of a string can be recursively expressed as follows:

$$\text{reverse}(\text{str1})=\begin{cases} 0 & \text{if } \text{str}==" \\ \text{str1}[-1]+\text{reverse}(\text{str}[:-1]) & \text{otherwise} \end{cases}$$

7. Checking whether a string is a palindrome can be recursively expressed as follows:

$$\text{isPalindrome}(\text{str1}) = \begin{cases} \text{True} & \text{if } \text{str1} == "" \\ \text{True} & \text{if } \text{str1}[0] == \text{str1}[-1] \\ & \text{and } \text{isPalindrome}(\text{str1}[1:-1]) \\ \text{False} & \text{otherwise} \end{cases}$$

8. The approach to flatten a list recursively can be expressed as follows:

```

for every element i in list lst1
 if i is not a list
 append it to list lst2
 otherwise,
 flatten list i

```

9. In Tower of Hanoi problem, if there are n disks to be transferred from pole 1 to pole 3, using pole 2 as the spare pole, the task can be accomplished as follows:

```

if n == 1
 transfer one disk from pole 1 to pole 3.
else
 transfer (n - 1) disks from pole 1 to pole 2 using pole 3 as
 the spare pole
 transfer one disk from pole 1 to pole 3
 transfer (n - 1) disks from pole 2 to pole 3 using pole 1 as
 the spare pole

```

#### EXERCISES

1. Write a recursive function that multiplies two positive numbers a and b, and returns the result.  
Multiplication is to be achieved as  $a + a + a$  (b times).
2. Write a recursive function that takes number n as an input parameter and prints n-digit strictly increasing numbers.
3. Write a recursive function that generates all binary strings of n-bit length.
4. Write a recursive function that takes two strings as input parameters and prints all interleaving strings of the given two strings preserving their order of occurrence. For example, interleaving of strings 'AB' and 'CD' will generate the strings: 'ABCD', 'ACBD', 'ACDB', 'CDAB', 'CADC' and 'CABD'.
5. Write a recursive function that inserts the element x at every kth position in the given list, and returns the modified list. For example, if we wish to insert element 50 at every 3rd position (counting 0, 1, 2, 3) in the list [1, 2, 3, 4, 5, 6, 7], the output list will be [1, 2, 3, 50, 4, 5, 6, 50, 7].
6. Write a recursive function that deletes every kth element, and returns the modified list. For example, if we wish to delete every 3rd element from the list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], the output list will be [1, 2, 4, 5, 7, 8, 10, 11].

7. Write a recursive function that recursively removes adjacent duplicates from a given list, and returns the modified list. For example, removing adjacent duplicates recursively from the list [1, 2, 4, 4, 5, 7, 7, 7, 8, 8, 9, 7] will yield list [1, 2, 5, 9, 7].
8. Write a recursive function that takes two numbers as input parameters and computes their greatest common divisor.
9. In Fig. 8.17, after line 25, include the following statement, execute the code again and justify the output:

```
lst2 = deepCopy(lst1)
```

# CHAPTER 9

## FILES AND EXCEPTIONS

### CHAPTER OUTLINE

9.1 File Handling

9.2 Writing Structures to a File

9.3 Errors and Exceptions

9.4 Handling Exceptions using try...except

9.5 File Processing Example

Programs that we have developed so far take data from the user in an interactive manner. Such data remain in memory only during the lifetime of the program. Often we want to store data permanently, in the form of files that usually reside on disks, so that it is available as and when required. For example, for login application, we would like to validate the user name and password entered by a user against the names and passwords of valid users stored permanently in a file. By default, Python provides a standard input file and a standard output file to deal with the transitory data. The standard input file is read from the keyboard, and the standard output file is displayed on the screen. Apart from these standard input/output files, we can also create files on disks that store data permanently for subsequent use. In this chapter, we will study file handling in detail. Moreover, programs necessarily have to deal with exception situations like unavailability of a file we want to read from or an error like division by zero. Therefore, in this chapter, we will also discuss how to deal with such situations smoothly.

files provide a means to store data permanently

## 9.1 FILE HANDLING

Files provide a means of communication between the program and the outside world. A file is nothing but a stream of bytes, comprising data of interest. Files can be used in any application where data is required to be stored permanently. Before performing a read or write operation in a file, we need to open the file. Built-in function `open()` is used for this purpose. This function takes the name of the file as the first argument. The second argument indicates the mode for accessing the file. A file may be opened in any of the three modes: `r` (read), `w` (write), and `a` (append). Read mode is used when an existing file is to be read. Write mode is used when a file is to be accessed for writing data in it. While opening a file in the read mode, if the specified file does not exist, it would lead to an error. As opposed to this, while opening a file in write mode, if the specified file does not exist, Python will create a new file. However, while opening a file in write mode, if the specified file already exists, this file gets overwritten. As the name suggests, append mode allows us to write into a file by appending contents at the end of the specified file. However, if the specified file does not exist, a new file is created. The absence of the second argument in the `open` function sets it to default value '`r`' (read mode). Successful execution of `open` function returns a file object. Failure in opening the file results in error. A file may be opened using the following syntax:

file is a stream of bytes

`open()` : to open a file

modes for opening a file:

`read(r)`: to read the file

`write(w)`: to write to the file

`append(a)`: to write at the end of the file

opening a non-existent file in `w` or `a` mode creates a new file with the given name

```
f = open(file_name, access_mode)
```

by default, a file is opened in read mode

In the following statement, the file object returned by the function `open` has been named `f` and we will be using this name for writing the contents into the file:

```
>>> f = open('PYTHON', 'w')
```

By default, the system creates a file in the default working directory. Instead of this relative path name, absolute (full) path name such as

`F:\PythonCode\Ch09` could also be specified. The functions `read` and `write` are used for reading from and writing into the file respectively. To use `read/write` function, we use the following notation: name of the file object, followed by the dot operator (`.`), followed by the name of the function. For example, in the following statement, we use `write()` function to write some text in the file `PYTHON`.

`read()`: to read a file

`write()`: to write into a file

```
>>> f.write('''Python:
```

Python is an interactive programming language.

Simple syntax of the language makes Python programs easy to read and write.''')

130

Note that apart from writing the given data into a file, the function `write` also returns the number of

characters written into the file (130 in the above example). Since the file is opened in write mode, Python disallows read operation on it by flagging an error as shown below:

write function returns the number of characters written in the file

reading a file opened for writing yields an error

```
>>> f.read()

Traceback (most recent call last):

 File "<pyshell#0>", line 1, in <module>

 f.read()

 io.UnsupportedOperation: not readable
```

When a file is no longer required for reading or writing, it should be closed by invoking the function `close` as shown below:

`close()`: saves and closes the opened file

```
>>> f.close()
```

The function `close` also saves a file, which was opened for writing. Once a file is closed, it cannot be read or written any further unless it is opened again and an attempt to access the file results in an I/O (input/output) error:

attempt to read/write a closed file results in an error

```
>>> f.write('Python was developed by Guido
Van Rossum in 1991.')

Traceback (most recent call last):

 File "<pyshell#21>", line 1, in <module>

 f.write('Python was developed by Guido
Van Rossum in 1991.')

ValueError: I/O operation on closed file.
```

To read the contents of the file `f`, we open the file again, but this time in read mode. Once the file has been opened in read mode, its contents can be read using `read` function:

```
>>> f = open('PYTHON', 'r')

>>> f.read()

'Python:\nPython is an interactive
programming language.\nSimple syntax of
the language makes Python programs easy to
read and write.'
```

```
>>> f.close()
```

While writing to the file, we entered a multiline string comprising three lines. When the file is read, the newline character marks the end of each line. Of course, we could have displayed the string using the `print` function as shown below:

```
>>> f = open('PYTHON', 'r')

>>> print(f.read())

Python:

Python is an interactive programming
language.
```

Simple syntax of the language makes Python programs easy to read and write.

```
>>> f.close()
```

As seen above, the `read()` function retrieves the contents of the entire file. More often, we need to read data in smaller chunks. As an alternative, we can read a fixed number of bytes from the file by specifying the number of bytes as the argument to `read` function. For example, reading only first 4 bytes will return `Pyth` as output.

function `read` retrieves the content of the entire file

reading fixed number of bytes

```
>>> f = open('PYTHON', 'r')
```

```
>>> f.tell()
```

0

```
>>> f.read(4)
```

'Pyth'

```
>>> f.tell()
```

4

We have used the function `tell` to know the current position while reading the file object `f`. Initially, the current position of file object `f` is set at 0, i.e., position of the first byte. A subsequent read operation will read the file `f` from the current position of the file object.

`tell()`: to find current position of the file object

```
>>> f.read(4)
```

```
'on:\n'
```

```
>>> f.tell()
```

```
9
```

Note that so far we have read only eight bytes from the input file. However, the function `f.tell()` shows 9 as the current position in the file. This is because the end of line character '`\n`' is stored by WINDOWS operating system as a pair of characters, namely, carriage return character (`CR`) to transfer control to beginning of line and line feed character (`LF`) to transfer control to the next line. However, if we are using a linux/UNIX® based systems, the function `tell` would return the position 8 as shown below:

`readline()`: to read one line at a time from the file

```
>>> f.tell()
```

```
4
```

```
>>> f.read(4)
```

```
'on:\n'
```

```
>>> f.tell()
```

```
8
```

Another way of reading from a file is reading one line at a time, Python function `readline()` is used for this purpose:

```
>>> f.readline()
```

```
'Python is an interactive programming
language.\n'
```

Note that the `readline` function reads a stream of bytes beginning the current position until a *newline* character is encountered. On reading a sequence of four bytes from the file twice, the current file position was set at the beginning of the second line. That is why, on invoking the function `readline`, the system returns the second line of text. Once we are done with reading this line, the next read operation will take place beginning the next line.

```
>>> f.readline()

'Simple syntax of the language makes
Python programs easy to read and write.'

>>> f.readline()

''
```

Note that, when all the contents of a file have been read, a further read operation on it will return a null string. One way to read from a particular position in a file is to use the function `seek()`. For example, for reading from the beginning of the file, we may invoke the `seek` function with the argument `0` as shown below:

reading the file that has been read entirely returns a null string

```
seek(): to reach desired position in a file

>>> f.seek(0)

0
```

The function `seek` returns the new absolute position. The function `readlines()` returns all the remaining lines of the file in the form of a list:

`readlines()`: reads and returns a list of lines read from a file

```
>>> f.readlines()

['Python:\n', 'Python is an interactive
programming language.\n', 'Simple syntax
of the language makes Python programs easy
to read and write.]
```

As each string in the list returned by the function `readlines` comprises one line, it terminates with the newline character. Just like the `readlines` function discussed above, the function `writelines` takes a list of lines to be written in the file as an argument.

```
writelines(): writes a list of lines to a file

>>> f = open('PYTHON', 'w')

>>> description = ['Python:\n', 'Python is
an interactive programming language.\n']

>>> f.writelines(description)

>>> f.close()

>>> f = open('PYTHON', 'r')

>>> f.read()

'Python:\nPython is an interactive
programming language.\n'

>>> f.close()
```

Suppose if we wish to copy the contents of a text file, say, `PYTHON` in another file `PythonCopy`. For this purpose, we open the source file `PYTHON` in read mode and the output file `PythonCopy` (yet to be created) in write mode, read the data from the file `PYTHON`, and write it into the file `PythonCopy` as follows:

```
>>> f1 = open('PYTHON', 'r')
```

```
>>> f2 = open('PythonCopy', 'w')

>>> data = f1.read()

>>> f2.write(data)

55

>>> f1.close()

>>> f2.close()
```

Note that if an application requires the file to be opened in both read and write mode, 'r+' mode can be used while opening it. If we wish to put the new content at the end of previously existing contents in a file, we need to open the file in append ('a') mode as shown below:

use r+ mode for opening a file both for reading and writing

opening a file in write mode deletes its previously existing contents

```
>>> f = open('PYTHON', 'a')

>>> f.write('Simple syntax of the language
')
```

31

```
>>> f.write('makes Python programs easy to
read and write')
```

44

```
>>> f.close()

>>> f = open('PYTHON', 'r')

>>> f.read()
```

```
'Python:\nPython is an interactive\nprogramming language.\nSimple syntax of\nthe language makes Python programs easy to\nread and write'
```

```
>>> f.close()
```

To ensure that the contents written to a file have been saved on the disk, it must be closed. While we are still working on a file, its contents may be saved anytime using the `flush` function.

#### 9.2 WRITING STRUCTURES TO A FILE

If we wish to write a structure such as a list or a dictionary to a file and read it subsequently, we may use the Python module `pickle`. Pickling refers to the process of converting the structure to a byte stream before writing to the file. While reading the contents of the file, a reverse process called unpickling is used to convert the byte stream back to the original structure. In the script `fileStructure` (Fig. 9.1), we will demonstrate how to write a list and a dictionary to a file. To begin with, we import the module `pickle`. In lines 9 and 10, we write a list and a dictionary respectively to file `f`, using the function `dump` of the `pickle` module which performs pickling. This function takes two arguments: the first argument is the structure to be written to the file, and the second argument is a file object. In lines 14 and 15, we read a list and a dictionary from the file using the `load` function of the `pickle` module that performs unpickling. This function takes file object as an argument and returns an object such as a list or a dictionary. In Python 3.x versions, binary mode is used for reading or writing pickled data. So, a file that involves reading or writing pickled data should be opened in binary mode by specifying `b` as the mode for opening it.

```
01 import pickle
02 def main():
03 """
04 Objective: To write and read a list and dictionary to
05 and from a file
06 Input Parameter: None
07 Return Value: None
08 """
09 f = open('file1', 'wb')
10 pickle.dump(['hello', 'world'], f)
11 pickle.dump({1:'one', 2:'two'}, f)
12 f.close()
13
14 f = open('file1', 'rb')
15 value1 = pickle.load(f)
16 value2 = pickle.load(f)
17 print(value1, value2)
18 f.close()
19
20 if __name__ == '__main__':
21 main()
```

**Fig. 9.1** Program to read and write a structure to the file  
(fileStructure.py)

`dump()`: to convert a structure to byte stream and write it to a file

`load()`: to read a byte stream from a file and convert it back to the original structure

On executing script in Fig. 9.1, Python responds with the following output:

```
['hello', 'world'] {1: 'one', 2: 'two'}
```

### 9.3 ERRORS AND EXCEPTIONS

Errors occur when something goes wrong. We have encountered several errors in our program such as index out of bound, division by zero, and invalid cast operation. The errors in Python programming may be categorized as syntax errors and exceptions. A syntax error occurs when a rule of Python grammar is violated, for example, a syntax error occurs in the following call to `print` function because of missing quote mark at the end of the string. Similarly, missing colon in the `for` statement results in a syntax error.

```
syntax error: violation of Python grammar rule
>>> print('Hello)
SyntaxError: EOL while scanning string
literal

>>> for i in range(0, 10)
SyntaxError: invalid syntax
```

Another common error in Python programming is caused by incorrect indentation. Often it leads to a syntax error, but incorrect indentation may sometimes

lead to a logical error that alters the intended meaning of the program that is difficult to locate.

#### Indentation error

In contrast to a syntax error, an exception occurs because of some mistake in the program that the system cannot detect before executing the code as there is nothing wrong in terms of the syntax, but leads to a situation during the program execution that the system cannot handle. For example, opening a non-existent file for reading, attempting to access value of a variable without assigning a value to it, and dividing a number by zero. These errors disrupt the flow of the program at a run-time by terminating the execution at the point of occurrence of the error. We have noticed that whenever an exception occurs, a `Traceback` object is displayed which includes error name, its description, and the point of occurrence of the error such as line number. Now we describe some commonly encountered exceptions:

exceptions: errors that occur at execution time

##### 1. `NameError`

This exception occurs whenever a name that appears in a statement is not found globally. For example, in the following statement, we intend to take `marks` as an input from the user. For doing so, we intended to use function `input` but instead typed `Input`. Python being case-sensitive fails to recognize the function `input` and the system responds with the error message `NameError: name 'Input' is not defined`. This message begins with the name of the exception. Note that the following `Traceback` object describes that error occurred in line 1 in Python shell, in the most recent call:

```
name not found globally

>>> marks = Input('Enter your marks')
Traceback (most recent call last):
 File "<pyshell#0>", line 1, in <module>
 marks = Input('Enter your marks')
NameError: name 'Input' is not defined
```

Note that the above error could not have been detected as a syntax error as it is perfectly fine to define a function `Input` as shown below:

```
>>> def Input():
 return input('Enter your marks: ')
>>> marks = Input()
Enter your marks: 78
>>> marks
'78'
```

Similarly, accessing the value of a variable before defining it raises an exception:

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

## 2. `TypeError`

This exception occurs when an operation or function is applied to an object of inappropriate type. For example, the expression '`sum of 2 and 3 is`' + 5 involves adding a number to a string which is not a valid operation resulting in an exception.

Invalid type of operands for the operation

```
>>> 'sum of 2 and 3 is ' + 5
Traceback (most recent call last):
 File "<pyshell#12>", line 1, in <module>
 'sum of 2 and 3 is ' + 5
TypeError: must be str, not int
```

## 3. `ValueError`

This exception occurs whenever an inappropriate argument value, even though of correct type, is used in a function call, for example:

Invalid argument value

```
>>> int('Hello')
Traceback (most recent call last):
 File "<pyshell#5>", line 1, in <module>
 int('Hello')
ValueError: invalid literal for int() with
base 10: 'Hello'
```

Note that in the above example, the function `int` has been invoked with a valid type of argument, i.e. `str`, but its value, i.e., 'Hello', cannot be converted to an integer. Hence the `ValueError`.

## 4. `ZeroDivisionError`

This exception occurs when we try to perform numeric division in which the denominator happens to be zero, for example:

attempt to divide by zero

```
>>> 78 / (2+3-5)
```

```
Traceback (most recent call last):
 File "<pyshell#13>", line 1, in <module>
 78/(2+3-5)
ZeroDivisionError: division by zero
```

#### 5. OSError

This exception occurs whenever there is a system related error such as disk full or an error related to input/output, for example, opening a non-existent file for reading or reading a file opened in write mode:

```
system related error
```

```
>>> f = open('passwordFile.txt')
Traceback (most recent call last):
 File "<pyshell#9>", line 1, in <module>
 f = open('passwordFile.txt')
FileNotFoundException: [Errno 2] No such file or
directory: 'passwordFile.txt'
```

#### 6. IndexError

This exception occurs whenever we try to access an index that is out of a valid range. For example, let us name the list of colors ['red', 'green', 'blue'], as colors. Now the valid range of indexes for colors is [-3, -2, -1, 0, 1, 2] and the valid index range of indexes for the string colors[2] is [-4, -3, -2, -1, 0, 1, 2, 3]. Accessing an index outside a valid range will cause IndexError exception to occur:

```
accessing an invalid index
```

```
>>> colors = ['red', 'green', 'blue']
>>> colors[4]
Traceback (most recent call last):
 File "<pyshell#10>", line 1, in <module>
 colors[4]
IndexError: list index out of range
>>> colors[2][4]
Traceback (most recent call last):
 File "<pyshell#11>", line 1, in <module>
 colors[2][4]
IndexError: string index out of range
```

### 9.4 HANDLING EXCEPTIONS USING TRY...EXCEPT

We have seen several examples of exceptions that would result in abrupt termination of program execution. Such exceptions are known as unhandled exceptions. To prevent a program from terminating abruptly when an exception is raised, we need to handle it by catching it and taking appropriate action using the `try...except` clause.

Whereas a `try` block comprises statements that have the potential to raise an exception, `except` block

describes the action to be taken when an exception is raised. In the `except` clause, we may specify a list of exceptions and the common action to be taken on occurrence of any of these exceptions. Alternatively, a specific action may be provided for each of the exceptions separately. We can also specify a `finally` block in the `try...except` clause, which is executed irrespective of whether an exception is raised.

`try` block: statements having potential to raise an exception

`except` block: action to be performed when exception is raised

In the script `try_except1` (Fig. 9.2), line 10 (within `except` block) will be executed only when any input–output related error is raised in line 8 that lies within the `try` block. While executing this script, an attempt to open a non-existent file will cause the `IOError` exception and will yield the following output:

`finally` block: executed irrespective of whether an exception is raised

```
01 def main():
02 """
03 Objective: To open a file for reading
04 Input Parameter: None
05 Return Value: None
06 """
07 try:
08 f = open('Temporary_File' , 'r')
09 except IOError:
10 print('Problem with Input Output... ')
11 print('Program continues smoothly beyond try...except block')
12
13 if __name__ == '__main__':
14 main()
```

**Fig. 9.2** Program to open non-existent file (`try_except1.py`)

Problem with Input Output...

Program continues smoothly beyond  
try...except block

Note that as `IOError` exception raised in line 8 had been handled (lines 9–10), the program did not terminate abruptly and continued smoothly to execute line 11 after the exception was handled.

We can access the description of the Traceback notice directly. For this purpose, the name of the exception such as `IOError` is followed by the keyword `as`, which is followed by a user-defined name such as `err` in the `except` clause (Fig. 9.3). So, when `IOError`

exception is raised during execution of the `open` function (line 9), the exception information is assigned to the variable `err` and the message [Errno 2] No such file or directory: 'Temporary\_File' is displayed (line 11). It is also possible to track the exception raised by Python using the expression `sys.exc_info()`, which yield the details of the exception as a tuple comprising the type of exception, and description of the exception, a reference to the exception object, for example, on executing the script `try_except2` (Fig. 9.3), the system responds as follows:

tracking raised exception

```
01 import sys
02 def main():
03 """
04 Objective: To open a file for reading
05 Input Parameter: None
06 Return Value: None
07 """
08 try:
09 f = open('Temporary_File', 'r')
10 except IOError as err:
11 print('Problem with Input Output...\\n', err)
12 print(sys.exc_info())
13 print('Program continues smoothly beyond try...except block')
14
15 if __name__ == '__main__':
16 main()
```

**Fig. 9.3** Program to open non-existent file (try\_except2.py)

Problem with Input Output...

```
[Errno 2] No such file or directory:
'Temporary_File'

(<class 'FileNotFoundException',
FileNotFoundException(2, 'No such file or
directory'), <traceback object at
0x02EB2530>)
```

Program continues smoothly beyond  
try...except block

There may be situations when a sequence of statements that appear in a `try` block can potentially throw more than one exceptions. We can deal with such situations in different ways: the first option is to enumerate all possible exceptions in the `except` clause in the form of a list, the second option is to define an `except` clause for handling each exception separately, and yet another option is to go for a defensive approach by placing the targeted group of statements in `try` block and specify an empty `except` clause. Such an `except` clause is capable of catching all possible errors.

an empty `except` clause catches all possible exceptions

In the script `pricePerUnit1` (Fig. 9.4), we wish to compute the price per unit weight of an item. For this purpose, we define a function `main` that takes `price` and `weight` as inputs and computes the price per unit. Note that `ValueError` and `TypeError` exceptions may be raised while converting an argument to a `float` value (lines 12 and 14) and `ZeroDivisionError` exception may be raised while dividing `price` by `weight` (line 16). For handling these exceptions, we include all of them in the `except` clause (line 17). An optional `else` clause (line 20) may also be specified in the `try...except` clause, which will be executed only if no exception is raised.

```
01 import sys
02 def main():
03 """
04 Objective: To compute price per unit weight of an item
05 Input Parameter: None
06 Return Value: None
07 """
```

```

08 price = input('Enter price of item purchased: ')
09 weight = input('Enter weight of item purchased: ')
10 try:
11 if price == '': price = None
12 price = float(price)
13 if weight == '': weight = None
14 weight = float(weight)
15 assert price >= 0 and weight >= 0
16 result = price / weight
17 except (ValueError, TypeError, ZeroDivisionError):
18 print('Invalid input provided by user \n' + \
19 str(sys.exc_info()))
20 else:
21 print('Price per unit weight: ', result)
22
23 if __name__ == '__main__':
24 main()

```

**Fig. 9.4** Program to compute price per unit weight of an item  
(pricePerUnit1.py)

On executing the script `pricePerUnit1` (Fig. 9.4), the system prompts the user to enter values of `price` and `weight`:

>>>

Enter price of item purchased: 400

Enter weight of item purchased: 0

```
Invalid input provided by user

(<class 'ZeroDivisionError'>,
ZeroDivisionError('float division by
zero'), <traceback object at 0x039BD2B0>)
```

As we have specified a common action to be taken on each of the exceptions `ValueError`, `TypeError`, and `ZeroDivisionError`, we could have replaced the `except` clause:

```
common actions for the exceptions

except (ValueError, TypeError,
ZeroDivisionError):
```

by the empty `except` clause:

```
except:
```

However, it is often preferable to handle different exceptions separately as shown in the script `pricePerUnit2` (Fig. 9.5). Also, we make use of the `assert` statement in line 13 to ensure that `price` and `weight` are positive values. An empty `except` clause is used (line 21) to catch all other exceptions (for example, invalid range).

It is preferable to handle different exceptions separately

```
01 import sys
02 def main():
03 """
04 Objective: To compute price per unit weight of an item
05 Input Parameter: None
06 Return Value: None
```

```

07 """
08 price = input('Enter price of item purchased: ')
09 weight = input('Enter weight of item purchased: ')
10 try:
11 if price == '': price = None
12 price = float(price)
13 if weight == '': weight = None
14 weight = float(weight)
15 assert price >= 0 and weight >= 0
16 result = price / weight
17 except ValueError:
18 print('Invalid inputs: ValueError')
19 except TypeError:
20 print('Invalid inputs: TypeError')
21 except ZeroDivisionError:
22 print('Invalid inputs: ZeroDivisionError')
23 except:
24 print(str(sys.exc_info()))
25 else:
26 print('Price per unit weight:', result)
27
28 if __name__ == '__main__':
29 main()

```

**Fig. 9.5** Program to compute price per unit weight of an item  
(pricePerUnit2.py)

Next, we present the system response on execution of the script `pricePerUnit2.py` on different inputs:

Enter price of item purchased: 20

```
Enter weight of item purchased: x
Invalid inputs: ValueError

Enter price of item purchased: -20

Enter weight of item purchased: 10
(<class 'AssertionError'>,
AssertionError(), <traceback object at
0x0356D328>

Enter price of item purchased: 20

Enter weight of item purchased: 0
Invalid inputs: (ZeroDivisionError)

Enter price of item purchased: 20

Enter weight of item purchased:
Invalid inputs: TypeError

Enter price of item purchased: -20

Enter weight of item purchased: 0
(<class 'AssertionError'>,
AssertionError(), <traceback object at
0x03B42508>)
```

When we enter the inputs -20 and 0, each of the assert statement (line 15) and division operation (line 16) can raise exceptions. However, as the assert statement is encountered before the division operation, in the try...except clause, AssertionError exception is raised. As none of the exceptions specified in lines 17, 19, and 21 matches the AssertionError exception, the empty except block (lines 23–24) got executed. Thus, line 16 got skipped.

A segment of the script that handles an exception is called a handler. In the above script, the exceptions were

raised and handled within a `try...except` block. However, if an exception is raised in a `try` block, but the script does not include the corresponding handler in that block, then search for a handler is continued in the outer `try...except` block, and so on. We illustrate this in the script `pricePerUnit3` (Fig. 9.6) – a slightly modified version of the script `pricePerUnit2` (Fig. 9.5). In this version, each of the lines 13 and 18 is placed under a separate `try...except` block, which catches only `ValueError` exception. So, when `TypeError` is raised in line 13 or line 18, Python does not find any matching `except` clause in the inner `try...except` block and the search for the corresponding handler continues in the outer `try...except` block, where we find the required handler and the function `main` prints the string

`Invalid inputs: TypeError'.`

If there is no handler for the exception raised in try block,  
search for handler takes place in outer try...except block

```
01 import sys
02 def main():
03 """
04 Objective: To compute price per unit weight of an item
05 Input Parameter: None
06 Return Value: None
07 """
08 price = input('Enter price of item purchased: ')
09 weight = input('Enter weight of item purchased: ')
10 try:
11 if price == '': price = None
12 try:
13 price = float(price)
14 except ValueError:
```

```
15 print('Invalid inputs: ValueError')
16 if weight == '': weight = None
17 try:
18 weight = float(weight)
19 except ValueError:
20 print('Invalid inputs: ValueError')
21 assert price >= 0 and weight >= 0
22 result = price / weight
23 except TypeError:
24 print('Invalid inputs: TypeError')
```

```

21 print('Invalid inputs. Try again')
22
23 except ZeroDivisionError:
24 print('Invalid inputs: ZeroDivisionError')
25
26 except:
27 print(str(sys.exc_info()))
28
29 else:
30 print('Price per unit weight: ', result)
31
32 if __name__ == '__main__':
33 main()

```

**Fig. 9.6** Program to compute price per unit weight of an item  
(pricePerUnit3.py)

If we wish to execute some action(s) irrespective of whether an exception is raised, such action(s) may be specified in the `finally` clause. Further, an exception may be raised explicitly using the keyword `raise`, followed by the name of the exception, followed by the string (enclosed in parentheses) to be displayed when the exception is raised. For example, in the script `raiseFinally1` (Fig. 9.7, line10), if the value of the variable `marks` is out of the valid range, i.e. `marks < 0` or `marks > 100`, the exception `ValueError('Marks out of range')` is thrown. As the exception is not handled using an `except` clause, the program terminates by throwing the unhandled exception on invoking the `print` function in the `finally` clause.

use `raise` keyword for explicitly raising an exception

an unhandled exception leads to program termination

>>>

Bye

Traceback (most recent call last):

File "F:/PythonCode/Ch09/except.py", line  
16

    main()

File "F:/PythonCode/Ch09/except.py", line  
10, in main

    raise ValueError('Marks out of range')

ValueError: Marks out of range

```
01 def main():
02 """
03 Objective: To illustrate the use of raise and finally clauses
04 Input Parameter: None
05 Return Value: None
06 """
07 marks = 110
08 try:
09 if marks < 0 or marks > 100:
10 raise ValueError('Marks out of range')
11 finally:
12 print('Bye')
13 print('Program continues after handling exception')
14
15 if __name__ == '__main__':
16 main()
```

**Fig. 9.7** To illustrate the use of raise and finally clauses  
(raiseFinally1.py)

Next, we modify the script `raiseFinally1` by including an `except` clause (Fig. 9.8) so that the program continues smoothly after the exception is raised.

>>>

Bye

Program continues after handling exception

```
01 def main():
02 """
03 Objective: To illustrate the use of raise and finally clauses
04 Input Parameter: None
05 Return Value: None
06 """
07 marks = 110
08 try:
09 if marks < 0 or marks > 100:
10 raise ValueError('Marks out of range')
```

```
11 except:
12 pass
13 finally:
14 print('Bye')
15 print('Program continues after handling exception')
16
17 if __name__ == '__main__':
18 main()
```

**Fig. 9.8** To illustrate the use of raise and finally clauses  
(raiseFinally2.py)

#### 9.5 FILE PROCESSING EXAMPLE

In this section, we will develop a program to moderate the results in an examination. We are given a file named

`studentMarks`. This file contains the student data that includes roll number (`rollNo`), name (`name`), and marks (`marks`) for each student. The data about each student is stored in a separate line and the individual pieces of information `rollNo`, `name`, and `marks` are separated by commas. Further, data about each student is stored in a separate line. Sample data in the file is shown below:

```
sample data for file studentMarks
```

```
4001,Nitin Negi,75
```

```
4002,Kishalaya Sen,98
```

```
4003,Kunal Dua,80
```

```
4004,Prashant Sharma,60
```

```
4005,Saurav Sharma,88
```

We define `addPerCent` as the percentage of `maxMarks` that should be added to the marks obtained to get the moderated marks, subject to the upper limit of `maxMarks`. To carry out the moderation, we prompt the user to enter the moderation percentage (`addPerCent`) and produce another file `moderatedMarks` containing moderated marks of the students. Next, we describe this task in the form of a pseudocode (Fig. 9.9).

```
1. Open file studentMarks in read mode.
2. Open file moderatedMarks in write mode.
3. Read one line of input (line1) from studentMarks.
4. while (line1 != ''):
 • Compute moderated marks and write one line of output in the file
 moderatedMarks.
 • Read one line of input (line1) from studentMarks.
```

**Fig. 9.9** Pseudocode for producing file moderatedMarks

As we need to ensure throughout the program that the file operations are being performed smoothly without any errors, we must perform checks on error conditions. The revised pseudocode is shown in Fig. 9.10.

check for possible exceptions

```
1. Open file studentMarks in read mode while checking for any errors.
2. Open file moderatedMarks in write mode while checking for any errors.
3. Read one line of input (line1) from studentMarks.
4. while (line != ''):
 • Retrieve the values of rollNo, name, and marks from the line1
 while checking for any errors.
 • Compute moderated marks and write one line of output in the file
 moderatedMarks.
 • Read one line of input (line1) from studentMarks.
```

**Fig. 9.10** Pseudocode for producing file moderatedMarks

The complete script is given in Fig. 9.11. When we execute the program in Fig. 9.11 and enter 3 as value of addPercent, the system outputs the contents of the file moderatedMarks as follows:

```
content of file moderatedMarks
4001,Nitin Negi,78.0
4002,Kishalaya Sen,100
4003,Kunal Dua,83.0
4004,Prashant Sharma,63.0
4005,Saurav Sharma,91.0
```

```
01 import sys
02 def computeModeratedMarks(file1, file2, addPercent):
03 """
04 Objective: To compute moderated marks of students
05 Input Parameters: file1, file2: file names - string values
06 addPercent - numeric value
07 Return Value: None
08 Side effect: A new file - file2 of moderated marks is produced
09 """
10 try:
11 fIn = open(file1, 'r')
12 fOut = open(file2, 'w')
13 except IOError:
14 print('Problem in opening the file'); sys.exit()
15 line1 = fIn.readline()
16 while(line1 != ''):
17 sList = line1.split(',')
18 try:
```

```
19 rollNo = int(sList[0])
20 name = sList[1]
21 marks = int(sList[2])
22 except IndexError:
23 print('Undefined Index'); sys.exit()
24 except (ValueError):
25 print('Unsuccessful conversion to int'); sys.exit()
```

```

26 maxMarks= 100
27 moderatedMarks = marks+((addPercent*maxMarks) /100)
28 if moderatedMarks > 100:
29 moderatedMarks = 100
30 fOut.write(str(rollNo) + ',' + name + ',' + \
31 str(moderatedMarks) + '\n')
32 line1 = fIn.readline()
33
34 fIn.close()
35 fOut.close()
36
37 def main():
38 """
39 Objective: To compute moderated marks based on user input
40 Input Parameter: None
41 Return Value: None
42 """
43
44 import sys
45 sys.path.append('F:\PythonCode\Ch09')
46 # To compute moderated marks of students
47 file1 = input('Enter name of file containing marks:')
48 file2 = input('Enter output file for moderated marks:')
49 addPercent = int(input('Enter moderation percentage:'))
50 computeModeratedMarks(file1, file2, addPercent)
51
52 if __name__=='__main__':
53 main()

```

**Fig. 9.11** Program to compute moderated marks  
(moderatedMarks.py)

In our next program, we wish to compute monthly wages to be paid to employees in an organization. The input data has been provided to us in two files named `empMaster` and `empMonthly`. The first file `empMaster` contains permanent data about employees (also called master data) that include employee id (`empID`),

employee name (`empName`), and hourly wages (`hrlyWages`). Individual elements of the data are separated by commas. Further, the data about each employee is stored in a separate line. Sample data in the file is shown below:

sample data for file `empMaster`

1001,Vinay Kumar,30

1002,Rohit Sen,35

1003,Vinita Sharma,28

1004,Bijoy Dutta,35

The second file `empMonthly` contains monthly information (often called transaction data) about employees. It stores two pieces of information about each employee, namely, employee id (`tEmpID`) and the number of hours worked (`hrsWorked`). Data in this file is also comma separated. Data about each employee is stored in a separate line. We assume that information about the employees is stored in each file in the ascending order of employee id. Further, we assume that the file `empMonthly` contains information about each employee even if he/she worked for zero hours. Sample data in this file is shown below:

sample data for file `empMonthly`

1001,245

1002,0

1003,0

1004,240

To compute the monthly wages of all employees, we need to read the files `employeeMaster` and `empMonthly` and produce a third file `monthlyWages` containing monthly wages of the employees. We describe this task in the form of a pseudocode (Fig. 9.12).

```
1. Open files empMaster and empMonthly in read mode.
2. Open file monthlyWages in write mode.
3. Read one line of input (line1) from empMaster.
4. while (line1 != ''):
 • Read one line of input (line2) from empMonthly.
 • Check that empID in empMaster and tEmpID in the empMonthly
 match.
 • Compute monthly wages and write one line of output in file
 monthlyWages.
 • Read one line of input (line1) from empMaster.
```

**Fig. 9.12** Pseudocode to compute contents for file `monthlyWages`

As we need to ensure throughout the program that file operations are being performed smoothly without any errors, we must perform checks on error conditions. The revised pseudocode is shown in Fig. 9.13.

check for possible exceptions

The complete script is given in Fig. 9.14. Instead of invoking the `exit` function in line 13, we could have executed a `return` statement to return the control to `main` function and ask the user to enter the name of the file again. On the execution of the program in Fig. 9.14,

the system creates an output file monthlyWages with the following content:

1001, 7350

1002, 0

1003, 0

1004, 8400

1. Open files empMaster and empMonthly in read mode while checking for any errors.
2. Open file monthlyWages in write mode while checking for any errors.
3. Read one line of input (line1) from empMaster.
4. while (line1 != ''):
  - Retrieve values of empID and hrlyWages from the line1 while checking for any errors.
  - Read one line of input (line2) from empMonthly.
  - Retrieve the values of tEmpID and hrsWorked from the line2 while checking for any errors.
  - Check that empID in empMaster and tEmpID in the empMonthly match.
  - Compute monthly wages and write one line of output in file monthlyWages.
  - Read one line of input (line1) from empMaster.

**Fig. 9.13** Pseudocode to compute contents for file monthlyWages

```
01 import sys
02 def generateSalary(file1, file2, file3):
```

```
03 ''
04 Objective: To compute monthly wages of all employees
05 Input Parameters: file1, file2, file3 - string value
06 Return Value: None
07 ''
08 try:
09 fMaster = open(file1, 'r')
10 fTrans = open(file2, 'r')
11 fWages = open(file3,'w')
12 except IOError:
13 print('Problem with opening the file');sys.exit()
14 line1 = fMaster.readline()
15 while((line1) != ''):
16 sList1 = line1.split(',')
17 try:
18 empID = int(sList1[0])
19 hrlyWages = int(sList1[2])
20 except IndexError:
21 print('Undefined Index');sys.exit()
22 except (ValueError, TypeError):
23 print('Unsuccessful conversion to int'); sys.exit()
24 line2 = fTrans.readline()
25 sList2 = line2.split(',')
26 try:
27 tEmpID = int(sList2[0])
```

```
28 hrsWorked = int(sList2[1])
```

```

29 except IndexError:
30 print('Undefined Index'); sys.exit()
31 except (ValueError, TypeError):
32 print('Unsuccessful conversion to int'); sys.exit()
33 if empID == tEmpID:
34 fWages.write(str(empID)+','+str(hrlyWages*\
35 hrsWorked)+'\n')
36 line1 = fMaster.readline()
37 fMaster.close()
38 fTrans.close()
39 fWages.close()
40 def main():
41 """
42 Objective: To determine monthly wages of all employees based on
43 user input
44 Input Parameter: None
45 Return Value: None
46 """
47 import sys
48 sys.path.append('F:\PythonCode\Ch09')
49 file1 = input('Enter name of file containing hourly rate:')
50 file2 = input('Enter name of file containing hours worked:')
51 file3 = input('Enter output file for salary generation:')
52 generateSalary(file1,file2,file3)
53 if __name__=='__main__':
54 main()

```

**Fig. 9.14** Program to generate salary (salaryGen.py)

In the above program, we assume that the information about each employee in `empMaster` is available in `empMonthly`. Suppose, the file `empMonthly` does not contain an entry for employees for whom `hrlyWorked` is equal to 0. Sample data in this file is shown below:

1001,245

1004,240

Next, we present a pseudocode for computing monthly wages of all employees (Fig. 9.15).

1. Open files empMaster and empMonthly in read mode while checking for any errors.
2. Open file monthlyWages in write mode while checking for any errors.

3. Read one line of input (line1) from empMaster.
4. while (line1 != ''):
  - Retrieve the values of empID and hrlyWages from line1 while checking for any errors.
  - Read one line of input (line2) from empMonthly.
  - If (line2 != ''):
    - Retrieve the values of tEmpID and hrsWorked from line2 while checking for any errors.
    - while (empID in empMaster != tEmpID in empMonthly):
      - Compute monthly wages and write as one line of output in file monthlyWages assuming hrsWorked = 0 for empID.
      - Read one line of input (line1) from empMaster and retrieve the values of empID and hrlyWages while checking for any error.
      - Compute monthly wages and write as one line of output in file monthlyWages.
    - else:
      - Compute monthly wages and write as one line of output in file monthlyWages assuming hrsWorked = 0 for empID.
  - Read one line of input (line1) from empMaster.

**Fig. 9.15** Pseudocode to compute contents for file monthlyWages

The complete script is given in Fig. 9.16. On the execution of the above program, the system outputs the contents of the file monthlyWages as follows:

1001, 7350

1002, 0

1003, 0

1004, 8400

```
01 import sys
02 def generateSalary(file1, file2, file3):
03 """
04 Objective: To compute monthly wages of all employees
05 Input Parameters: file1, file2, file3 - string value
06 Return Value: None
07 """
08 try:
09 fMaster = open(file1, 'r')
10 fTrans = open(file2, 'r')
```



```
11 fWages = open(file3,'w')
12 except IOError:
13 print('Problem with opening the file'); sys.exit()
14 line1 = fMaster.readline()
15 while((line1) != ''):
16 sList1 = line1.split(',')
17 try:
18 empID = int(sList1[0])
19 hrlyWages = int(sList1[2])
20 except IndexError:
21 print('Undefined Index'); sys.exit()
22 except (ValueError, TypeError):
23 print('Unsuccessful conversion to int'); sys.exit()
24 line2 = fTrans.readline()
25 if line2 != '':
26 sList2 = line2.split(',')
27 try:
28 tEmpID = int(sList2[0])
29 hrsWorked = int(sList2[1])
30 except IndexError:
31 print('Undefined Index'); sys.exit()
32 except (ValueError, TypeError):
33 print ('Unsuccessful conversion to int'); sys.exit()
34 while empID != tEmpID:
35 fWages.write(str(empID)+','+str(hrlyWages*0)+'\n')
36 line1 = fMaster.readline()
37 sList1 = line1.split(',')
38 try:
39 empID = int(sList1[0])
40 hrlyWages = int(sList1[2])
41 except IndexError:
42 print('Undefined Index'); sys.exit()
43 except (ValueError, TypeError):
44 print('Unsuccessful conversion to int'); sys.
45 exit()
46 fWages.write(str(empID)+','+str(hrlyWages*\
47 hrsWorked)+'\n')
48 line1 = fMaster.readline()
49 else:
50 fWages.write(str(empID)+','+str(hrlyWages*0)+'\n')
51 line1 = fMaster.readline()
52 fMaster.close()
53 fTrans.close()
54 fWages.close()
```

```
55 def main():
56 """
57 Objective: To determine monthly wages of all employees based on
58 user input
59 Input Parameter: None
60 Return Value: None
61 """
62 import sys
63 sys.path.append('F:\PythonCode\Ch09')
64 file1 = input('Enter name of file containing hourly rate:')
65 file2 = input('Enter name of file containing hours worked:')
66 file3 = input('Enter output file for salary generation:')
67 generateSalary(file1,file2,file3)
68 if __name__=='__main__':
69 main()
```

**Fig. 9.16** Program to generate salary (salaryGen.py)

#### SUMMARY

1. A file is a stream of bytes, comprising data of interest.
2. Built-in function `open()` is used for opening a file. It returns a file object. This function takes the name of the file as the first argument. The second argument indicates mode for accessing the file.
3. A file may be opened in any of three modes: `r` (read), `w` (write), and `a` (append). Read mode is used when an existing file is to be read. Write mode is used when a file is to be accessed for writing data in it. Append mode allows one to write into a file by appending contents at the end. If the specified file does not exist, a file is created.
4. The absence of the second argument in `open` function sets it to default value '`r`' (read mode).

5. When an existing file is opened in write mode, previous contents of the file get erased.
6. The functions `read` and `write` are used for reading from and writing into the file, respectively. To use `read/write` function, we use the following notation: name of the file object, followed by the dot operator (.), and followed by the name of the function.
7. Function `close` is used for closing the file. The function `close` also saves a file, which was opened for writing. Once a file is closed, it cannot be read or written any further unless it is opened again and an attempt to access the file results in an I/O (input/output) error.
8. Function `tell` yields current position in the file.
9. The `readline` function reads a stream of bytes beginning the current position until a newline character is encountered.
10. Read operation on a file whose all the contents have been read will return a null string.
11. Function `seek()` is used for accessing the file from a particular position.
12. Function `readlines()` returns all the remaining lines of the file in the form of a list of strings. Each string terminates with a newline character.
13. Function `writelines` takes a list of lines to be written to the file as an argument.
14. Pickling refers to the process of converting a structure to a byte stream before writing to the file. The reverse process of converting a stream of bytes into a structure is known as unpickling.
15. Python module `pickle` is used for writing an object such as list or dictionary to a file and reading it subsequently. Function `dump` of the `pickle` module performs pickling. Function `load` of the `pickle` module performs unpickling.
16. A syntax error occurs when a rule of Python grammar is violated.
17. The exception occurs because of some mistake in the program that the system cannot detect before executing the code as there is nothing wrong in terms of the syntax, but leads to a situation during the program execution that the system cannot handle. These errors disrupt the flow of the program at a runtime by terminating the execution at the point of occurrence of the error.
18. `NameError` exception occurs whenever a name specified in the statement is not found globally.
19. `TypeError` exception occurs when an operation in an expression is incompatible with the type of operands.
20. `ValueError` exception occurs whenever an invalid argument is used in a function call.
21. `ZeroDivisionError` exception occurs when we try to perform numeric division in which denominator happens to be zero.
22. `IOError` exception occurs whenever there is any error related to input or output.
23. `IndexError` exception occurs whenever we try to access an index out of the valid range.
24. `try...except` clause is used for handling the exception. Whereas a `try` block comprises statements, which have the potential to raise an exception, `except` block describes the action to be taken when exception is

- raised. In the `except` clause, we may specify a list of exceptions and the action to be taken on occurrence of each exception.
25. `finally` block is associated with `try...except` clause and is executed irrespective of whether an exception is raised.
  26. Details of the exception raised by Python can be accessed from the object: `sys.exc_info()`.

#### EXERCISES

1. Write a function that takes two file names, `file1` and `file2` as input. The function should read the contents of the file `file1` line by line and should write them to another file `file2` after adding a newline at the end of each line.
2. Write a function that reads a file `file1` and displays the number of words and the number of vowels in the file.
3. Write a function that takes data to be stored in the file `file1` as interactive input from the user until he responds with nothing as input. Each line (or paragraph) taken as input from the user should be capitalized, and stored in the file `file1`.
4. Write a function that reads the file `file1` and copies only alternative lines to another file `file2`. Alternative lines copied should be the odd numbered lines. Handle all exceptions that can be raised.
5. Write a function that takes two files of equal size as input from the user. The first file contains weights of items and the second file contains corresponding prices. Create another file that should contain price per unit weight for each item.
6. Write a function that reads the contents of the file `Poem.txt` and counts the number of alphabets, blank spaces, lowercase letters and uppercase letters, the number of words starting with a vowel, and the number of occurrences of word 'beautiful' in the file.
7. What will be the output produced on executing function `inversel` when the following input is entered as the value of variable `num`:  
 (a)5 (b)0 (c)2.0 (d)x (e)None  

```
def inversel():
 try:
 num = input('Enter the number: ')
 num = float(num)
 inverse = 1.0 / num
 except ValueError:
 print('ValueError')
 except TypeError:
 print('TypeError')
 except ZeroDivisionError:
 print('ZeroDivisionError')
 except:
 print('Any other Error')
 else:
 print(inverse)
 finally:
 print('Function inverse completed')
```
8. Examine the following function percentage:

```
def percentage(marks, total):
 try:
 percent = (marks / total) * 100
 except ValueError:
 print('ValueError')
 except TypeError:
 print('TypeError')
 except ZeroDivisionError:
 print('ZeroDivisionError')
 except:
 print('Any other Error')
 else:
 print(percent)
 finally:
 print('Function percentage completed')
```

Determine the output for the following function calls:

1. percentage(150.0, 200.0)
  2. percentage(150.0, 0.0)
  3. percentage('150.0', '200.0')
9. Identify two exceptions that may be raised while executing the following statement:
- ```
result = a + b
```
10. What will be the output for the following code snippets if the file being opened does not exist:
- ```
1. try:
 f = open('file1.txt', 'r')
 except IOError:
 print('Problem with Input
Output...\n')
 else:
 print('No Problem with Input
Output...\n')

2. try:
 f = open('file1.txt', 'w')
 except IOError:
 print('Problem with Input
Output...\n')
 else:
 print('No Problem with Input
Output...\n')
```

<sup>1</sup>  
\_ UNIX® is a registered trademark of The Open Group

# CHAPTER 10

## CLASSES I

### CHAPTER OUTLINE

[10.1 Classes and Objects](#)

[10.2 Person: An Example of Class](#)

[10.3 Class as Abstract Data Type](#)

[10.4 Date Class](#)

We have already studied basic building blocks of Python language such as variables for naming the data objects, control structures for control flow, and functions for providing a systematic way of problem solving by dividing the given problem into several sub-problems. A program of moderate size would comprise several groups of related information. Keeping track of them in an ad-hoc manner would be a challenging task. Packaging together related code and data in the form of classes would make our job easier. A class is a template that provides a logical grouping of data and methods that operate on them. Instances of a class are called objects.

class: logical grouping of related data and methods that operate on them

Variables used so far took values of types (also called classes) string (`str`), integer (`int`), floating point (`float`), Boolean (`bool`), list, tuple, or dictionary (`dict`). As these classes are already available in Python, they are called built-in classes. Python also allows us to define new classes (i.e., types), called user-defined classes. Data and methods associated with a class are collectively known as class attributes.

class attributes: data and methods associated with a class

## 10.1 CLASSES AND OBJECTS

We begin our study of classes with the built-in class `str`. The strings '`Raman`', '`CBSE`', and '`Jan 11 2014`' are instances of class `str`. The following assignment statement assigns the object '`Raman`' of class `str` to the variable name:

```
the variable name refers to object of class str
```

functions of a class are known as methods

```
>>> name = 'Raman'
```

A class would usually have several functions (called methods in the context of classes), which can act on objects of the class. We have already seen several methods that can be applied to objects of the class `str`. For example, the method `lower` returns an `str` object that has all uppercase letters in the given string object replaced by lowercase letters:

```
>>> name.lower()
'raman'
```

Here, the method `lower` defined in class `str` has been invoked for the object `name`. To specify an attribute of a class (or class instance), we write the name of the class (or class instance) followed by a dot, followed by the name of that attribute. In the above example, `name` – an instance of class `str` is followed by a dot (.), followed by `lower()` – the attribute of interest. The method `lower` may also be invoked as follows:

specifying attribute of a class

```
>>> str.lower(name)
'raman'
```

In this format, we specify the name of the class (`str`), followed by the dot operator (`.`), followed by the name of the method (`lower`), followed by an object (`name`). The object name being an argument is enclosed in parentheses.

#### 10.2 PERSON: AN EXAMPLE OF CLASS

In this section, we shall discuss the concept of class with the help of an example. To keep the matter simple, we shall describe a person using three attributes, viz., `name`, `DOB` (date of birth), and `address`, each of which takes values of type `str`. In Fig. 10.1, we define the class `Person` that describes a person with the above-mentioned attributes `name`, `DOB`, and `address`.

data attributes of class `Person`: `name`, `DOB`, and `address`

```
01 class Person:
02 ''' The class Person describes a person'''
03 count = 0
04 def __init__(self, name, DOB, address):
05 '''
06 Objective: To initialize object of class Person
07 Input Parameters:
08 self (implicit parameter) - object of type Person
09 name - string
10 DOB - string (Date of Birth)
11 address - string
12 Return Value: None
13 '''
14 self.name = name
15 self.DOB = DOB
16 self.address = address
17 Person.count += 1
18
19 def getName(self):
```

```
20 '''
21 Objective: To retrieve name of the person
22 Input Parameter: self (implicit parameter) - object of
23 type Person
24 Return Value: name - string
25 '''
26 return self.name
```

```
20
27 def getDOB(self):
28 """
29 Objective: To retrieve the date of birth of a person
30 Input Parameter: self (implicit parameter) - object of
31 type Person
32 Return Value: DOB - string
33 """
34 return self.DOB
35
36 def getAddress(self):
37 """
38 Objective: To retrieve address of person
39 Input Parameter: self (implicit parameter) - object of
40 type Person
41 Return Value: address - string
42 """
43 return self.address
44
45 def setName(self, name):
46 """
47 Objective: To update name of person
48 Input Parameter: self (implicit parameter) - object of
49 type Person
50 name - string value
51 Return Value: None
52 """
53 self.name = name
54
55 def setDOB(self, DOB):
56 """
57 Objective: To update DOB of person
58 Input Parameter: self (implicit parameter) - object of
59 type Person
60 DOB - string value
61 Return Value: None
62 """
63 self.DOB = DOB
```

```
60
61
62 def setAddress(self, address):
63 """
```

```

64 Objective: To update address of person
65 Input Parameter: self (implicit parameter) - object of
66 type Person
66 address - string value
67 Return Value: None
66 """
68 self.address = address
69
70 def getCount(self):
71 """
72 Objective: To get count of objects of type Person
73 Input Parameter: self (implicit parameter) - object of type
73 Person
74 Return Value: count: numeric
75 """
76 return Person.count
77
78
79 def __str__(self):
80 """
81 Objective: To return string representation of object of type
81 Person
82 Input Parameter: self (implicit parameter)- object of type
82 Person
83 Return Value: string
84 """
85 return 'Name:' +self.name+'\nDOB:' +str(self.DOB) \
86 +'\nAddress:' +self.address

```

**Fig. 10.1** Class Person (person.py)

A class definition begins with the keyword `class` followed by the name of the class, and a colon. By

convention, the first letter of the class name is capitalized. The syntax for `class` definition is as follows:

syntax for class definition

```
class ClassName:
 classBody
```

In the docstring in line 2, we describe the purpose of the class. In line 3, we introduce the class variable `count` and initialize it to 0. It is used to keep count of the number of `Person` objects created. The class variables are also called static variables. The `__init__` method is used to initialize data for the instance of the class being constructed and is called class initializer or class constructor. However, for the sake of completeness, we would like to mention that there may be a class not having any `__init__` method. Such a class would use the default constructor of Python. Names of the special methods begin and end with a double underscore. The `__init__` method is called automatically when an object of the class is created, for example, the following instruction creates an object named `p1`:

class variables are also called static variables

`__init__()`: to initialize data for an instance of the class

names of special methods begin and end with double underscore

creating an object of class `Person`

```
>>> p1 = Person('Amir','24-10-1990',
'38/4, IIT Delhi 110016')
```

The execution of the above statement does three things:

1. Creates an instance of class `Person`
2. Initializes it by invoking the method `__init__` defined in lines 4–17
3. Returns a reference to it, so the name `p1` now refers to the instance of the class `Person` that has just been created

Execution of statements in lines 14, 15, and 16 sets data members `name`, `DOB`, and `address` of the object `p1` equal to parameter values '`'Amir'`', '`'24-10-1990'`', and '`'38/4, IIT Delhi 110016'` respectively with which the method `__init__` was invoked. Execution of the assignment statement in line 17 increments the class variable `count`, which now holds value 1. Note that all instances of a class `Person` share this attributes. A reference to `count` may be made using either the class name or a specific object of the class `Person`.

a class attribute is shared among all instances of the class

a class attribute can be referenced using the name of the class or an object of the class

Note that while creating an object (such as `p1`) of class `Person`, although we mention only three arguments to be used by method `__init__`, in effect four arguments are used for initializing an instance of `Person`. By default, Python passes object itself (such as `p1`) as the first argument to the method `__init__`. In the formal parameter list for the method `__init__`, it appears by the name `self`. Although being a formal parameter, any other name (like `me`) would do equally well, there is a strong convention to use `self` as the first parameter. Indeed, when we execute the statement:

by default, the object itself is passed as the first parameter to the method `__init__`

```
p1 = Person('Amir','24-10-1990','38/4, IIT
Delhi 110016')
```

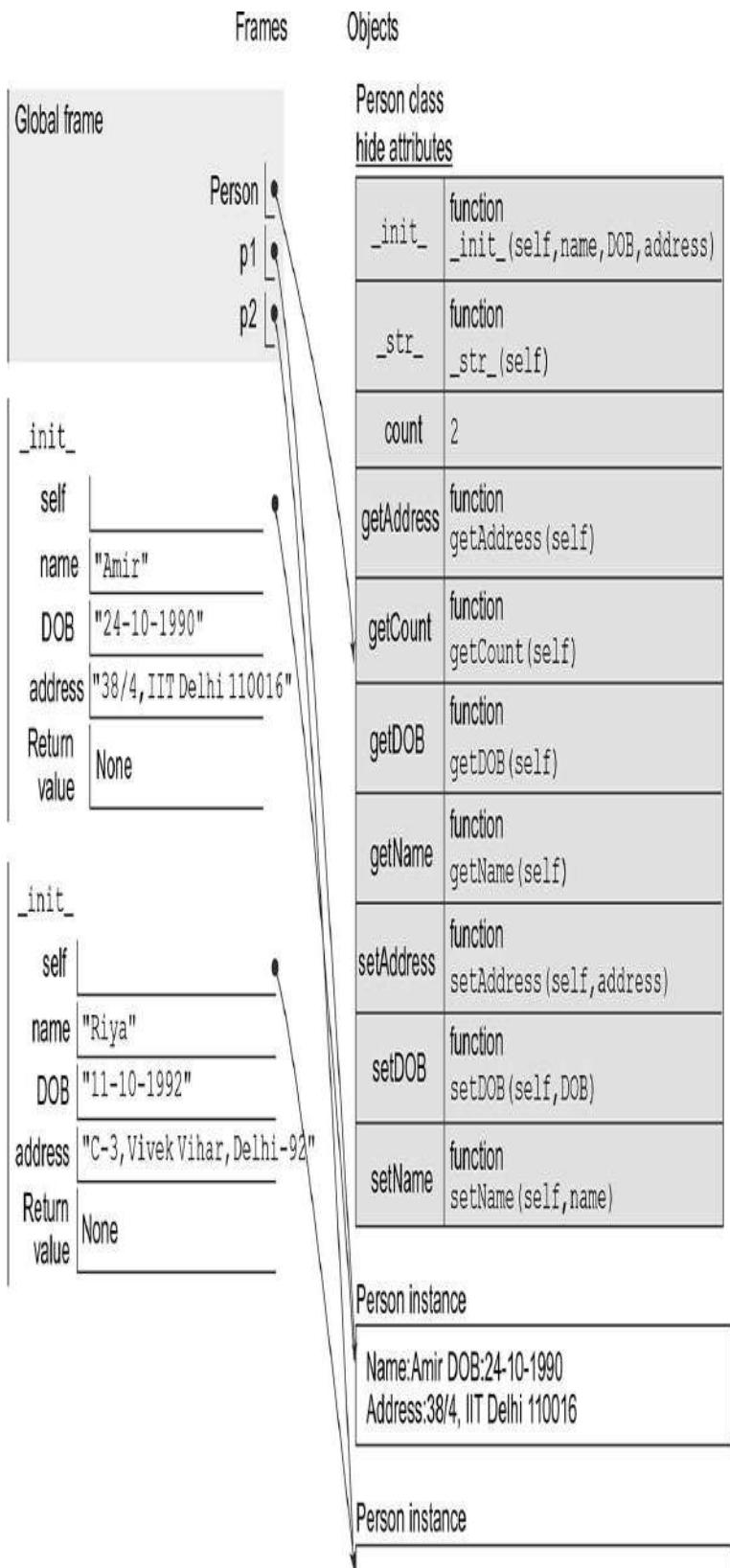
Python invokes the method `Person.__init__` with `p1` (the object being created) as the first parameter. Note that while the name, DOB, and address mentioned in line 4 are formal parameters for the method `__init__`, variables `self.name`, `self.DOB`, and `self.address` used in lines 14, 15, and 16 are associated with the specific instance, such as `p1`, of the object being created. Next, we create another object of class `Person`:

```
>>> p2 = Person('Riya','11-10-1992','C-
3,Vivek Vihar, Delhi-92')
```

Execution of the above statement creates an object named `p2` of class `Person` and initializes it by invoking the method `__init__`. Execution of statements in lines 14, 15, and 16 sets data members `name`, `DOB`, and `address` of object `p2` equal to parameters '`Riya`', '`11-10-1992`', and '`C-3,Vivek Vihar, Delhi-92`' respectively. Execution of the assignment statement in line 17 increments the class variable `count`, which now holds value 2.

Data members and methods associated with an instance of a class are called attributes of the object. The set of attributes that are associated with an object forms its namespace. [Figure 10.2](#) shows Python tutor visualization of objects `p1` and `p2` along with the class attributes. The data members of the object `p1` are referred to as `p1.name`, `p1.DOB`, `p1.address`, and the methods of the object `p1` are referred to as `p1.getName`, `p1.getDOB`, `p1.getAddress`, `p1.setName`, `p1.setDOB`, `p1.setAddress`, `p1.getCount`, and `p1.__str__`. We refer to the attributes of the object `p2` in a similar manner.

object attributes: data members and methods associated with an instance of a class





**Fig. 10.2** Python Tutor visualization of objects `p1` and `p2`, and class attributes

To execute the script `person` and generate some objects, let us include the following two statements in the script:

```
p1 = Person('Amir','24-10-1990','38/4, IIT
Delhi 110016')
```

```
p2 = Person('Riya','11-10-1992','C-3,Vivek
Vihar, Delhi-92')
```

Before we go ahead to execute the modified script `person` in Python Tutor, it is important to know that currently, Python Tutor does not allow modification of the value of a class variable. So, if we wish to execute the modified script `person` in Python Tutor, we need to delete line 17 that modifies the value of the class variable `count`. Indeed, Fig. 10.2 is a manually edited version of the figure produced by the Python Tutor.

It is important to know that whereas each instance of a class has its copy of data members, the class variables and methods are shared. In summary, operations supported by classes may be categorized as follows:

1. Instantiation: It refers to the creation of an object, i.e. an instance of the class.
2. Attribute references: Methods and data members of an object of a class are accessed using the notation: name of the object, followed by dot operator, followed by the member name.

operations supported by a class: instantiation and attribute references

For the system-defined classes like `int`, `str`, `list`, and `tuple`, Python provides built-in method `attribute`

`__str__` that transforms internal representation of the object to a printable string as illustrated below:

`__str__()`: to transform internal representation of the object to a printable string

```
>>> hasattr(7, '__str__')
```

True

```
>>> hasattr(int, '__str__')
```

True

```
>>> hasattr(list, '__str__')
```

True

```
>>> int.__str__(7)
```

'7'

```
>>> print(int.__str__(7))
```

7

The method `hasattr(obj, attr)` returns True if instance `obj` contains the attribute `attr`, and False otherwise. Note that the method `__str__` when applied to int object 7 indeed returns equivalent `str` value '7'. Thus, the statement

`hasattr()`: does an object possess the given attribute?

```
print(7)
```

is effectively equivalent to each of the following statements:

```
print(int.__str__(7))

print(str(7))
```

When the `print` function is called to print an object, Python invokes `__str__` method of the corresponding class to obtain a string representation of the object. So, to print the values of data attributes of an object of type `Person`, we define the method `__str__` (lines 79–86, Fig. 10.1) for this class. This method takes an object of class `Person` as the parameter and returns a string describing the data attributes: `name`, `DOB`, and `address` associated with the object. Thus, execution of statement

`__str__` method is automatically invoked when `print` function is invoked to print an object

```
p1.__str__()
```

would yield the string:

```
'Name:Amir\nDOB:24-10-1990\nAddress:38/4,
IIT Delhi 110016'
```

and

```
print(p1.__str__())
```

would yield the output:

```
Name:Amir
```

```
DOB:24-10-1990
```

```
Address:38/4, IIT Delhi 110016
```

Execution of the following statement also yields the same output as shown above:

```
>>> print(p1)
```

In Fig. 10.1, we have defined methods `getName`, `getDOB`, and `getAddress`, which are responsible for

returning the name, DOB, and address associated with an object. We have also defined methods `setName`, `setDOB`, and `setAddress` for updating name, date of birth, and address, respectively. For example, suppose, DOB for person `p1`, needs to be modified to '24-10-1991'. To achieve this, we invoke the method `setDOB` associated with the object `p1`.

```
methods of the class Person
```

```
>>> p1.setDOB('24-10-1991')
```

```
>>> print(p1)
```

```
Name:Amir
```

```
DOB:24-10-1991
```

```
Address:38/4, IIT Delhi 110016
```

### 10.2.1 Destructor

When an object is no more required, we use the `del` statement. However, an object may be referenced by multiple names. Execution of the `del` statement reduces the reference count by one. When the reference count becomes zero, the `__del__` method is invoked. Next, we define the `__del__` method for the class `Person`. As the number of `Person` objects reduces by one on execution of the destructor, we decrement `Person.count` by one (line 9, Fig. 10.3).

```
1 def __del__(self):
2 """
3 Objective: To be invoked on deletion of an instance of the
4 class Person
5 Input Parameter:
6 self (implicit parameter) - object of type Person
7 Return Value: None
8 """
9 print('Deleted !!')
10 Person.count -= 1
```

**Fig. 10.3** Person class method `__del__`

The use of `del` statement is illustrated below:

```
>>> p1 = Person('Amir','24-10-1990','38/4,
IIT Delhi 110016')

>>> p2 = Person('Riya','11-10-1992','C-
3,Vivek Vihar, Delhi-92')

>>> p3 = p2

>>> print(Person.count)

2

>>> del p1

Deleted !!

>>> print(p1)

Traceback (most recent call last):
```

```
File "<pyshell#23>", line 1, in <module>
 print(p1)
NameError: name 'p1' is not defined

>>> print(Person.count)

1

>>> del p2

>>> print(Person.count)

1

>>> del p3

Deleted !!

>>> print(Person.count)

0
```

the method `__del__` is invoked before destroying the instance object, when all the references to the given instance have been removed

Note that we created two Person objects `p1` and `p2`. Name `p3` refers to the same object as `p2` and is just another reference for the object `p2` created earlier, and the reference count of the object `p2` as well as that of `p3` is 2. Execution of the instruction `del p1` invokes `__del__` method defined in the class `Person`, the object `p1` gets deleted, `Person.count` gets decremented by one, and the message '`Deleted !!`' is printed. However, the instruction `del p2` does not delete the object `p2` as the name `p3` still refers to it. It only dissociates the name `p2` from the object.

#### 10.3 CLASS AS ABSTRACT DATA TYPE

Oxford dictionary describes *abstract* as follows:

- existing in thought or idea but not having a physical or concrete existence,
- the quality of dealing with ideas rather than events,
- freedom from representational qualities,
- the process of considering something independently of its associations or attributes.

abstract data type

Indeed, a class in Python introduces an abstract data type in the sense mentioned above. A class definition provides a framework for creating objects. Objects of a class can be created, modified, and destroyed in the program.

objects can be created, modified, and destroyed

Once a class is defined, it can be saved as an independent module that can be imported for use in any module. For example, let us save the above class definition in the file `person.py` in the directory `F:\PythonCode\Ch10`. We illustrate the use of the class `Person` in another script, `personEx` (Fig. 10.4). The first step is to include the appropriate directory (`F:\PythonCode\Ch10`) in the path (lines 4–6). Next, in line 7, we import all attributes of class `Person` from the module `person.py`, so that they become available in the current module `personEx.py`.

The execution of the first statement in the main function (line 16) prints the directory of class `Person` as shown in Fig. 10.5 (lines 1–8). Note that directory contains the docstring `__doc__`, the class variable `count`, and the list of methods of the class, but does not contain the data attributes of the class as they belong to the class objects and not directly to the class. Also, note that the directory includes some attributes that we have not discussed. These are system defined attributes. Once you are comfortable with the basic concepts of classes,

you can experiment with them. As expected, use of print function in line 17 prints the docstring:

directory of a class

The class Person describes a person

associated with the class Person (lines 9–10, Fig. 10.5). In line 18 (Fig. 10.4), we print the name of the module that contains the class Person (lines 11–12, Fig. 10.5). In line 22 (Fig. 10.4), we create an object p1 of the class Person.

```
01 """
02 Objective: To illustrate the use of class Person
03 """
04 import sys
05 import os
06 sys.path.append('F:\PythonCode\Ch10')
07 from person import Person
08 def main():
09 """
10 Objective: To illustrate the use of class Person
11 Input Parameter: None
12 Return Value: None
13 """
14 # About class Person : dir, __doc__, __module__
15
16 print('***** dir(Person):\n', dir(Person))
17 print('***** Person.__doc__:\n', Person.__doc__)
18 print('***** Person.__module__:\n', Person.__module__)
19
20 # create an object p1
21
22 p1 = Person('Amir', '24-10-1991','38/4, IIT Delhi 110016')
23 print('***** Person.count:\n', Person.count) # Violation of
principle of abstraction
24 print('***** p1.getCount():\n', p1.getCount()) # This is fine
25 print('***** p1.__doc__:\n', p1.__doc__)
26 print('***** p1.__module__:\n', p1.__module__)
27 print('***** p1:\n', p1)
28 print('***** dir(p1):\n', dir(p1))
```

```
20 print('''+ str(p1) + '''+ str(p2))
21
22
23
24
25
26
27
28
29 # create an object p2
30
31
32 p2 = Person('Riya', '11-10-1992', 'C-3,Vivek Vihar, Delhi-92')
33 print('***** Person.count:\n', Person.count) # Violation of
34 principle of abstraction
35 print('***** p2.getCount():\n', p2.getCount())
36 print('***** p1.getCount():\n', p1.getCount())
37 print('\n***** p2.__doc__:\n', p2.__doc__)
38 print('***** p2.__module__:\n', p2.__module__)
39 print('***** p2:\n', p2)
40 print('***** dir(p2):\n', dir(p2))
41
42 # illustration: instances of __doc__, p1.__module__, dir
43
44 print('\n***** id: Person.__doc__:\n', id(Person.__doc__))
45 print('***** id: Person.__module__:\n', id(Person.__module__))
```

```

45 print('***** id: p1.__doc__:\\n', id(p1.__doc__))
46 print('***** id: p1.__module__:\\n', id(p1.__module__))
47 print('***** id: p2.__doc__:\\n', id(p2.__doc__))
48 print('***** id: p2.__module__:\\n', id(p2.__module__))
49 print('***** id: dir(Person):\\n', id(dir(Person)))
50 print('***** id: dir(p1):\\n', id(dir(p1)))
51 print('***** id: dir(p2):\\n', id(dir(p2)))
52
53 # use of __dict__
54
55 print('\\n***** Person.__dict__\\n', Person.__dict__)
56 print('\\n***** p1.__dict__:\\n', p1.__dict__)
57 print('\\n***** p2.__dict__:\\n', p2.__dict__)
58
62 if __name__ == '__main__':
63 main()

```

**Fig. 10.4** Using class Person (personEx.py)

```

001 ***** dir(Person):
002 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
003 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
004 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
005 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
006 __repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
007 ...

```

```
007 '__weakref__', 'count', 'getAddress', 'getCount', 'getDOB', 'getName',
008 'setAddress', 'setDOB', 'setName']
009 ***** Person.__doc__:
010 The class Person describes a person
011 ***** Person.__module__:
012 person
013 ***** Person.count:
014 1
015 ***** p1.getCount():
016 1
017 ***** p1.__doc__:
018 The class Person describes a person
019 ***** p1.__module__:
020 person
021 ***** p1:
022 Name:Amir
023 DOB:24-10-1991
024 Address:38/4, IIT Delhi 110016
025 ***** dir(p1):
```

```
026 ['DOB', '__class__', '__delattr__', '__dict__', '__dir__', '__
027 doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
028 '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
029 '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
030 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
031 '__str__', '__subclasshook__', '__weakref__', 'address',
032 'count', 'getAddress', 'getCount', 'getDOB', 'getName', 'name',
033 'setAddress', 'setDOB', 'setName']
034 ***** Person.count:
035 2
036 ***** n2.getCount():
```

```
*** P2.py:line 1, in <module>
037 2
038 ***** pl.getCount():
039 2
040
041 ***** p2.__doc__:
042 The class Person describes a person
043 ***** p2.__module__:
044 person
045 ***** p2:
046 Name:Riya
047 DOB:11-10-1992
048 Address:C-3,Vivek Vihar, Delhi-92
049 ***** dir(p2):
050 ['DOB', '__class__', '__delattr__', '__dict__', '__dir__', '__
051 doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
052 '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
053 '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
054 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
055 '__str__', '__subclasshook__', '__weakref__', 'address',
056 'count', 'getAddress', 'getCount', 'getDOB', 'getName', 'name',
057 'setAddress', 'setDOB', 'setName']
058
059 ***** id: Person.__doc__:
060 52379232
061 ***** id: Person.__module__:
062 52374176
063 ***** id: pl.__doc__:
064 52379232
065 ***** id: pl.__module__:
066 52374176
067 ***** id: p2.__doc__:
068 52379232
069 ***** id: p2.__module__:
070 52374176
071 ***** id: dir(Person):
```

```
072 52332984
073 ***** id: dir(pl):
074 52332184
075 ***** id: dir(p2):
076 52374176
```

```

070 52331344
077 ***** Person.__dict__
078 {'__module__': 'person', '__doc__': ' The class Person describes
079 a person', 'count': 2, '__init__': <function Person.__init__ at
080 0x036D8F60>, 'getName': <function Person.getName at 0x036D8F18>,
081 'getDOB': <function Person.getDOB at 0x036D8ED0>, 'getAddress':
082 <function Person.getAddress at 0x0341E108>, 'setName': <function
083 Person.setName at 0x01273228>, 'setDOB': <function Person.setDOB
084 at 0x036E0030>, 'setAddress': <function Person.setAddress at
085 0x036E0078>, 'getCount': <function Person.getCount at 0x036E00C0>,
086 '__str__': <function Person.__str__ at 0x036E0108>, '__dict__':
087 <attribute '__dict__' of 'Person' objects>, '__weakref__':
088 <attribute '__weakref__' of 'Person' objects>}
089
090 ***** p1.__dict__:
091 {'name': 'Amir', 'DOB': '24-10-1991', 'address': '38/4, IIT Delhi
092 110016'}
093
094 ***** p2.__dict__:
095 {'name': 'Riya', 'DOB': '11-10-1992', 'address': 'C-3,Vivek
096 Vihar, Delhi-92'}

```

**Fig. 10.5** Output of the script personEx (Fig. 10.4)

In line 23 (Fig. 10.4), we print the value of class variable count (lines 13–14, Fig. 10.5). Here, we would like to mention that to access data attributes of a class directly outside of the class definition is in violation of the abstraction principle and it is best avoided. In line 24

(Fig. 10.4), we invoke the method `getCount` for the object `p1` that fetches the class variable `count` (lines 15–16, Fig. 10.5). This is consistent with the abstraction principle as we are making use of the function `getCount` to retrieve the value of `count`. In lines 25–26 (Fig. 10.4), we print the values of `p1.__doc__` and `p1.__module__` (lines 17–20, Fig. 10.5). Note that `p1.getCount()`, `p1.__doc__`, and `p1.__module__` yield the same values as `Person.count`, `Person.__doc__`, and `Person.__module__`, respectively, because `p1` is an object of class `Person`. In line 27 (Fig. 10.4), we print the object `p1` (see Fig. 10.5, lines 21–24). The `print` function invokes the method `Person.__str__()` for the object `p1`. More explicitly, we may invoke the `__str__` method defined in class `Person` as follows:

while `Person.count` can also be accessed using `p1.count`, but assigning a value to `p1.count` would not change the value of corresponding class variable, instead it will create a new data attribute for the object `p1`

```
>>> print(p1.__str__())
Name :Amir
DOB :24-10-1991
Address :38/4, IIT Delhi 110016
```

The above statement invokes the method `__str__` of the object `p1`. Here, the object `p1` that corresponds to the formal parameter `self` is passed to the method `__str__` implicitly. In the following call to the `print` function, the call to method `__str__` of class `Person` is even more explicit and matches the syntactic description of `__str__(self)`. However, in practice, this form is rarely used.

```
explicit call to method __str__
>>> print(Person.__str__(p1))

Name:Amir

DOB:24-10-1991

Address:38/4, IIT Delhi 110016
```

In line 28 ([Fig. 10.4](#)), we print the directory of the object p1 (lines 25–33, [Fig. 10.5](#)). Note that in addition to attributes of the class Person that automatically get associated with the object p1, the directory for object p1 also contains its data attributes: name, DOB, and address.

In lines 32–39 ([Fig. 10.4](#)), we create another object p2 and print Person.count, p2.getCount(), p1.getCount(), p2.\_\_doc\_\_, p2.\_\_module\_\_, p2, and dir(p2) (lines 34–57, [Fig. 10.5](#)). Note that each of Person.count, p2.getCount(), and p1.getCount() yields value 2. You must have noticed that p1.getCount() yields the value 2 because Python maintains only one copy of a class variable that gets associated with all instances of that class. For the same reason, p1.\_\_doc\_\_ and p2.\_\_doc\_\_ have identical values as each of them corresponds to the string Person.\_\_doc\_\_ ([Fig. 10.5](#), lines 9–10, 17–18, 41–42). Similarly, p1.\_\_module\_\_ and p2.\_\_module\_\_ have identical values as each of them corresponds to the string Person.\_\_module\_\_ ([Fig. 10.5](#), lines 11–12, 19–20, 43–44). Indeed, dir(p1) and dir(p2) also have identical values ([Fig. 10.5](#), lines 25–33, 49–57) as p1 and p2 have the same set of attributes. Further, in output lines 59–70 ([Fig. 10.5](#)) corresponding to lines 41–51 ([Fig. 10.4](#)) of source code, we see that Person.\_\_module\_\_, p1.\_\_module\_\_, and p2.\_\_module\_\_ have the same object ids. Similarly, Person.\_\_doc\_\_, p1.\_\_doc\_\_, and p2.\_\_doc\_\_ have identical object ids.

Python maintains only one copy of class variable

In lines 55–57 (Fig. 10.4), we print Person.\_\_dict\_\_, p1.\_\_dict\_\_, and p2.\_\_dict\_\_. The object Person.\_\_dict\_\_ is a dictionary comprising key-value pairs (Fig. 10.5, lines 77–88). A key may either be a class data member or a method, and the corresponding value would be the value of the data member or the reference associated with the class method. Similarly, each of the dictionaries p1.\_\_dict\_\_ and p2.\_\_dict\_\_ comprises mapping between data members associated with it and their values (Fig. 10.5, lines 90–92, 94–96).

\_\_dict\_\_ attribute

#### 10.4 DATE CLASS

We often need to deal with date and time in our programs. Although Python provides a built-in class for this purpose, in this section, we shall discuss how to develop our class to deal with dates. We know that the date comprises the day, month, and year. In the script date (Fig. 10.6), we define the class MyDate. In this example we have struggled to ensure that the class MyDate creates an instance of a valid date only. However, you may skip the details of validating the date on first reading.

```
01 import sys
02 class MyDate:
03
04 """
05 MyDate: A simple implementation of date as a class.
06
07 """
08 def __init__(self, day = 1, month = 1, year = 2000):
09 """
```

```
10 Objective: To initialize data members of object MyDate
11 Input Parameters:
12 self (implicit parameter) - object of type MyDate
13 day, month, and year - int
14 Return Value: None
15 ''
16 if not(type(day)==int and type(month)==int and
17 type(year)==int):
18 print('Invalid data provided for date')
19 sys.exit()
20 if month > 0 and month <= 12:
21 self.month = month
22 else:
23 print('Invalid value for month')
24 sys.exit()
25 if year > 1900:
26 self.year = year
27 else:
28 print('Invalid value for year. Year should be greater\
29 than 1900.')
30 sys.exit()
31 self.day = self.checkDay(day) # validate day
32
33 def checkDay(self, day):
34 ''
35 Objective: To validate day component
36 Input Parameters:
37 self (implicit parameter) - object of type MyDate
38 day - numeric
```

```
38 Return Value: day if it is correct else message 'Invalid
39 value for the day' is printed and the program is
40 terminated
41 ''
42 # currentYear: list of no of days in months of current year
43
44 if self.year%400==0 or (self.year%100!=0 and self.
45 year%4==0): #if leap year
46 currentYear = [31,29,31,30,31,30,31,31,30,31,30,31]
```

```
45 else:
46 currentYear = [31,28,31,30,31,30,31,31,30,31,30,31]
47 if (day > 0 and day <= currentYear[self.month - 1]):
48 return day
49 else:
50 print('Invalid value for day')
51 sys.exit()
52
53 def __str__(self):
54 """
55 Objective: To return string representation of object
56 Input Parameter:
57 self (implicit parameter) - object of type MyDate
58 Return Value: string
59 """
60 # Approach: use dd-mm-yyyy format
61 if self.day <= 9:
62 day = '0' + str(self.day)
63 else:
64 day = str(self.day)
65 if self.month <= 9:
66 month = '0' + str(self.month)
67 else:
68 month = str(self.month)
69 return day+'-'+month+'-'+str(self.year)
70
71 def main():
72 """
73 Objective: To create objects of class Date and to perform
74 operations on it
75 Input Parameter: None
76 Return Value: None
77 """
78 today = MyDate(3,9,2014)
79 print(today)
80 defaultDate = MyDate()
```

```
80 print(defaultDate)
81
82 if __name__=='__main__':
83 main()
```

**Fig. 10.6** Definition and use of class MyDate (date.py)

The execution of line 77 (Fig. 10.6) creates an object named today and initializes it to date 03-09-2014 by invoking the constructor of the class MyDate as MyDate(3, 9, 2014). Execution of the lines 20, 25, and 30 in the constructor achieves the desired initialization of the data members day, month, and year of the object today of the class MyDate. Execution of the statement in line 79 creates the object defaultDate. Since no arguments are provided while creating the object defaultDate, the method `__init__` uses the default parameter values 01, 01, and 2000 for initializing the instance variables day, month, and year, respectively. So, on executing line 80, Python will print:

```
default date
```

```
01-01-2000
```

Note that day, month, and year provided while invoking the constructor of the class MyDate may be invalid. For example, the following statement attempts to create an object of the class MyDate by initializing the instance variables day, month, and year as 30, 02, and 2012, respectively:

```
>>> feb30 = MyDate(30, 2, 2012)
```

```
Invalid value for day
```

Therefore, when we create an object of class `MyDate`, we need to ensure that the values of the `day`, `month`, and `year` passed as arguments to the constructor `__init__` constitute a valid date. Firstly, values of data attributes `day`, `month`, and `year` should be a positive and non-zero integer. Also, the value of `month` should not be more than 12, and the value of `day` should be valid as per the `month` and the specified `year`. In line 30 (Fig. 10.6), we invoke the method `checkDay` (lines 32–51). It first determines whether the given `year` is a leap year (line 43) and then assigns the list of days in months of that `year` to the variable `currentYear`. In line 47 of the method `checkDay`, we determine whether the `day` specified by the user is within correct range: less than or equal to the maximum number of days in the given `month`, and greater than 0. If any of the attributes `day`, `month`, or `year` provided by the user happens to be incorrect, the relevant message '`Invalid value for day`', '`Invalid value for month`', or '`Invalid value for year`. `Year` should be greater than 1900.' is printed and the program terminates.

#### validating date

As mentioned earlier that the `print` function in line 78 invokes the method `__str__` that returns the string representation of this object in the dd-mm-yyyy format to print the value of object `today` of the class `MyDate` as 03-09-2014.

#### SUMMARY

1. A class is a template that provides a logical grouping of data and methods that operate on them. Instances of a class are called objects.
2. Data members and methods associated with an instance of a class are called attributes of the object. The set of attributes that are associated with an object forms its namespace. To use an attribute of a class (or class instance), we specify the name of the class (or class instance) followed by a dot operator, followed by the name of that attribute.
3. A class definition begins with the keyword `class` followed by the name of the class, and a colon. By

- convention, the first letter of the class name is capitalized.
4. Functions defined in a class are called methods. By default, Python passes the object itself as the first parameter to the method.
  5. Special methods begin and end with a double underscore.
  6. The `__init__` method is used to initialize an instance of a class.
  7. When we want to delete an object using `del` statement, Python special method `__del__`, if defined, is invoked for the object.
  8. Operations supported by classes may be categorized as follows:
    1. Instantiation: It refers to the creation of an object, i.e. an instance of the class.
    2. Attribute references: Methods and data members of an object of a class are accessed using the notation: name of the object, followed by dot operator, followed by the member name.
  9. Python provides a built-in method attribute `__str__` for transforming the internal representation of the object to a printable string. When the `print` function is executed to print an object, Python invokes the method `__str__` of the corresponding class to obtain a string representation of the object.
  10. Python built-in data member `__doc__` stores docstring of the class.
  11. Python built-in data member `__dict__` is a dictionary comprising key-value pairs. A key may either be a data member or a method, and the corresponding value would be the value of the data members or the reference associated with the class method.
  12. Class attributes are shared among all instances of the class.

#### EXERCISES

1. Define a class `Rectangle`. The class should contain sides: `length` and `breadth` of the rectangle as the data members. It should support the following methods:
  1. `__init__` for initializing the data members: `length` and `breadth`.
  2. `setLength` for updating the length of the rectangle.
  3. `setBreadth` for updating breadth of the rectangle.
  4. `getLength` for retrieving the length of the rectangle.
  5. `getBreadth` for retrieving the breadth of the rectangle.
  6. `area` to find the area of the rectangle.
  7. `perimeter` for finding perimeter of the rectangle.
2. Define the following methods for `MyDate` class:
  1. `addDays` – To add  $n$  days to the date.
  2. `addMonths` – To add  $n$  months to date.
  3. `addYears` – To add  $n$  years to date.
  4. `weekday` – To return weekday of the date.

5. `diffDates` – To find difference between two dates in terms of the years, months, and days.  
 6. `futureDate` – To find a future date after a given number of days, months, and years.  
 7. `pastDate` – To find a date in the past before a given number of days, months, and years.  
 3. Define a class `Student` that keeps track of academic record of students in a school. The class should contain the following data members:  
     `rollNum` – Roll number of student  
     `name` – Name of student  
     `marksList` – List of marks in five subjects  
     `stream` – A: Arts, C: Commerce, S: Science  
     `percentage` – Percentage computed using marks  
     `grade` – Grade in each subject computed using marks  
     `division` – Division computed on the basis of overall percentage  
 The class should support the following methods:  
     1. `__init__` for initializing the data members.  
     2. `setMarks` to take marks for five subjects as an input from the user.  
     3. `getStream` for accessing the stream of the student.  
     4. `percentage` for computing the overall percentage for the student.  
     5. `gradeGen` that generates grades for each student in each course on the basis of the marks obtained. Criteria for computing the grade is as follows:

| Marks                      | Grade |
|----------------------------|-------|
| $\geq 90$                  | A     |
| $<90 \text{ and } \geq 80$ | B     |
| $<80 \text{ and } \geq 65$ | C     |
| $<65 \text{ and } \geq 40$ | D     |
| $<40$                      | E     |

6. `division` for computing division on the basis of the following criteria based on overall percentage of marks scored:

| Percentage                 | Division |
|----------------------------|----------|
| $\geq 60$                  | I        |
| $<60 \text{ and } \geq 50$ | II       |
| $<50 \text{ and } \geq 35$ | III      |

7. `__str__` that displays student information.

4. Define a class `Bank` that keeps track of bank customers.

The class should contain the following data members:

- name – Name of the customer
- accountNum – Account Number
- type – Account Type (Savings or Current)
- amount – amount deposited in the bank account
- interest – Interest earned by the customer

The class should support the following methods:

1. `__init__` for initializing the data members.
2. `deposit` for depositing money in the account.
3. `withdrawal` for withdrawing money from the account.
4. `findInterest` that determines the interest on the basis of amount in the account:

| Amount                                  | Interest per annum (%) |
|-----------------------------------------|------------------------|
| $\geq 5,00,000$                         | 8                      |
| $\geq 3,00,000 \text{ and } < 5,00,000$ | 7                      |
| $\geq 1,00,000 \text{ and } < 3,00,000$ | 5                      |
| $< 1,00,000$                            | 3                      |

5. `__str__` that displays information about bank customer.

5. Define a class `Item` that keeps track of items available in the shop. The class should contain the following data members:

- name – Name of the item
- price – Price of the item
- quantity – Quantity of the item available in the stock

The class should support the following methods:

1. `__init__` for initializing the data members.
  2. `purchase` for updating the quantity after a purchase made by the customer. The method should take the number of items to be purchased as an input.
  3. `increaseStock` for updating the quantity of an item for which new stock has arrived. The method should take the number of items to be added as an input.
  4. `display` that displays information about an item.
6. Fill in the details (Fig. 10.7) to define the method to compute the date on the following day of a given date

for the class MyDate:

```
def nextDay(self):
 """
 Objective: To display next day's date
 Input Parameter:
 self - object of type MyDate
 Return Value: string representations of next day's date
 """
 """
 Approach:
 Increment day if next day is within same month,
 else adjust month and year
 """
 pass
```

**Fig. 10.7** Program to method to compute the date on the following day  
of a given date

# CHAPTER 11

## CLASSES II

### CHAPTER OUTLINE

[11.1 Polymorphism](#)

[11.2 Encapsulation, Data Hiding, and Data Abstraction](#)

[11.3 Modifier and Accessor Methods](#)

[11.4 Static Method](#)

[11.5 Adding Methods Dynamically](#)

[11.6 Composition](#)

[11.7 Inheritance](#)

[11.8 Built-in Functions for Classes](#)

We have already studied user-defined data types in the form of classes. The classes lie at the heart of a programming methodology called object-oriented paradigm or *Object-oriented Programming* (OOP). It includes several concepts like polymorphism, encapsulation, data hiding, data abstraction, and inheritance. In this chapter, we shall study these concepts.

### object-oriented programming

#### 11.1 POLYMORPHISM

Object-oriented programming revolves around the notion of objects. A method/operator may be applied to objects of different types (classes). This feature of object-oriented programming is called polymorphism. We have already seen that `len` function operates on various types of objects such as `str`, `list`, and `tuple`. Based on the type of object whose length we want to find out, Python invokes `__len__` method of the appropriate class. As another example, the `print` function may be used to

print object of any type whether user-defined or pre-defined. Based on the type (class) of the object to be printed, Python would invoke `__str__` method of the associated class (pre-defined classes such as `str`, `int`, `dict`, `tuple`, or user-defined class such as `MyDate`).

applying a method/ operator to objects of different types

### 11.1.1 Operator Overloading

When we add, subtract, multiply, or divide two `int` or `float` objects using operators `+` `-` `*` `/`, the corresponding Python special method `__add__`, `__sub__`, `__mul__`, or `__div__` gets invoked for the class (type) of objects on which the operator is to be applied. For example, in the expressions `3.5+4.5`, `9.7-5.5`, `6.5*2.1`, and `11.5/5.5`, the methods `__add__`, `__sub__`, `__mul__`, and `__div__` of `float` class gets invoked. In the same spirit, in the expressions `3+4`, `[1,2]+[3,4]`, `(1,2)+(3,4)`, and `'12' + '34'`, the method `__add__` of the respective class `int`, `list`, `tuple`, and `str` gets invoked, as illustrated in Fig. 11.1. Use of the same syntactic operator such as `+` in the examples shown above for objects of different classes (types) is called operator overloading.

overloading + operator

```
>>> int.__add__(3,4) >>> 3+4
7 7
>>> list.__add__([1,2], [3,4]) >>> [1,2] + [3,4]
[1, 2, 3, 4] [1, 2, 3, 4]
>>> tuple.__add__((1,2), (3,4)) >>> (1,2) + (3,4)
(1, 2, 3, 4) (1, 2, 3, 4)
>>> str.__add__('12', '34') >>> '12' + '34'
'1234' '1234'
```

**Fig. 11.1** Use of special methods

Next, we define a class `Point` (Fig. 11.2) for representing 2D points on the plane. The class contains methods `__init__` and `__str__` for initializing and displaying the string representation of `Point` object respectively.

```
01 class Point:
02
03 def __init__(self, x, y):
04 '''
05 Objective: To initialize object of class Point
06 Input Parameters:
07 self (implicit parameter) - object of type Point
08 x - x co-ordinate of point (numeric value)
09 y - y co-ordinate of point (numeric value)
10 Return Value: None
11 '''
12 self.x = x
13 self.y = y
14
15 def __str__(self):
16 '''
17 Objective: To return string representation of object of
18 type Point
19 Input Parameter: self (implicit parameter)- object of type
20 Point
21 Return Value: string
22 '''
23 return str((self.x,self.y))
```

**Fig. 11.2** Class Point (point.py)

Suppose we wish to add two points to determine co-ordinates of the new point. For this purpose, we may overload the special method `__add__` by defining an implementation of addition for objects of type `Point`. In Fig. 11.3, we modify the class `Point` by incorporating the method `__add__` which overloads the operator `+`.

overloading + operator for adding two points

```
01 class Point:
02
03 def __init__(self, x, y):
04 '''
05 Objective: To initialize object of class Point
06 Input Parameters:
07 self (implicit parameter) - object of type Point
08 x - x co-ordinate of point (numeric value)
09 y - y co-ordinate of point (numeric value)
10 Return Value: None
11 '''
12 self.x = x
13 self.y = y
14
15 def __add__(self, other):
16 '''
17 Objective: To add two Point objects
18 Input Parameters:
19 self (implicit parameter) - object of type Point
20 other - object of type Point
21 Return Value: result - object of type Point
22 '''
23 x = self.x + other.x
24 y = self.y + other.y
25 return Point(x,y)
```

```

26
27 def __str__(self):
28 """
29 Objective: To return string representation of object of
30 type Point
31 Input Parameter: self (implicit parameter)- object of type
32 Point
33 Return Value: string
34 """
35 return str((self.x,self.y))

```

**Fig. 11.3** Class Point (point.py)

Next, we create the objects point1 and point2 of the class Point, and apply the + operator on these objects as follows:

```

>>> point1 = Point(3,6)

>>> point2 = Point(2,1)

>>> print(point1 + point2)

(5,7)

```

#### *Comparing Dates*

In addition to the special methods discussed above, Python also provides special methods such as `__eq__`, `__lt__`, `__le__`, `__gt__`, and `__ge__` for overloading comparison operators `==`, `<`, `<=`, `>`, and `>=` respectively. Suppose we wish to compare two dates. Let us define two objects of MyDate class and compare them as follows:

```

>>> date1 = MyDate(31,12,2014)

>>> id(date1)

49078096

>>> date2 = MyDate(31,12,2014)

```

```
>>> id(date2)
10775792

>>> date1 == date2

False
```

Note that in spite of two dates being same, the comparison yields `False` as the result. It is so because the default implementation of

`__eq__` method compares ids of the two objects. So, we define our own implementation of the equality operator `==` for the class `MyDate` (Fig. 11.4) that would consider two dates to be equal if and only if they agree on each of the day, month, and year components. In line 10 (Fig. 11.4) we invoke the built-in function `isinstance` to check whether the second object received as a parameter is an object of `MyDate` class. If the condition holds `True`, it returns the result of the comparison. Otherwise, it displays the message 'Type Mismatch' and the program terminates. The method `__ne__` corresponding to the operator `!=` may be developed on similar lines.

Next, we illustrate the use of the method `__eq__`

default implementation of `__eq__()` compares ids of two objects

overloading operator `==` for comparing two dates

define method `__ne__` for overloading operator `!=`

if `__ne__()` is not defined in a class, then usually, it returns not `__eq__()`

```
>>> date1 = MyDate(31,12,2014)
```

```
>>> date2 = MyDate(31,12,2014)
```

```
>>> date3 = MyDate(1,12,2014)
```

```
>>> date1 == date2
```

```
True
```

```
>>> date1 == date3
```

```
False
```

```
01 def __eq__(self, other):
02 '''
03 Objective: To compare two MyDate objects for equality
04 Input Parameters:
05 self (implicit parameter) - object of type MyDate
06 other - object of type MyDate
07 Return Value: True if equal else False
08 '''
09 # check for equality of year, month, day
10 if isinstance(other, MyDate):
11 return(self.day == other.day) and (self.month == \
12 other.month) and (self.year == other.year)
13 else:
14 print('Type Mismatch')
15 sys.exit()
```

**Fig. 11.4** Class methods `__eq__`

Next, we present the implementation of `__lt__` method for the `MyDate` class (see Fig. 11.5). This method may be used to sort a list of objects of type `Date`.

overloading operator < for comparing date objects

```
>>> dates = [MyDate(31,12,2014),
 MyDate(31,10,2014), MyDate(17,12,2014)]
```

```
>>> dates.sort()
```

```
>>> for date in dates:
```

```
 print(date)
```

31-10-2014

17-12-2014

31-12-2014

```
01 def __lt__(self, other):
02 """
03 Objective: To compare two MyDate objects
04 Input Parameters:
05 self (implicit parameter) - object of type MyDate
06 other - object of type MyDate
07 Return Value: True if value of first MyDate object is less
08 than seconds, else False
09 """
10 # compare by year, month, day successively
```

```

11 if isinstance(other, MyDate):
12 if self.year < other.year:
13 answer = True
14 elif self.year == other.year and \
15 self.month < other.month:
16 answer = True
17 elif self.year == other.year and self.month == other.month\
18 and self.day < other.day:
19 answer = True
20 else:
21 answer = False
22 return answer
23 else:
24 print('Type Mismatch')
25 sys.exit()

```

**Fig. 11.5** Date class methods `__lt__`

When we invoke the `sort` function with a list of objects of type `Date`, it makes use of the method `__lt__` defined for the `Date` class.

### 11.1.2 Function Overloading

Function overloading provides the ability of writing different functions having the same name, but with a different number of parameters, possibly of the various types. For example, we may like to define two functions by the name `area` to compute the area of a circle or rectangle:

function overloading: different functions having the same name but different number and type of parameters

```
def area(radius):

 # Computes the area of circle

 areaCirc = 3.14 * radius * radius

 return areaCirc

def area(length, breadth):

 # Computes the area of rectangle

 areaRect = length * breadth

 return areaRect
```

When we require a function to behave differently depending on the number and/or type of parameters, we like to define separate functions that have the same name, but can be differentiated by the number and type of parameters. However, Python does not support function overloading. Indeed, Python cannot distinguish between parameters based on their type as the type of a parameter is inferred implicitly when a function is invoked with actual arguments. Further, when we define a number of functions that have the same name, Python retains only the most recent definition. Thus, whereas invoking the function `area` with a single argument results in error, invoking it with two arguments yields the expected output as illustrated below:

Python does not support function overloading

out of all the definitions in a scope having the same function name, Python retains the most recent definition

```
>>> area(4)
```

```
Traceback (most recent call last):

 File "<pyshell#3>", line 1, in
<module>

 area(4)

TypeError: area() missing 1 required
positional argument: 'breadth'

>>> area(4, 5)

20
```

Method overloading may be implemented indirectly, via use of default parameters. Let us assume that we wish the same function named `area` to compute the area of each of the circle and rectangle depending upon the number of arguments passed. We may define it as given below:

```
indirect implementation of method overloading

def area(a, b = None):
 """
 Objective: To compute area of a circle or
 rectangle depending on the number of parameters.

 Inputs:
 a: radius, in the case of a circle side1, in case
 of rectangle
 b: None, in the case of a circle side2, in the
 case of rectangle
 Output: area of circle or rectangle as applicable
 """

 if b == None:
 # Computes the area of circle
 areaCirc = 3.14 * a * a
```

```

 return areaCirc
else:

 # Computes the area of rectangle
 areaRect = a * b

 return areaRect

```

Thus, when the function `area` is invoked with two arguments, it computes the area of the rectangle, and when it is invoked with only one argument, it computes the area of the circle.

#### 11.2 ENCAPSULATION, DATA HIDING, AND DATA ABSTRACTION

*Encapsulation* enables us to group together related data and its associated functions under one name. Classes provide an *abstraction* where essential features of the real world are represented in the form of interfaces, *hiding* the lower-level complexities of implementation details. Object-oriented languages such as C++ restrict access to data attributes of an instance or a class outside the class. However, Python permits us to do so freely, for example:

encapsulation: grouping together related data and its associated functions under one name

abstraction: representing essential features of the real world, hiding lower level details

```
today = MyDate(31,1,2014)
```

```
today.day = 15
```

Note that setting attribute `day` of the object `today` is a violation of the principle of data abstraction, which states that the data attributes of the instance variables should only be accessed by the methods of the class. Python allows us to prevent accidental access to the data and method attributes from outside the class via the

notion of private members. We can tell Python that an attribute is a private attribute by prefixing the attribute name by at least two consecutive underscore characters. Further, the attribute name should not have more than one underscore character at the end. This technique of restricting access to private members from outside the class is known as *name mangling*. For example, in the class `MyDate`, to indicate that the attributes `day`, `month`, and `year` are private members of the class, we would prefix each of these attribute names by two consecutive underscore characters `__`, thus replacing the attribute names `day`, `month`, and `year` by `__day`, `__month`, and `__year`, respectively. Now, these attributes are not directly accessible from outside the class, and an attempt to access them will result in error. For example, an attempt to access `today.__day` from outside the scope of the class `MyDate` generates an error message indicating that no attribute with name `__day` exists:

accessing data and method attributes outside the class is a violation of the principle of abstraction

name mangling: a technique for defining private attributes

```
>>> today = MyDate(3, 9, 2014)

>>> print(today.__day)

Traceback (most recent call last):

 File "<pyshell#1>", line 1, in
<module>

 print(today.__day)

AttributeError: 'MyDate' object has no
attribute '__day'
```

However, access to this attribute within the class will continue to be allowed using notation `self.__day`. The restriction on the use of private variables from outside the scope of the class may be bypassed using the syntax:

syntax for accessing private members from outside of the class

`<instance>._<className><attributeName>`

Thus, the attribute `__day` of the object `today` may be accessed as `today._MyDate__day`. However, in accordance with the principle of data abstraction, we suggest that the data attributes of a class should be accessed only through class methods by defining the operations on the data attributes in the form of methods. Next, suppose we wish to add an attribute `weekDay` to the namespace of the object `today` of the class `MyDate`. For doing so, we may use the following statement:

adding an attribute to the namespace of an object

`today.weekDay = 'Wednesday'`

However, the inclusion of the attribute `weekDay` in the namespace of `today` will not affect the namespaces of other objects of the class `MyDate`, which might have been created already or may be created later. Since the class `MyDate` does not have any interface to deal with the new attribute `weekDay`, all modifications to this attribute must be applied by the programmer explicitly. Again, one should normally define all the attributes in the class definition only. The inclusion of an attribute such as `weekDay` for an instance of the class should only be an exception.

The methods in the class definition that modify the value of one or more arguments are known as modifiers. For example, the methods `setAddress` and `setName` in the class `Person` are modifiers since they change the values of the data attributes `address` and `name` respectively associated with parameter `self`. However, methods such as `getAddress` and `getName` of the class `MyDate` only access (do not modify) the data attributes `address` and `name` of the object `self`. Such methods are known as accessors.

modifiers: methods that modify the arguments

accessors: only access (do not modify) the arguments

#### 11.4 STATIC METHOD

In the methods discussed so far, the object that invokes the method is passed as the first implicit argument (`self`). These methods are called instance methods. An instance method typically defines operations on the data members of an instance of a class. However, there are situations when we want to define a method for a class to modify the class data members. Such a method does not require a class object to be passed as the first parameter and is called a static method. A static method is invoked as an attribute of a class. It is usually invoked as `className.staticMethodName`.

instance methods: invoked for the particular instance of the class, which is passed as the first argument (corresponding to `self`)

static methods: used for modifying class data members and do not require passing object as the first parameter

Let us examine the class `Person` once again. To maintain uniformity of naming conventions used in this chapter, we shall rename the variable `count` as `personCount`. Thus, in this class, we use the class variable `personCount` in the method `__init__` to keep track of the number of instances of `Person`.

The value of the variable `personCount` is incremented by one every time an instance of `Person` is created. However, since the information `personCount` is not associated with any specific instance of the class `Person`, it should be accessed using a static method without reference to a specific instance of the class `Person`. In the script `person` (Fig. 11.6), we define two static methods, namely, `incPersonCount` and `getPersonCount`. The `incPersonCount` method is invoked by `__init__` and increments `personCount` by one whenever an instance of the class is created. The method `getPersonCount` (replacement for function `getCount`) may be used to retrieve the value of data member `personCount`. To tell Python that a method is a static method, the function decorator `@staticmethod` precedes the method definition.

`@staticmethod`: decorator that precedes the definition of a static method

```
01
02 class Person:
03
04 personCount = 0
05 def __init__(self, name, DOB, address):
06 """
07 Objective: To initialize object of class Person
08 Input Parameters:
09 self (implicit parameter) - object of type Person
10 name - string
11 DOB - string (Date of Birth)
12 address - string
13 Return Value: None
```

```

14 """
15 self.name = name
16 self.DOB = DOB
17 self.address = address
18
19 Person.incPersonCount()
20
21 @staticmethod
22 def incPersonCount():
23 """
24 Objective: To increment value of class variable personCount
25 Input Parameter: None
26 Return Value: None
27 """
28 Person.personCount += 1
29
30 @staticmethod
31 def getPersonCount():
32 """
33 Objective: To determine value of class variable personCount
34 Input Parameter: None
35 Return Value: personCount
36 """
37 return Person.personCount

```

**Fig. 11.6** Class Person (person.py)

The following instructions illustrate the use of static methods:

```

>>> p1 = Person('Smita', '20 May,1980',
'House-177 Block-10, Shalimar Road,
Delhi')

>>> p2 = Person('Aarushi', '20
April,1990', 'House-175 Block-10, Shalimar
Road, Delhi')

```

```
>>> Person.getPersonCount()
```

2

#### 11.5 ADDING METHODS DYNAMICALLY

Once all the methods of the class have been defined, one may realize the need to add another method to the class or a particular instance of the class. Python allows us to add methods dynamically to a class using the syntax:

syntax for adding methods dynamically to a class

```
<className>. <newMethodName> =
<existingFunctionName>
```

For the sake of an illustration, we define a class `Student` having only the constructor method `__init__` (Fig. 11.7). The script also includes the methods `percentage` and `result`. However, these methods do not belong to the class `Student`. The following instruction will add the method `percentage` to the class.

```
>>> Student.percentage = percentage
```

```
01 from types import MethodType
02
03 class Student:
04 maxMarks = 500
05
06 def __init__(self, name, rollNum, marks):
07 """
08 Objective: To initialize object of class Student
09 Input Parameters:
10 self (implicit parameter) - object of type Student
11 name - string
12 rollNum, totalMarks - numeric value
13 Return Value: None
14 """
15 self.name = name
16 self.rollNum = rollNum
17 self.marks = marks
18
19
20 def percentage(self):
```

```
21 """
22 Objective: To compute the percentage of marks obtained by a
23 student
24 Input Parameter:
25 self (implicit parameter) - object of type Student
26 Return Value: percent
27 """
28 percent = (float(self.marks) / Student.maxMarks) * 100
29 return percent
30
31 def result(self):
32 """
33 Objective: To compute the result of a student
34 Pass if percetage>=40, fail otherwise
35 Input Parameter:
36 self (implicit parameter) - object of type Student
37 Return Value: string
38 """
39 return 'pass' if self.percentage() >= 40 else 'fail'
```

**Fig. 11.7** Class Student (student.py)

The method `percentage` can now be invoked like any other method of the class `Student` as shown below:

```
>>> s1 = Student('Amey', 47, 450)
>>> print(s1.percentage())
```

90.0

Instead, if we were to associate a method with a particular instance of a class, we need to import the function `MethodType` from the module `types` (line 1, Fig. 11.7). Subsequently, we use the following syntax to add a method to an instance of a class:

syntax for adding methods dynamically to an instance of a class

```
<instance>.
<newMethodName>=MethodType (<existingFunctionName>, <instance>)
```

Now we are ready to add the method `result` to the instance `s1` of the class `Student`.

```
>>> s1.result = MethodType(result, s1)

>>> s1.result()

'pass'
```

Invoking this method from a class instance to which we have not added the `result` method will lead to an error:

```
>>> s2 = Student('Vihan', 23, 490)
```

```
>>> s2.percentage()
```

98.0

```
>>> s2.result()
```

Traceback (most recent call last):

```
 File "<pyshell#99>", line 1, in
<module>
```

```
 s2.result()
```

```
AttributeError: 'Student' object has no
attribute 'result'
```

#### 11.6 COMPOSITION

In this section, we will refine the class `Person`, introduced in the last chapter, that comprised the instance attributes `name`, `DOB`, and `address` in addition to the class attribute `personCount`. While creating objects of class `Person`, we used string objects such as '`24-10-1990`' as values of the date of birth (`DOB`). In the last chapter, we also defined `MyDate` class comprising three components `day`, `month`, and `year`. To use this representation of the date in the class `Person`, we only need to import the `MyDate` class in the script, and there is no change in the description of class `Person` (Fig. 11.8).

```
01 from date import MyDate
02 class Person:
03 """
04 Copy details from Figure 10.1
05 """
06
```

**Fig. 11.8** Class `Person` (`person.py`)

Now we illustrate the use of `MyDate` class to create instances of `Person` class:

```
>>> dob = MyDate(24,10,1990)

>>> p1 = Person('Rajat Mittal', dob, \
23,Malviya Nagar,Delhi')

>>> print(p1)

Name:Rajat Mittal
```

DOB: 24-10-1990

Address: B-23, Malviya Nagar, Delhi

Note that data attribute `dob` of object `p1` is of type `MyDate`. Here, we have used an object of another class `MyDate` as an attribute of the class `Person`. The process of using objects of the other classes as attribute values is called object *composition*. It is important to observe that we have already used this concept subconsciously. While creating objects of class `Person` earlier, we used objects of system-defined class `str` as values of attributes `name`, `DOB`, and `address`. We could have, equally well, used the object `MyDate(24, 10, 1990)` directly as an argument, while invoking the constructor, instead of first creating it as `dob`, as shown below:

using objects of other classes as attributes

```
>> p1 = Person('Rajat Mittal',
 MyDate(24,10,1990), 'B-23, Malviya
 Nagar, Delhi')
```

#### 11.7 INHERITANCE

Inheritance is an important feature of object oriented programming that imparts ability to a class to inherit properties and behavior of another class. The class `object` includes a rich set of special Python functions as seen below:

inheriting properties and behavior of another class

```
>>> dir(object)
```

```
['__class__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__',
 '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
```

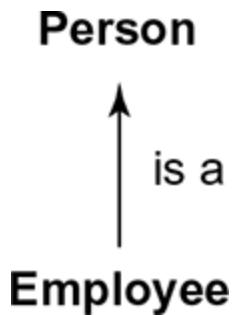
It is interesting to note that various types such as `int`, `list`, `tuple`, and `dict` have some common methods such as `__eq__`, `__lt__`, and `__str__`. These types have derived these common methods (attributes) from the `object` class. The `object` class serves as the base class of each of these classes.

`object` class is the base class of all classes

Suppose we want to develop a program that deals with employees in an organization. Data attributes of an employee may comprise his/her name, address, date of birth, employee id, and salary. As a first reaction, one may wish to create class `Employee` having these attributes and define all the methods required for this class. However, notice that every employee is a person, and we have already done much hard work to develop the `Person` class. It makes sense to design the `Employee` class as an extension of the `Person` class. In this way, all the data and method attributes of the `Person` class would be available to `Employee` class. In the language of Object-oriented Programming (OOP), we say that `Employee` class inherits or derives the data and method attributes from the `Person` class. Here, `Person` class is called *base*, *super*, or *parent* class, and `Employee` class is called *derived*, *sub*, or *child* class. Thus, the notion of inheritance allows us to express the parent-child relationship among the classes as shown in Fig. 11.9.

relationship between base class and derived class

"is a" relationship



**Fig. 11.9** Parent and child class

#### 11.7.1 Single Inheritance

When inheritance involves a derived class that derives its properties from a single base class (see Fig. 11.9), it is called *single inheritance*. As another example, think of a manager in a factory. The manager would have all the attributes of an employee and may have some new attributes like managerial pay. This relationship is expressed in Fig. 11.10.

Manager class derives properties from the base class Employee



**Fig. 11.10** Employee and manager class

Details of the class `Employee` appear in Fig. 11.11. As the `Employee` class is a derived class of the base class `Person`, we import the base class `Person` from module `person` (stored as `person.py` in the current directory). Next, we must tell Python that the class `Employee` is a derived class of base class `Person`. We do this by enclosing the name of base class `Person` in parenthesis following the name of derived class `Employee` (line 3, Fig. 11.11). Based on the assumption that the *Employee* ids begin with 1001, the class attribute `nextId` is initialized as 1001 and is incremented by the constructor method whenever an instance of class `Employee` is created.

```
01 from person import Person
02 from date import MyDate
03 class Employee(Person):
04 nextId = 1001
05 employeeCount = 0
06
07 def __init__(self, name, DOB, address, basicSalary, dateOfJoining):
08 """
09 Objective: To initialize an object of class Employee
10 Input Parameters:
```

```
11 self (implicit parameter) - object of type Employee
12 name - string, address - string
13 DOB - Date of Birth - object of type MyDate
14 basicSalary - numeric value
15 dateOfJoining - object of type MyDate
16 Return Value: None
17 """
18 Person.__init__(self, name, DOB, address)
19 self.idNum = Employee.nextId
20 Employee.nextId += 1
```

```
19 self.idNum = Employee.nextId
20 self.basicSalary = basicSalary
21 self.dateOfJoining = dateOfJoining
22 Employee.nextId += 1
23 Employee.employeeCount += 1
24
25 def getId(self):
26 """
27 Objective: To retrieve id of the Employee
28 Input Parameter: self (implicit parameter)- object of type
29 Employee
30 Return Value: id - numeric value
31 """
32
33 def getSalary(self):
34 """
35 Objective: To retrieve salary of the Employee
36 Input Parameter: self (implicit parameter) - object of type
37 Employee
38 Return Value: basicSalary - numeric value
39 """
40
41 def reviseSalary(self, newSalary):
42 """
43 Objective: To update salary of the Employee
44 Input Parameters: self (implicit parameter) - object of
45 type Employee
46 newSalary - numeric value
47 Return Value: None
48 """
49
50 def getJoiningDate(self):
```

```
51 """
52 Objective: To retrieve joining date of the Employee
53 Input Parameter: self (implicit parameter) - object of
54 type Employee
55 Return Value: dateOfJoining - object of type MyDate
56 """
```

```

56 return self.dateOfJoining
57
58 def __str__(self):
59 """
60 Objective: To return string representation of object of type
61 Employee.
62 Input Parameter: self (implicit parameter) - object of
63 type Employee
64 Return Value: string
65 """
66 return Person.__str__(self)+'\nId:'+str(self.getId())+\
67 '\nSalary:'+str(self.getSalary())+\
68 '\nDate of Joining:'+str(self.getJoiningDate())
69
70 def __lt__(self, other):
71 """
72 Objective: To check if id of the first employee is less
73 than the other
74 Input Parameters:
75 self (implicit parameter) - object of type Employee
76 other - object of type Employee
77 Return Value: True if self.getId() < other.getId(), else False
78 """
79 return self.getId() < other.getId()
80
81 def __del__(self):
82 """
83 Objective: To be invoked on deletion of an instance of a class
84 Employee
85 Input Parameter:
86 self (implicit parameter) - object of type Employee
87 Return Value: None
88 """
89 Person.__del__(self)
90 Employee.employeeCount -= 1

```

**Fig. 11.11** Class Employee (employee.py)

As the Employee class is a derived class of the base class Person, all the data and method attributes defined in the Person class become available to the Employee class. Recall that we had defined the methods

`__init__`, `__str__`, and `__del__` in the class `Person`. However, we need to redefine these methods in the derived class `Employee`. When an object of a derived class makes a call to a method that is also defined in the base class, Python invokes the method of derived class. Thus, the derived class methods override the base class methods, and this process is known as method *overriding*.

#### method overriding: redefining a method in the derived class

The constructor for the `Employee` class, needs to initialize the variables of the `Person` class as well as the new variables introduced in the `Employee` class. Recall that we say that `Employee` is a `Person`. Therefore, to initialize an object of class `Employee`, we invoke the constructor of class `Person` that initializes the `Person` class variables for an object of the `Employee` class. For this purpose, we pass `self` (`self` refers to an object of `Employee` class) as an argument while invoking the `Person` class constructor. To be more specific, invoking the constructor of the `Person` class in the `Employee` class constructor does not create a new instance of the `Person` class, instead it only initializes the variables of the `Person` class for the `Employee` class object.

Within the definition of the constructor for class `Employee`, if we attempt to invoke `__init__`, it would lead to a recursive call to itself. So we qualify `__init__`, by prefixing the name of the base class, as `Person.__init__`. Similarly, in line 65, the method `Person.__str__` of the base class is invoked to return the string representation for the inherited variables. Note that invoking the method `Person.__init__` increments the class variable `personCount` of the class `Person`.

#### invoking a base class method

In the destructor method `__del__` of class `Employee`, the class variable `employeeCount` would be decremented by 1. Recall that the variables of the `Person` class for the `Employee` class object were also initialized while creating the `Employee` class object, thus, when `__del__` method of class `Employee` is invoked, we invoke the destructor method `Person.__del__` of the base class `Person` (line 86). Consequently, the class variable `personCount` of the base class `Person` would also get decremented by 1. This is illustrated by the following example:

```
>>> person1 = Person('Shyam Kapoor',
MyDate(24,10,1990), 'B-23, Malviya Nagar,
Delhi')

'B-23,Malviya Nagar,Delhi')

>>> print(Person.personCount)

1

>>> emp1 = Employee('Rehman',
MyDate(5,6,1970),

'D-9, Vivek Vihar, Delhi', 50000,
MyDate(2,8,2013))

>>> print(Employee.employeeCount)

1

>>> print(Person.personCount)

2

>>> del emp1

Deleted!!

>>> print(Employee.employeeCount)
```

0

```
>>> print(Person.personCount)
```

1

In line 18 (Fig. 11.11), call to the method `__init__` is made using a superclass name and the object instance is explicitly passed as an argument to the superclass method. Alternatively, we may use the `super` function to access a method of the superclass. Thus, line 18 (Fig. 11.11) can be replaced by the following statement:

using `super` function for accessing a method of a base class

```
super(Employee, self).__init__(name, DOB,
address)
```

or

```
super().__init__(name, DOB, address)
```

Similarly, line 86 (Fig. 11.11) can be replaced by the following statement:

```
super(Employee, self).__del__()
```

or

```
super().__del__()
```

#### *Scope Rule*

To access an instance attribute, Python will first look for it in the instance's namespace, then in the class namespace, and then in the superclass namespace recursively going up in the hierarchy. We illustrate this in Fig. 11.12.

scope rule for accessing an instance attribute

```
01 class A:
02 z = 30
03
04 def __init__(self):
05 '''
06 Objective: To initialize object of class A
07 Input Parameter:
08 self (implicit parameter) - object of type A
09 Return Value: None
10 '''
11
12 self.x = 2
13 self.y = 10
14 self.w = 100
15
```

```
16 def __str__(self):
17 '''
18 Objective: To return string representation of object of
19 type A
20 Input Parameter: self (implicit parameter) - object of
21 type A
22 Return Value: string
23 '''
24 return 'w:' + str(self.w) + ', x:' + str(self.x) + ', y:' +\n25 str(self.y) + ', z:' + str(self.z)
26
27 class B(A):
28 # B is subclass of A
29 x = 4
```

```

29 y = 20
30
31 def __init__(self):
32 A.__init__(self)
33 self.x = 6
34 self.v = 50
35
36 def __str__(self):
37 return A.__str__(self) + ', v:' + str(self.v)
38
39
40 def main():
41 """
42 Objective: To illustrate scope of attributes
43 Input Parameter: None
44 Return Value: None
45 """
46 ob1 = A()
47 ob2 = B()
48 print('ob1:\n' + str(ob1))
49 print('ob2:\n' + str(ob2))
50 print('dir(A):\n', dir(A))
51 print('dir(ob1):\n', dir(ob1))
52 print('dir(B):\n', dir(B))
53 print('dir(ob2):\n', dir(ob2))
54
55 if __name__=='__main__':
56 main()

```

**Fig. 11.12** Scope of variables (`scope.py`)

In the script `scope`, we create the objects `ob1` and `ob2` of the classes `A` and `B`, respectively (lines 46 and 47, Fig. 11.12). These objects are displayed using `print` functions in lines 48 and 49. Next, we display the directories of class `A`, object `ob1`, class `B`, and object `ob2` (lines 50–53, Fig. 11.12).

```

ob1:
w:100, x:2, y:10, z:30

ob2:

```

```
w:100, x:6, y:10, z:30, v:50
```

```
dir(A):
```

```
['__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__',
 '__weakref__', 'z']
```

```
dir(ob1):
```

```
['__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__',
 '__weakref__', 'w', 'x', 'y', 'z']
```

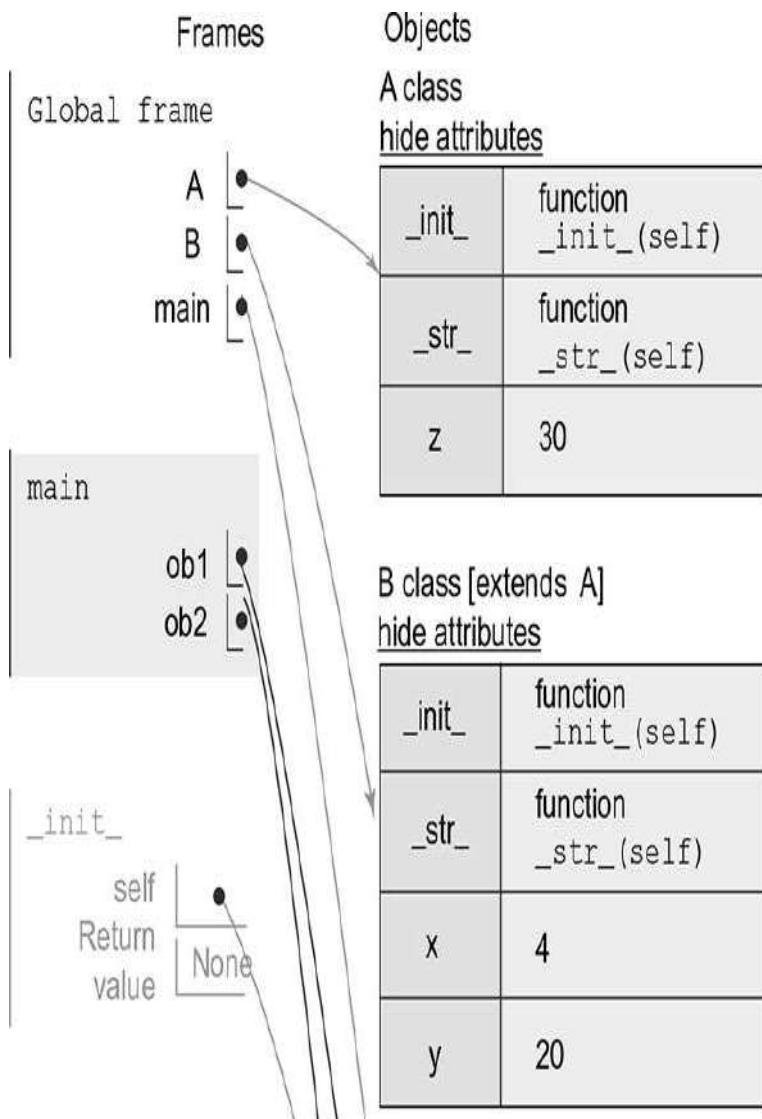
```
dir(B):
```

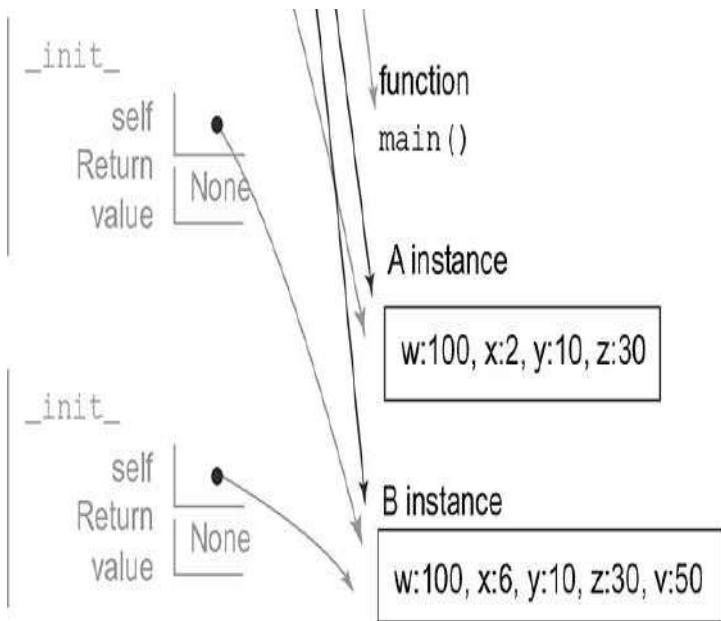
```
['__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__',
 '__weakref__', 'x', 'y', 'z']
```

```
dir(ob2):
```

```
['__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__',
 '__weakref__', 'v', 'w', 'x', 'y', 'z']
```

The values of data attributes of the objects `ob1` and `ob2` can be read from the namespaces of these objects (Fig. 11.13).





**Fig. 11.13** Representation of class A, class B, object `ob1`, and object `ob2`

In the expression '`ob1:\n' + str(ob1)`', when the `__str__` method is invoked, it uses the values of data attributes `x`, `y`, and `w` directly from the namespace of object `ob1`. The data attribute `ob1.z` is also accessible as it is defined as a class variable of the class `A` of which `ob1` is an object. According to scope rule, an attribute is first searched in object's namespace. Therefore, in the expression '`ob2:\n' + str(ob2)`', when the `__str__` method is invoked, it uses the values of data members `x` and `v` directly from the namespace of the object `ob2` and the variables `x` and `v` are bound to values 6 and 50, respectively. The attribute `y` is an instance attribute inherited from the parent class `A` and is bound to value 10. Similarly, the attribute `w` is an instance attribute inherited from the superclass `A` and is bound to value 100. However, for accessing `z`, Python first searches in object `ob2`'s namespace, then in its class `B`'s namespace, and finally in superclass `A`'s namespace, where it is found and gets bound to value 30.

We may obtain the name(s) of the class(es) from which a class inherits properties as follows:

`__bases__`: used to find base class(es) of a class

```
>>> str.__bases__
```

```
(<class 'object'>,)
```

```
>>> B.__bases__
```

```
(<class '__main__.A'>,)
```

```
>>> B.__bases__[0]
```

```
<class '__main__.A'>
```

Note that class `B` inherits from only one class, i.e. `A`.

Therefore, tuple `B.__bases__` contains only one class, i.e. `<class '__main__.A'>`. Given an object, its class name may be obtained as shown below:

`__name__`: used to retrieve the name of a class

```
>>> 'Python'.__class__.__name__
```

```
'str'
```

### *Extending Scope of `int` Class Using a User Defined Class*

Built-in methods such as `__add__`, `__sub__`, `__mult__`, and `__div__` are already defined for objects of class `int`. However, there is no implementation of method `len`. We define the length of an integer `x` to be the number of digits in `abs(x)`. For this purpose, we define a class `MyInt` as a derived class of Python class `int` (Fig. 11.14) and define a method `len` that computes the length of an integer.

extending `int` class as `MyInt` and defining a method `len` for it

```
01 class MyInt(int):
02
03 def __len__(self):
04 """
05 Objective: To find the number of digits in an integer
06 Examples: len(2695) --> 4, len(-2695) --> 4
07 Input Parameter:
08 self (implicit parameter) - object of type MyInt
09 Return Value: digits - numeric
10 """
11
12 num = self
13
14 if num < 0:
15
16 num = -num
17
18 digits = 0
19
20 while num > 0:
21
22 num = num // 10
23
24 digits += 1
25
26 return digits if digits > 0 else 1 # return 1 if number ==0
```

**Fig. 11.14** MyInt class (myint.py)

Next, we illustrate the use of `MyInt` class to display the number of digits in an integer:

```
>>> n1 = MyInt(3456)

>>> n2 = MyInt(-18)

>>> type(n1), type(n2)
```

```
(<class '__main__.MyInt'>, <class
'__main__.MyInt'>)

>>> len(n1)

4

>>> len(n2)

2

>>> len(MyInt(0))

1
```

Note that the methods of the `int` class such as `__add__`, `__sub__`, `__mult__`, and `__div__` can still be applied to objects of type `MyInt` as it inherits all the attributes of the `int` class. The following examples illustrate this:

```
>>> n1 + n2
```

```
3438
```

```
>>> n1 - n2
```

```
3474
```

### 11.7.2 Hierarchical Inheritance

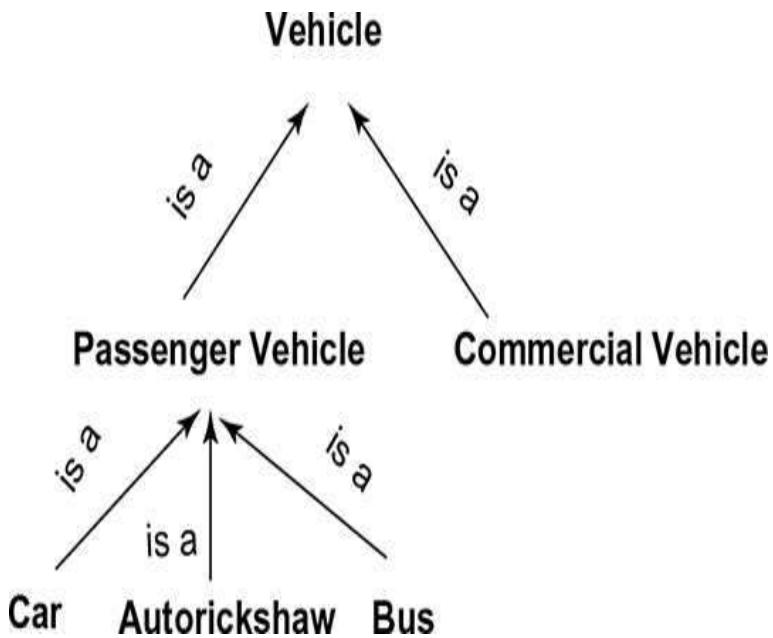
In Section 11.7.1, we discussed the derived classes. Further, a derived class is like any other class and may serve as base class for another class derived from it and so on. Each derived class may define its new attributes. Thus, inheritance allows us to create a class hierarchy, also called type hierarchy. When inheritance involves more than one level of hierarchy such as `A -> B -> C`, it is called multilevel inheritance, and inheritance involving multiple classes deriving from single base class is known as hierarchical inheritance. For example, examine the base class `Vehicle`, having attributes like registration number, make, model, and color. From this class, we may derive classes `PassengerVehicle` and

CommercialVehicle. The class PassengerVehicle may have additional attributes like maximum passenger capacity. The class CommercialVehicle, may have other attributes like maximum load capacity. From the PassengerVehicle class, we may derive classes Car, Autorickshaw, and Bus. The Car and Bus may have attributes like the number of doors, not shared by Autorickshaw. The Bus may have a Boolean attribute doubleDecker not shared by Car and Autorickshaw. Note that every car has all the attributes of a PassengerVehicle, and every PassengerVehicle has all the attributes of a Vehicle. So we can say Car is a PassengerVehicle and PassengerVehicle is a Vehicle. This relationship between the derived class and the base class is called *is an/is a* relationship. The relationships among Vehicle, PassengerVehicle, Car, Autorickshaw, and Bus are shown in Fig. 11.15. The figure demonstrates that several derived classes may inherit from the same base class, for example, the classes PassengerVehicle and CommercialVehicle inherit from the same base class Vehicle, and the classes Car, Autorickshaw, and Bus inherit from the same base class PassengerVehicle.

multilevel inheritance

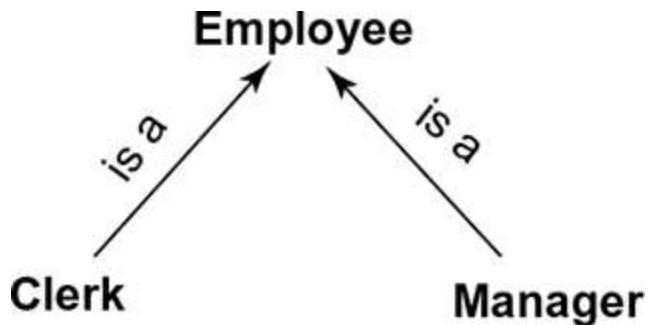
hierarchical inheritance

"is a" relationship



**Fig. 11.15** Example of inheritance hierarchy

As another example of class inheritance, the two classes **Clerk** and **Manager** may be derived from single base class **Employee** (Fig. 11.16). Whereas the **Clerk** class has an additional class attribute **clerkCount**, the **Manager** class has two additional attributes: a class attribute **managerCount** and an instance attribute **managerialPay**. The complete definition of these classes is given in Fig. 11.17. Note that class **Manager** defines method **getSalary** that overrides the superclass method **getSalary** and computes the manager's salary by adding **managerialPay** to an employee's **Salary**.



**Fig. 11.16** Inheritance relationship between **Employee**, **Clerk**, and **Manager**

Next, let us study some examples:

```
>>> clerk1 = Clerk('Arun',
MyDate(5,6,1930),
'D-9, Sarojni Nagar, Delhi', 20000,
MyDate(5,6,1950))
```

```
>>> print(clerk1)
```

Name:Arun

DOB:05-06-1930

Address:D-9, Sarojni Nagar, Delhi

Id:1001

Salary:20000

Date of Joining:05-06-1950

```
01 from employee import Employee
02 from date import MyDate
03 import sys
04 class Clerk(Employee):
05 clerkCount = 0
06 def __init__(self, name, DOB, address, basicSalary, dateOfJoining):
07 """
08 Objective: To initialize object of class Clerk
09 Input Parameters:
10 self (implicit parameter) - object of type Clerk
11 name - string, address - string
12 DOB - Date of Birth - object of type MyDate
13 basicSalary - numeric value
14 dateOfJoining - object of type MyDate
15 Return Value: None
16 """
17 super().__init__(name, DOB, address, basicSalary, \
18 dateOfJoining)
19 Clerk.clerkCount += 1
20
21 def __del__(self):
22 """
```

```
23 Objective: To be invoked on deletion of an instance of a
24 class Clerk
25 Input Parameter:
26 self (implicit parameter) - object of type Clerk
27 Return Value: None
28 """
29 super().__del__()
30 Clerk.clerkCount -= 1
31
32
33 managerCount = 0
34 def __init__(self, name, DOB, address, basicSalary, dateOfJoining, \
35 managerialPay):
36 """
37 Objective: To initialize object of class Manager
38 Input Parameters:
39 self (implicit parameter) - object of type Manager
40 name - string, address - string
```

```
41 DOB - Date of Birth - object of type MyDate
42 basicSalary, managerialPay - numeric value
43 dateOfJoining - object of type MyDate
44 Return Value: None
45 """
46 super().__init__(name, DOB, address, basicSalary, \
47 dateOfJoining)
48 self.managerialPay = managerialPay
49 Manager.managerCount += 1
50
51 def updateManagerialPay(self, newManagerialPay):
52 """
53 Objective: To update managerialPay of the Manager
54 Input Parameters:
55 self (implicit parameter) - object of type Manager
56 newManagerialPay - numeric value
57 Return Value: None
58 """
59 self.managerialPay = newManagerialPay
60
61 def getSalary(self):
62 """
```

```

63 Objective: To get salary of the Manager
64 Input Parameter:
65 self (implicit parameter) - object of type Manager
66 Return Value: salary - numeric value
67 """
68 return Employee.getSalary(self)+self.managerialPay
69
70 def __str__(self):
71 """
72 Objective: To return string representation of object of type
73 Manager
74 Input Parameter:
75 self (implicit parameter) - object of type Manager
76 Return Value: string
77 """
78 returnEmployee.__str__(self)+'\nManagerialPay:' +str(self.
79 managerialPay)
80
81 def __del__(self):
82 """
83 Objective: To be invoked on deletion of an instance of a
84 class Manager
85 Input Parameter:

```

```

83 self (implicit parameter) - object of type Manager
84 Return Value: None
85 """
86 super().__del__()
87 Manager.managerCount -= 1

```

**Fig. 11.17** Classes Clerk and Manager

Since there is no `__str__` method in the class `Clerk`, to print the object `clerk1`, `__str__` method of the superclass `Employee` is invoked. Also, because Arun is the first employee, Id 1001 is assigned to him. Next, we create an object of class `Manager`:

invoking a method of a superclass

```
>>> manager1 = Manager('Rehman',
MyDate(5, 6, 1970), 'D-9, Vivek Vihar,
Delhi',
50000, MyDate(5, 6, 1990), 2000)
```

```
>>> print(manager1)
```

Name:Rehman

DOB:05-06-1970

Address:D-9, Vivek Vihar, Delhi

Id:1002

Salary:52000

Date of Joining:05-06-1990

ManagerialPay:2000

As the class Manager has its own `__str__` method (lines 70–78), it will be invoked to print `manager1`. As Manager is an employee, the method `__str__` of Manager class concatenates

'\nManagerialPay:' + str(self.managerialPay) to Employee.`__str__(self)` to return string representation of an object of the class Manager. Next, let us print salary (`basicSalary + managerialPay`) of `manager1`:

```
>>> print(manager1.getSalary())
```

52000

As a Manager is also an Employee, the Manager class method `getSalary` invokes the method `getSalary` of the Employee class to compute the `basicSalary` of

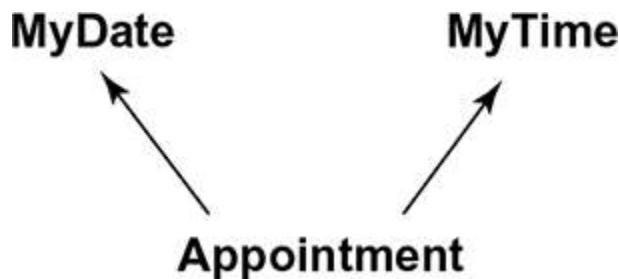
the Manager and adds managerialPay to compute the salary of the manager.

### 11.7.3 Multiple Inheritance

In the case of multiple inheritance, a subclass derives its attributes from two or more classes. In this section, we make use of multiple inheritance to schedule appointments by creating a class Appointment. Since an appointment will comprise date and time, we may inherit the classes MyDate and MyTime in the class Appointment (Fig. 11.18).

a subclass may derive its properties from two or more classes

the class Appointment derives properties from two classes:  
MyDate and MyTime



**Fig. 11.18** Classes Clerk and Manager

We have already developed the class MyDate. In Fig. 11.19, we give details of the class MyTime having data attributes hours, minutes, and seconds.

```
01 import sys
02 class MyTime:
03 def __init__(self, hours = 0, minutes = 0, seconds = 0):
04 """
05 Initiates the initializing data members of an object of class
```

```
05 Objective: To initialize data members of an object of class
06 MyTime
07
08 Input Parameters:
09 self (implicit parameter) - object of type MyTime
10 hours, minutes, seconds - numeric
11
12 Return Value: None
13
14 """
15
16 def setHours(self, hours):
17 """
18 Objective: To set hours of object of class MyTime
19 Input Parameters:
20 self (implicit parameter) - object of type MyTime
21 hours - numeric
22 Return Value: None
23
24 if hours >= 0 and hours <= 23:
25 self.hours = hours
26 else:
```

```
27 print('Invalid value for hours')
28 sys.exit()
29
30 def setMinutes(self, minutes):
31 """
```

```
32 Objective: To set minutes of object of class MyTime
33 Input Parameters:
34 self (implicit parameter) - object of type MyTime
35 minutes - numeric
36 Return Value: None
37 """
38 if minutes >= 0 and minutes <= 59:
39 self.minutes = minutes
40 else:
41 print('Invalid value for minutes')
42 sys.exit()
43
44 def setSeconds(self, seconds):
45 """
46 Objective: To set seconds of object of class MyTime
47 Input Parameters:
48 self - object of type MyTime
49 seconds - numeric
50 Return Value: None
51 """
52 if seconds >= 0 and seconds <= 59:
53 self.seconds = seconds
54 else:
55 print('Invalid value for seconds')
56 sys.exit()
57
58 def __str__(self):
59 """
60 Objective: To return string representation of object of
61 type MyTime
62 Input Parameter:
63 self (implicit parameter) - object of type MyTime
64 Return Value: string
65 """
66
67 if self.hours <= 9:
68 hours = '0' + str(self.hours)
69 else:
```

```
69 hours = str(self.hours)
70 if self.minutes <= 9:
71 minutes = '0' + str(self.minutes)
72 else:
73 minutes = str(self.minutes)
74 if self.seconds <= 9:
75 seconds = '0' + str(self.seconds)
76 else:
77 seconds = str(self.seconds)
78 return hours+':'+minutes+':'+seconds # hh:mm:ss format
```

**Fig. 11.19** Class MyTime (time1.py)

We may create an object of class MyTime as follows:

```
>>> currentTime = MyTime(23, 50, 30)

>>> print(currentTime)

23:50:30
```

Finally, we present the derived class Appointment ([Fig. 11.20](#)).

```
01 import sys
02 from date import MyDate
03 from timel import MyTime
04 class Appointment(MyDate, MyTime):
05
06 def __init__(self, day, month, year, hours, minutes, seconds,\n description):
07 """
08 Objective: To initialize data members of object
09 Input Parameters:
10 self (implicit parameter) - object of type Appointment
11 day, month, year, hours, minutes, year - numeric
12 description - string
13
14 Return Value: None
15 """
16 MyDate.__init__(self, day, month, year)
17 MyTime.__init__(self, hours, minutes, seconds)
18 self.description = description
19
20 def __str__(self):
21 """
22 Objective: To return string representation of object of
23 type Appointment
```

```
23 Input Parameter:
24 self (implicit parameter) - object of type Appointment
25 Return Value: string
26 !!!
27 return MyDate.__str__(self)+', '+MyTime.__str__(self)\\
28 +'\n'+self.description
```

**Fig. 11.20** Class Appointment (appointment.py)

Suppose we wish to create two appointments for the Principal of a college. The first is a meeting with office staff regarding admissions scheduled on 15 July 2014 at 10:00 and the second is a meeting with teachers regarding timetable scheduled on 18 July 2014 at 13:30. For doing so, we may create two objects as follows:

```
>>> meetStaff =Appointment(15, 7, 2014,
10,\n
0, 0,' Meeting with staff regarding
admissions')

>>> meetTeachers =Appointment(18, 7,
2014,\n
13, 30, 0, ' Meeting with teachers
regarding timetable')
```

Details of an appointment may be printed using the `str` representation of an appointment, for example:

```
>>> print(meetStaff)

15-7-2014, 10:00:00

Meeting with staff regarding admissions
```

If required, hierarchical inheritance may be combined with multiple and multilevel inheritance, to yield hybrid inheritance.

#### 11.7.4 Abstract Methods

An abstract method in a base class identifies the functionality that should be implemented by all its subclasses. However, since the implementation of an abstract method would differ from one subclass to another, often the method body comprises just a `pass` statement. Every subclass of the base class will override this method with its implementation. A class containing abstract methods is called abstract class. It is important to point out that although it is not possible to instantiate an abstract class, a subclass of an abstract class that defines the abstract methods may be instantiated as usual.

abstract methods and abstract classes identify functionality that should be implemented by all the subclasses

abstract class: class containing abstract methods

To use Python Abstract Base Classes (ABCs), one needs to import `ABCMeta` and `abstractmethod` from the `abc` module. For example, let us examine the class `Shape` and its subclasses `Rectangle` and `Circle` defined in the script `shape` (Fig. 11.21). Each of the subclasses `Rectangle` and `Circle` needs methods for computing area and perimeter. However, the procedure for computing area and perimeter would differ for `Rectangle` and `Circle`. Therefore, we define the classes `Rectangle` and `Circle` as subclasses of the class `Shape`. We define methods `area` and `perimeter` as abstract methods in the class `Shape`. The subclasses `Rectangle` and `Circle` override the methods `area` and `perimeter` by defining their implementation as shown in Fig. 11.21. If the subclasses `Rectangle` and `Circle` fail to override any of the methods `area` or

perimeter, Python will yield a `TypeError` while instantiating them.

abstract classes cannot be instantiated

`abc` module: provides support for Python abstract base classes

```
01 from abc import ABCMeta, abstractmethod
02 class Shape:
03
04 __metaclass__ = ABCMeta
05
06 def __init__(self, shapeType):
07 """
08 Objective: To initialize object of class Shape
09 Input Parameters:
10 self (implicit parameter) - object of type Shape
11 shapeType - string
12
13 Return Value: None
14
15 """
16
17 self.shapeType = shapeType
18
19 @abstractmethod
20 def area(self):
21 pass
22
23 @abstractmethod
24 def perimeter(self):
25 pass
```

```
22
23 class Rectangle(Shape):
24
25 def __init__(self, length, breadth):
26 """
27 Objective: To initialize object of class Rectangle
28 Input Parameters:
29 self (implicit parameter) - object of type Rectangle
```

```
30 length, breadth - numeric value
31 Return Value: None
32 """
33 Shape.__init__(self, 'Rectangle')
34 self.length = length
35 self.breadth = breadth
36
37 def area(self):
38 """
39 Objective: To compute area of the Rectangle
40 Input Parameter:
41 self (implicit parameter) - object of type Rectangle
42 Return Value: numeric value
43 """
44 return self.length * self.breadth
45
46
47 def perimeter(self):
48 """
49 Objective: To compute perimeter of the Rectangle
50 Input Parameter:
51 self (implicit parameter) - object of type Rectangle
52 Return Value: numeric value
53 """
54 return 2 * (self.length + self.breadth)
55
56
57 class Circle(Shape):
58
59 pi = 3.14
```

```
60 def __init__(self, radius):
61 """
62 Objective: To initialize object of class Circle
63 Input Parameters:
64 self (implicit parameter) - object of type Circle
65 radius - numeric value
66 Return Value: None
67 """
68 Shape.__init__(self, 'Circle')
69 self.radius = radius
70
71 def area(self):
72 """
73 Objective: To compute the area of the Circle
```

```

74 Input Parameter:
75 self (implicit parameter) - object of type Circle
76 Return Value: area - numeric value
77 """
78 return round(Circle.pi * (self.radius ** 2),2)
79
80
81 def perimeter(self):
82 """
83 Objective: To compute the perimeter of the Circle
84 Input Parameter:
85 self (implicit parameter) - object of type Circle
86 Return Value: perimeter - numeric value
87 """
88 return round(2 * Circle.pi * self.radius, 2)

```

**Fig. 11.21** Class Shape and its derived classes (`shape.py`)

We indicate that a method is abstract by preceding its definition by

`@abstractmethod` (function decorator). Note that the definition of an abstract class begins with

```

@abstractmethod - method: decorator for specifying an
abstract method

```

```

__metaclass__ = ABCMeta

```

To illustrate the use of abstract classes, we create an object `rectangle` of the class `Rectangle` having length and breadth 30 and 15, respectively, and an object `circle` of the class `Circle` having radius 5. The methods `area` and `perimeter` defined in the subclasses `Rectangle` and `Circle` override the corresponding abstract methods defined in the superclass `Shape`. Next, we see some examples of the use of these methods:

```
>>> rectangle = Rectangle(30, 15)
```

```
>>> rectangle.area()
```

```
450
```

```
>>> rectangle.perimeter()
```

```
90
```

```
>>> circle = Circle(5)
```

```
>>> circle.area()
```

```
78.5
```

```
>>> circle.perimeter()
```

```
31.4
```

### 11.7.5 Attribute Resolution Order for Inheritance

Python classes are categorized as old and new style classes. Python 3 is purely based on new-style classes. A new-style class inherits from the built-in class `object`. In the new style classes, to access an attribute of an object, Python typically looks for it in the namespace of the object itself, then in the namespace of the class, then in the namespace of the superclasses in the order in which they are derived and so on. In the script `newStyleClasses` ([Fig. 11.22](#)), class `C` inherits from classes `B1` and `B2`, which further inherit from the class `A`. It is important to point out that the class `C` inherits

from classes `B1` and `B2` in the order `B1` and `B2`.  
However, if the definition of class `C` were to begin with

new-style classes

attribute resolution order: for accessing an attribute of an object

```
class C(B2, B1):
```

then the class `C` would inherit from classes `B1` and `B2` in the order `B2` and `B1`.

```
01 class A:
02 test = 20
03
04 class B1(A):
05 pass
06
07 class B2(A):
08 test = 50
09
10 class C(B1, B2):
11 def __str__(self):
12 return str(self.test)
13
14 def main():
15 !!!
```

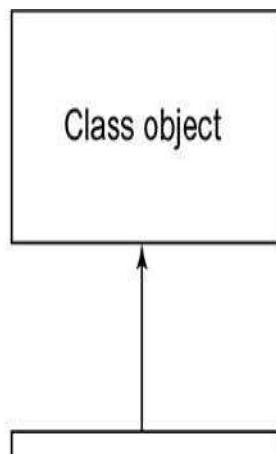
```

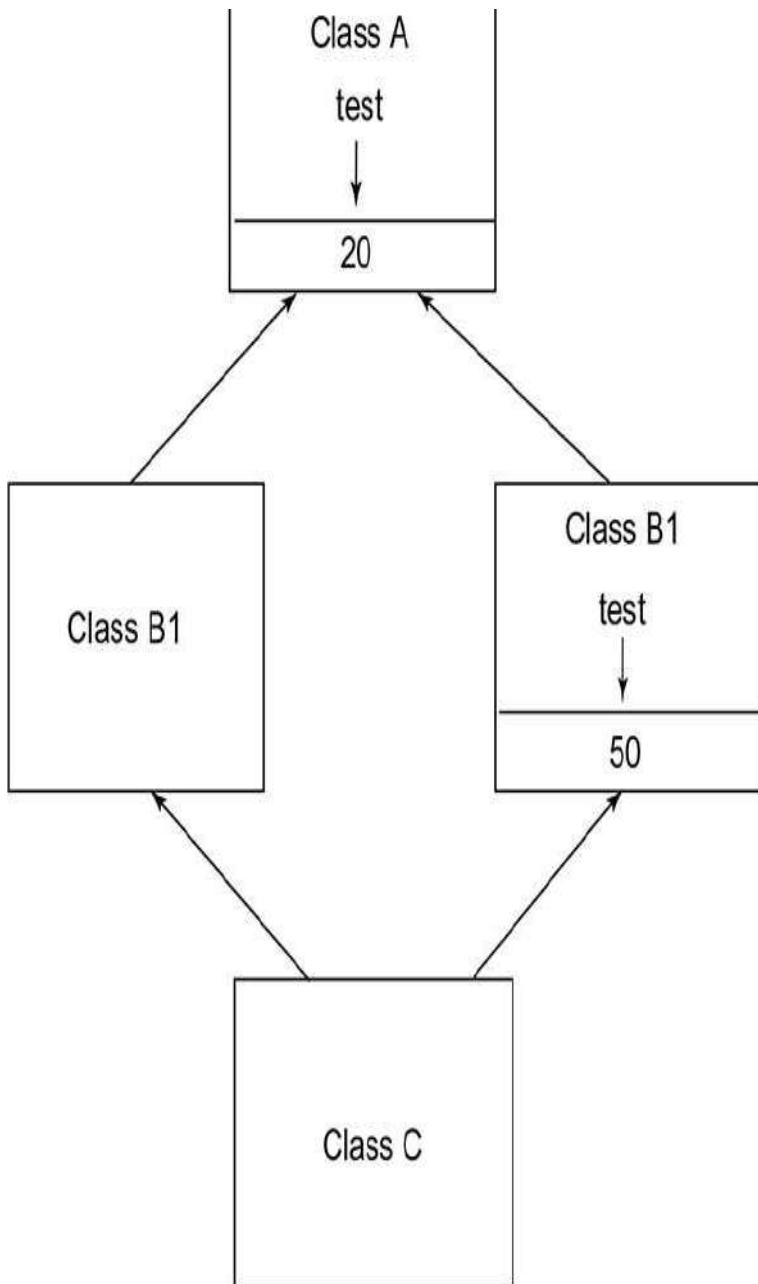
16 Objective: To illustrate attribute resolution order in new-style
17 classes
18
19 Input Parameter: None
20
21 Return Value: None
22
23 """
24
25 c1 = C()
26
27 print(c1)
28
29
30 if __name__=='__main__':
31
32 main()

```

**Fig. 11.22** Inheritance in new-style classes  
(newStyleClasses.py)

In Fig. 11.23, the order of resolution would be the object `c1` of class `C`, class `C`, class `B1`, class `B2`, class `A`, class `object`. On executing the script in Fig. 11.22, value `50` is printed. When the `print` function is executed in the `main` function (line 21), Python invokes pre-defined method `__str__` where it needs to reference variable `test`. Since `test` is not an instance variable of the object `c1`, the search is made in class `C`, subsequently followed by a search in superclass `B1`, and finally in `B2` where it is found.





**Fig. 11.23** Inheritance hierarchy for classes in Fig. 11.22

For determining the order in which methods of classes will be accessed, Python provides the attribute `__mro__` which stands for method resolution order. It returns a list ordered by the classes in which search for methods is to take place. For example:

`__mro__`: finds order in which methods of classes will be accessed

```
>>> C.__mro__
(<class '__main__.C', <class
'__main__.B1', <class '__main__.B2',
<class '__main__.A', <class 'object'>)

>>> B2.__mro__
(<class '__main__.B2', <class
'__main__.A', <class 'object'>)

>>> B1.__mro__
(<class '__main__.B1', <class
'__main__.A', <class 'object'>)

>>> A.__mro__
(<class '__main__.A', <class 'object'>)
```

#### 11.8 BUILT-IN FUNCTIONS FOR CLASSES

Python provides several built-in functions that deal with classes and their instances. For example, to find whether a class is a subclass of another class, one may use the function `issubclass` that takes two arguments: `sub` and `super`:

```
issubclass()
issubclass(sub, super)
```

The function `issubclass` returns `True` if `sub` is the subclass of class `super`, and `False` otherwise. For example:

```
>>> issubclass(Appointment, MyDate)
```

```
True
```

To find whether the instance *obj* is an object of class *class1*, we use the function `isinstance`:

```
isinstance()
isinstance(obj, class1)
```

This function returns `True` if either *obj* is an instance of class *class1* or it is an instance of a subclass of class *class1*, and `False` otherwise, for example:

```
>>> isinstance(meetStaff, MyDate)
```

```
True
```

```
>>> isinstance(manager1, Employee)
```

```
True
```

```
>>> isinstance(manager1, MyDate)
```

```
False
```

```
>>>
```

To find whether an instance *obj* contains an attribute *attr*, we use the function `hasattr`:

```
hasattr()

>>> hasattr(obj, attr)
```

This function returns `True` if instance *obj* contains an attribute *attr*, and `False` otherwise, for example:

```
>>> hasattr(manager1, 'hours')
```

```
False
```

```
>>> hasattr(manager1, 'DOB')
```

```
True
```

The functions `getattr` and `setattr` may be used to retrieve and set respectively, the value `val` of an attribute `attr` of an instance `obj`, as illustrated below:

```
getattr() and setattr()

getattr(obj, attr)

setattr(obj, attr, val)

>>> getattr(meetStaff, 'day')
```

```
15
```

```
>>> setattr(meetStaff, 'day', 23)

>>> print(meetStaff)
```

```
23-07-2014, 10:00:00
```

```
Meeting with parents regarding admissions
```

We may delete an attribute `attr` of instance `obj` as follows:

```
delattr()

delattr(obj, attr)

>>> delattr(meetStaff, 'day')

>>> print(meetStaff)
```

```
Traceback (most recent call last):
```

```
 File "<pyshell#5>", line 1, in
<module>
```

```
print(appointment1)

File "F:\PythonCode\Ch11\appointment.py",
line 28, in __str__
 +'\\n'+self.description

File " F:\PythonCode\Ch11\date.py", line
103, in __str__
 if self.day <= 9:

AttributeError: 'Appointment' object has
no attribute 'day'
```

As we had deleted the `day` attribute, subsequent attempts to access it in the `print` method resulted in error.

#### SUMMARY

1. Classes are based on an object-oriented paradigm known as Object-oriented Programming (OOP). Object-oriented programming revolves around the notion of objects.
2. A method/operator may be applied to objects of different types (classes). This feature of object-oriented programming is called polymorphism.
3. In operator overloading, a standard operator is redefined as per the requirement of a class, for example, the special methods that support operator overloading include `__add__`, `__sub__`, `__mult__`, `__div__`, `__lt__`, `__le__`, `__gt__`, and `__ge__`.
4. While using a binary operator, the method associated with the object on the *left-hand side* of the operator is invoked.
5. Function overloading provides the ability of writing different functions having the same name, but with a different number of parameters, possibly of various types. Python does not support function overloading. Indirectly, method overloading may be realized using default parameters.
6. Python special method `__del__` is used for defining the class destructor.
7. Encapsulation enables us to group related data and its associated functions under one name. Encapsulation is realized via classes. Classes provide an abstraction where essential features of the real world are represented in the form of interfaces, hiding the lower level complexities of implementation details.
8. Python allows us to prevent accidental access to the data and method attributes from outside the class. This is achieved via the notion of private members. We can

tell Python that an attribute is a private attribute by prefixing the attribute name by two consecutive underscore characters `__`. Further, name of a private attribute should not have more than one underscore character at the end. This technique of restricting access to private members from outside the class is known as *name mangling*. The restriction on the use of private variables from outside the scope of the class, may be bypassed using the syntax:

```
<instance>.__<className><attributeName>
```

9. Methods in the class definition that modify the value of one or more arguments are known as modifiers.
10. Methods in the class definition which only access (do not modify) the data attributes associated with arguments are known as accessors.
11. The methods in which the object that invokes the method is passed as the first implicit parameter (`self`) are called instance methods. An instance method defines operations on the data members of an instance of a class.
12. The method that does not require a class object to be passed as the first parameter is called a static method. Call to this method should be made as `className.staticMethodName`, although Python allows a static method to be invoked as an attribute of an object also.
13. Python allows to add methods dynamically to a class or an instance of a class using the syntax:  

```
<className>.<newMethodName> =
<existingFunctionName>
```
14. Python also allows to add a method to an instance of a class:  

```
<instance>.
<newMethodName>=MethodType(<existingFunction
Name>,<instance>)
```
15. The process of using objects of other classes as attribute values is called object *composition*.
16. Inheritance makes it possible to create a class (also called type) hierarchy. The class whose properties are inherited is known as a base class, parent class, or superclass. The class that inherits from another class is called derived class, child class, or subclass. A derived class may serve as base class for another class derived from it and so on. Each derived class may define its new attributes.
17. When inheritance involves a derived class that derives its properties from a single base class (see Fig. 11.9), it is called *single inheritance*.
18. When an object of a derived class makes a call to a method that is also defined in the base class, it is the method of the derived class that is invoked by Python. Thus, the derived class methods override the base class methods, and this process is known as method *overriding*.
19. When inheritance involves more than one level of hierarchy such as `A -> B -> C`, it is called multilevel inheritance, and inheritance involving multiple classes deriving from single base class is known as hierarchical inheritance.
20. In multiple inheritance, a subclass derives its attributes from two or more classes.

21. An abstract method in a base class identifies the functionality that should be implemented by all its subclasses. Every subclass of the base class should override this method with its implementation. A class containing abstract methods is called abstract class. To use Python Abstract Base Classes (ABCs), one needs to import ABCMeta and abstractmethod from abc module.
22. A new-style class inherits from the built-in class `object`. In the new-style classes, to access an attribute of an object, Python first looks for it in the namespace of the object itself, then in the namespace of the class, then in the namespace of the superclasses in the order in which they are derived, and so on.
23. Python provides several built-in functions that deal with classes and their instances:
1. The function `issubclass` is used to find whether a class is a subclass of another class.
  2. The function `isinstance` is used to find whether an object is an instance of a class.
  3. The function `hasattr` is used to determine whether an instance contains a particular attribute.
  4. The functions `getattr` and `setattr` are used to retrieve or set the value of an attribute of an instance respectively.
  5. The function `delattr` is used to delete an attribute of an instance.

#### EXERCISES

1. Write a class `Point` having `x` and `y` coordinates as data members. Write another class `LineSegment` that derives the class `Point`. Also, list appropriate methods.
  2. Define the method `__ne__` for the class `MyDate`.
  3. Define the method `__sub__` (Fig. 11.24) for the class `MyDate`.
  4. Define a class `ComplexNumbers`. Write operations for addition, subtraction, and multiplication, using the notion of operator overloading.
  5. Examine the following class `Shape`:
- ```

class Shape:
    def __init__(self, shapeType, x, y):
        self.shapeType = shapeType
        self.length = x
        self.width = y
    def computeArea():
        pass
  
```

```

def __sub__(self, other):
    ...
    Objective: To subtract two date objects
    Input Parameters:
        self (implicit parameter) - object of type MyDate
        other - object of type MyDate
    Return Value: string representation of difference of two objects
        if second object passed as parameter is of MyDate type
        else message 'Undefined type' is printed and program is terminated
    ...
    ...
    Examples:
        yyyy-mm-dd      yyyy-mm-dd      yyyy-mm-dd
        2014-04-04      2012-01-12      2014-06-06
        - 2012-06-06   - 2010-05-13   - 2012-04-04
        -----
        0001-09-28      0001-07-30      0002-02-02
    ...
    pass

```

Fig. 11.24 Method `__sub__` for the class `MyDate`

The class `Shape` should be inherited by the classes `Rectangle` and `Triangle`. Both the derived classes should invoke the method `__init__` to initialize data members. Note that `length` and `width` correspond to `height` and `base` for a triangle. The derived classes should override the method `computeArea` of the superclass `Shape`.

6. Examine the class `Person` defined in this chapter. Define a class `Student` that derives this class and defines `name`, `rollNum`, `class`, `totalMarks`, and `Year` as the data members. The class should contain the instance method `__init__` and the abstract method `percentage`. Define two classes `Grad` and `PostGrad` which inherit from the base class `Student`. Both the classes should define their `__init__` method and should override the abstract method `percentage` of the superclass. Note that `totalMarks` obtained are out of 600 and 400 for `Grad` and `PostGrad` classes respectively.
7. Define a base class `Vehicle`, having attributes `registration number`, `make`, `model`, and `color`. Also, define classes `PassengerVehicle` and `CommercialVehicle` that derive from the class `Vehicle`. The `PassengerVehicle` class should have additional attribute for maximum passenger capacity. The `CommercialVehicle` class should have an additional attribute for maximum load capacity. Define `__init__` method for all these classes. Also, define get and set methods to retrieve and set the value of the data attributes.
8. Define classes `Car`, `Autorickshaw`, and `Bus` which derive from the `PassengerVehicle` class mentioned in the previous question. The `Car` and `Bus` should have attributes for storing information about the number of doors, not shared by `Autorickshaw`. The `Bus` should have Boolean attribute `doubleDecker` not shared by `Car` and `Autorickshaw`. Define `__init__` method for all

- these classes. Also, define get and set methods to determine and set the value of the data attributes.
9. Define a class `Account`, having attributes account holder's name, account number, account type, the amount deposited and minimum deposit amount. Define two classes, namely `Savings` and `Current`. The `Savings` class should have a property `interest`. Define `__init__` method for all these classes. Also, define get and set methods to determine and set the value of the data attributes.
10. Using Python built-in functions, write the statements to
1. Determine whether `A` is a subclass of `B`.
 2. Determine whether attribute `attr` exists in the namespace of object `ob` of class `A`.
 3. Assign value `70` to attribute `attr` of object `ob` of class `A`.
 4. Delete an attribute `attr` of object `ob` of class `A`.

CHAPTER 12

LIST MANIPULATION

CHAPTER OUTLINE

12.1 Sorting

12.2 Searching

12.3 A Case Study

12.4 More on Sorting

In several applications, we need to sort data or search for some item of interest in the given data. For example, we may be required to arrange the answer sheets in the order of roll number, to arrange the student data on the basis of marks, to search for a phone number in a telephone directory, or to find an answer sheet with a given roll number. We already know that given the roll number of a student, it is easier to find his/her answer sheet if the answer sheets are arranged according to roll number. Similarly, it is easier to find how many students have secured pass marks, if the answer sheets are arranged according to marks. The process of arranging data in ascending/descending order is called sorting. In this chapter, we will study how to sort the data using several techniques. We will also study searching techniques for unsorted data as well as for the data that is already sorted. This chapter concludes with a case study relating to student data.

sorting: the process of arranging data in ascending/descending order

searching is more efficient when data is sorted

Suppose we need to arrange a list of names in lexicographic order, i.e., as they appear in a dictionary. Examine the sample data that appears in the list names (Fig. 12.1).

'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
0	1	2	3	4	5

Fig. 12.1 The list of names and the corresponding indexes at which they appear

We have already seen that the strings can be compared using relational operators $<$, \leq , $>$, \geq , $=$, and \neq . So, we shall frequently use the terms such as less, less than equal to, greater, greater than equal to, equal to, not equal to, least, smallest, largest, maximum in the context of strings.

The attribute or property of data that forms the basis of sorting is called the key. There are several techniques of sorting the data such as selection sort, bubble sort, insertion sort, heap sort, merge sort, and quick sort, which can be studied for their relative efficiency. However, in this chapter, we shall discuss only first three techniques. Without any loss of generality, we discuss how to sort the data in ascending order. The other case of sorting in descending order follows similarly. We just need to change the $<$, \leq operators to $>$, \geq operators, respectively and vice versa.

key : attribute that forms the basis of sorting

12.1.1 Selection Sort

It is the simplest but not a very efficient method of sorting. To begin with, we find the lexicographically smallest value in the list and interchange this entry in the list with the first value in the list. Next, we find the

entry with the smallest value out of the remaining entries, and interchange it with the second value in the list, and proceed in this manner. If there are n values, only $n-1$ values need to be placed in order because when $(n-1)$ values have been put in order, then the n th value would automatically be in order. This technique of sorting is called selection sort.

selection sort: a simple method of sorting

selection sort requires $n-1$ passes of the entire sequence of values

Now, we discuss selection sort technique in the context of sample data in Fig. 12.1. In the first iteration ($i=0$), we find the smallest name in the list. To do this, we begin with the name at index $j=0$. When we have examined only the name at index 0, it is also the smallest of the names examined so far. We keep track of the index of the smallest name examined so far using the variable `minIndex`. Thus, when $j=0$, `minIndex` is equal to 0 (Fig. 12.2). Next, compare value at index $j=1$ with the value at index `minIndex`. Since 'Sanvi' is smaller than 'Vijaya', `minIndex` will be set equal to 1. Next, consider the next value in the list at index $j=2$. We compare it with the value at index `minIndex`. Since 'Ruby' is smaller than 'Sanvi', we will set `minIndex` to 2. When $j=3$, 'Zafar' will be compared with 'Ruby' (value at `minIndex` 2). Since 'Zafar' is greater than 'Ruby', the value of `minIndex` will remain unchanged. When $j=4$, 'Maya' will be compared with 'Ruby' (value at `minIndex` 2), and `minIndex` will be set to 4 since 'Maya' is smaller than 'Ruby'. Finally, when $j=5$, 'Anya' is found to be smaller than 'Maya' (value at `minIndex` 4). Thus, after scanning the entire list, `minIndex` will have value 5.

Now, we need to exchange this smallest name entry 'Anya' at index 5 (minIndex) with the name at index i=0. This may be accomplished as follows:

placing i th smallest value at i th index in the sorted list

```
lst[i], lst[minIndex] = lst[minIndex],
lst[i]
```

j=0	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

minIndex

j=1	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

minIndex

j=2	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

minIndex

j=3	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

minIndex

j=4	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

Vijaya	Sanvi	Ruby	Zafar	Maya	Anya
0	1	2	3	4	5
minIndex					

j=5

'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
0	1	2	3	4	5
minIndex					

Fig. 12.2 Steps in the selection sort method for first iteration ($i = 0$)

At this stage, we have scanned data from index 0 to index 5, and swapped the name at index 0, with the smallest name, found at index 5. This completes the first iteration, and the modified list is shown in Fig. 12.3.

'Anya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Vijaya'
0	1	2	3	4	5
minIndex					

Fig. 12.3 Modified data at the end of first iteration ($i = 0$)

Now that the smallest name 'Anya' is at the proper position (index 0), we only have to sort the names in the updated list from index 1 to 5. For this purpose, we need to find the index of the second smallest name in the list (i.e. the smallest name in the index range 1 to 5). In Fig. 12.4, we illustrate this process.

$j=1$	'Anya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Vijaya'
	0	1	2	3	4	5
	minIndex					
$j=2$	'Anya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Vijaya'
	0	1	2	3	4	5
	minIndex					
$j=3$	'Anya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Vijaya'
	0	1	2	3	4	5
	minIndex					
$j=4$	'Anya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Vijaya'
	0	1	2	3	4	5
	minIndex					
$j=5$	'Anya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Vijaya'
	0	1	2	3	4	5
	minIndex					

Fig. 12.4 Steps in the selection sort method for second iteration ($i = 1$)

Since name 'Maya' at index 4 is the smallest name across the indexes ranging from 1 to 5, we swap name at index 4 with the name at index 1. The modified list at the end of the second iteration is shown in Fig. 12.5.

'Anya'	'Maya'	'Ruby'	'Zafar'	'Sanvi'	'Vijaya'
0	1	2	3	4	5

Fig. 12.5 Modified data at the end of second iteration ($i = 2$)

In the third iteration, we have to sort names in the updated list from index 2 to 5. Since the smallest name in the remaining unsorted list is at `minIndex 2`, there will be no change in the relative position of entries in the third iteration. At the end of the fourth iteration, names at indexes 3 and 4 will be swapped, and the modified list is shown in Fig. 12.6.

if the i th smallest element is already at its position in an iteration, there will be no change in relative position of entries

'Anya'	'Maya'	'Ruby'	'Sanvi'	'Zafar'	'Vijaya'
0	1	2	3	4	5

Fig. 12.6 Modified data at the end of fourth iteration ($i = 3$)

In the fifth iteration, we begin the search for the smallest name from the index 4. Since the name at index 5 is smaller than the name at index 4, the names at indexes 4 and 5 will be swapped, and the modified list is shown in Fig. 12.7.

Now that five names (at indexes 0, ..., 4) have been placed in order, the last name at index 5 is automatically in order. In this manner, we have been able to sort a list of names (Fig. 12.7).

'Anya'	'Maya'	'Ruby'	'Sanvi'	'Vijaya'	'Zafar'
0	1	2	3	4	5

Fig. 12.7 Modified data at the end of fifth iteration ($i = 4$)

We summarize the sorting procedure discussed above in Fig. 12.8.

```

01 def selectionSort(lst):
02     """
03     Objective: To sort the list lst in ascending order using
04     selection sort
05     Input Parameter: lst - list
06     Return Value: None
07     """
08     for i in range(0, len(lst)-1):
09
10         # Find minimum value in range [i, len(lst)-1]
11         # Swap the minimum value with ith position if not already at
12             position i

```

Fig. 12.8 Sorting procedure for selection sort
(selectionSort.py)

Using the above ideas, we give the complete script selectionSort (Fig. 12.9).

```
01 def selectionSort(lst):
02     """
03     Objective: To sort the list lst in ascending order using
04     selection sort
05     Input Parameter: lst - list
06     Return Value: None
07     """
08     for i in range(0, len(lst)-1):
```

```

09     minIndex = i
10     for j in range(i + 1, len(lst)):
11         if lst[j] < lst[minIndex]:
12             minIndex = j
13         if minIndex != i:
14             lst[minIndex], lst[i] = lst[i], lst[minIndex]
15
16 def main():
17     """
18     Objective: To sort the list provided as an input by the user
19     Input Parameter: None
20     Return Value: None
21     """
22     lst = eval(input('Enter a list: '))
23     print('Sorted List')
24     selectionSort(lst)
25     print(lst)
26
27 if __name__ == '__main__':
28     main()

```

Fig. 12.9 Program for selection sort (`selectionSort.py`)

On execution of the script `selectionSort` (Fig. 12.9), the user is prompted to enter a list (`lst`). This list will be passed as an argument to function `selectionSort`, which will sort the list. Since a list (like other objects) is passed by reference in Python, and the function only modifies the components of the list,

there is no need to return the sorted list. Sample runs of the script `selectionSort` are shown below:

```
Enter a list: ['Vijaya', 'Sanvi', 'Ruby',
'Zafar', 'Maya', 'Anya']
```

Sorted List

```
['Anya', 'Maya', 'Ruby', 'Sanvi',
'Vijaya', 'Zafar']
```

```
Enter a list: [1001, 1006, 1002, 1005,
1004, 1003]
```

Sorted List

```
[1001, 1002, 1003, 1004, 1005, 1006]
```

the function `selectionSort` works equally well on numeric and string data

12.1.2 Bubble Sort

Just like selection sort, in this method also, the data is sorted by making several passes (iterations) through the list. In each pass, we compare values in adjacent positions and interchange them, if they are out of order. To begin with, we have a list of n values. The n th value and $(n-1)$ th value are compared and interchanged if the n th value is smaller than $(n-1)$ th value, then $(n-1)$ th value and $(n-2)$ th value are compared and interchanged if found out of order, and so on. Observe that in the first pass, the smallest value will move to the front of the list at index 0; on subsequent passes, it will be ignored. Thus, in the second iteration, we need to sort the remaining list of $n-1$ values excluding the value at index 0. After $n-1$ iterations, the list will be completely sorted, and the algorithm will halt. We use the list in Fig. 12.10 to show the working of the bubble sort procedure:

bubble sort: compare keys at adjacent positions

'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
0	1	2	3	4	5

Fig. 12.10 List of names and the corresponding indexes at which they appear

In the first iteration of bubble sort, we start at index 5 and keep on comparing adjacent locations in order to move the smallest name to the beginning of the list. Steps performed in the first iteration are shown below in Fig. 12.11:

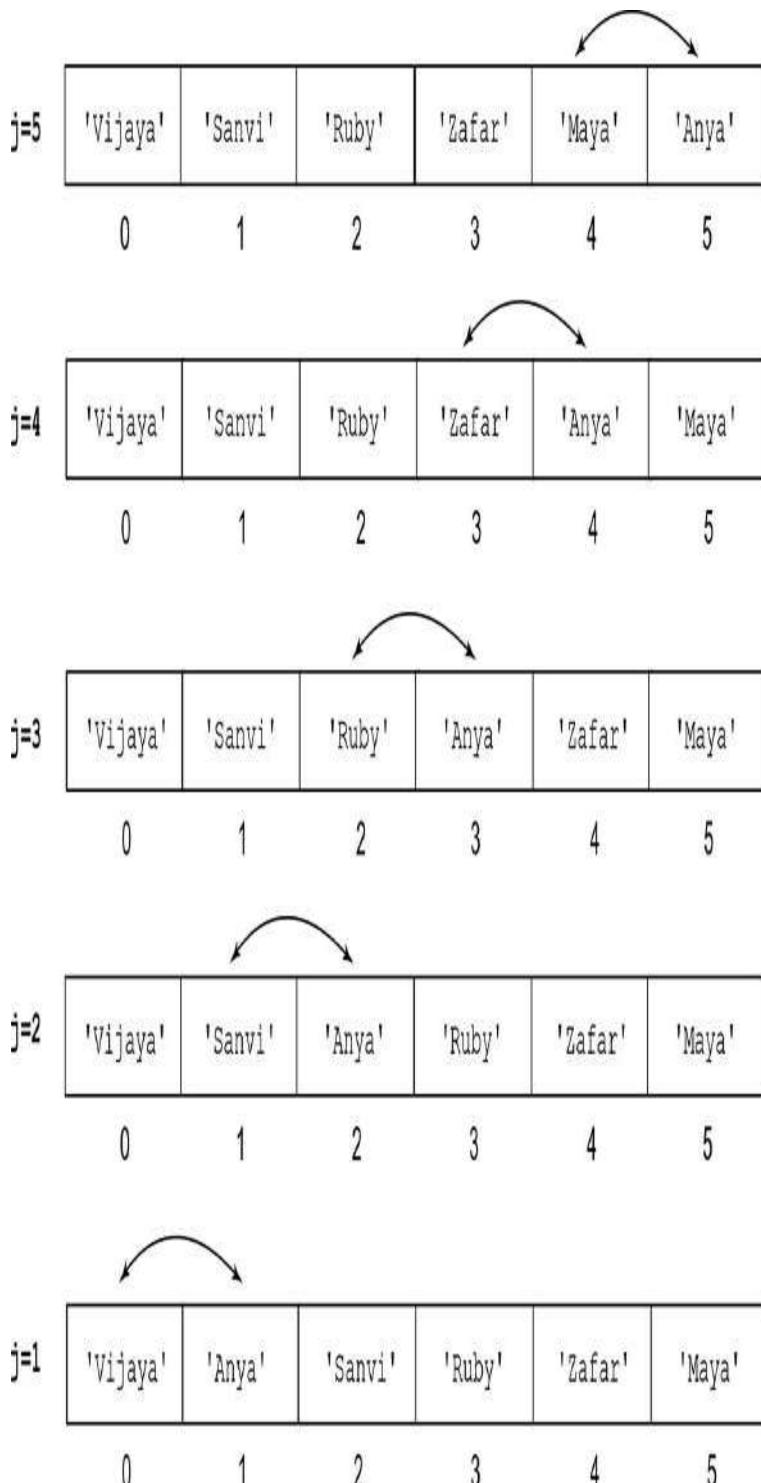


Fig. 12.11 Steps in the first iteration of bubble sort ($i = 0$). The curved double-headed arrows mark the list elements being swapped

The modified list, thus modified, at the end of the first iteration is shown in Fig. 12.12:

'Anya'	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'
0	1	2	3	4	5

Fig. 12.12 Modified list at the end of first iteration ($i = 0$)

Note that at the end of the first iteration, the smallest name moves to the first position (index 0). Similarly in the second iteration, above procedure will be performed on names in the index range 1 to 5 and at the end of it, the second smallest name will move to index 1 as shown in Fig. 12.13.

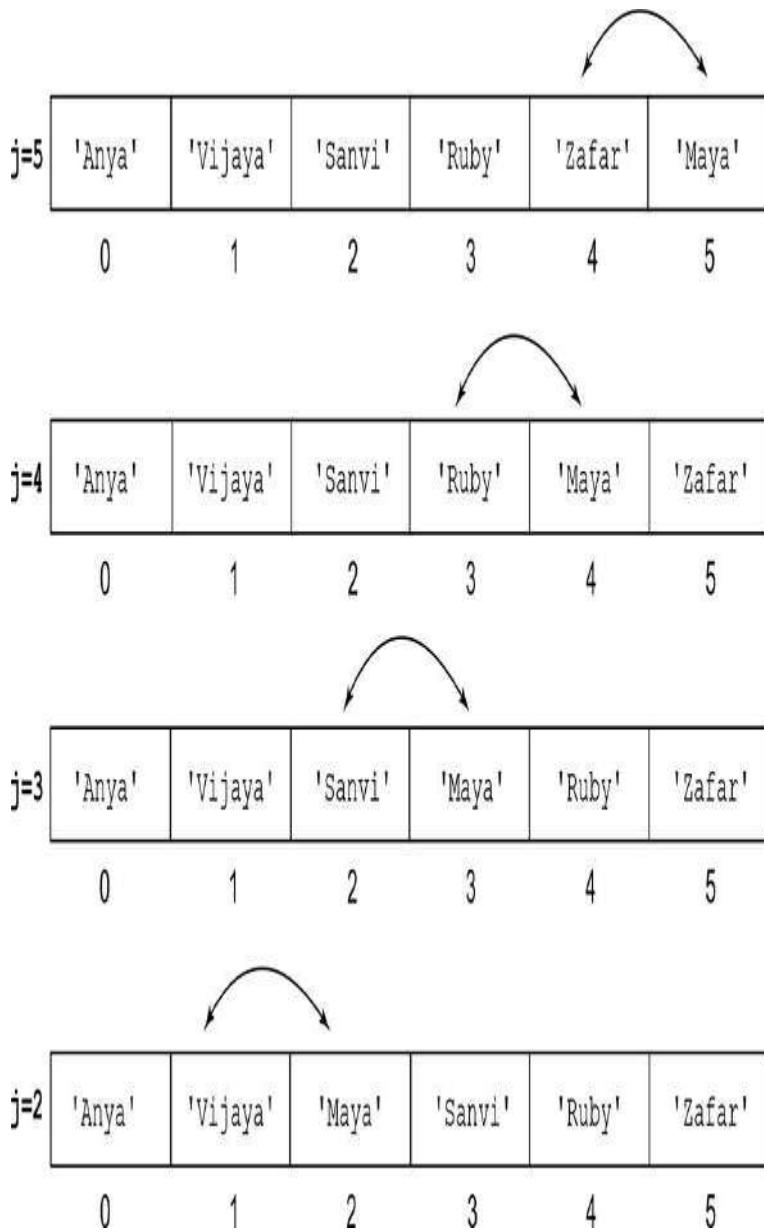


Fig. 12.13 Steps in the second iteration of bubble sort. The curved double-headed arrows mark the list elements being swapped

The modified list after the second iteration is shown in Fig. 12.14.

'Anya'	'Maya'	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'
0	1	2	3	4	5

Fig. 12.14 Modified list at the end of second iteration ($i=1$)

In the third iteration, we consider the names in the index range 2 to 5 and the modified list at the end of the third iteration is shown in Fig. 12.15.

'Anya'	'Maya'	'Ruby'	'Vijaya'	'Sanvi'	'Zafar'
0	1	2	3	4	5

Fig. 12.15 Modified list at the end of third iteration ($i=2$)

In the fourth iteration, we consider the names in the index range 3 to 5 and the modified list at the end of the fourth iteration is shown in Fig. 12.16.

'Anya'	'Maya'	'Ruby'	'Sanvi'	'Vijaya'	'Zafar'
0	1	2	3	4	5

Fig. 12.16 Modified list at the end of fourth iteration ($i=3$)

In the fifth iteration, we consider the names in the index range 4 to 5. As the names at indexes 4 to 5 are already in sorted order, the list at the end of the fifth iteration will be the same as the list at the end of the fourth iteration (Fig. 12.16). Thus, at the end of the fifth iteration, all the six names have been arranged in order. Based on the preceding discussion, we present the complete script for the bubble sort method (Fig. 12.17).


```
01 def bubbleSort(lst):
02     """
03         Objective: To sort the list lst in ascending order using bubble
04         sort
05         Input Parameter: lst - list
06         Return Value: None
07     """
08     n = len(lst)-1 # highest valid index in the list
09     for i in range(0, n):
10         for j in range(n, i, -1):
11             if lst[j] < lst[j - 1]: # interchange if out of order
12                 lst[j], lst[j - 1] = lst[j - 1], lst[j]
13
14 def main():
15     """
16         Objective: To sort the list provided as an input by the user
17         Input Parameter: None
18         Return Value: None
```

```
19     """
20     lst = eval(input('Enter the list: '))
21     print('Sorted List')
22     bubbleSort(lst)
23     print(lst)
24
```

```
|25 if __name__=='__main__':
26     main()
```

Fig. 12.17 Program for bubble sort (`bubbleSort.py`)

Sample runs of the script `bubbleSort` are shown as follows:

```
Enter the list: ['Vijaya', 'Sanvi',
'Ruby', 'Zafar', 'Maya', 'Anya']
```

Sorted List

```
['Anya', 'Maya', 'Ruby', 'Sanvi',
'Vijaya', 'Zafar']
```

```
Enter the list: [1001, 1006, 1002, 1005,
1004, 1003]
```

Sorted List

```
[1001, 1002, 1003, 1004, 1005, 1006]
```

examples of input lists and the corresponding sorted output lists

Next, let us apply bubble sort algorithm on the list in [Fig. 12.18](#). The modified list at the end of the first, second, and third iteration is shown in [Figs. 12.19](#), [12.20](#), and [12.21](#), respectively.

'Ruby'	'Sanvi'	'Maya'	'Vijaya'	'Anya'	'Zafar'
0	1	2	3	4	5

Fig. 12.18 List of names and the corresponding indexes at which they appear

'Anya'	'Ruby'	'Sanvi'	'Maya'	'Vijaya'	'Zafar'
0	1	2	3	4	5

Fig. 12.19 Modified list at the end of first iteration ($i = 0$)

'Anya'	'Maya'	'Ruby'	'Sanvi'	'Vijaya'	'Zafar'
0	1	2	3	4	5

Fig. 12.20 Modified list at the end of second iteration ($i = 1$)

325

'Anya'	'Maya'	'Ruby'	'Sanvi'	'Vijaya'	'Zafar'
0	1	2	3	4	5

Fig. 12.21 Modified list at the end of third iteration ($i = 2$)

Note that when we apply bubble sort on the list in Fig. 12.18, the list gets sorted at the end of the second iteration and no data is swapped in the third iteration. Indeed, if no data is swapped in an iteration, we conclude that the list has already been sorted. As such, the control should break out of the loop, instead of unnecessarily proceeding with the remaining iterations. Based on this observation, we modify the script `bubbleSort` (see Fig. 12.22).

if no swapping takes place in an iteration in bubble sort, we conclude that the list is now sorted, and skip rest of the iterations

```
01 def bubbleSort(lst):
02     """
03         Objective: To sort the list lst in ascending order using bubble
04         sort
05         Input Parameter: lst - list
06         Return Value: None
07     """
08     n = len(lst)-1 # highest index of an element in list
09     for i in range(0, n):
10         swap = False
11         for j in range(n, i, -1):
12             if lst[j] < lst[j - 1]: # swap if out of order
13                 swap = True
14                 lst[j], lst[j - 1] = lst[j - 1], lst[j]
15             if swap == False: # break, if entire list is sorted
16                 break
17
18 def main():
19     """
20         Objective: To sort the list provided as an input by the user
21         Input Parameter: None
22         Return Value: None
23     """
24     lst = eval(input('Enter the list: '))
25     print('Sorted List')
```

```

26     bubbleSort(lst)
27     print(lst)
28
29 if __name__=='__main__':
30     main()

```

Fig. 12.22 Program for bubble sort (bubbleSort1.py)

12.1.3 Insertion Sort

In insertion sort, the list is logically divided into two parts. Whereas the left part is the sorted part, the right part is the unsorted part comprising the elements yet to be arranged in sorted order. In each iteration, we increase the length of the sorted part by one in the following manner: insert the first element from the unsorted part into the sorted part at the correct position. To find the correct position of the value to be inserted, we compare it with values in the sorted part (starting from the rightmost value in the sorted part) and shift each value to the right by one position, until the correct position is found. For example, examine the list in Fig. 12.23.

maintain two divisions of the list: sorted part and unsorted part.

in each iteration, insert first value from the unsorted part into the sorted part

'Anya'	'Sanvi'	'Vijaya'	'Zafar'	'Maya'	'Ruby'
0	1	2	3	4	5

Fig. 12.23 List of names and the corresponding indexes

In this example, the left sorted part (grey portion) comprises elements at indices 0, 1, 2, 3, and the right unsorted part (white portion) comprises the elements at indices 4 and 5. As of now, the lengths of the sorted and unsorted parts are 4 and 2, respectively. So, at the end of the next iteration, lengths of sorted and unsorted parts will be 5 and 1, respectively. Our task in this iteration is to insert the element `lst[4]` at the correct position. As '`Maya`' < '`Zafar`', we shift '`Zafar`' one position right, but before we do so, we must save '`Maya`' in a temporary variable, say, `temp`. The modified list is shown in Fig. 12.24.

```
temp = 'Maya'
```

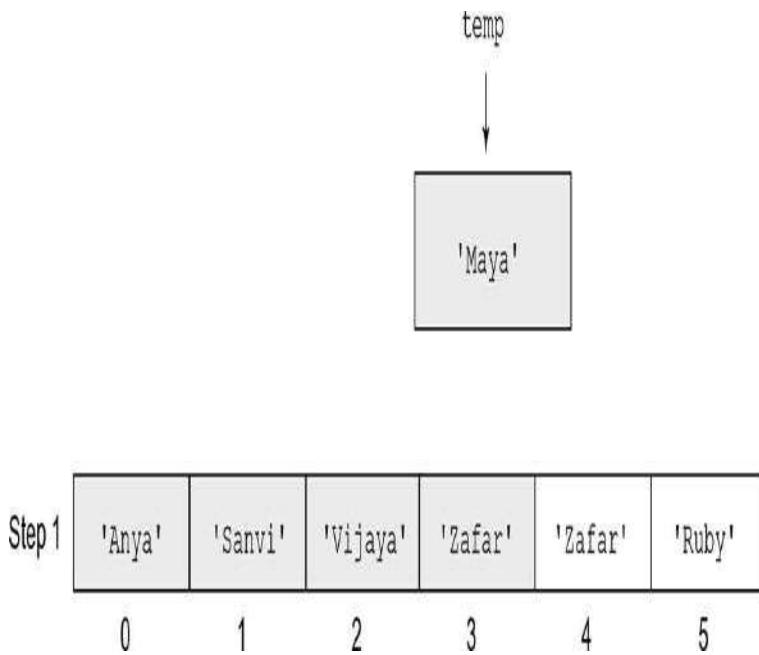


Fig. 12.24 List of names and the corresponding indexes

Next, we compare the value of the variable `temp`, i.e. '`Maya`' with `lst[2]`, i.e., '`Vijaya`'. As '`Maya`' < '`Vijaya`', we shift '`Vijaya`' one position right. The modified list is shown in Fig. 12.25.

Step 2	'Anya'	'Sanvi'	'Vijaya'	'Vijaya'	'Zafar'	'Ruby'
	0	1	2	3	4	5

Fig. 12.25 List of names and the corresponding indexes

Next, we compare the value of the variable `temp`, i.e.

`'Maya'` with `lst[1]`, i.e. `'Sanvi'`. As `'Maya' < 'Sanvi'`, we shift `'Sanvi'` one position right. The modified list is shown in Fig. 12.26.

Step 3	'Anya'	'Sanvi'	'Sanvi'	'Vijaya'	'Zafar'	'Ruby'
	0	1	2	3	4	5

Fig. 12.26 List of names and the corresponding indexes

Next, we compare, `'Maya'` with `lst[0]`, i.e. `'Anya'`.

As `'Maya' > 'Anya'`, we do not need to shift `'Anya'`, and `'Maya'` can now be placed at index 1. The modified list is shown in Fig. 12.27.

'Anya'	'Maya'	'Sanvi'	'Vijaya'	'Zafar'	'Ruby'
0	1	2	3	4	5

Fig. 12.27 List of names and the corresponding indexes

Recall that the element `'Maya'` to be inserted in the list sorted up to index 3 was at index 4. The following code achieves the desired effect of shifting `'Maya'` to its correct position at index 1:

```
i = 4
```

```
temp = lst[i]
```

```
j = i - 1
```

```
while j >= 0 and lst[j] > temp:
```

```
    lst[j + 1] = lst[j]
```

```
    j = j-1
```

```
lst[j + 1] = temp
```

code to insert value at index 4 at correct position in the list
(sorted up to index 3)

Once again, we use the following list to illustrate insertion sort (Fig. 12.28):

'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
0	1	2	3	4	5

Fig. 12.28 The list of names and the corresponding indexes

initially, the sorted list comprise value at index 0, followed by remaining indexes to be part of unsorted list

As shown in Fig. 12.29, in the first iteration, 'Vijaya' is shifted towards the right by one position and 'Sanvi' (that appeared at index 1 earlier) is inserted at index 0. Thus, 'Sanvi' now belongs to the sorted part of the list. Similarly, in the second iteration, names at indices 0 and 1 are shifted towards the right and the name 'Anya' is placed at index 0 in the left part. This process is continued till the fifth iteration when the right partition becomes empty and the left partition contains the sorted list. On the basis of the foregoing discussion, we present the complete program for insertion sort (Fig. 12.30).

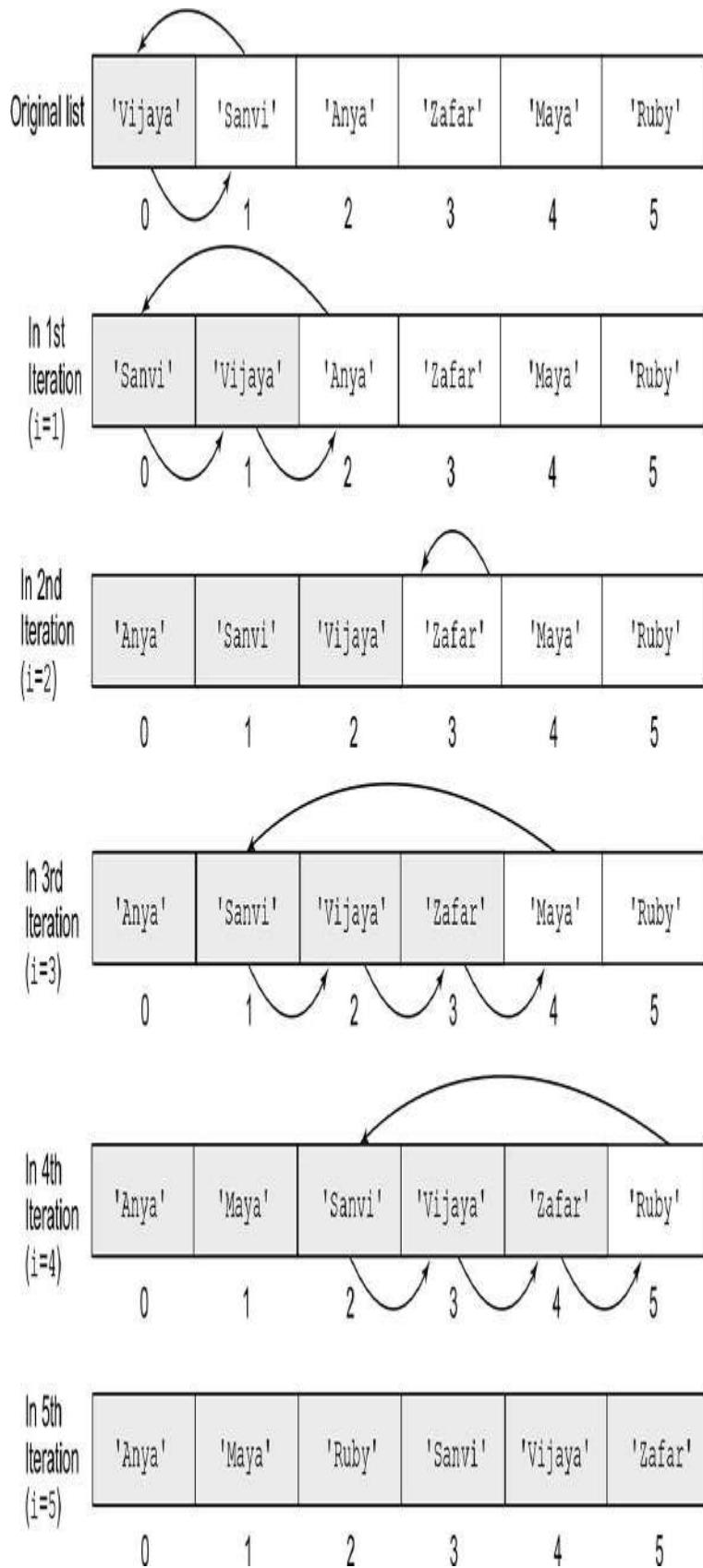


Fig. 12.29 Steps in insertion sort on the list in Fig. 12.28

```
01 def insertionSort(lst):
02     """
03     Objective: To sort the list lst in ascending order using
04     insertion sort
05     Input Parameter: lst - list
06     Return Value: None
07     """
08     for i in range(1, len(lst)):
```

```

09     temp = lst[i]
10     j = i - 1
11     while j >= 0 and lst[j] > temp: # shift lst[j] right
12         lst[j + 1] = lst[j]
13         j = j-1
14     lst[j + 1] = temp
15
16 def main():
17     """
18     Objective: To sort the list provided as an input by the user
19     Input Parameter: None
20     Return Value: None
21     """
22     lst = eval(input('Enter a list: '))
23     print('Sorted List')
24     insertionSort(lst)
25     print(lst)
26
27 if __name__=='__main__':
28     main()

```

Fig. 12.30 Program for insertion sort (`insertionSort.py`)

Sample runs of the script `insertionSort` are shown below:

```
Enter a list: ['Vijaya', 'Sanvi', 'Ruby',
'Zafar', 'Maya', 'Anya']
```

Sorted List

```
['Anya', 'Maya', 'Ruby', 'Sanvi',
'Vijaya', 'Zafar']
```

```
Enter a list: [1001, 1006, 1002, 1005,
1004, 1003]
```

Sorted List

```
[1001, 1002, 1003, 1004, 1005, 1006]
```

examples of input lists and the corresponding sorted output lists

12.2 SEARCHING

In this section, we shall discuss how to find out whether a data value appears in a list. For example, given a list of names of students in a class, we wish to find whether a particular name appears in the list. In the following sections, we discuss two techniques for searching a value in a list, namely, linear search, and binary search. While the former technique scans the list from the beginning till the data value is found or the list is exhausted, the latter technique is considerably more efficient in terms of time complexity but works for sorted lists only.

searching for a value in a list

12.2.1 Linear Search

To search for an element in a list, we scan the list from the beginning till the required data value is found or the list is exhausted. This searching technique is known as linear search as the search process is sequential. For example, given a list of names of students, to find whether a particular name appears in the list, we examine each name in the list starting from the first position (i.e., index 0). When the target value is found in the list, we terminate the search. However, if the entire list is exhausted and the target value is not found, we

conclude that the value being searched is not there in the list. For example, examine the following list of names of students (Fig. 12.31):

linear search: sequential search that scans the list from the beginning till the required data value is found or the list is exhausted

'Shilpi'	'Ankita'	'Shweta'	'Gaurav'	'Shubham'	'Rashmi'
0	1	2	3	4	5

Fig. 12.31 List of names of students

To search for the name 'Shubham' in the list, we look for this value starting at index 0. After iterating over values at the indexes 0, 1, 2, and 3, when the value at index 4 is compared to the target value 'Shubham', we find that it matches the target value, and the search terminates successfully. However, if we were searching for 'Sarthak', the entire list is exhausted without finding a match. So, we conclude that the name 'Sarthak' is not in the list. The complete script for linear search appears in Fig. 12.32.

```
01 def linearSearch(lst, searchValue):  
02     '''  
03     Objective: To search for an element in a list using linear  
04     search  
05     Input Parameters:  
06         lst - list  
07         searchValue - any type  
08     Return Value: index - numeric value, if searchValue is found,  
09                 None otherwise  
10     '''  
11     for i in range(0, len(lst)):  
12         if lst[i] == searchValue:  
13             return i  
14     return None  
15 def main():  
16     '''  
17     Objective: To search for an element in the list provided as an  
18     input by the user  
19     Input Parameter: None
```

```
20     Return Value: None
21
22     ''
23
24     lst = eval(input('Enter a list: '))
25     searchVal = eval(input('Enter the value to be searched: '))
26     searchResult = linearSearch(lst, searchVal)
27     print(searchVal, ' found at index ', searchResult)
28
29
30 if __name__=='__main__':
31     main()
```

Fig. 12.32 Program for linear search (linearSearch.py)

On executing the script in Fig. 12.32, Python prompts the user for the list and the value to be searched and will display the appropriate message indicating presence or absence of the value, for example:

```
Enter a list: ['Reena', 'Stuti', 'Monika',
'Deepika']
```

```
Enter the value to be searched: 'Monika'
```

```
Monika found at index 2
```

```
Enter a list: ['Shilpi', 'Ankita',
'Shweta', 'Gaurav', 'Shubham', 'Rashmi']
```

```
Enter the value to be searched: 'Sarthak'
```

```
Sarthak found at index None
```

```
Enter a list: [1001, 1006, 1002, 1005,
1004, 1003]
```

```
Enter the value to be searched: 1005
```

```
1005 found at index 3
```

```
Enter a list: [1001, 1006, 1002, 1005,  
1004, 1003]
```

```
Enter the value to be searched: 1008
```

```
1008 found at index None
```

examples of linear search

12.2.2 Binary Search

If the list to be searched is already sorted, we can use a faster method, called binary search. For searching a name, in a list of names that has been sorted in ascending order, we proceed as follows:

binary search works on a sorted sequence of values

First, we examine the middle name of the list, if it matches the name that we are looking for, we stop. If we do not find the name at the middle position, we will only have to search on the left or right side of the middle name depending on whether the name at the middle position is greater or smaller than the name we are looking for. Based on this idea, we present the `binarySearch` function in Fig. 12.33.

central idea in binary search

```
01 def binarySearch(lst, searchValue):
02     """
03         Objective: To search for an element in a sorted list using
04             binary search
05         Input Parameters:
06             lst - sorted list
07             searchValue - any type
08         Return Value: Index of the matched element, if searchValue is
09                         found, None otherwise
10         Assumption: List is sorted in increasing order
11     """
12     if lst[mid] == searchValue
13         # announce that search is successful
14     else if lst[mid] < searchValue
15         # perform binary search in the list to the right of
16             mid element
17     else
18         # perform binary search in the list to the left of
19             middle element
```

Fig. 12.33 Function for binary search

In Fig. 12.34, we present complete script
(binarySearch) for binary search.

```
01 def binarySearch(lst, searchValue):  
02     """  
03         Objective: To search for an element in a sorted list using  
04             binary search  
05         Input Parameters: lst - sorted list, searchValue - any type  
06         Return Value: Index of the matched element, if searchValue is  
07                         found, None otherwise  
08         Assumption: List is sorted in increasing order  
09     """  
10     low, high = 0, len(lst) - 1  
11     while low <= high:  
12         mid = int((low + high)/2)  
13         if lst[mid] == searchValue:  
14             return mid  
15         elif searchValue < lst[mid]:  
16             high = mid - 1  
17         else:
```

```
18             low = mid + 1  
19     return None  
20  
21 def isSorted(lst):  
22     """  
23         Objective: To find whether the list is sorted in ascending  
24             order.  
25         Input Parameter: lst - sorted list
```

```

25     Return Value: True if list is sorted
26             False otherwise
27     """
28     for i in range(1,len(lst)):
29         if lst[i] < lst[i-1]:
30             return False
31     return True
32
33 def main():
34     """
35     Objective: To search for an element in the sorted list provided
36             as an input by the user
37     Input Parameter: None
38     Return Value: None
39     """
40     lst = eval(input('Enter a sorted list (ascending order): '))
41     if not(isSorted(lst)):
42         print('Given list is not sorted')
43     else:
44         searchVal = eval(input('Enter the value to be searched: '))
45         print(searchVal, 'found at index', \
46               binarySearch(lst, searchVal))
47
48 if __name__=='__main__':
49     main()

```

Fig. 12.34 Program for binary search (binarySearch.py)

On execution of the script `binarySearch`, the user is prompted to enter a sorted list (line 40). In line 41, we invoke the function `isSorted` to find out whether the list entered by the user is sorted. If the list entered by the user happens to be sorted, the user is asked for the value to be searched and the control is transferred to the

function `binarySearch`. In line 10, values of the variables `low` and `high` are set to the first and the last indexes of the list, respectively. The value of `mid` is computed by taking an average of `low` and `high` in line 12. If we find `lst[mid] == searchValue` in line 13, the search terminates, else we reduce the range to be searched by modifying either `low` or `high` depending on the `searchValue`. But, we have yet to consider the case when the required value is not in the list. As the search always takes place between indices `low` and `high`, the search should not proceed if `low > high`. This fact is used to terminate the search loop in the function `binarySearch`.

stopping criteria for binary search: `low > high`

On executing the script `binarySearch` (Fig. 12.34), Python will prompt the user to enter list and the value to be searched and will display the appropriate message indicating presence or absence of value, for instance:

```
Enter a sorted list (ascending order):  
['Reena', 'Stuti', 'Monika', 'Deepika']
```

```
Given list is not sorted
```

```
Enter a sorted list (ascending order):  
[1001, 1002, 1003, 1004, 1005]
```

```
Enter the value to be searched: 1004
```

```
1004 found at index 3
```

```
Enter a sorted list (ascending order):  
['Ankit', 'Gaurav', 'Nitin', 'Rashmi',  
'Shilpi', 'Shubham', 'Shweta']
```

```
Enter the value to be searched: 'Gaurav'
```

```
Gaurav found at index 1
```

```
Enter a sorted list (ascending order):  
['Gaurav', 'Nitin', 'Rashmi', 'Shilpi',  
'Shubham', 'Shweta']
```

```
Enter the value to be searched: 'Shubham'
```

```
Shubham found at index 4
```

examples of binary search

Now we study the last example in detail (Fig. 12.35). In this example, we used binary search to look for the name 'Shubham' in the list ['Gaurav', 'Nitin', 'Rashmi', 'Shilpi', 'Shubham', 'Shweta']:

'Gaurav'	'Nitin'	'Rashmi'	'Shilpi'	'Shubham'	'Shweta'
0	1	2	3	4	5

Fig. 12.35 List of names of students

In the first iteration, the variables `low` and `high` are initialized to 0 to 5, respectively. When the name to be searched is compared against the name at index `mid` ($=2$), the name 'Shubham' is found to be greater than 'Rashmi'. Now the value of `low` is set to 3 so that the search can be continued in the index range 3 to 5. In the second iteration, taking 3 as `low` and 5 as `high`, the name 'Shubham' is compared with the name at index `mid` ($=4$). Since the name 'Shubham' is found at index 4, search terminates by exiting the loop. The search process has been illustrated in Fig. 12.36.

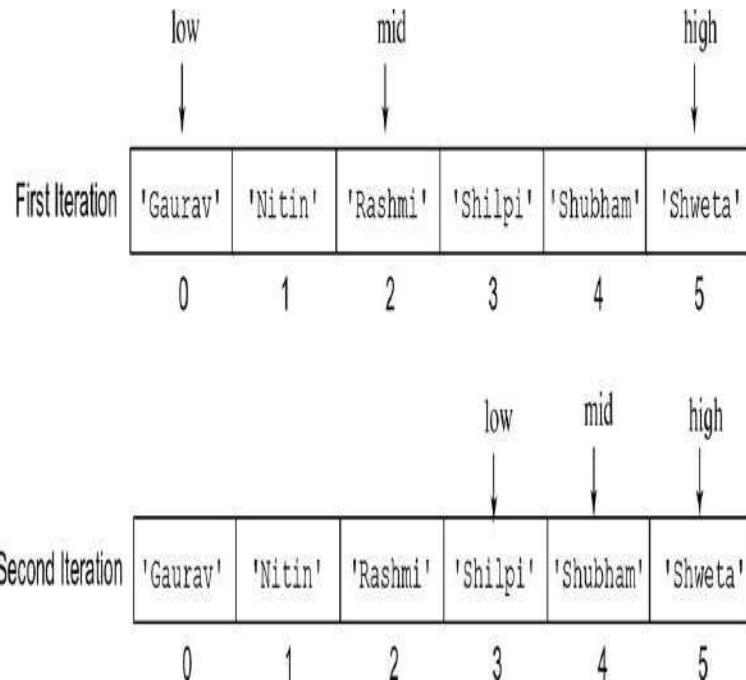


Fig. 12.36 Steps for binary search for data given in Fig. 12.35

Note that binary search on the above-sorted list took only two steps. However, if the same sorted list were to be searched for the name 'Shubham' using linear search, it would have taken five steps. To see how fast binary search works, we only have to observe that every time search fails, the search interval is reduced to half. Thus, in the worst case, we would require no more than $(\log_2 N) + 1$ steps.

binary search is more efficient than linear search

number of steps taken by binary search: $(\log_2 N) + 1$

! 12.3 A CASE STUDY

In this section, we will discuss how to prepare a merit list of students who took an examination. For each student, we store information about roll number, name, and marks obtained. Sample data is shown in Table 12.1.

details about student data

Table 12.1 Sample data for student information

Roll No.	Name	Marks
1	'Amit'	36
2	'Umang'	27
3	'Shweta'	85
4	'Alok'	50
5	'Kriti'	64
6	'Shashi'	55

We define a class `Student` having a data attribute `rollNo`, `name`, and `marks` (Fig. 12.37). Note that the class `Student` has two methods – constructor

`__init__` used for initializing data members of the class and the special method `__str__` for constructing the string representation of an object of class `Student`, to be used later for printing.

Let us introduce another class `Section` (Fig. 12.38) having data attribute `records` that store information about students in the form of a list of `Student` objects. In this class, we define various methods for operations on the list `records`. As we have not yet discussed the implementation of these methods, we use `pass` statement as the body of the function.


```
01 class Student:  
02  
03     def __init__(self, rollNo, name, marks):  
04         '''  
05             Objective: To initialize data members of object of type  
06             Student  
07             Input Parameters:  
08                 self (implicit parameter)- object of type Student  
09                 rollNo, marks - numeric  
10                 name - string  
11             Return Value: None  
12         '''  
13         self.rollNo = rollNo  
14         self.name = name  
15         self.marks = marks  
16  
17     def __str__(self):  
18         '''  
19             Objective: To return string representation of object of  
20             type Student  
21             Input Parameter: self (implicit parameter)- object of type  
22             Student  
23             Return Value: string  
24         '''  
25         return 'Roll No: ' + str(self.rollNo) + ', Name: ' +\br/>26             self.name + ', Marks: ' + str(self.marks) + '\n'
```

Fig. 12.37 Class Student (student.py)

```
01 import pickle
02 from student import Student
03 class Section:
04
05     def __init__(self):
06         # initialize data members of object of class Section
07         pass
08
09     def readList(self, source):
10         # read objects of Student class from source file
11         pass
```

337

```
12
13     def writeList(self, destination):
14         # write objects of Student class in list records
15         pass
16
17     def insertEnd(self, rollNo, name, marks):
18         # insert a Student instance at the end of list records
19         pass
20
21     def isSorted(self):
22         # determine whether the list records is sorted
23         # according to roll number in increasing order
24         pass
25
26     def binarySearch(self, rollNo):
```

```

27     # Find whether a Student instance with given rollNo exists
28     # in the list records
29     pass
30
31     def linearSearch(self, rollNo):
32         # Find whether a Student instance with given rollNo exists
33         # in the list records
34         pass
35
36     def insertionSort(self):
37         # sort the list records
38         pass
39
40     def sortedInsert(self, rollNo, name, marks):
41         # insert a Student instance in the sorted list records
42         pass
43
44     def delete(self, rollNo):
45         # delete a Student instance from the sorted list records
46         pass
47
48     def __str__(self):
49         pass

```

Fig. 12.38 Class Section (`section.py`)

We will use an object of class `Section`, to store the sample data in [Table 12.1](#) as list records (Fig. 12.39):

As first step, we develop a function `textToObject` (Fig. 12.40) that takes the comma-separated student data as input from a text file (`source`), converts it to structured data (objects of class `Student`), and writes to another file (`destination`). We also develop another function `objectToText` that takes a file (`source`) of objects of class `Student` as input and outputs the comma-separated student data as a text file (`destination`).

converting comma-separated student data to objects of class
Student

converting objects of class Student to comma-separated
student data.

Section instance

records	list					
	0	1	2	3	4	5
	Student instance					
	marks 36	marks 27	marks 85	marks 50	marks 50	marks 55
	name "Amit"	name "Umang"	name "Shweta"	name "Alok"	name "kriti"	name "Shashi"
	rollNo 1	rollNo 2	rollNo 3	rollNo 4	rollNo 5	rollNo 6

Fig. 12.39 Memory representation records of object section

```
01 import pickle
02 from student import Student
03 def textToObject(source, destination):
04     """
05     Objective: To convert text data in source file to data in the
06     form of Student objects and write to destination file
07     Input Parameters: source, destination - string
08     Return Value: None
09     """
10     f1 = open(source, 'r')
11     f2 = open(destination, 'wb')
12     line = f1.readline()
13     while line != '':
14         let = line.split(',')
```

```

11     line = f1.readline()
12
13     student = Student(int(lst[0]), lst[1], float(lst[2]))
14     pickle.dump(student, f2)
15
16     line = f1.readline()
17
18     f1.close()
19
20     f2.close()
21
22
23     def objectToText(source, destination):
24         """
25             Objective: To convert Student object data in source file to
26             text data and write to destination file
27             Input Parameters: source, destination - string
28             Return Value: None
29
30         """
31         f1 = open(source, 'rb')
32         f2 = open(destination, 'w')

```

339

```

31     try:
32         while True:
33             student = pickle.load(f1)
34             line = str(student.rollNo) + ',' + student.name + ','\
35             +str(student.marks)+ '\n'
36             f2.write(line)
37     except EOFError:
38         # Exception is raised when we attempt to read even after
39         # all objects have been read
40         pass
41     f1.close()
42     f2.close()

```

Fig. 12.40 Reading and writing structured data to and from a file
(impExpObject.py)

12.3.1 Operations for Class Section

The data about students is stored in an object of the class `Section` as a list `records` of the objects of the class `Student`. On this list, we will perform the list manipulation operations: insert a student in a section, search for a student in a section, sort a section, delete a student, and list all students in a section. Next, we describe various methods for performing these operations and finally present a complete script for processing student data.

list manipulation operations on the list of objects of the class `Student`

1. `__init__()`
The `__init__` method (Fig. 12.41) initializes an instance of the `Section` class by initializing its list `records` (of the `Student` class) as an empty list.

initializing an instance of class `Section`

```
01  def __init__(self):
02      """
03          Objective: To initialize data members of object of class
04          Section
05          Input Parameters:
06              self (implicit parameter)- object of type Section
07          Return Value: None
08      """
09      self.records = list()
```

Fig. 12.41 Method `__init__`

2. `readList(self, source)`
The `readList` method (Fig. 12.42) accepts the name of the `source` file containing `Student` objects as an input parameter and appends the `Student` objects read from the `source` file to the list `records` of the `Section` instance at hand.

reading the `Student` objects for list `records` from `source` file

```

01 def readList(self, source):
02     """
03         Objective: To read objects of Student class from source file
04         and insert them to list records
05         Input Parameters:
06             self (implicit parameter)- object of the type Section
07             source - string (file name)
08         Return Value: None
09     """
10     self.records = list()
11     f = open(source, 'rb')
12     try:
13         while True:
14             student = pickle.load(f)
15             self.records.append(student)
16     except EOFError:
17         pass
18     print(self)
19     f.close()

```

Fig. 12.42 Method readList

3. writeList(self, destination)

The writeList method (Fig. 12.43) accepts the name of the destination file as an input parameter and writes to it one by one the Student objects in the list records of the Section instance at hand.

writing the Student objects in list records to destination file

```

01 def writeList(self, destination):
02     """
03         Objective: To write objects of Student class in list records
04         to destination file
05         Input Parameters:
06             self (implicit parameter)- object of the type Section
07             destination - string (file name)
08         Return Value: None
09     """
10     f = open(destination, 'wb')
11     for student in self.records:
12         pickle.dump(student, f)
13     f.close()

```

Fig. 12.43 Method writeList

4. insertEnd(self, rollNo, name, marks)

For inserting information about a new student, we define the method insertEnd (Fig. 12.44). The method insertEnd accepts rollNo, name, and marks as parameters, creates an object student of type Student, and inserts it in the list records. In this method, we first check whether the list records already has an object with the same roll number as the object being inserted. If so, an error message indicating duplicate roll number is returned, otherwise the record is appended to the list records and a message indicating successful record insertion is returned.

inserting Student instance at the end of list records

```
01  def insertEnd(self, rollNo, name, marks):
02      """
03          Objective: To insert a student record of type Student to
04          list records
05          Input Parameters:
06              self (implicit parameter) - object of the type Section
07              rollNo, marks - numeric
08              name - string
09          Return Value: string :indicates whether the record is
10          successfully inserted
11      """
12      student = Student(rollNo, name, marks)
13      if student.rollNo not in [s.rollNo for s in self.records]:
14          self.records.append(student)
15      return 'Record inserted successfully'
16      return 'Record cannot be inserted! Duplicate roll no.'
```

Fig. 12.44 Method insertEnd

5. `isSorted(self)`

If the list `records` is arranged in ascending order of the values of the key `rollNo`, the method `isSorted` (Fig. 12.45) returns True, otherwise it returns False.

checking whether the Student objects are sorted with respect to
roll numbers in list records

```
01  def isSorted(self):
02      """
03          Objective: To determine whether the list records is sorted
04          according to the field roll number in increasing
05          order
06          Input Parameter: self (implicit parameter)- object of type
07              Section
08          Return Value: True if list records is sorted
09                  False otherwise
10      """
```

```
11      for i in range(1,len(self.records)):
12          if self.records[i].rollNo < self.records[i-1].rollNo:
13              return False
14      return True
```

Fig. 12.45 Method isSorted

6. `binarySearch(self, rollNo)`

The `binarySearch` method (Fig. 12.46) is applicable if the list `records` is already sorted. It accepts `rollNo` as an input parameter. The `Student` objects are searched for the given `rollNo` in the list `records` using the binary search procedure described in the previous section. If a `Student` object with the given `rollNo` is found in the list `records`, the method returns the index of the matched object, otherwise it returns `None`.

searching for the roll number of a student in sorted list records

```

01 def binarySearch(self, rollNo):
02     """
03         Objective: To search for a roll number in the sorted list
04         records using binary search
05         Input Parameters:
06             self (implicit parameter) - object of type Section
07             rollNo - Numeric(search value)
08         Return Value: Index of the matched object, if value is
09             found else None
10         Assumption: List records is sorted in ascending order by
11             rollNo
12         """
13     if not(self.isSorted()):
14         print('List is not sorted')
15         return False
16     low, high = 0, len(self.records) - 1
17     while low <= high:
18         mid = int((low + high)/2)
19         if self.records[mid].rollNo == rollNo:
20             return mid
21         elif rollNo < self.records[mid].rollNo:
22             high = mid - 1
23         else:
24             low = mid + 1
25     return None

```

Fig. 12.46 Method `binarySearch`

7. `linearSearch(self, rollNo)`
The `linearSearch` method (Fig. 12.47) is used if the list `records` is not sorted. It accepts `rollNo` as an input parameter. The `Student` objects are searched sequentially in the list `records` for the given `rollNo` in the list `records` using the linear search procedure described in the previous section. If a `Student` object with the given `rollNo` is found in the list `records`, the method returns the index of the matched object, otherwise it returns `None`.

searching for the roll number of a student in the unsorted list
`records`

```

01 def linearSearch(self, rollNo):
02     """
03     Objective: To search for a roll number in list of records
04     using linear search
05     Input Parameters:
06         self (implicit parameter)- object of type Section
07         rollNo - Numeric(search value)
08     Return Value: Index of the matched object, if value is
09     found else None
10     """
11     for i in range(0, len(self.records)):
12         if self.records[i].rollNo == rollNo:
13             return i
14     return None

```

Fig. 12.47 Method linearSearch

8. insertionSort(self)

In the previous section, we discussed three techniques of sorting a list of objects. Now, we make use of the insertion sort technique (Fig. 12.48) to sort the list records of Student objects, based on rollNo.

sorting list records of Student objects based on roll number using insertion sort

```

01 def insertionSort(self):
02     """
03     Objective: To sort the list in ascending order of rollNo
04     using insertion sort
05     Input Parameter:
06         self (implicit parameter)- object of type Section
07     Return Value: None
08     """

```

```

09     for i in range(1, len(self.records)):
10         temp = self.records[i]
11         j = i - 1
12         while j >= 0 and self.records[j].rollNo > temp.
13             self.records[j+1] = self.records[j]
14             j = j-1
15         self.records[j + 1] = temp

```

Fig. 12.48 Method insertionSort

9. delete(self, rollNo)

The delete method (Fig. 12.49) accepts rollNo as the input parameter. It searches the entire list records sequentially for the Student object having the given rollNo. If the desired Student object is found, it is removed from the list records and the method returns the message 'Record deleted successfully' to indicate successful termination. However, if the desired Student object having the given rollNo is not found, the method returns the message 'Record not found'.

deleting a Student instance with the given roll number from the list records

```
01  def delete(self, rollNo):
02      """
03          Objective: To remove a student record of type Student from
04          list records having roll number rollNo
05          Input Parameters:
06              self (implicit parameter) - object of type Section
07              rollNo - Numeric
08          Return Value: string (indicates whether the record is
09              successfully deleted)
10      ...
11      for i in range(0, len(self.records)):
12          if self.records[i].rollNo == rollNo:
13              del self.records[i]
14              return 'Record deleted successfully'
15      return 'Record not found'
```

Fig. 12.49 Method delete

10. sortedInsert(self, rollNo, name, marks)
The method accepts as input parameters rollNo, name, and marks of the student for whom we want to insert a Student object in the list records. The method assumes that the list record is already sorted with respect to roll numbers. If the list is an empty list, the method appends the student instance to the empty list. Otherwise, it finds an appropriate index at which record is to be inserted and inserts the record at that place. The method rejects a record if the roll number is duplicated. In Fig. 12.50, we define this method.

inserting Student instance at appropriate place in the sorted list records

```

01 def sortedInsert(self, rollNo, name, marks):
02     """
03         Objective: To insert a student record of type Student to
04             sorted list records
05         Input Parameters:
06             self (implicit parameter) - object of type Section
07             rollNo, marks - numeric
08             name - string
09         Return Value: string (indicates whether the record is
10             successfully inserted)
11         Assumption: List is already sorted with respect to roll
12             number field
13             """
14
15     student = Student(rollNo, name, marks)
16     if len(self.records) == 0:
17         self.records.append(student)
18         return 'Record inserted successfully '
19
20     (low, high) = (0, len(self.records) - 1)
21     mid = int((low + high)/2)
22     while low <= high:
23         mid = int((low + high)/2)
24         if self.records[mid].rollNo == rollNo:
25             return 'Record cannot be inserted! Duplicacy in
26             roll no.'
27         elif rollNo < self.records[mid].rollNo:
28             high = mid - 1
29         else:
30             low = mid + 1
31
32         if self.records[mid].rollNo < rollNo:
33             self.records.insert(mid+1, student)
34         else:
35             self.records.insert(mid, student)
36         return 'Record inserted successfully '

```

Fig. 12.50 Method sortedInsert

11. __str__ (self)
The method __str__ (Fig. 12.51) of class Section constructs the string representation for an instance of the class. It traverses the list records invoking __str__ function for each Student object.

returning string representation of object of type Section

```

01     def __str__(self):
02         """
03             Objective: To return string representation of object of
04             type Section
05             Input Parameter: self (implicit parameter)- object of type
06             Section
07             Return Value: string
08             """
09         strl= ''
10         for i in self.records:
11             strl += Student.__str__(i)
12

```

Fig. 12.51 Method `__str__`

Complete Script (Fig. 12.52)

Now we are ready to present a complete script for processing student data. Before doing any list processing, we need to read the raw data from a text file and convert it to a file of objects. Similarly, after the processing, we need to write the structured data back to a text file. The script works as follows:

1. Invoke the function `textToObject` (line 14) from the module `impExpObject` for converting the student information in a text file to a file of `Student` objects. In future, we may use this file of `Student` objects instead of the text file. The data in the text file appears in the following format:
 1,Amit,36
 5,Kriti,50
 4,Alok,50
 2,Umang,27
 3,Shweta,85
 working of the script for processing student data
2. Create an instance `section` of the `Section` class and initialize its list `records` of `Student` instances from the file created in step 1 (lines 16–18). This list of records is unsorted as of now.
3. Search for a student in the `section` using linear search (lines 20–26).
4. Sort the `Student` objects in the `section` based on `rollNo` and write in a file. In future, we may use the sorted file (lines 28–30).
5. Read the sorted file created in step 4 to create an instance `section` of the `Section` class and use binary search to look for a `rollNo` (lines 32–42).
6. Read the sorted file created in step 4 to create a text file for viewing it using a text editor (line 44).

```

01 from student import Student
02 from section import Section
03 from impExpObject import *
04
05 def main():
06     """
07     Objective: To provide list manipulation functionality
08     Input Parameter: None
09     Return Value: None
10     """
11     source = input('Enter source file name: ')
12     destination = input('Enter destination file name: ')
13     # read text file and create a file of Student objects.
14     textToObject(source, destination)
15
16     section = Section()
17     #Reading a a file of Student objects and
18     section.readList(destination)
19
20     #searching for a record
21     rollNo = int(input('Enter RollNo for record to be searched '))
22     message = section.linearSearch(rollNo)
23     if message != None:
24         print(rollNo, ' found at index ', message)
25     else:
26         print(rollNo, ' not found')
27
28     # sorting and writing it back.
29     section.insertionSort()
30     section.writeList(destination)
31
32     #Reading a list and searching for a record
33     section.readList(destination)
34     rollNo = int(input('Enter RollNo for record to be searched '))
35     message = section.binarySearch(rollNo)
36     if message == False:
37         print('List is not sorted')
38     else:
39         if message != None:
40             print(rollNo, ' found at index ', message)

```

```

41     else:
42         print(rollNo, ' not found')
43
44     objectToText(destination, source)
45
46 if __name__=='__main__':
47     main()

```

Fig. 12.52 Program for processing student data
(listManipulation.py)

Next, we show the working of the above script (Fig. 12.52) with the help of an example:

```
Enter source file name: student.txt

Enter destination file name:
studentObjects.txt

Roll No: 1, Name: Amit, Marks: 36.0

Roll No: 5, Name: Kriti, Marks: 50.0

Roll No: 4, Name: Alok, Marks: 50.0

Roll No: 2, Name: Umang, Marks: 27.0

Roll No: 3, Name: Shweta, Marks: 85.0

Enter RollNo for record to be searched 3

3 found at index 4

Roll No: 1, Name: Amit, Marks: 36.0

Roll No: 2, Name: Umang, Marks: 27.0

Roll No: 3, Name: Shweta, Marks: 85.0

Roll No: 4, Name: Alok, Marks: 50.0

Roll No: 5, Name: Kriti, Marks: 50.0

Enter RollNo for record to be searched 6

6 not found
```

Suppose we wish to sort the student data on the basis of the particular key attribute. For this purpose, we develop method `insertionSort` (Fig. 12.53) that takes the choice as an input parameter and sorts the student data based on the value of attributes `rollNo`, `name`, or `marks` depending upon 1, 2, or 3 as the user choice respectively. This method uses the method `key` to retrieve the value of the key attribute based on the user choice. Recall that the method `insertionSort` is a member of class `Section`. Also, the method `key` should be included as a member of the class `Student`.

```

01     def key(self, choice):
02         """
03             Objective: To return key value based on user choice
04             Input Parameter:
05                 self (implicit parameter)- object of type Student
06                 choice: for specifying sorting field

```

```

07         Return Value: string or numeric data based on user input
08         """
09         if choice == 1:
10             return self.rollNo
11         elif choice == 2:
12             return self.name
13         else:
14             return self.marks
15
16     def insertionSort(self,choice):
17         """
18             Objective: To sort the list in ascending order of field
19             using insertion sort
20             Input Parameters:
21                 self (implicit parameter)- object of type Section
22                 choice - for specifying sorting field
23             Return Value: None
24         """
25         for i in range(1, len(self.records)):
26             temp = self.records[i]
27             j = i - 1
28             while j >= 0 and self.records[j].key(choice) > \
29                 temp.key(choice):
30                 self.records[j+1] = self.records[j]
31                 j = j-1
32             self.records[j + 1] = temp

```

Fig. 12.53 Methods `insertionSort` and `key`

12.4 MORE ON SORTING

12.4.1 Merge Sort

The merge sort algorithm is based on the divide and conquer strategy that takes a list as an input and keeps on dividing it into two lists until a list of length 1 is obtained. Notice that when a list of even length is divided into two lists, each of them will be of the same size, but in case the original list is of odd length, the two lists would differ in size by one. It conquers the sorting problem by merging pairs of length one lists to obtain lists of length two, merging pairs of lists of length two to obtain lists of length four, and so on until the final sorted list of the size of the original list is obtained. It is

important to point out that the lists being merged may not always be of identical size. It would become clear when we show the working of the algorithm with the help of an example.

merge sort is based on divide and conquer strategy

We have provided a function `mergeSort` (Fig. 12.54) that takes a list as an input parameter and returns the sorted list. Note that if the length of a list is 0 or 1, the function `mergeSort` returns it as it is (the trivial case as the list is already sorted). Otherwise, it divides the list into two parts. When each of these two parts is sorted, it invokes the function `merge` that takes two sorted lists (`lst1` and `lst2`) and merges them into a new single sorted list `sortedLst`. The function `merge` works by comparing front elements of the two sorted lists and removing the one with the smaller value, which is placed in `sortedLst`. When any of the two lists becomes empty, the values in the other list are appended to the list `sortedLst`.

```
01 def merge(lst1, lst2):
02     """
03     Objective: To merge two sorted lists into a single sorted list
04     Input Parameter: lst1, lst2 - list
05     Return Value: sorted list
06     """
07     sortedLst = []
08     while len(lst1) != 0 and len(lst2) != 0:
09         if lst1[0] < lst2[0]:
10             sortedLst.append(lst1[0])
11             lst1.remove(lst1[0])
12         else:
13             sortedLst.append(lst2[0])
14             lst2.remove(lst2[0])
15     if len(lst1) == 0:
```

```

16         sortedLst += lst2
17     else:
18         sortedLst += lst1
19     return sortedLst
20
21 def mergeSort(lst):
22     """
23     Objective: To sort the list lst in ascending order using merge
24     sort
25     Input Parameter: lst - list
26     Return Value: sorted list
27     """
28     if len(lst) == 0 or len(lst) == 1:
29         return lst
30     else:
31         mid = len(lst)//2
32         lst1 = mergeSort(lst[:mid])
33         lst2 = mergeSort(lst[mid:])
34         return merge(lst1,lst2)
35

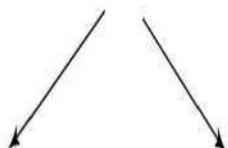
```

Fig. 12.54 Program for merge sort (`mergeSort.py`)

The splitting and merging process of merge sort on the list [15, 12, 14, 17, 13, 11, 12, 16, 14] is illustrated using Figs. 12.55 and 12.56.

Splitting Process

15	12	14	17	13	11	12	16	14
0	1	2	3	4	5	6	7	8



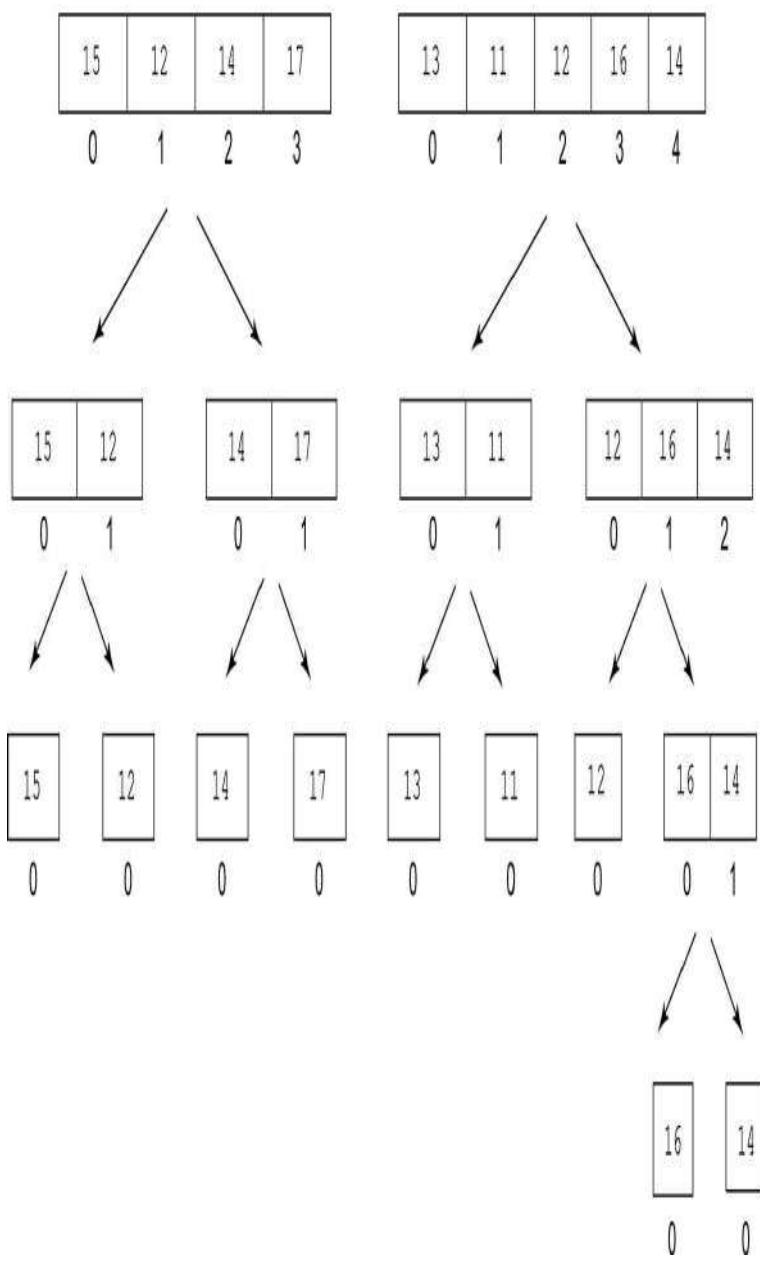


Fig. 12.55 Dividing the list until singleton lists are obtained

12.4.2 Quick Sort

As compared to merge sort technique, discussed in previous section, quick sort is more efficient in terms of the space requirement as it does not need extra space for sorting. It works by choosing a value called pivot element, using which we split the list to be sorted in two parts: the first part comprising elements smaller than or

equal to the pivot element and the second part comprising elements greater than the pivot element. The two parts are further sorted using quick sort in a similar manner until the partitions of length 1 or 0 are left. Although there are different methods of choosing the pivot element, for the sake of simplicity, we choose the last element of the list/partition under consideration as the pivot element.

quick sort uses a pivot element for partitioning the list around it

We have provided a function `quickSort` (Fig. 12.57) that takes a list as an input parameter along with start and end indexes. If the length of the list is greater than 2, the function `quickSort` invokes the function `partition` that divides the list to be sorted (list `lst` from index `start` to `end`) into two parts around the pivot element, followed by call to function `quicksort` for each of the two parts. The function `partition` maintains two regions: `start` to `i` that keeps track of elements smaller than or equal to pivot element and the region from `i+1` to `j-1` (highlighted region in Fig. 12.58) that keeps track of elements greater than the pivot element. The function works by comparing every element `lst[j]` in the sub-list under consideration with the pivot element, and if it is less than or equal to the pivot element, value of index `i` (initialized to `start - 1`) is incremented and the element at the incremented index `i` is swapped with the element at index `j`. Thus, we ensure that elements smaller than or equal to the pivot element are in the region `start` to `i`. Finally, in line 15, we swap the elements `lst[i+1]` and `lst[end]` so that on the left of the pivot element lie elements smaller or equal to it and on its right lie the elements larger than it. For example, examine the execution of function `partition` for the list [12, 18, 17, 11, 13, 14] (Fig. 12.58). Figure 12.58 illustrates how the function `partition` works for the first call to the function.

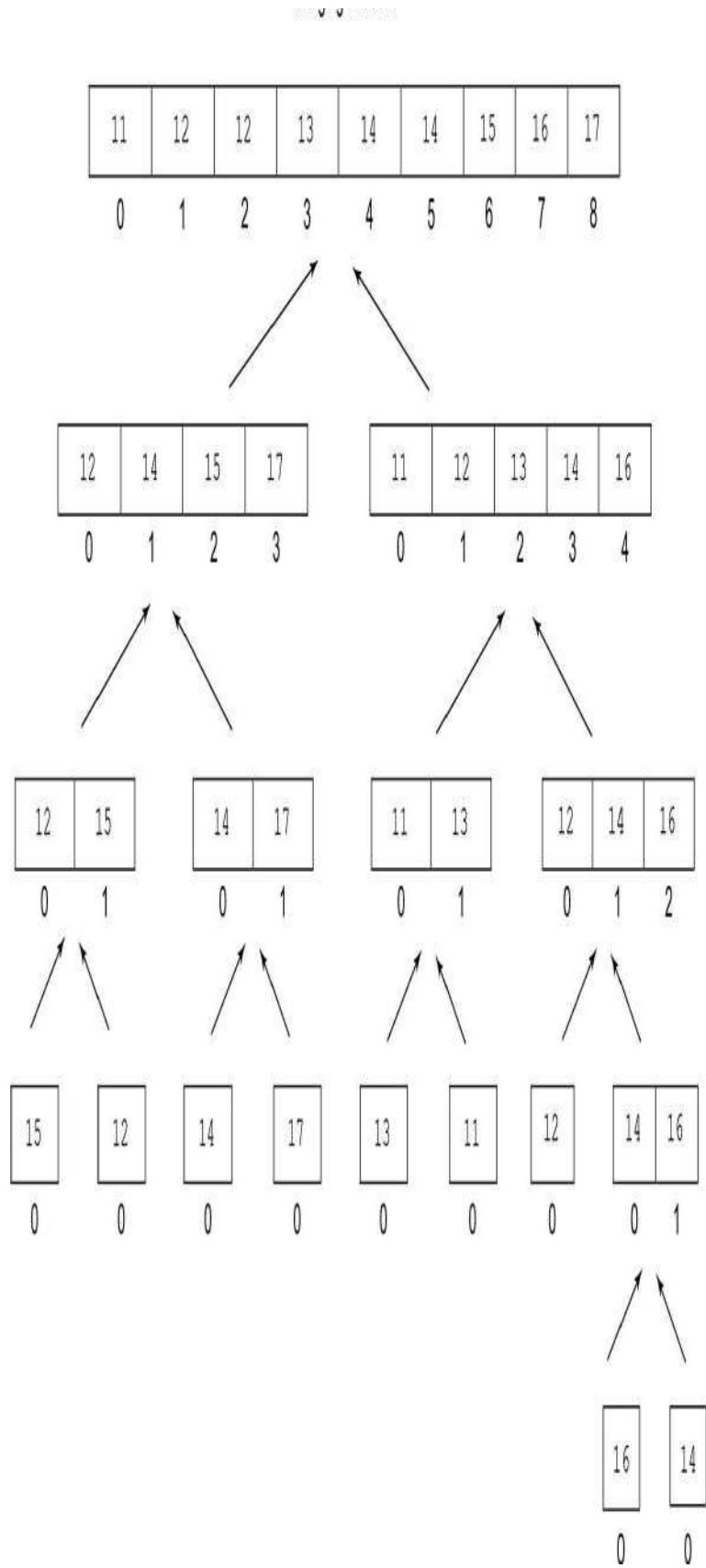


Fig. 12.56 Merging the sorted lists to obtain a single sorted list

```
01 def partition(lst, start, end):
02     """
03         Objective: To partition lists into two parts around pivot
04         element
05         Input Parameter: lst - list; start, end - numeric value
06         Return Value: index - integer value (index of pivot element
07                         in sorted list)
08     """
09     pivotElement = lst[end]
10     i = start-1
11     for j in range(start, end):
12         if lst[j] <= pivotElement:
13             i += 1
14             lst[i], lst[j] = lst[j], lst[i]
15     lst[i+1], lst[end] = lst[end], lst[i+1]
16     return i+1
17
18
19 def quickSort(lst, start = 0, end = None):
20     """
21         Objective: To sort the list lst in ascending order using quick
22         sort
23         Input Parameter: lst - list
24         Return Value: sorted list
25     """
26     if end == None: #Quick sort invoked for the first time
27         end = len(lst) - 1
28     if start < end:
29         splitPoint = partition(lst, start, end)
30         quickSort(lst, start, splitPoint-1)
```

31

```
quickSort(lst, splitPoint+1, end)
```

Fig. 12.57 Program for quick sort (`quickSort.py`)

(a)

12	18	17	11	13	14
0	1	2	3	4	5
i=-1	start, j				end

(b)

12	18	17	11	13	14
0	1	2	3	4	5
start, i	j				end

(c)

12	18	17	11	13	14
0	1	2	3	4	5
start, i		j			end

(d)

12	18	17	11	13	14
0	1	2	3	4	5
start, i		j			end

(e)

12	11	17	18	13	14
0	1	2	3	4	5
start	i		j		end

(f)

12	11	13	18	17	14
0	1	2	3	4	5
start		i			end

(g)

12	11	13	14	17	18
0	1	2	3	4	5
start		i			end

Fig. 12.58 Program for quick sort (quickSort.py)

1. The process of arranging data in ascending/descending order is called sorting. The attribute that forms the basis of sorting is called key.
2. In selection sort, to begin with, we find the smallest value in the list, and interchange this entry with the first entry in the list. Next, we find the entry with the smallest value out of the remaining entries, and interchange it with the second value in the list, and proceed in this manner. If there are n values, only $(n - 1)$ values need to be placed in order because when $(n - 1)$ values have been put in the proper place, then the n th value would automatically be in order.
3. In bubble sort, in each pass, we compare values in adjacent positions and interchange them, if they are out of order. Suppose we have a list of n values. To begin with, the n th and $(n - 1)$ th values are compared and interchanged if found out of order; then $(n - 1)$ th and $(n - 2)$ th values are compared and interchanged if found out of order, and so on. Observe that in the first pass the smallest value will move to the front of the list at index 0; on subsequent passes, it will be ignored. After $n - 1$ iterations, the list will be completely sorted and the algorithm will halt.
4. In insertion sort, the list is logically divided into two parts. Whereas the left part is the sorted part, the right part is unsorted part comprising the elements yet to be arranged in sorted order. In each iteration, we increase the length of the sorted part by one in the following manner: insert the first element from the unsorted part in the sorted part at the correct position. To find the correct position of the value to be inserted, we compare it with values in sorted part and shift each value to the right by one position, until the correct position is found.
5. Searching is the task of finding whether a data value appears in a list.
6. In linear search of a data value in a list, we scan the list from the beginning till the data value is found, or the list is exhausted. This technique is known as linear search as the order of search is linear or sequential in nature.
7. Binary search is used in case the list to be searched is already sorted. For searching a data value in this list, we proceed as follows: first, we examine the middle element of the list, if it is equal to the data value that we are looking for, we will stop. If we do not find the element at the middle position, we will only have to search on the left or the right of the middle element depending on whether the data value at the middle position is greater or smaller than the data value we are looking for.
8. Merge sort and quick sort methods make significantly less number of comparisons as compared to selection sort, insertion sort, and bubble sort methods.

EXERCISES

1. Develop a program to sort the employee data on the basis of pay of the employees using (i) selection sort, (ii) bubble sort algorithm, and (iii) insertion sort. Consider

- a list `L` containing objects of class `Employee` having `empNum`, `name`, and `salary`.
2. Write the recursive version of linear search and binary search algorithms, discussed in the text.
 3. Rewrite selection sort, bubble sort and insertion sort functions using recursion.
 4. For the list shown below, show the values of the indexes `low`, `high`, and `mid` at each step of the binary search method discussed in the text when we are searching for the key:

10	15	22	24	45	55
0	1	2	3	4	5

- 1. 15
- 2. 25
- 3. 55
- 4. 40
- 5. 22

5. Write a function `leftCirculate` that takes a list as an input and left circulates the values in the list so that in the final list, each value is left shifted by one position and leftmost value in the original list now appears as the rightmost value. For example, on execution of the function on the list `[1, 2, 3, 4, 5]` it would be transformed to the list `[2, 3, 4, 5, 1]`. Modify the function to include a numeric argument to specify the number of positions by which left rotation is to be carried out.
6. Write a program that defines a class `Card` which can be used to instantiate cards with a particular rank and suit. Create another class `DeckOfCards` for maintaining a sorted list of cards using a method `sortedInsert` that takes an object of class `Card` as an input parameter and inserts it at the suitable position in the sorted list.

!
This section may be skipped without loss of continuity.

CHAPTER 13

DATA STRUCTURES I: STACK AND QUEUES

CHAPTER OUTLINE

13.1 Stacks

13.2 Queues

Oxford dictionary defines data as follows:

Facts and statistics collected together for reference or analysis:

The quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

Philosophy : Things known or assumed as facts, making the basis of reasoning or calculation.

dictionary meaning of data

In this chapter, we will be concerned with the first meaning of the term data. Further, Oxford Dictionary defines *structure* as an arrangement of and relations between the parts of something, a building or other object constructed from several parts. The terms such as construction, form, formation, shape, combination, conformation, pattern, and mold are used as synonyms for structure. Thus, the phrase *data structure* refers to construction, formation, shape, combination, conformation, pattern, mold, or structure made from simple forms of data. In this chapter, we shall study the data structures stacks and queues, and methods to implement them using lists in Python.

dictionary meaning of structure

data structures

13.1 STACKS

In a stack, objects are stored one over the other, and these objects leave in the reverse order of their arrival. For example, in a restaurant, the waiter puts the used plate one above the other. The cleaner takes a plate from the top, cleans it, and puts it onto another stack, from which the plate placed on top can be picked up for eating. Note that the used plate that is placed last is cleaned first and the plate that is cleaned last is taken first for eating. This is called *Last In First Out* (LIFO) arrangement. We observe another example of *Last In First Out* arrangement in the library, where the readers stack used books one above other. Subsequently, a staff member in the library picks up books one by one from the top of the stack and places them in appropriate shelves.

stack: objects leave in the reverse order of their arrival

stack: Last In First Out (LIFO) arrangement

Based on the above discussion, a stack is characterized by the following operations (amongst others, to be discussed shortly):

- push: Place (push) a new element on top of the stack. The push operation is also called insertion of an element in a stack.
- pop: Remove (pop) top element from the stack. The pop operation is also called deletion of an element from a stack.

push: to place new element on top of the stack

pop: to remove top element from the stack

The easiest way to represent a stack is using a list. As stack operations push and pop relate to the top element of the stack, we need to keep track of the top element of the stack. Remember, it is only the top element which is accessible. At any point in time, insertion or removal of an object takes place at the top of the stack. For example, beginning with an empty stack, we perform the following operations in a sequence.

insertion and deletion of elements take place from top of the stack

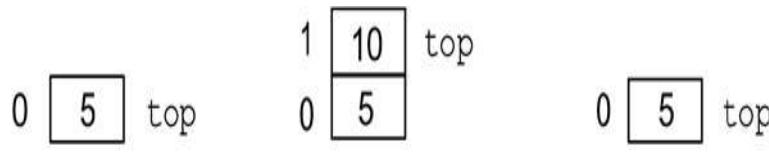
1. push 5
2. push 10
3. pop
4. push 20
5. push 25
6. push 30
7. pop
8. pop
9. pop
10. pop
11. pop

In Fig. 13.1, we show the stack, in the form of a list, on execution of each of the above operations. Note that in step 11, we tried to pop an element from the stack when it was already empty. When an attempt is made to pop from an empty stack, it is called underflow condition.

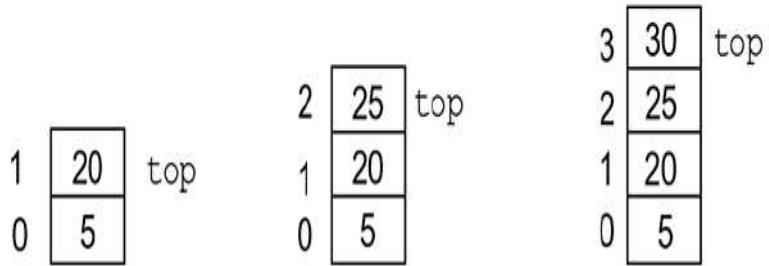
underflow condition: pop operation on an empty stack

In light of the above discussion, we define the class Stack (Fig. 13.2). The following operations are often required to be performed on a stack: check if the stack is empty, push an object, pop an object, retrieve value at the top of the stack without removing it from the stack, find the number of elements in the stack, print contents of the stack. Note that we do not discuss an operation to

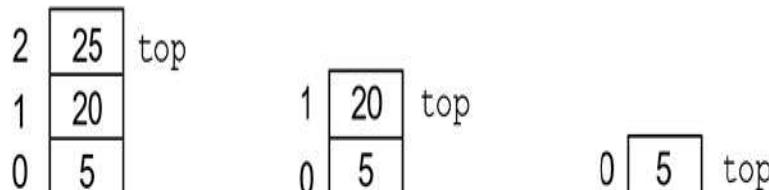
test whether the stack is full as append operation in a list is limited only by the physical storage available to the program.



1. push 5 2. push 10 3. pop



4. push 20 5. push 25 6. push 30



7. pop 8. pop 9. pop

10. pop 11. pop

Stack underflow

Fig. 13.1 Stack operations

stack underflow

In Fig. 13.2, we present the above-mentioned stack operations using inbuilt list operations `append` and `pop`. The constructor for the class `Stack` defines an empty list – `values` – to store the data objects that would be pushed on the stack. Next, we describe each of the above-mentioned operations for the class `Stack`.

use of list operations `append` and `pop` for implementing stack operations

```
01 class Stack:  
02  
03     def __init__(self):  
04         '''  
05             Objective: To initialize data members of the Stack  
06             Input Parameter:  
07                 self (implicit parameter) - object of type Stack  
08             Return Value: None  
09         '''  
10         self.values = list()  
11  
12     def push(self, element):  
13         '''  
14             Objective: To put an element on top of the stack  
15             Input Parameters:  
16                 self (implicit parameter) - object of type Stack  
17                 element - value to be inserted  
18             Return Value: None
```



```
19     """
20     self.values.append(element)
21
22     def isEmpty(self):
23         """
24             Objective: To determine if the stack is empty
25             Input Parameter:
26                 self (implicit parameter) - object of type Stack
27             Return Value: True if the stack is empty, else False
28         """
29         return len(self.values) == 0
30
31     def pop(self):
32         """
33             Objective: To remove an element from the top of stack
34             Input Parameter:
35                 self (implicit parameter) - object of type Stack
36             Return Value: top element of the stack, if stack is
37                         not empty, else None
38         """
39         if not(self.isEmpty()):
40             return self.values.pop()
41         else:
42             print('Stack Underflow')
43             return None
44
45
46     def top(self):
47         """
48             Objective: To return top element of the stack
49             Input Parameter:
50                 self (implicit parameter) - object of type Stack
51             Return Value: Top element of the stack, if stack is not empty,
52                         else None
53         """
54         if not(self.isEmpty()):
55             return self.values[-1]
56         else:
57             print ('Stack Empty')
58             return None
59
60     def size(self):
61         """
62             Objective: To return the number of elements in the stack
63             Input Parameter:
```

```

 03      Input Parameter:
 64          self (implicit parameter) - object of type Stack
 65      Return Value: number of elements in stack - numeric
 66      """
 67      return len(self.values)
 68
 69
 70  def __str__(self):
 71      """
 72          Objective: To return string representation of a stack
 73          Input Parameter: self (implicit parameter) - object of
 74          type Stack
 75          Return Value: string
 76          """
 77          stringRepr = ''
 78          for i in reversed(self.values):
 79              stringRepr += str(i) + '\t'
 80          return stringRepr

```

Fig. 13.2 Class Stack (stack.py)

1. Is Stack Empty

The method `isEmpty` determines whether the stack is empty by checking whether the length of the list `values` is zero, i.e. whether `len(self.values) == 0`. The method returns `True` if the stack is empty, and `False` otherwise.

2. Push an Object

The method `push` takes the element to be pushed as an input parameter and puts it at the end of the list `values` by invoking the method `append`.

3. Pop an Object

The method `pop` first checks whether the stack is empty by invoking the method `isEmpty`. If the stack is not empty, the object at the top of the stack (i.e., the element that was pushed on the stack last) is popped and returned. Otherwise, the message 'Stack Underflow' is displayed and `None` is returned indicating empty stack.

4. Retrieve Value at the Top of Stack

The method `top` returns value at the top of the stack if the stack is not empty. Otherwise, `None` is returned indicating empty stack. The stack contents remain unchanged.

5. Find the Number of Elements in the Stack

The method `size` returns the number of elements in the stack by returning the length of the list `values`. For the purpose of demonstration, we have provided this method to return number of elements in the stack, even though stack operations do not require this method.

6. Print Contents of the Stack

The method `__str__` produces a string representation of the entire stack by concatenating the string representation of each value in the list `values`. The string representation is used to print the contents of the stack. For the purpose of demonstration, we have provided this method to display the contents of the entire stack, even though stack operations do not require this method.

On executing the script `stackOperations` (Fig. 13.3), it prompts the user to choose from various actions that can be performed on a stack.


```
01 from stack import Stack
02 def main():
03     """
04     Objective: To provide stack functionality
05     Input Parameter: None
06     Return Value: None
07     """
08
09     while 1:
10         print('Choose an option \n')
11         print('1: Create stack')
12         print('2: Delete stack')
13         print('3: Push')
14         print('4: Pop')
15         print('5: Print Stack Data')
16         print('6: Top element')
17         print('7: No. of elements')
18         choice = int(input('Enter Choice: '))
19         if choice == 1:
20             stk = Stack()
21             print('Stack Created')
22         elif choice == 2:
23             del stk
24             print('Stack Deleted')
25         elif choice == 3:
26             element = int(input('Enter element to be inserted: '))
27             stk.push(element)
28         elif choice == 4:
29             print('Element poped:', stk.pop())
30         elif choice == 5:
31             print(stk)
32         elif choice == 6:
```

```
33     print('Top element',stk.top())
34 elif choice == 7:
35     print('No. of elements', stk.size())
36 proceed = input('enter y if you wish to continue: ')
37 if proceed != 'y' and proceed != 'Y':
38     break
39
40 if __name__=='__main__':
41     main()
```

Fig. 13.3 main function illustrating stack operations
(stackOperations.py)

13.1.1 Evaluating Arithmetic Expressions

Next, we discuss the evaluation of arithmetic expressions. The usual form in which we write an arithmetic expression is called infix form because the operator lies in between the two operands. While writing expressions in the infix form, we use pairs of parentheses to ensure correct evaluation order. For example, examine the following expression:

infix expression: operator lies in between two operands

a + (b + c) * (k + (d + e) * (f + g * h))

Next, we show the evaluation of the above expression (Fig. 13.4). Note that in order to evaluate the expression correctly, we need to keep track of precedence of operators and matching parentheses. In this process, we have to scan parts of the expression several times. An

alternative form of expressions, called postfix form removes this difficulty.

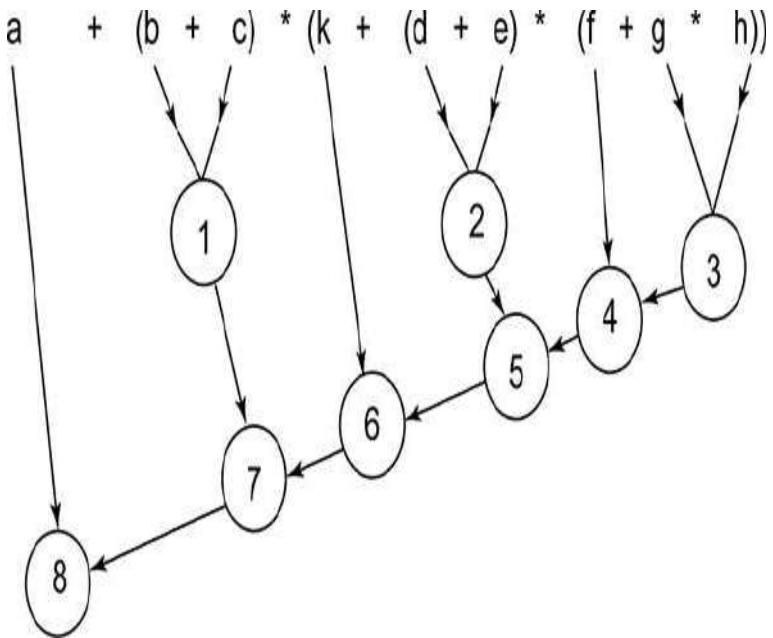


Fig. 13.4 Evaluation of expression $a + (b + c) * (k + (d + e) * (f + g * h))$

evaluation of infix expression

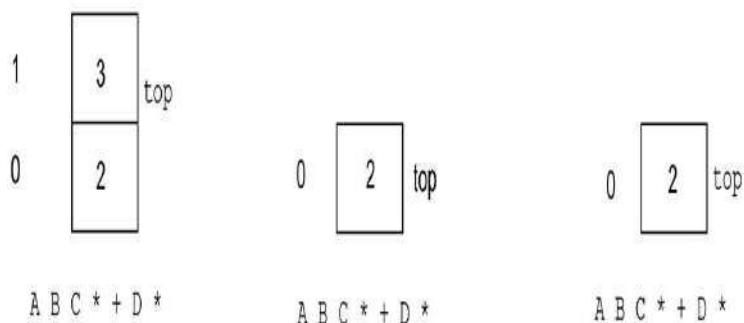
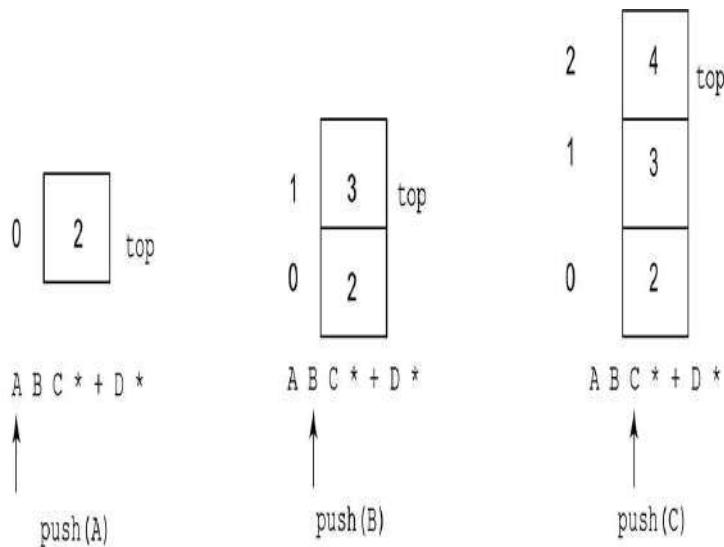
Postfix form: In the postfix form, the operator appears after the operands. For example, the infix expression $a + b$ would be written as $a\ b\ +$ in the postfix form. Now, as $a\ b\ +$ is the postfix expression for $a + b$, the postfix expression for $(a + b) + c$ would be $a\ b\ +\ c\ +$. Before we get into the details of converting expressions from infix to postfix form, we discuss the evaluation of postfix form expressions. To evaluate a postfix form expression, we make use of a stack and scan the postfix expression from left to right, applying the following rules:

postfix form : the operator appears after the operands

- On seeing an operand in the expression, push it onto the stack.
- On seeing an operator in the expression, pop the necessary number of operands from the stack, apply the operator on the popped operands, and push the result on the stack.

evaluating a postfix expression

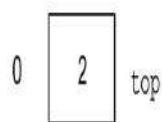
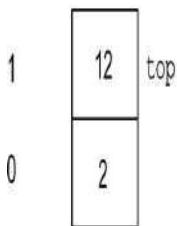
Next, we examine evaluation of a postfix expression $A \ B \ C \ * \ + \ D \ *$, where the variables A, B, C, and D take values 2, 3, 4, and 5, respectively (see Fig. 13.5). The arrow marks the position of the symbol being processed currently. It is evident from the computation shown in Fig. 13.5 that the postfix expression $A \ B \ C \ * \ + \ D \ *$ is equivalent to the infix expression $(A + B * C) * D$.



operand2 = pop()
 \Rightarrow operand2 $\leftarrow 4$

operand1 = pop()
 \Rightarrow operand1 $\leftarrow 3$

result = operand1 *
 operand2
 \Rightarrow result $\leftarrow 3 * 4$

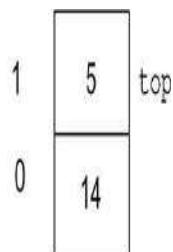
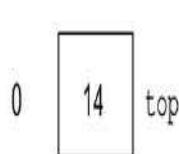


result = operand1 *
 operand2
 \Rightarrow result $\leftarrow 3 * 4$

$A \ B \ C \ * \ + \ D \ *$
 \uparrow
 push(result)

$A \ B \ C \ * \ + \ D \ *$
 \uparrow
 operand2 = pop()
 \Rightarrow operand2 $\leftarrow 12$

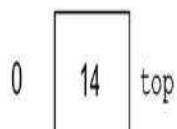
$A \ B \ C \ * \ + \ D \ *$
 \uparrow
 operand1 = pop()
 \Rightarrow operand1 $\leftarrow 2$



$A \ B \ C \ * \ + \ D \ *$
 \uparrow
 result = operand1 +
 operand2
 \Rightarrow result $\leftarrow 2 + 12$

$A \ B \ C \ * \ + \ D \ *$
 \uparrow
 push(result)

$A \ B \ C \ * \ + \ D \ *$
 \uparrow
 push(5)

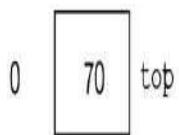


$A \ B \ C \ * \ + \ D \ *$

$A \ B \ C \ * \ + \ D \ *$

$A \ B \ C \ * \ + \ D \ *$

A B C * + D * \uparrow operand2 = pop() $\Rightarrow \text{operand2} \leftarrow 5$	A B C * + D * \uparrow operand1 = pop() $\Rightarrow \text{operand1} \leftarrow 14$	A B C * + D * \uparrow $\text{result = operand1 *}$ operand2 $\Rightarrow \text{result} \leftarrow 14 * 5$
---	--	--



A B C * + D * \uparrow push(result)	A B C * + D * \uparrow result = pop()
---	---

Fig. 13.5 Evaluation of postfix expression A B C * + D *

In light of the above discussion, we develop script `evaluatePostfix` (Fig. 13.6). In this program, we deal with only one digit operands. The function `main` in this program invokes the function `evaluatePostfixExp`, which takes a postfix expression as the input parameter and returns the result obtained on evaluating the input expression.

```
01 from stack import Stack
02 def evaluatePostfixExp(exp):
03     """
04     Objective: To evaluate postfix expression
05     Input Parameter: exp - string (postfix expression)
06     Return Value: result - integer
07
08     Limitation: operands are single digit integers,
09                 Only binary arithmetic operands are allowed
```

```
10     """
11     stk = Stack()
12     operations = ['+', '-', '*', '/']
13     for symbol in exp:
14         if symbol in operations:
15             operand2 = stk.pop()
16             operand1 = stk.pop()
17             result = str(eval(operand1 + symbol + operand2))
18             stk.push(result)
19         else:
20             stk.push(symbol)
21     result = int(stk.pop())
22     return result
23
24 def main():
25     """
26     Objective: To evaluate postfix expression entered by the user
27     Input Parameter: None
```

```

28     Return Value: None
29     """
30     """
31     Assumption: Input expression is correctly formed
32     """
33     exp = input('Enter a postfix expression-use single digit numbers
34     only: ')
35     result = evaluatePostfixExp(exp)
36     print(result)
37
38 if __name__=='__main__':
39     main()

```

Fig. 13.6 Script for evaluating postfix expression
(evaluatePostfix.py)

On executing the script `evaluatePostfix` (Fig 13.6), it prompts the user to enter a postfix expression and returns the result obtained on evaluating it. For example, on entering the expression $75-6*4+$, corresponding to the infix expression $(7-5)*6+4$, Python responds with the result 16.

13.1.2 Conversion of Infix Expression to Postfix Expression

In this section, we discuss how to convert an infix expression into a postfix expression. Once again a stack will come to our help. Recall that we evaluate the postfix expression from left to right using a stack. To begin with, examine the expression $A + B * C$. As multiplication has higher precedence than addition, the corresponding postfix expression would be $A B C * +$ because when we use a stack to evaluate it, it gets evaluated as $A + (B * C)$. Note that the expression $A B + C *$ will not work because it would get evaluated as $(A+B) * C$. The preceding example gives us a general clue for converting an infix expression into equivalent postfix expression. As we scan the infix expression, we keep on appending

operands to the output string as they are encountered. To deal with the operators, we use a stack. To decide how to process the current symbol, we apply the following rules:

rules for infix to postfix conversion

- If the character currently being scanned is an operator, and precedence (operator currently scanned) > precedence (operator on top of the stack), push it on the stack
- If the character currently being scanned is an operator, and precedence (operator currently scanned) <= precedence (operator on top of the stack), pop off and append to the output string, each operator having precedence higher than or equal to the precedence of the operator currently being scanned, until an operator having lower precedence is encountered on top of the stack, or the stack becomes empty. Now, push the operator being scanned currently on the stack.
- If the character being scanned is an operand, append it to the output string.
- If the end of the input string is reached, pop off all the operators one by one and append to the output string.

So far, we have not discussed how to deal with the parentheses. As parentheses have the highest precedence, as soon as '(' is encountered, it should be placed on the stack. '(' marks beginning of a sub-expression of the given infix expression that would be subsequently terminated by matching ')'. Now, we need to convert this sub-expression to postfix form. However, the precedence of '(' being highest, no operator can be pushed on it and '(' would be popped off the stack without processing any other operator. To prevent this situation, we modify the rules for push and pop operations as follows:

infix to postfix: expressions involving parentheses

- push: if operator currently scanned is '(', or the top of the stack is '(', or precedence (operator currently scanned) > precedence (operator on top of the stack)
- pop: if operator currently scanned == ')', then pop off all the operators up to and including the first '(' encountered in the stack. If the character currently being scanned is an operator other than a parenthesis, and precedence (operator currently scanned) <= precedence (operator on top of the stack), pop off and append to the output string, each operator having precedence higher than or equal to

the precedence of the operator currently being scanned, one by one, until ')' or an operator having lower precedence is encountered on top of the stack, or the stack becomes empty. Now, push the operator being scanned currently on the stack.

- If the character being scanned is an operand, concatenate it to the output string.
- If the end of the input string is reached, pop off all operators one by one. Concatenate the operators to output string as they are popped off.

Note that a right parenthesis would never be pushed on the stack. Since ')' terminates a sub-expression, on encountering ')', we pop off all operators up to the matching '(', that is, the first '(' encountered on the stack. As the role of '(' is now over, it can also be popped out of the stack. However, it is not appended to the output string as postfix form expressions do not use parentheses. We put together above ideas in the form of the function `postfix` (Fig. 13.7). Note that we have considered expressions involving +, -, *, /, (,) and single digit operands only.

On executing the script in Fig. 13.7, it outputs the postfix expression for a given infix expression:

```
Enter the expression in infix notation:  
( (7-5)*6+4) 75-6*4+
```

```
01 from stack import Stack
02
03 def postfix(exp):
04     """
05         Objective: To convert infix expression to postfix
06         Input Parameter: exp - string (infix expression)
07         Return Value: result - string (postfix expression)
08     """
09     # precedence: dictionary to store operator:precedence pairs
10
11     precedence = {'+":1, '-':1, '*':2, '/':2}
12
13     operand = '0123456789' #string of valid operands
14
15     result = ''
16     stk = Stack()
17     for symbol in exp:
18         if symbol == '(':
19             stk.push(symbol)
20         elif symbol == ')':
21             val = stk.pop()
22             while val != '(':
```

```
23             result += val
24             val = stk.pop()
```

```

25     elif symbol in operand:
26         result += symbol
27     elif symbol in precedence:
28         if not(stk.isEmpty()):
29             stkTop = stk.top()
30             while not(stk.isEmpty()) and stkTop != '(' \
31                 and precedence[stkTop] >= precedence[symbol]:
32                 result += stkTop
33                 stk.pop()
34             if not(stk.isEmpty()):
35                 stkTop = stk.top()
36                 stk.push(symbol)
37             while not(stk.isEmpty()):
38                 result += stk.pop()
39         return result
40
41 def main():
42     """
43     Objective: To convert infix expression provided by user to
44     postfix expression
45     Input Parameter: None
46     Return Value: None
47     """
48     """
49     Assumption: Input expression is correctly formed
50     """
51     exp = input('Enter the expression in infix notation: ')
52     exp = postfix(exp)
53     print(exp)
54
55 if __name__=='__main__':
56     main()

```

Fig. 13.7 Program to find postfix expression for the given infix expression (`postfix.py`)

13.2 QUEUES

A queue is a data structure that behaves in First In First Out (FIFO) manner, i.e. objects leave in the same order in which they arrive. All insertions occur at one end called `rear` end, and all deletions occur at another end called `front` end. Insertion and deletion operations are also known as *enqueue* and *dequeue* operations respectively.

`queue`: first in, first out

`enqueue`: insertion (at `rear` end)

`dequeue`: deletion (at `front` end)

Beginning with an empty queue, let us perform the following operations on this queue in sequence: *enqueue 5, enqueue 10, dequeue, enqueue 20, enqueue 25, dequeue, dequeue, dequeue, dequeue*. In Fig. 13.8, we show the queue on execution of each of the aforementioned operations:

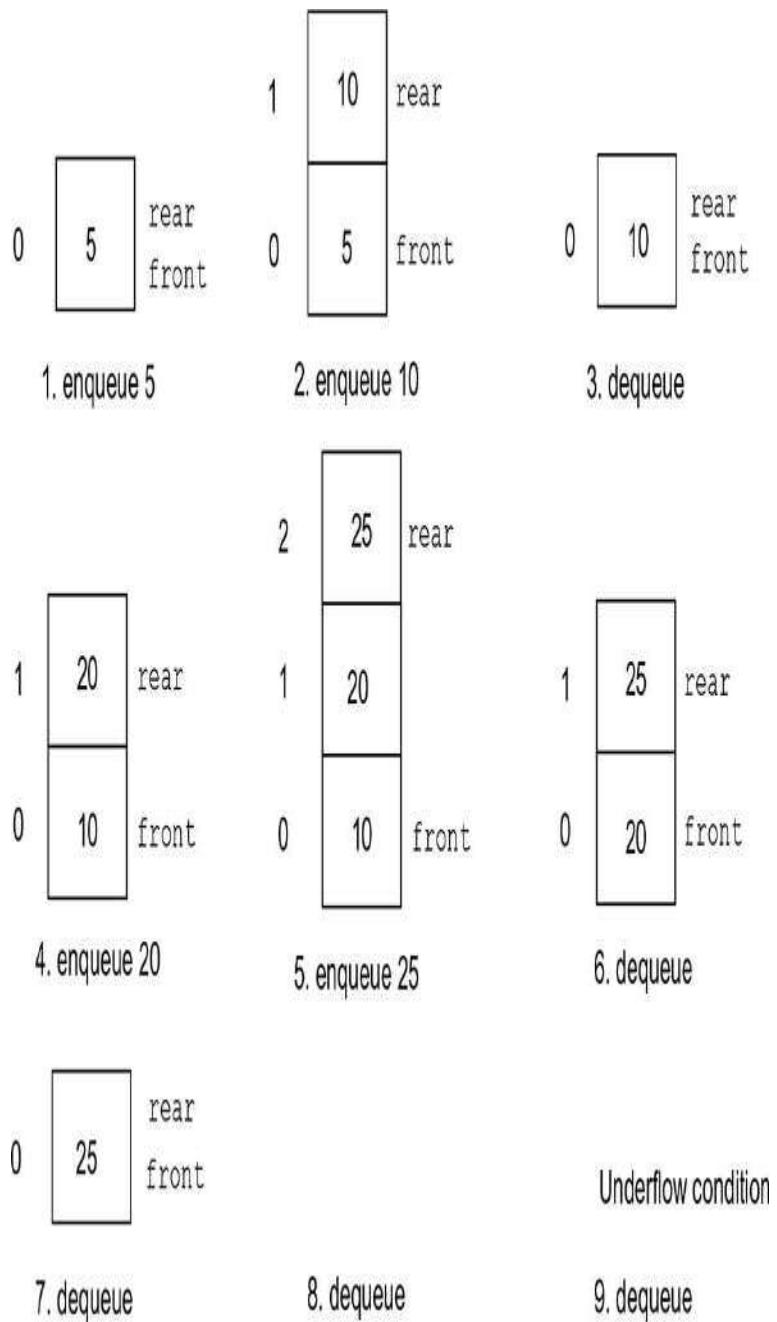


Fig. 13.8 Queue operations

Underflow denotes a condition when an attempt is made to delete from an empty queue. As mentioned earlier, whenever queue is empty, the number of elements in it is zero.

underflow: dequeue operation on an empty queue

A queue may be implemented in Python using lists. In the script `queue` (Fig. 13.9), we define the class `Queue`. We describe the *enqueue* and *dequeue* operations using inbuilt list operations `append` and `pop(0)`.

use of list operations `append` and `pop(0)` for implementing queue operations

```
01 class Queue:  
02  
03     def __init__(self):  
04         '''  
05             Objective: To initialize data members of object of  
06             type Queue  
07             Input Parameter:  
08                 self (implicit parameter)- object of type Queue  
09             Return Value: None  
10         '''  
11         self.values = list()  
12
```

```
13     def enqueue(self, element):  
14         '''  
15             Objective: To insert an element at the rear end  
16             Input Parameters:  
17                 self (implicit parameter)- object of type Queue  
18                 element - value to be inserted
```

```
19     Return Value: None
20     ''
21     self.values.append(element)
22
23     def dequeue(self):
24         '''
25             Objective: To remove an element from the front of queue
26             Input Parameter:
27                 self (implicit parameter)- object of type Queue
28             Return Value: Front element of the queue, if queue is
29                 not empty, else None
30         '''
31         if not(self.isEmpty()):
32             return self.values.pop(0)
33         else:
34             print('Queue Underflow')
35             return None
36
37     def isEmpty(self):
38         '''
39             Objective: To determine whether the queue is empty
40             Input Parameter:
41                 self (implicit parameter)- object of type Queue
42             Return Value: True if the queue is empty else False
43         '''
44         return len(self.values) == 0
45
46     def front(self):
47         '''
48             Objective: To return element at the front of queue
49             Input Parameter:
50                 self (implicit parameter)- object of type Queue
51             Return Value: Front element of the queue, if queue is
52                 not empty, else None
53         '''
54         if not(self.isEmpty()):
55             return self.values[0]
56         else:
57             print('Queue Empty')
```

```
58     return None
59
60
```

```

60     def size(self):
61         """
62             Objective: To return no. of elements in the queue
63             Input Parameter:
64                 self (implicit parameter)- object of type Queue
65             Return Value: number of elements in queue - numeric
66         """
67         return len(self.values)
68
69
70     def __str__(self):
71         """
72             Objective: To return string representation of object of
73             type Queue
74             Input Parameter: self (implicit parameter)- object of type
75             Queue
76             Return Value: string
77         """
78         stringRepr = ''
79         for i in self.values:
80             stringRepr += str(i) + '\t'
81         return stringRepr

```

Fig. 13.9 Class Queue (queue.py)

The constructor of class Queue introduces the list values for storing the queue objects. Initially, the list values is an empty list. We define the following Queue operations:

1. `isEmpty`: The method `isEmpty` returns `True` if the length of the list `values` is zero, and `False` otherwise.
2. `enqueue`: It takes the element to be inserted in the queue as an input argument and adds the element at the rear end of the list `values` by invoking the method `append`.
3. `dequeue`: The method `dequeue` first checks whether the queue is empty by invoking the method `isEmpty`. If the queue is empty, the message '`Queue Underflow`' is displayed and `None` is returned indicating underflow. If the queue is not empty, the value from the front end of the list `values` is deleted using the method `pop` and returned.
4. `front`: If the queue is not empty, the method `front` returns the value at the front end (index zero in the list `values`), without deleting it from the queue.

Otherwise, it returns None indicating empty queue.

```
01  from myQueue import Queue
02  def main():
03      """
04          Objective: To provide queue functionality
05          Input Parameter: None
06          Return Value: None
07      """
08
09      while 1:
10          print('Choose an option \n')
11          print('1: Create queue')
12          print('2: Delete queue')
13          print('3: Enqueue')
14          print('4: Dequeue')
15          print('5: Print Queue Data')
16          print('6: Front element')
17          print('7: No. of elements')
18          choice = int(input('Enter Choice: '))
19
20          if choice == 1:
21              q = Queue()
22              print('Queue Created')
23          elif choice == 2:
24              del q
25              print('Queue Deleted')
26          elif choice == 3:
27              element = int(input('Enter element to be inserted: '))
28              q.enqueue(element)
29          elif choice == 4:
30              print('Element deleted:', q.dequeue())
31          elif choice == 5:
32              print(q)
33          elif choice == 6:
34              print('Front element', q.front())
35          elif choice == 7:
36              print('No. of elements', q.size())
37              proceed = input('enter y if you wish to continue: ')
38              if proceed != 'y' and proceed != 'Y':
39                  break
40
41  if __name__=='__main__':
42      main()
```

Fig. 13.10 main function illustrating queue operations
(queueOperations.py)

5. `size`: The method `size` returns the number of elements in the queue by returning the length of the list values. For the purpose of demonstration, we have provided this method to return the number of elements in the queue, even though queue operations do not require this method.
6. `__str__`: The method `__str__` produces a string representation of the entire queue by concatenating the string representation of each value in the list values. For the purpose of demonstration, we have provided this method to display the contents of the entire queue, even though queue operations do not require this method.

SUMMARY

1. The stack is a data structure in which objects are stored one over the other, and these objects leave in the reverse order of their arrival. Such an arrangement of objects is called *Last In First Out* (LIFO) arrangement.
2. A stack is characterized by the following operations:
 1. push: Place (push) a new element on top of the stack. The push operation is also called insertion of an element.
 2. pop: Remove (pop) top element from the stack. The pop operation is also called deletion of an element.
3. Popping an element from the stack when it is empty results in an underflow condition.
4. Python's built-in list methods `append` and `pop` are typically used for implementing a stack.
5. A queue is a data structure that behaves in First In First Out (FIFO) manner i.e. objects leave in the same order in which they arrive. All insertions take place at one end called `rear` end, and all deletions occur at another end called `front` end. Insertion and deletion operations are also known as `enqueue` and `dequeue` operations respectively.
6. When a queue is empty, `dequeue` operation on it results in an underflow condition.
7. Python provides inbuilt list operations `append` and `pop` for implementing a queue.

EXERCISES

1. Imagine that Python list does not support methods `append` and `pop`. Examine the script in Fig. 13.11 and define your implementation of push and pop operation of the stack by filling up the code.

```
class Stack:  
  
    def __init__(self, size):  
        ...  
        Objective: To initialize data members of object of type Stack  
        Input Parameters:  
            self (implicit parameter) - object of type Stack  
            size - numeric (Number of elements in stack)  
        Return Value: None  
        ...
```



```

        self.stackSize = size
        self.values = [None for i in range(0, self.stackSize)]
        self.top = -1

    def isFull(self):
        """
        Objective: To determine if the stack is full
        Input Parameter:
            self (implicit parameter) - object of type Stack
        Return Value: True if the stack is full else False
        ...
        return (self.top + 1) == self.stackSize

    def isEmpty(self):
        """
        Objective: To determine if the stack is empty
        Input Parameter:
            self (implicit parameter) - object of type Stack
        Return Value: True if the stack is empty else False
        ...
        return self.top == -1

    def push(self, element):
        """
        Objective: To put an element on top of the stack
        Input Parameters:
            self (implicit parameter) - object of type Stack
            element - value to be inserted
        Return Value: None
        ...
        ...

        Approach:
        Check to see if stack is full or not
        If stack is not full, value of top is incremented and
        the element is inserted at the top
        Else
        Stack overflow message is printed.
        ...
        ##### Fill in the code #####
        """

    def pop(self):
        """
        Objective: To remove an element from the top of stack
        Input Parameter:

```

```

            self (implicit parameter) - object of type Stack
        Return Value: top element of the stack if stack is
                    not empty else None.
        ...
        ...

        Approach:
        Check to see if stack is empty or not.
        If stack is not empty, value at top is returned,
        moreover, the value of top is also decremented to reflect
        new top.
        Else
        Stack underflow message is printed and value None is returned.
        ...

```

```
##### Fill in the code #####
def topElement(self):
    ...
    Objective: To return top element of the stack
    Input Parameter:
        self (implicit parameter) - object of type Stack]
    Return Value: top element of the stack if stack is
        not empty else None
    ...
    ...
    Approach:
        Check to see if stack is empty.
        If stack is not empty, value at top is returned,
        Else
        Value None is returned.
    ...
##### Fill in the code #####
def size(self):
    ...
    Objective: To return no. of elements in the stack
    Input Parameter:
        self (implicit parameter) - object of type
    Return Value: number of elements in stack - numeric
    ...
    return self.top + 1
```

Fig. 13.11 Class Stack(stackDefined.py)

2. Imagine that Python list does not support methods append and pop. Examine the script in Fig. 13.12 and define your own implementation of enqueue and dequeue operation on a queue by filling up the missing

code:

```
class Queue:

    def __init__(self, size):
        """
        Objective: To initialize data members of object of type Queue
        Input Parameter:
            self (implicit parameter) - object of type Queue
        Return Value: None
        """

        self.qSize = size
        self.values = [None for i in range(0, self.qSize)]
        self.front, self.rear = -1, -1

    def enqueue(self, element):
        """
        Objective: To insert an element at the rear end
        Input Parameters:
            self (implicit parameter) - object of type Queue
            element - value to be inserted
        Return Value: None
        """

        Approach:
        Check to see if queue is full or not.
        If the queue is not full, the value of rear is incremented, and
        the element is inserted at the rear. Also, if the front is -1,
        assign index 0 to front.
        Else
        Queue overflow message is printed.
        """

        ##### Fill in the code #####
        #print("Enqueue operation performed")

    def dequeue(self):
        """
        Objective: To remove an element from the front of queue
        Input Parameter:
            self (implicit parameter) - object of type Queue
        Return Value: Front element of the queue if queue is not empty
                      else None
        """

        Approach:
```



```

Check to see if queue is empty or not.
If queue is not empty,
    If front is equal to rear, value at front is returned,
    moreover, front and rear are assigned the value -1.
    Else
        Value at front is returned as well as its value is
        incremented by 1.
    Else
        Queue underflow message is printed and value None is returned.
    ...
    ##### Fill in the code #####
def isFull(self):
    ...
    Objective: To determine whether the queue is full
    Input Parameter:
        self (implicit parameter) - object of type Queue
    Return Value: True if the queue is full else False
    ...
    return self.rear + 1 == self.qSize

def isEmpty(self):
    ...
    Objective: To determine whether the queue is empty
    Input Parameter:
        self (implicit parameter) - object of type Queue
    Return Value: True if the queue is empty else False
    ...
    return self.rear == -1

def frontValue(self):
    ...
    Objective: To return element at the front of queue
    Input Parameter:
        self (implicit parameter) - object of type Queue
    Return Value: Front element of the queue if queue is not empty
        else None
    ...
    ...
    Approach:
    Check to see if queue is empty or not.
    If queue is not empty, value at front is returned,
    Else
        Value None is returned.

```

```

    ...
    if not(self.isEmpty()):
        return self.values[self.front]
    else:
        print('Queue Empty')
        return None

def size(self):
    ...
    Objective: To return no. of elements in the queue
    Input Parameter:
        self (implicit parameter) - object of type Queue

```

```

Return Value: number of elements in queue - numeric
...
...
Approach:
Check to see if queue is empty or not.
If queue is not empty, return rear minus front plus 1 as size
Else
Return 0
...
if not(self.isEmpty()):
    return self.rear - self.front + 1
else:
    return 0

```

Fig. 13.12 Class Queue (queueDefined.py)

3. The queue implementation defined in question 2 does not utilize the storage space effectively since if `front` points to index `j` and `rear` points to index `i >= j`, empty space at indexes 0 to `i-1` cannot be utilized as shown in Fig. 13.13.

After deletion of elements 10 and 20

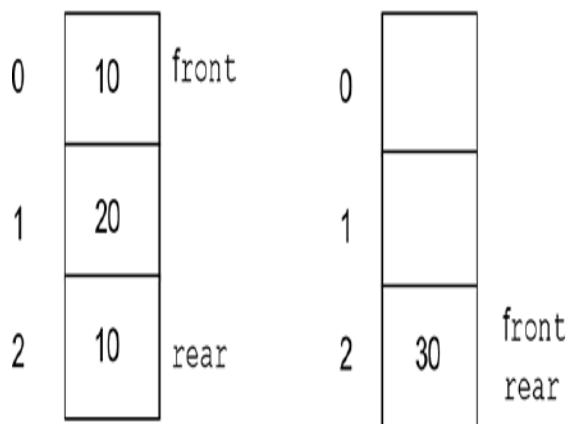
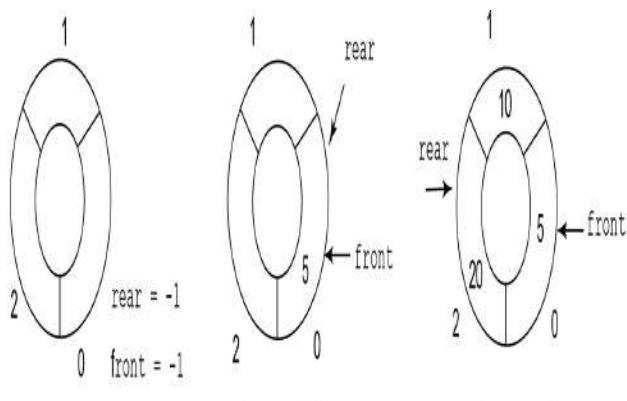


Fig. 13.13 Queue



1. Empty Queue

2. enqueue 5

3. enqueue 10

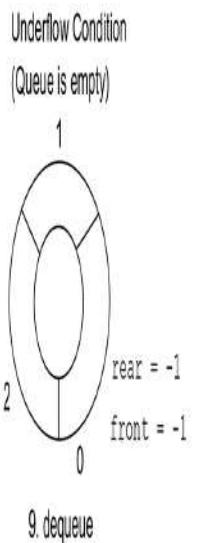
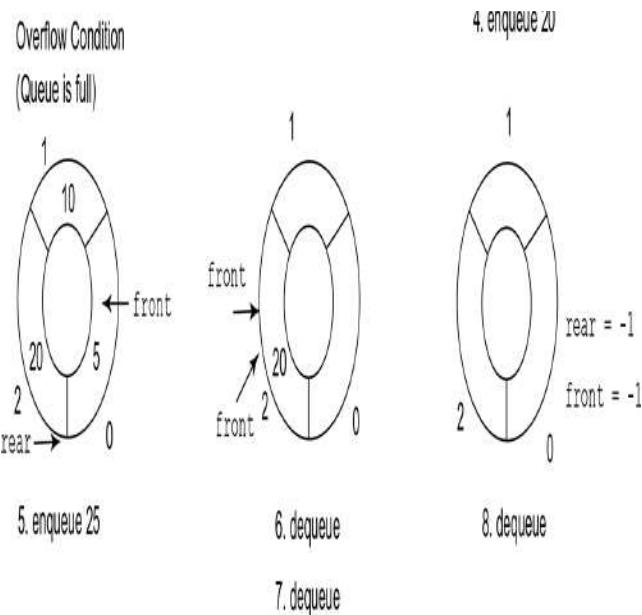


Fig. 13.14 Queue

The problem can be eliminated by maintaining circular queue (Fig. 13.14). Examine the skeleton of the script circular queue (Fig. 13.15) and fill up the left out code. The variable count keeps track of the current size of the queue and also enables us to discover whether the queue is empty or full. As indexes are used in a circular manner, whenever front or rear is incremented, we apply the modulo qSize operation, we update front and rear as $(front + 1) \% qSize$, and $(rear + 1) \% qSize$ respectively. The queue is said to be full if $(rear + 1) \% qSize == front$. Note that if we insert another element at this stage, front and rear would become equal.

4. Develop a program using a stack to find out whether the given string is a palindrome.

5. Rewrite the code in exercise 2, so that whenever there is a deletion, all the elements in the queue are shifted towards the front by one position.

```
class CircularQueue:

    def __init__(self, size):
        """
            Objective: To initialize data members of object of type CircularQueue
            Input Parameter:
                self (implicit parameter)- object of type CircularQueue
            Return Value: None
        """
        self.qSize = size
        self.count = 0
        self.values = [None for i in range(0, self.qSize)]
        self.front, self.rear = -1, -1

    def enqueue(self, element):
        """
            Objective: To insert an element in queue at the rear
            Input Parameters:
                self (implicit parameter)- object of type CircularQueue
                element - value to be inserted
            Return Value: None
        """
        """
            Approach:
            Check to see if queue is full.
            If queue is not full, value of rear is incremented by 1 modulus
            queue size and element is inserted at the rear. Also,
            the value of front is incremented by 1 modulus queue size
            if it is equal to -1. Further, count is incremented by 1.
            Else
                Queue overflow message is printed.
        """
        ###### Fill in the code ######

    def dequeue(self):
        """
            Objective: To remove an element from the front of queue
            Input Parameter:
                self (implicit parameter)- object of type CircularQueue
            Return Value: Front element of the queue if queue is not empty
                         else None
        """
        """
            Approach:
        """
```



```

Check to see if queue is empty or not.
If queue is not empty,
    Value at front is returned as well as its value is
    incremented by 1 modulus queue size.
    Also, the count is decremented by 1.
    If queue becomes empty,
        Set front and rear to -1.
Else
Queue underflow message is printed and value None is returned.
...
##### Fill in the code #####
def isFull(self):
...
Objective: To determine if the queue is full
Input Parameter:
    self (implicit parameter)- object of type CircularQueue
Return Value: True if the queue is full else False
...
return self.count == self.qSize

def isEmpty(self):
...
Objective: To determine if the queue is empty.
Input Parameter:
    self (implicit parameter)- the object of type CircularQueue
Return Value: True if the queue is empty else False.
...
return self.count == 0

def frontValue(self):
...
Objective: To return element at the front of queue.
Input Parameter:
    self (implicit parameter)- the object of type CircularQueue
Return Value: Front element of the queue if queue is not empty
    else None.
...
...
Approach:
Check to see if queue is empty.
If queue is not empty, value at front is returned,
Else

```

```

Value None is returned.
...
if not(self.isEmpty()):
    return self.values[self.front]
else:
    print('Queue Empty')
    return None

def size(self):
...
Objective: To return no. of elements in the queue
Input Parameter:
    self (implicit parameter)- object of type CircularQueue

```

```
Return Value: number of elements in queue - numeric
...
...
Approach:
Return count
...
return self.count
```

Fig. 13.15 Class CircularQueue (circQueue.py)

6. Rewrite the code in exercise 2 so that whenever there is a queue overflow scenario, all the elements in the queue are shifted (if possible) so that the front element is at index 0.
7. Write a program to reverse a string using stacks.
8. Evaluate the following postfix expression. Show the status of the stack on the execution of each operation.
 1. $99/52*9-+$
 2. $25*94/5-+$
 3. $526*93/-*$
 4. $51+31-*$
 5. $687+*82/-$
 6. $152-*25*+$
 7. $55+93-*3/$
 8. $651+21-+*$
 9. $836+27-9+*/$
9. Convert the following infix expression to its equivalent postfix expression, showing the stack contents at each step.
 1. $8+6/(5-3)*2$
 2. $(8-3-1)*6/2+8/4$
 3. $8+(9-9/3-4*7)-5$
 4. $7+6-7/2-7*4-3$

CHAPTER 14

DATA STRUCTURES II: LINKED LIST

CHAPTER OUTLINE

[14.1 Introduction](#)

[14.2 Insertion and Deletion at the Beginning of a Linked List](#)

[14.3 Deleting a Node with a Particular Value from a Linked List](#)

[14.4 Traversing a Linked List](#)

[14.5 Maintaining Sorted Linked List while Inserting](#)

[14.6 Stack Implementation Using Linked List](#)

[14.7 Queue Implementation Using Linked List](#)

In the previous chapter, we studied stacks and queues. Whereas a stack allows insertion and deletion of objects at one end, in a queue new objects are inserted at one end, and the deletion takes place from the other end. In this chapter, we will discuss a more flexible data structure called linked list that allows insertion and deletion of an object at an arbitrary position.

14.1 INTRODUCTION

A linked list is a sequence of objects called nodes. The relative position of the objects in a linked list is maintained using `next` link. Thus, a linked list comprises a number of objects, each of which contains a link to the next object (node). For example, let us examine the list `lst` shown in Fig. 14.1. The first node of the linked list comprises an `int` object 20 and a `next` link, which is a reference to the second node in the linked list. Similarly, the second node comprises an `int` object 15 and the `next` link, which is a reference to the third node. Since the third node is the last node of the linked list, there is no `next` object ahead of it. Therefore, it comprises an `int` object 25 and the `next` link refers to `None`.

linked list: sequence of objects called nodes, each of which contains a link to the next object

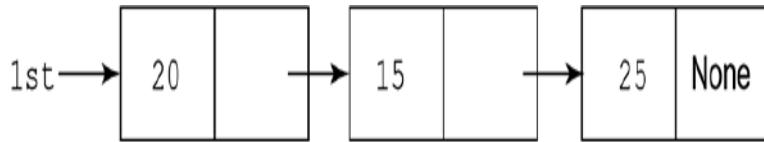


Fig. 14.1 Linked list visualization

Next, we describe a class `Node` for dealing with nodes of a linked list in a program. An instance of this class comprises two members, namely, `data` and `next`. We define the method `__init__` (Fig. 14.3) for the class `Node` that creates an instance of a node of the linked list. In this method, `self` refers to the `Node` object being created, `self.data` refers to the object `value`, being passed as an argument (an instance of class `int` in Fig. 14.1), and `self.next` refers to an instance of the class `Node`. Note that when we create a node object, by default, `next` is assigned the value `None`. Now, we show a more accurate representation of the linked list in Fig. 14.1 (Fig. 14.2). However, for reasons of brevity and simplicity of the figures, we will continue to use the representation of the linked list shown in Fig. 14.1.

```
class Node: for creating nodes of a linked list
```

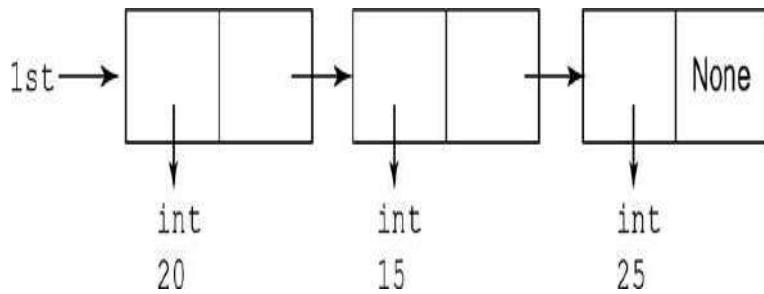


Fig. 14.2 Visualization of a linked list

```

01 class Node:
02     def __init__(self, value):
03         """
04             Objective: To initialize an object of class Node
05             Input Parameter:
06                 self (implicit parameter) - object of type Node
07             Return Value: None
08         """
09         self.data = value
10         self.next = None

```

Fig. 14.3 Class Node (node.py)

Now let us examine the script `linkedList1` (Fig. 14.4) that creates the linked list shown in Fig. 14.1. In Fig. 14.5, we show the run-time stack as the execution of the code proceeds in Python Tutor. Since Python Tutor does not allow us to import user defined modules, in order to execute the script in Python Tutor, import statement should be replaced by the complete definition of the class.

Execution of line 8 creates a node instance (Fig. 14.5(a)), comprising `data` (int object 20) and `next` link (object `None`). This node instance is named as `1st`. In this context, we use the following phrases

interchangeably: `lst` refers to the node, `lst` is a reference to the node, and `lst` points to the node.

Execution of line 9 creates another node instance (Fig. 14.5(b)), comprising `data` (int object 15) and `next` link (object `None`). Now, the name `lst.next`, which referred to `None` on execution of line 8, refers to the new node just created. Finally, execution of line 10 creates one more node instance (Fig. 14.5(c)), comprising `data` (int object 25) and `next` link (object `None`). Now, `lst.next.next`, which referred to `None` on execution of line 9, refers to the new node just created. Before proceeding further, it is important to note the following:

when a name refers to a node, we say : it is a reference to the node or it points to the node

```
01 from node import Node
02 def main():
03     """
04     Objective: To create a linked list comprising of three nodes
05     Input Parameter: None
06     Return Value: None
07     """
08     lst = Node(20)
09     lst.next = Node(15)
10     lst.next.next = Node(25)
11     print('lst: ', lst)
12     print('lst.data: ', lst.data)
13     print('lst.next: ', lst.next)
14     print('lst.next.data: ', lst.next.data)
15     print('lst.next.next: ', lst.next.next)
16     print('lst.next.next.data: ', lst.next.next.data)
17     print('lst.next.next.next: ', lst.next.next.next)
18
19 if __name__ == '__main__':
20     main()
```

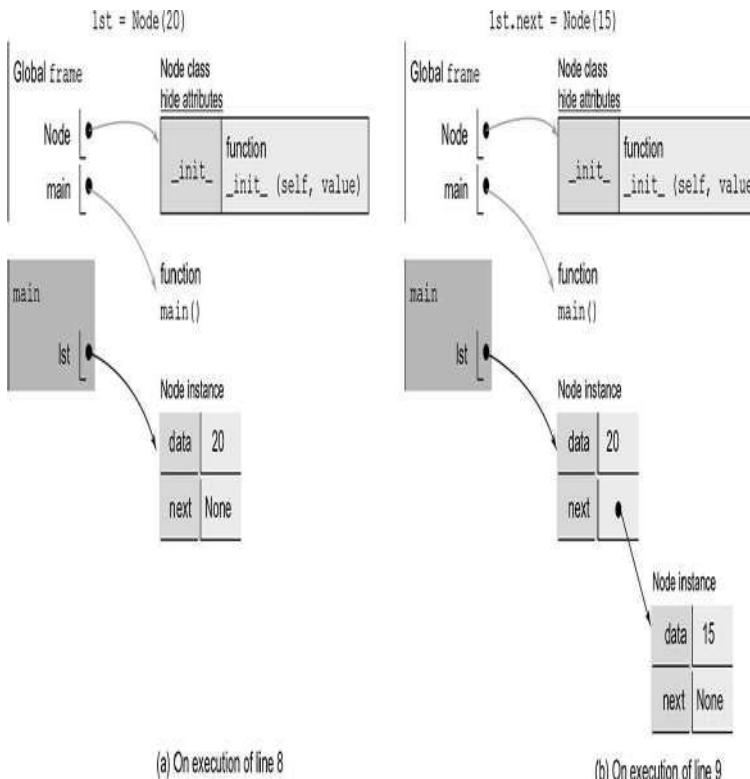
Fig. 14.4 Linked list creation (linkedList1.py)

`lst` refers to the first node of the linked list. We may also say that `lst` refers to the linked list. `lst.data` refers to the data attribute of the first node (i.e., int object 20) and `lst.next` refers to the `next` attribute of the first node (i.e., second node).

As `lst.next` refers to the second node, `lst.next.data` refers to the data attribute of the second node (i.e., int object

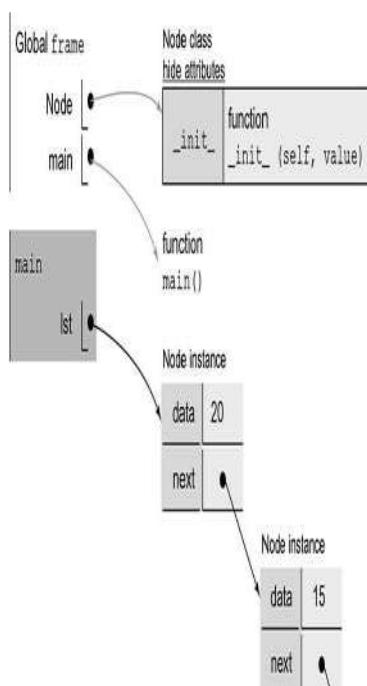
15) and `lst.next.next` refers to the `next` attribute of the second node (i.e. third node).

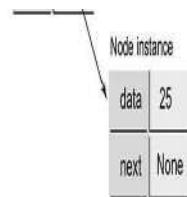
As `lst.next.next` refers to the third node,
`lst.next.next.data` refers to the `data` attribute of the third node (i.e., int object 25) and `lst.next.next` refers to the `next` attribute of the third node (i.e., object `None`).



(a) On execution of line 8

(b) On execution of line 9





(c) On execution of line 10

Fig. 14.5 Creation of a linked list `lst`

In lines 11 to 17 we print data values and node instances. On executing these lines, Python produced the following output:

```
lst: <node.Node object at 0x0305BD90>
lst.data: 20
lst.next: <node.Node object at 0x0305BDB0>
lst.next.data: 15
lst.next.next: <node.Node object at
0x0305BDD0>
lst.next.next.data: 25
lst.next.next.next: None
```

output on executing the script `linkedList1`

You must have noted that the above method of accessing nodes of a linked list as a sequence of `next` links (e.g., `lst.next.next.next`) beginning the first node is not just cumbersome, but also unworkable for linked lists having several nodes. As an alternative, we typically introduce, a variable `current` that keeps track of the node with which we are currently dealing. We insert a new node in the linked list by assigning the new node to `current.next`. In the script `linkedCubes` (Fig. 14.6), we use this method to create a linked list of cubes of four numbers beginning 1.

creating a linked list of cubes: keep track of the current node

```
01 from node import Node
02 def main():
03     """
04     Objective: To create a linked list comprising of four nodes
05     Input Parameter: None
06     Return Value: None
07     """
08     start, finish = 2, 5
09     lst = Node(1)
10     current = lst
11     for i in range(start,finish):
12         current.next = Node(i**3)
13         current = current.next
14
15 if __name__ == '__main__':
16     main()
```

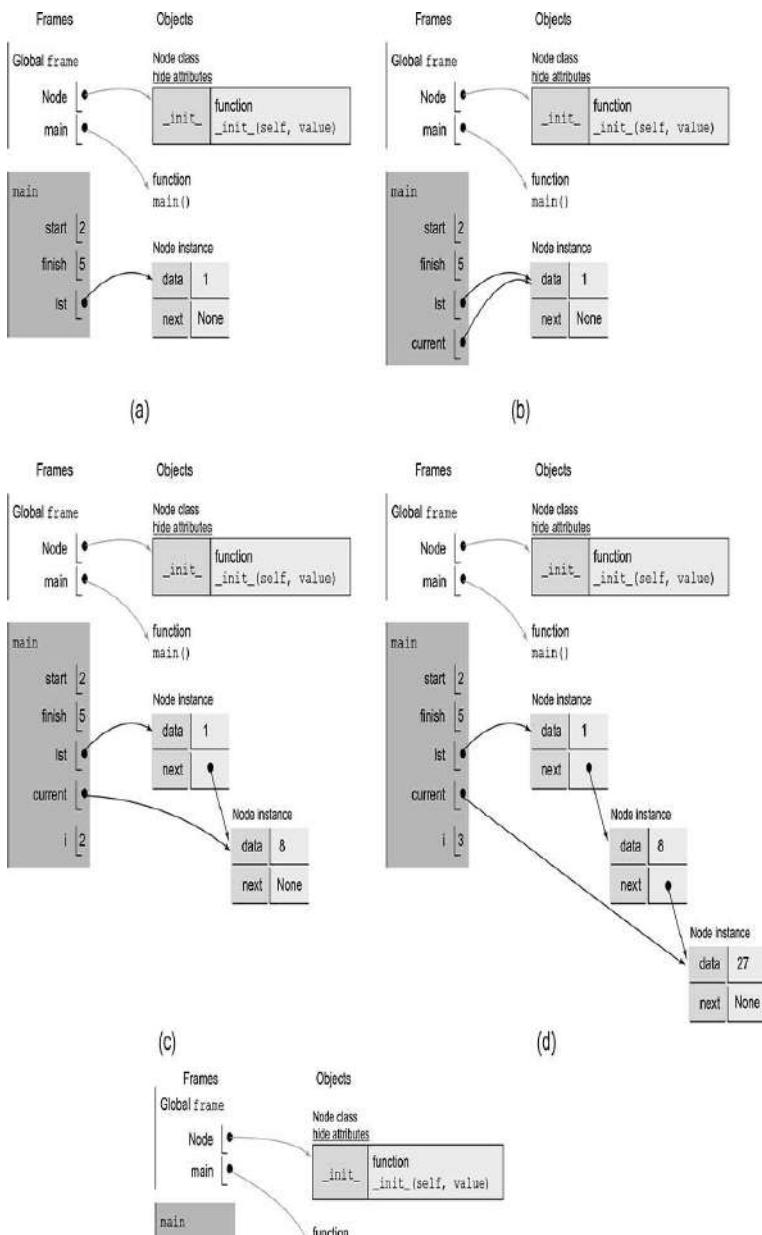
Fig. 14.6 Linked list of cubes (`linkedCubes.py`)

Execution of line 9 creates the node `lst` having the data attribute `lst.data` equal to 1 (Fig. 14.7(a)). On execution of line 10, both `current` and `lst` refer to this node (Fig. 14.7(b)). Every time line 12 is executed, a new node is created with data attribute `i**3` and assigned to `current.next`. Thus the new node gets inserted in the linked list. Subsequently, in line 13, `current` is assigned

`current.next` to make the node just inserted the current node for the next iteration. Thus, a new node is always inserted after the node that was inserted last in the linked list. The linked list on insertion of each of the nodes having data values 8, 27, and 64, respectively, is shown in Fig. 14.7(c), 14.7(d), and 14.7(e), respectively.

14.2 INSERTION AND DELETION AT THE BEGINNING OF A LINKED LIST

In this section, we will discuss linked lists in some more detail. We develop a class `LinkedList` (Fig. 14.8) that provides the following functionality:



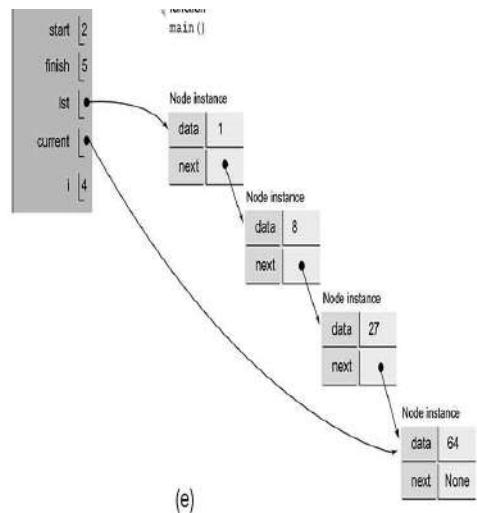


Fig. 14.7 Linked list 1st visualization

- create an empty linked list,
 - insert a node at the beginning of linked list
 - delete a node from the beginning of linked list.

This class makes use of the `Node` class discussed earlier. We introduce a variable `head` to refer to the first node of the linked list. As the linked list is initially empty, `head` is initialized to be `None` in the constructor method (line 10).

variable head refers to first node of the linked list

```
01 from node import Node
02 class LinkedList:
03     def __init__(self):
04         """
05             Objective: To initialize object of class LinkedList
06             Input Parameter:
07                 self (implicit parameter) - object of type LinkedList
08             Return Value: None
09
10             """
11             self.head = None
12
13     def insertBegin(self, value):
```

```
12     def insertBegin(self, value):
13         """
14             Objective: To insert a node at the beginning of LinkedList
15             Input Parameters:
16                 self (implicit parameter) - object of type LinkedList
17                 value - data for the node to be inserted
18             Return Value: None
19             """
20
21             if self.head is None:
22                 self.head = Node(value)
23             else:
24                 temp = Node(value)
25                 temp.next = self.head
26                 self.head = temp
27                 print('Value Inserted!!!')
28
29     def delBegin(self):
30         """
31             Objective: To delete a node from the beginning of LinkedList
32             Input Parameter:
33                 self (implicit parameter) - object of type LinkedList
34             Return Value: value in the deleted node
35             """
36
37             if self.head is None:    #Empty List
38                 print('Empty List')
```



```
37         return None
38     else:
39         temp = self.head
40         value = self.head.data
41         self.head = self.head.next
42         del temp
43         return value
44
```

Fig. 14.8 Class LinkedList (linkedList.py)

```
01 from linkedList import LinkedList
02 def main():
03     """
04     Objective: To carry out linked list operations based on user
05     inputs
06     Input Parameter: None
07     Return Value: None
08     """
09     lst = LinkedList()
10     while 1:
11         choice = int(input('1:Insert in Beginning \n2:Delete \
12         from Beginning \n3:Quit\n'))
13         if choice == 1:
14             value = eval(input('Enter value to be inserted: '))
15             lst.insertBegin(value)
16             print("Value inserted")
```

```

15     elif choice == 2:
16         value = lst.delBegin()
17         print('Value Deleted!!', value)
18     elif choice == 3:
19         break
20
21 if __name__ == '__main__':
22     main()

```

Fig. 14.9 Creating a linked list (`linkedListMain.py`)

In the script `linkedListMain` (Fig. 14.9), we demonstrate the use of class `LinkedList`. The execution of line 8 invokes the constructor that creates a `LinkedList` instance named `lst`. This step assigns the value `None` to the attribute `head` of this linked list. The linked list `lst` at this stage is shown in Fig. 14.10(a) as visualized in Python Tutor. Since, the Python Tutor does not support `eval` (Fig. 14.9, line 13), we have replaced it with the function `int` for executing the code in Python Tutor. However, in order to be able to use the program for data values of arbitrary type, we do not incorporate this change in Fig. 14.9. Next, the control enters the `while` loop. The user is asked to enter his/her choice of action out of the following:

1. insert a node at the beginning of the linked list
2. delete a node from the beginning of the linked list
3. quit the linked list operations

linked list operations

When the option 1 is chosen, the user is asked to enter the value for the node to be inserted in the linked list. We demonstrate the linked list operations for the following sequence of choices:

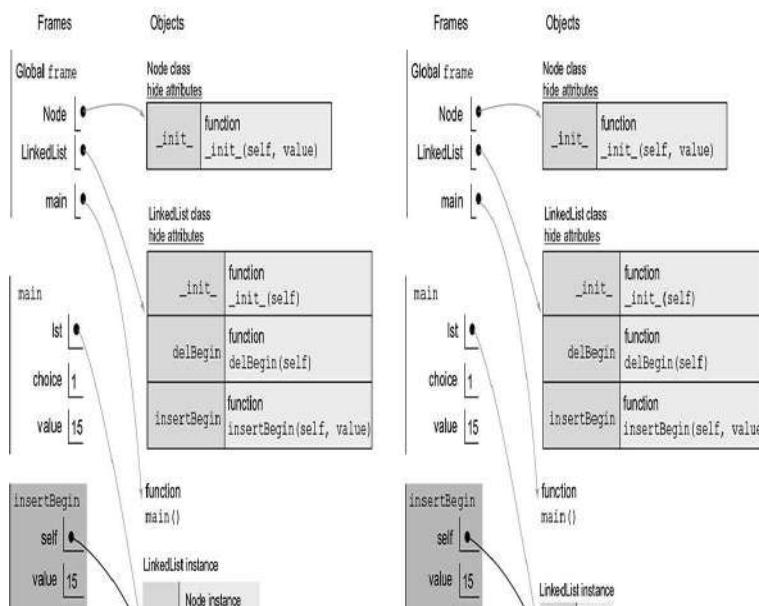
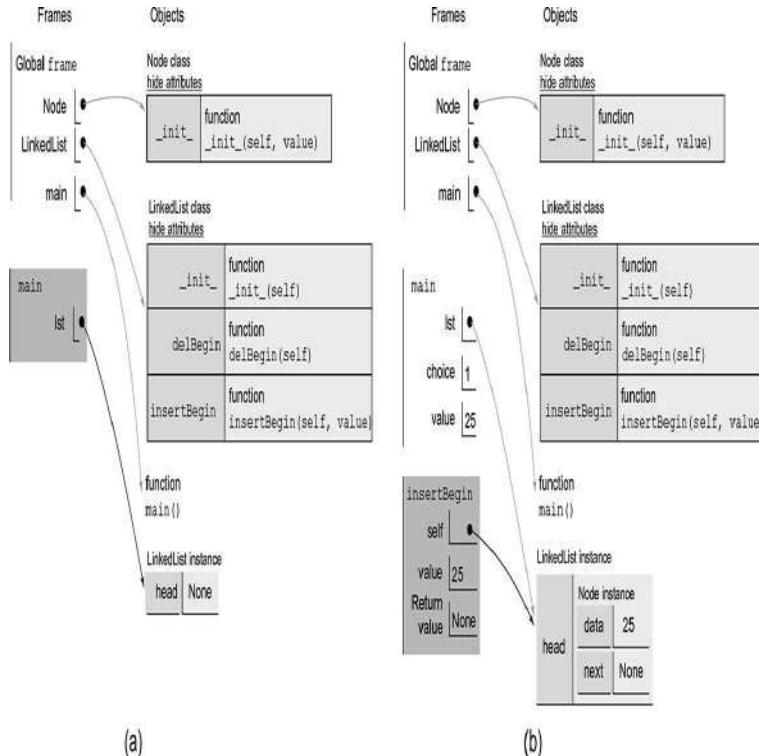
Choice	value
1	25
1	15
1	20
2	
3	

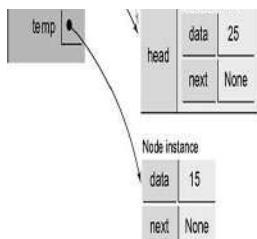
On entering choice 1 and value 25, the method `insertBegin` is invoked with the argument 25. It inserts a node having value 25 at the beginning of the linked list. For this purpose, it first checks whether linked list is empty, i.e., `head` contains `None` as the associated value. As the linked list is, indeed, currently empty, it invokes the constructor of the `Node` class with the argument 25 (Fig. 14.8, line 21) to create a node having 25 as the value of `data` and `None` as the value of the `next` link. This node instance is referred to via `head` of the linked list (Fig. 14.10(b)).

inserting a node at the beginning

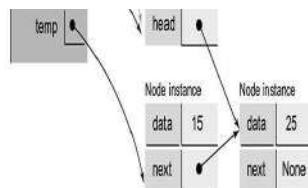
Next, we wish to insert another node having `data` value 15 in the linked list. However, the linked list is non-empty now. Again, we invoke the method `insertBegin` with the argument 15. Execution of line 23 (Fig. 14.8) creates a node instance named `temp` having 15 as the value of `data` and `None` as the value of the `next` link (Fig. 14.10(c)). On execution of line 24, the node instance `temp` gets added to the linked list by assigning the value of `head` of the linked list `lst` to the `next` link of `temp` (Fig. 14.10(d)). Note, however, that we still need to update `head` to point to the beginning of the modified list. Execution of line 25 (Fig. 14.8) achieves this (Fig. 14.10(e)). In Fig. 14.10(f), we see a compact representation of the same linked list on exit from the function `insertBegin` as visualized in Python

Tutor. To add another node having value 20 at the beginning of the linked list, we invoke the method `insertBegin` with the argument 20. On execution of the method `insertBegin`, the final linked list and its compact representation are shown in Fig. 14.10(g) and Fig. 14.10(h), respectively.

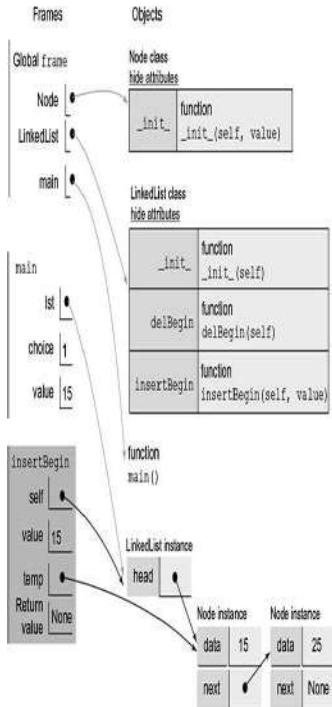




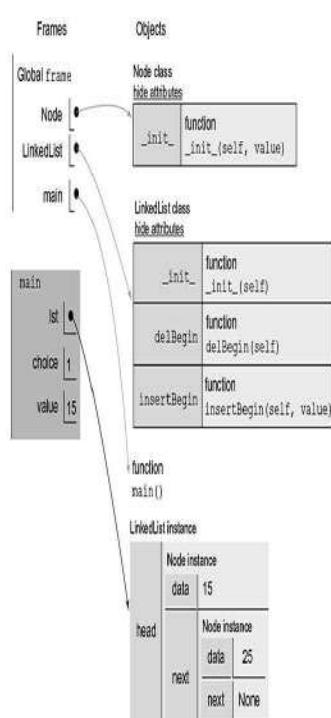
(c)



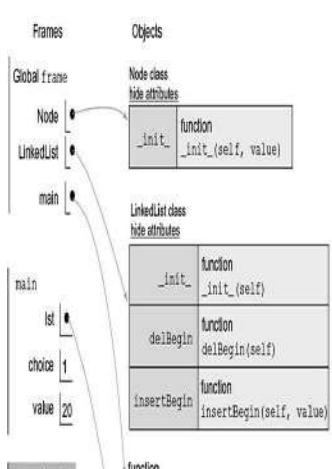
(d)



(e)



(f)



(g)

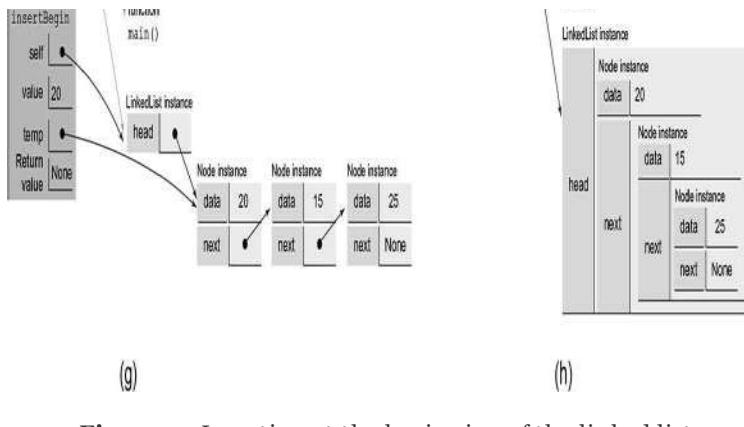
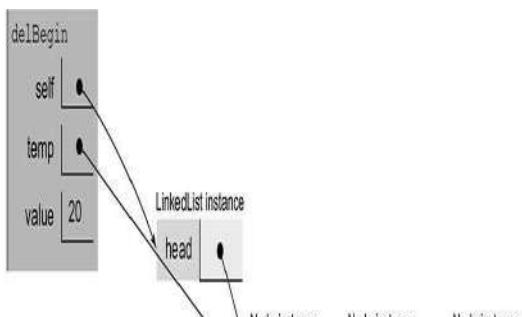
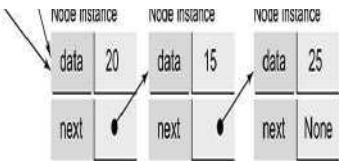


Fig. 14.10 Insertion at the beginning of the linked list

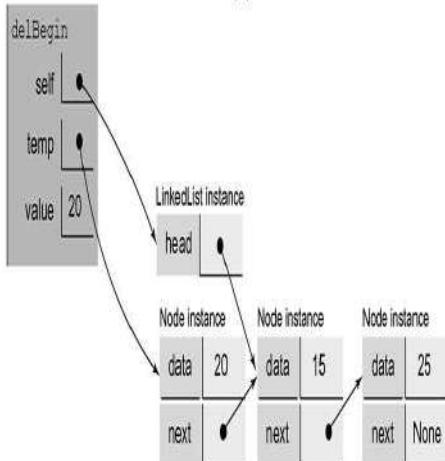
Next, we wish to delete a node from the beginning of the linked list. On entering choice 2, method `delBegin` is invoked. It first checks whether linked list is empty, i.e. whether the value of `head` is `None`. If the linked list is empty, the method displays the message '`Empty List`' and returns the control to the `main` function (lines 36–37, Fig. 14.8). However, the linked list `lst` is non-empty as it contains three nodes. To delete the first node, we first save a reference to it as `temp` and refer to its `data` as `value` (lines 39–40: Fig. 14.8, Fig. 14.11(a)). On execution of line 41 (Fig. 14.8), `head` of the linked list is updated to point to the second node, which now becomes the first node of the linked list (Fig. 14.11(b)). On the execution of line 42, the node pointed by `temp` is deleted using `del` (Fig. 14.11(c)). Finally, on the execution of line 43, method `delBegin` exits while returning `value` (20) (Fig. 14.11(d)).

deleting a node from the beginning

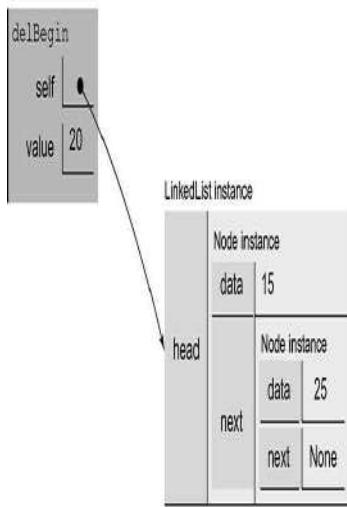




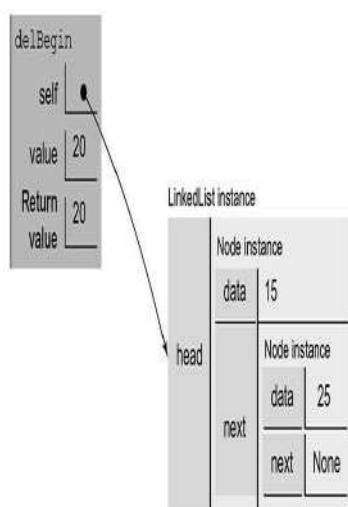
(a)



(b)



(c)



(d)

Fig. 14.11 Deletion from the beginning of linked list

14.3 DELETING A NODE WITH A PARTICULAR VALUE FROM A LINKED LIST

Often, we are interested in deleting a node having a particular value from a linked list. The method `delVal` (Fig. 14.12) takes the `value` as an input parameter and

deletes the first node with this `value`, if found in the linked list. If the list is empty, the function displays a message 'Empty List' (line 10) and exits. In line 12, we check whether the first node itself has the given `value` as its `data` attribute. If so, the execution of lines 13–16 deletes the first node of the linked list. Recall that this is what we did in the function `delBegin`.

attempt to delete from an empty list

deletion of first node

```
01 def delVal(self, value):
02     """
03         Objective: To delete a particular value from LinkedList
04         Input Parameters:
05             self (implicit parameter) - object of type LinkedList,
06             value - data for the node to be deleted
07         Return Value: value deleted
08     """
09     if self.head is None: #Empty List
10         print('Empty List')
11         return
12     if self.head.data == value: #First element is to be deleted
13         temp = self.head
14         self.head = self.head.next
15         del temp
16         print("Value Deleted!!")
17     else:
18         current = self.head
```

```

19     prev = None
20     while current != None and current.data != value:
21         prev = current
22         current = current.next
23     if current != None:
24         prev.next = current.next
25         del current
26         print('Value', value, 'Deleted!!!')
27     else:
28         print('Value not Found!!!')
29

```

Fig. 14.12 Method `delVal` of class `LinkedList` (`linkedList.py`)

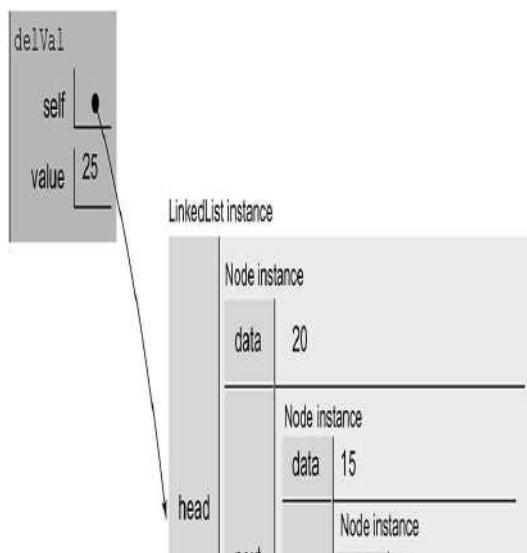
If the node to be deleted is not the first node, we traverse the linked list to search for the target node having `value` as its `data` attribute. To illustrate the process of deleting a node (not the first node) having a given value of `data`, we consider a linked list having four nodes comprising `data` 20, 15, 25, and 10 (Fig. 14.13(a)). Suppose we wish to delete the node having `data` 25. In order to delete the target node, we need to mark the target node as well as the node preceding the target node. A variable `current` is used that keeps track of the node we are currently dealing with. Another variable `prev` keeps track of the node previous to the current node. Initially, `current` is set to point to the first node, i.e. `head` of the linked list (line 18), and `prev` is assigned value `None` (line 19, Fig. 14.13(b)). Execution of `while` loop (lines 20–22) causes traversal of the linked list. Everytime lines 21 and 22 are executed, `prev` is assigned `current` and `current` is assigned `current.next`. i.e., `current` is updated to point to the next node until `current.data == value` or we reach the end of the linked list. (Fig. 14.13(c and d)).

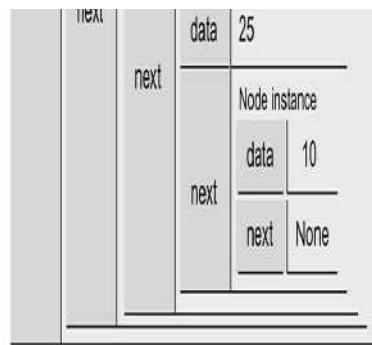
deleting a node somewhere in the middle: traverse the list to search for target value

keep track of the node previous to the target node

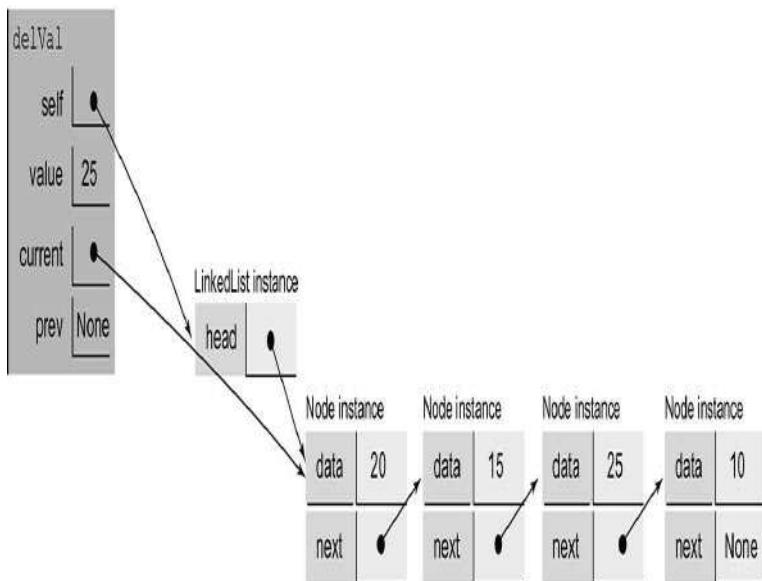
Since `value 25` is present in the linked list, the condition in line 23 evaluates to `True` when `current` refers to the node having `data 25`, and `prev` refers to the previous node having `data value 15`. Now, we have found the node to be deleted. At this point in time, `current` as well as `prev.next` refer to the target node to be deleted. Execution of line 24 updates the `next` link of the `prev` node instance to point to the node next to `current` node (Fig. 14.13(e)). Finally, the target node pointed to by `current` is deleted on the execution of line 25 (Fig. 14.13(f)). In case we reach the last node of the list without encountering any node with the given `value`, the condition in line 23 evaluates to `False` and the execution of line 28 displays the message '`Value not Found!!`'.

if a node having the target value is not found, display '`Value not Found!!`'.

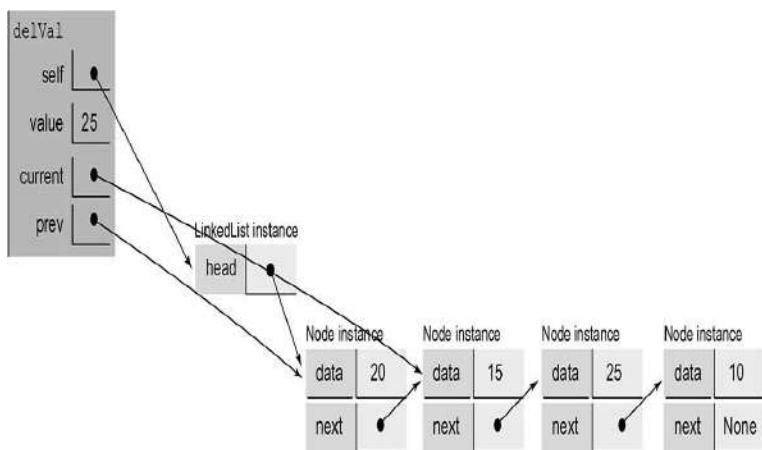




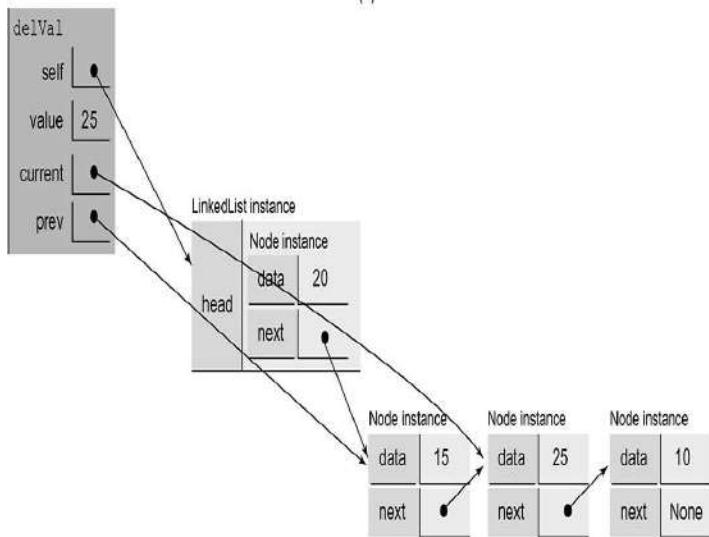
(a)



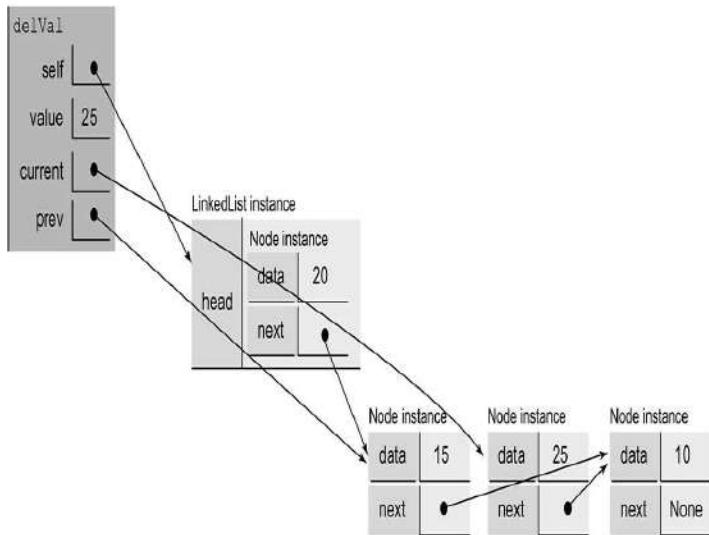
(b)



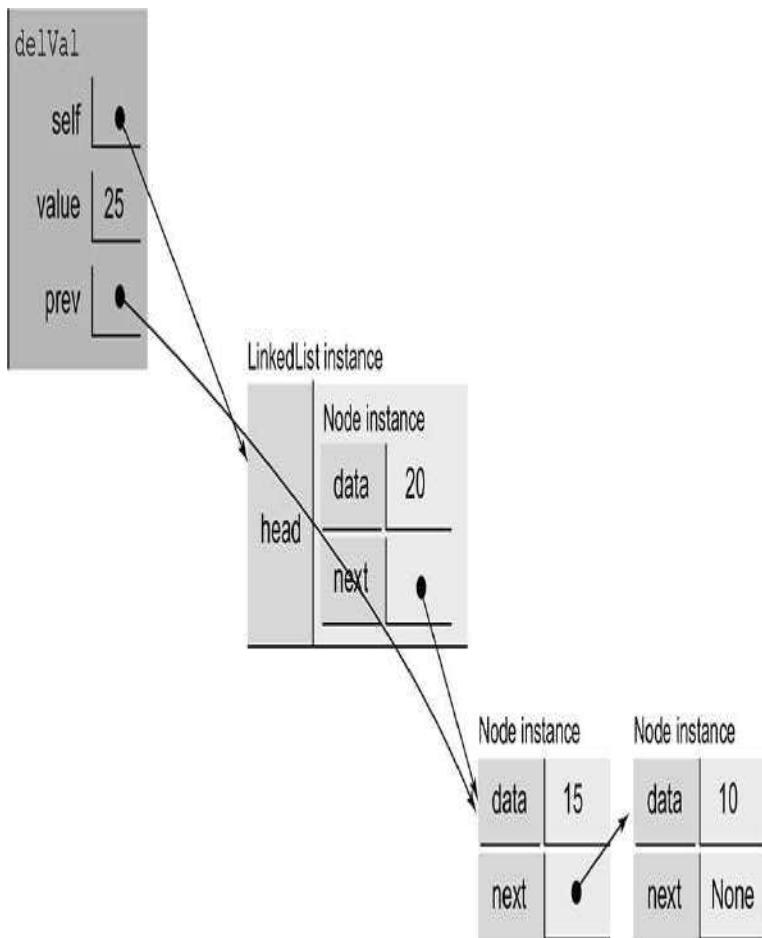
(c)



(d)



(e)



(f)

Fig. 14.13 Deletion of a particular value from the linked list

14.4 TRAVERSING A LINKED LIST

Next, we define the string representation for a linked list. Consider a linked list `lst` with three nodes having successive `data` values 20, 15, and 25. We define the string representation of this linked list to be the string: '`20->15->25`' as shown below:

```
>>> print(lst)
```

`20->15->25`

string representation of a linked list

The method `__str__` (Fig. 14.14) yields the string representation of the linked list. When the method `__str__` defined in Fig. 14.14 is included in a script being executed in Python Tutor that involves the class `LinkedList`, it suppresses the visualization of the nodes being used by Python Tutor. Therefore, when we need the visualization of linked lists in Python Tutor, we should remove this method from the class `LinkedList`.

```
01 def __str__(self):
02     """
03         Objective: To return string representation of an object of
04         type LinkedList
05         Input Parameter: self (implicit parameter) - object of type
06             LinkedList
07         Return Value: string
08     """
09     current = self.head
10     result = ''
11     if current != None:
12         while current.next != None:
13             result += str(current.data)+ '->'
14             current = current.next
15             result+= str(current.data)
16     else:
17         result = 'Empty List'
18     return result
```

Fig. 14.14 Method `__str__` of class `LinkedList`
(`linkedList.py`)

14.5 MAINTAINING SORTED LINKED LIST WHILE INSERTING

Many times, we are interested in maintaining data in sorted order to facilitate easy retrieval. In this section we will discuss how to use a linked list for maintaining the data in ascending order. For this purpose, we insert a new node at the correct position so that data in the modified linked list is always in sorted order. Let us examine the method `insertSort` (Fig. 14.15) for

maintaining a sorted linked list. On execution of the function `main` (Fig. 14.16), the user is repeatedly prompted to provide `value` for the node to be inserted in the linked list until the user enters an empty string ('').

maintaining a sorted linked list facilitates retrieval

As first step, the method `insertSort` checks whether linked list is empty i.e. whether `head` has the value `None` (line 21). If the linked list is currently empty, it invokes the constructor for the `Node` class with the argument `value` (line 22) to create a node having `value` as `data` and `None` as the value of the `next` link. The attribute `head` of the linked list now refers to this node instance (line 22). If the linked list is not empty and the `value` to be inserted is smaller than the value of the `data` of the first node, execution of lines 24–26 creates a new node and inserts it at the beginning of the linked list and the modified list remains in ascending order of `data` values. Otherwise, we need to find the proper position for inserting the new node with given `value`.

insertion in empty list

inserting a node with lowest value

inserting a node somewhere in the middle requires traversing the list

To illustrate the process of inserting a node (not the first node) having a given `value` of `data`, we consider a linked list, having three nodes comprising `data` 1, 3, and 5 (Fig. 14.17(a)). Suppose we wish to insert a node having `data` value 4 in the linked list. As usual, the

variable `current` is used to keep track of the node with which we are currently dealing. Initially, it is set to point to the first node, i.e. head of the linked list (line 28). We use another variable `prev` to keep track of the node previous to the current node. Initially, it is set to `None` (line 29 (Fig. 14.15), Fig. 14.17(b)). Execution of while loop (lines 30–34) causes traversal of the linked list until the control reaches the node before which the new node is to be inserted (i.e. `current.data > value`) (Fig. 14.17(c) and Fig. 14.17 (d)), or end of the linked list is reached. Execution of line 35 invokes the constructor for the `Node` class with the argument 4 to create a node `temp` having 4 as the value of `data` and `None` as the value of the `next` link (Fig. 14.17(e)). Execution of line 36 updates the `next` link of node instance pointed by `prev` to point to the new node (being referred to as `temp`) (Fig. 14.17(f)). Further, on execution of line 37, the `next` link of the `temp` node instance is updated to refer to the node instance `current` (Fig. 14.17(g)). In Fig. 14.17(h), we see a compact representation of the same linked list on exit from the function `insertSort`.

```

01  from node import Node
02  class LinkedList:
03      def __init__(self):
04          """
05              Objective: To initialize object of class LinkedList
06              Input Parameter:
07                  self (implicit parameter) - object of type LinkedList
08              Return Value: None
09          """
10      self.head = None
11
12
13  def insertSort(self, value):
14      """
15          Objective: To insert a node with given value in the sorted
16          LinkedList
17          Input Parameters:
18              self (implicit parameter) - object of type LinkedList,
19              value - data for the node to be inserted
20          Return Value: None
21          """
22      if self.head is None:

```

```
22         self.head = Node(value)
23     elif value < self.head.data:
24         temp = Node(value)
25         temp.next = self.head
26         self.head = temp
27     else:
28         current = self.head
29         prev = None
30         while current != None:
31             if current.data > value:
32                 break
33             prev = current
34             current = current.next
35         temp = Node(value)
36         prev.next = temp
37         temp.next = current
38         print('Value Inserted!!')
39
40
41     def __str__(self):
42         """
43             Objective: To return string representation of object of
44             type LinkedList
```

```
45     Input Parameter: self (implicit parameter) - object of type
46     LinkedList
47     Return Value: string
48     """
49
50     temp = self.head
51     result = ''
52     if temp != None:
53         while temp.next != None:
54             result += str(temp.data)+ '->'
55             temp = temp.next
56         result+= str(temp.data)
57     else:
58         result = 'Empty List'
59
60     return result
```

Fig. 14.15 Class LinkedList (linkedListSorted.py)

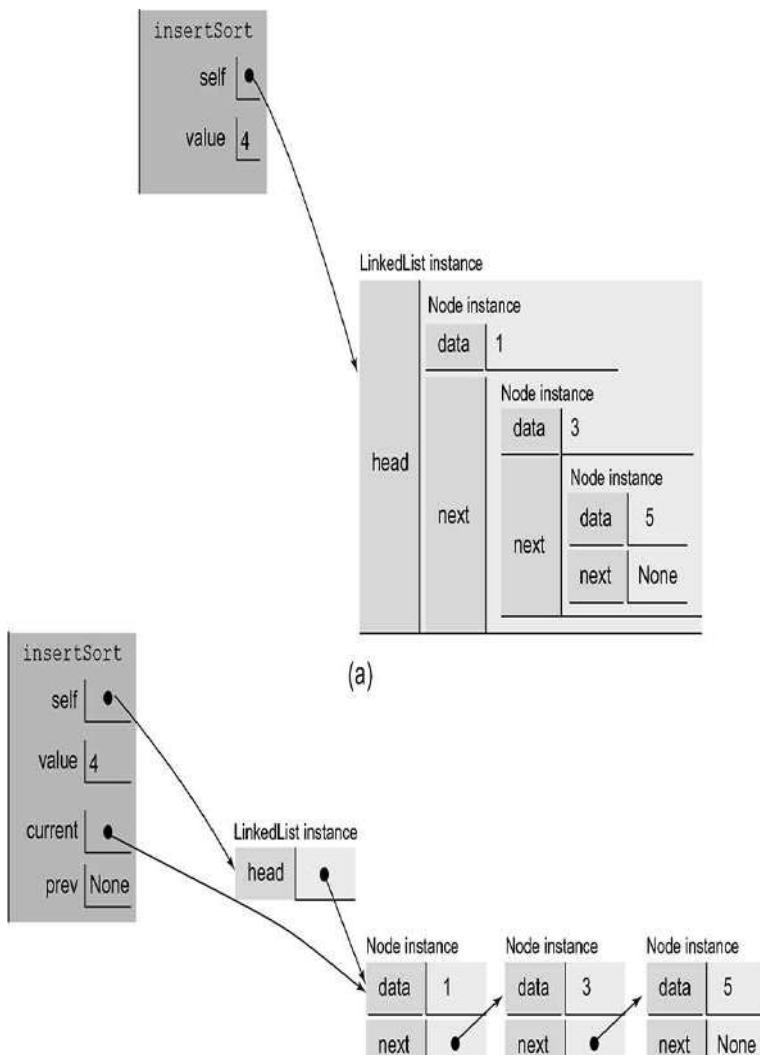
```
01 from linkedListSorted import LinkedList
02 def main():
03     """
04     Objective: To maintain a linked list in sorted order
05     Input Parameter: None
06     Return Value: None
07     """
08     lst = LinkedList()
09     while 1:
10         ...value = input("Enter value to be inserted: ")
```

```

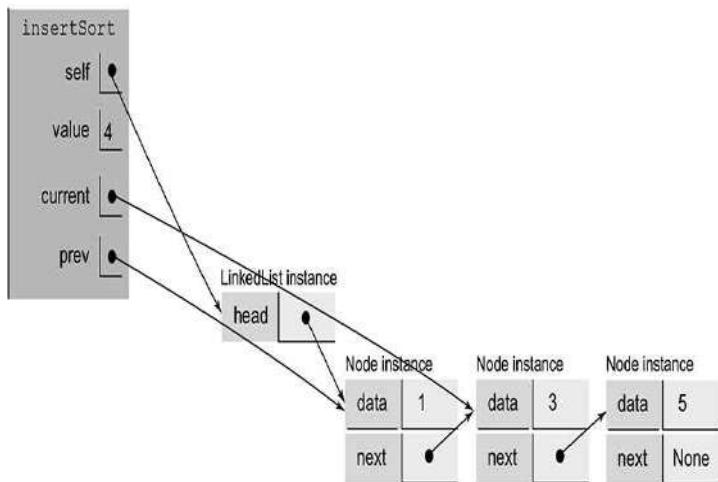
10         value = input('Enter value to be inserted: ')
11     if value == '':
12         break
13     lst.insertSort(eval(value))
14     print('Current Linked List Status')
15     print(lst)
16
17 if __name__ == '__main__':
18     main()

```

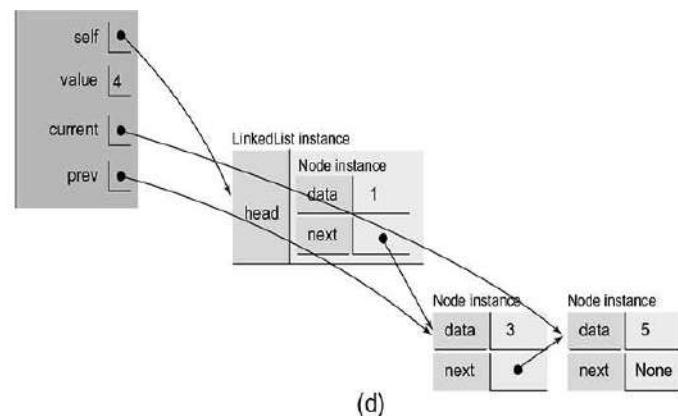
Fig. 14.16 Class LinkedList (`linkedListSortedMain.py`)



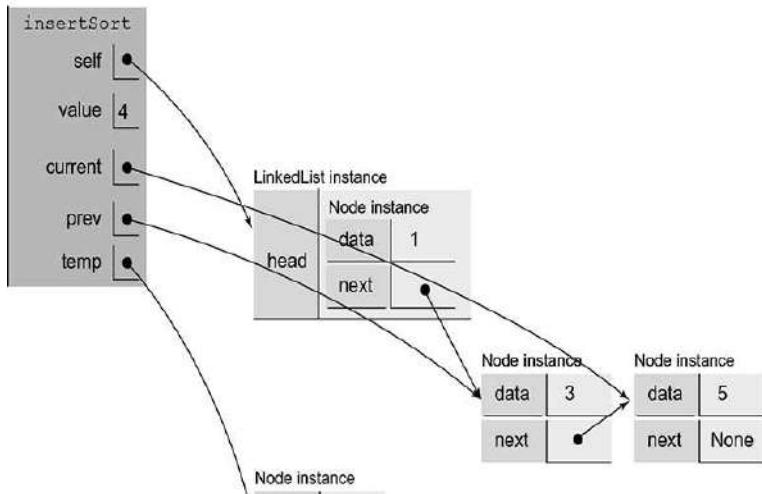
(b)

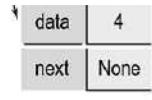


(c)

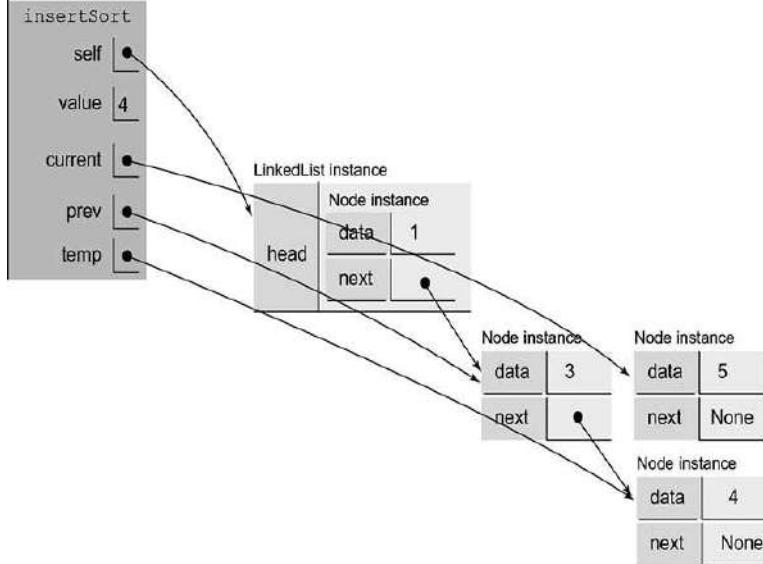


(d)

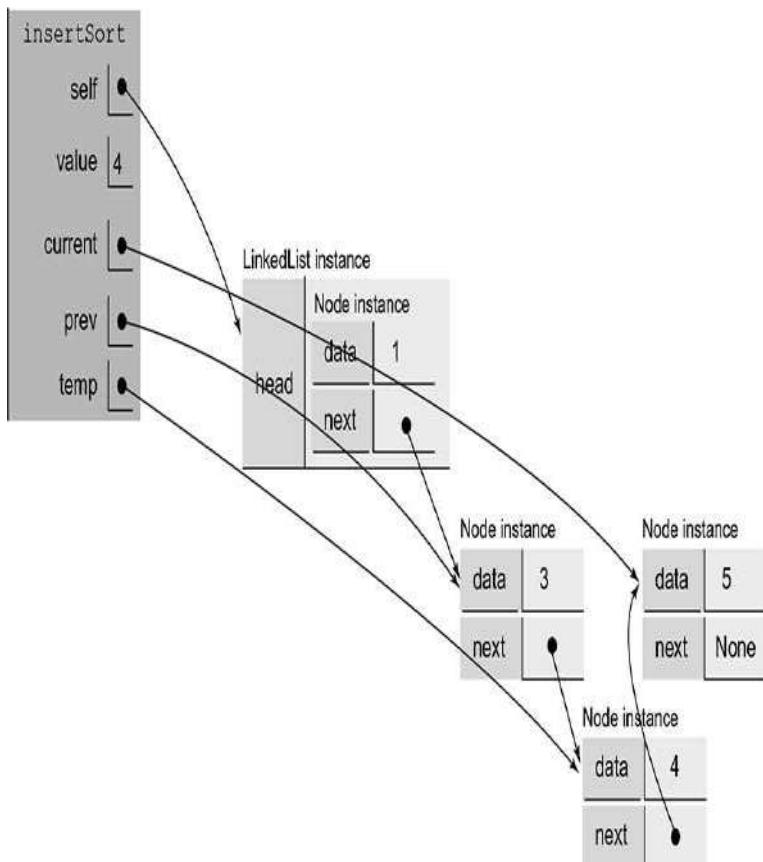




(e)



(f)



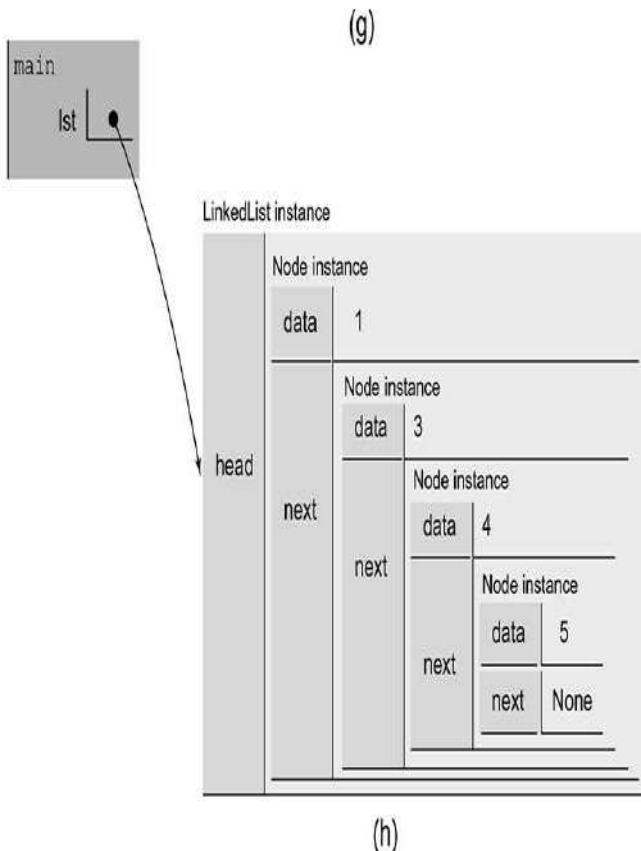


Fig. 14.17 Sorted linked list `lst`

`variable current:` keeps track of the current node of interest

`variable prev:` provides a link/reference to the node before
the `current` node

14.6 STACK IMPLEMENTATION USING LINKED LIST

In the script `linkedStack` (Fig. 14.18), we present an implementation of the stack operations using linked list. For this purpose, we define the class `LinkedStack`. We introduce a variable `top` to keep track of the node that was added most recently to the stack and initialize it to `None`. We also define the following methods:

```
01  from node import Node
```

```
02 class LinkedStack:
03
04     def __init__(self):
05         """
06             Objective: To initialize a LinkedStack object
07             Input Parameter:
08                 self (implicit parameter) - object of type LinkedStack
09             Return Value: None
10             """
11         self.top = None
12
13     def push(self, value):
14         """
15             Objective: To insert a node on top of the stack
16             Input Parameters:
17                 self (implicit parameter) - object of type LinkedStack
18                 value - data for the node to be inserted
19             Return Value: None
20             """
21         if self.top is None:
22             self.top = Node(value)
23         else:
24             temp = Node(value)
25             temp.next = self.top
26             self.top = temp
27
28     def pop(self):
29         """
30             Objective: To remove a node from the top of stack
31             Input Parameter:
32                 self (implicit parameter) - object of type LinkedStack
33             Return Value: value of the data attribute of the Top element
34             of the stack if stack is not empty, otherwise None
35             """
36         if self.top is None:    #Empty List
37             print('Stack Underflow')
38             return None
39         else:
40             temp = self.top
41             value = self.top.data
42             self.top = self.top.next
43             del temp
44             return value
45
46     def isEmpty(self):
```

```
47     """
48     Objective: To determine whether the stack is empty
49     Input Parameter:
50         self (implicit parameter) - object of type LinkedStack
51     Return Value: True if the stack is empty, False otherwise
52     """
53     return self.top is None
54
55 def getTop(self):
56     """
57     Objective: To return top element of the stack
58     Input Parameter:
59         self (implicit parameter) - object of type LinkedStack
60     Return Value: value of the data attribute of the Top element
61     of the stack if stack is not empty, otherwise None
62     """
63     if not(self.isEmpty()):
64         return self.top.data
65     else:
66         print('Stack Empty')
67         return None
68
69 def __str__(self):
70     """
71     Objective: To return string representation of a LinkedStack
72     object
73     Input Parameter: self (implicit parameter) - object of
74     type LinkedStack
75     Return Value: string
76     """
77     temp = self.top
78     result = ''
79     if temp != None:
80         while temp.next != None:
81             result += str(temp.data)+ '->'
82             temp = temp.next
83             result+= str(temp.data)
84     else:
85         result = 'Empty Stack'
86     return result
```

Fig. 14.18 Class `LinkedStack` (`linkedStack.py`)

- `push`: Push a node at the top of the stack
- `pop`: Pop off a node from the stack
- `isEmpty`: Check whether the stack is empty
- `getTop`: Retrieve the value at the top node, without affecting the stack contents
- `__str__`: String representation of the stack that comprises the concatenation of string representation of data in each node beginning the top node. For the purpose of demonstration, we have provided this method to display the contents of the entire stack, even though stack operations do not require this method.

`push`: analogous to insertion at the beginning

`pop`: analogous to deletion from the beginning

As we always push the new node on the stack as the top node, we must insert it at the beginning of the linked list. Similarly, we pop off a node from the stack by deleting the node at the beginning of the linked list. In the script `linkedStackMain` (Fig. 14.19), we present the main function for demonstrating various operations of the class `LinkedStack`.

```
01 from linkedStack import LinkedStack
02 def main():
03     """
04     Objective: To study stack functionality
05     Input Parameter: None
06     Return Value: None
07     """
08     while 1:
09         print('Choose an option \n')
10         print('1: Create stack')
11         print('2: Delete stack')
12         print('3: Push')
13         print('4: Pop')
14         print('5: Print Stack Data')
15         print('6: Top element')
16         choice = int(input('Enter Choice: '))
17         if choice == 1:
18             stk = LinkedStack()
19             print('Stack Created')
20         elif choice == 2:
21             del stk
22             print('Stack Deleted')
23         elif choice == 3:
```

```

24     element = int(input('Enter integer value to be pushed: '))
25     stk.push(element)
26     print('Value pushed!!!')
27 elif choice == 4:
28     value = stk.pop()
29     print('Element popped:', value)
30 elif choice == 5:
31     print(stk)
32 elif choice == 6:
33     value = stk.getTop()
34     print('Top element', value)
35 proceed = input('enter y if you wish to continue: ')
36 if proceed != 'y' and proceed != 'Y':
37     break
38
39 if __name__=='__main__':
40     main()

```

Fig. 14.19 main function using operations of class LinkedStack
(linkedStackMain.py)

14.7 QUEUE IMPLEMENTATION USING LINKED LIST

In the script `linkedQueue` (Fig. 14.20), we present an implementation of the queue operations using a linked list. For this purpose, we define the class `LinkedQueue`. We introduce variables `front` and `rear` to keep track of the nodes that were inserted first and last, respectively, in the queue and initialize these variables to `None`. We also define the following methods:

- `enqueue`: Insert a node at the rear end of the queue
- `dequeue`: Delete a node from the front end of the queue
- `isEmpty`: Check whether the queue is empty
- `getFront`: Retrieve the value at the front end of the queue, without affecting its contents
- `__str__`: String representation of the queue that comprises the concatenation of string representation of data in each node beginning the front node. For the purpose of demonstration, we have provided this method to display the contents of the entire queue, even though queue operations do not require this method.

`enqueue`: add an element to a queue

In the script `linkedQueue` (Fig. 14.20), we define the class `LinkedQueue`. In the method `enqueue`, we first check whether the linked list is empty, i.e. whether `front == None`. If the linked list is empty, we invoke the constructor for the `Node` class with the argument `value` (line 22). Now we can refer to the node just created via `front` and `rear` of the linked list (line 22). However, if the linked list is non-empty, we need to insert the new node after the last node in the linked list which is being referred by `rear` (line 24). Also, we update the `rear` to point to this newly created node (line 25). The method `dequeue` deletes the first node of the linked queue referred by `front` using the following approach:

```

001 from node import Node
002 class LinkedQueue:
003
004     def __init__(self):
005         """
006             Objective: To initialize an object of class LinkedQueue
007             Input Parameter:
008                 self (implicit parameter)- object of type LinkedQueue
009             Return Value: None
010         """
011         self.front = self.rear = None
012
013     def enqueue(self, value):
014         """
015             Objective: To insert a node in the queue at the rear end
016             Input Parameters:
017                 self (implicit parameter)- object of type LinkedQueue
018                 value - data for the node to be inserted

```

```
010         value - data for the node to be inserted
019     Return Value: None
020     """
021     if self.front is None:
022         self.front = self.rear = Node(value)
023     else:
024         self.rear.next = Node(value)
025         self.rear = self.rear.next
026
027     def dequeue(self):
028         """
029             Objective: To remove a value from the front of Linkedqueue
030             Input Parameter:
031                 self (implicit parameter)- object of type Queue
032             Return Value: Value of the data attribute of the front node
033             if the queue is not empty, None otherwise
034         """
035
036         if self.front is None:    #Empty List
037             print('Queue Underflow')
038             return None
039         else:
040             temp = self.front
041             value = self.front.data
042             self.front = self.front.next
043             del temp
044             return value
045
046     def isEmpty(self):
```

```
047     """
048     Objective: To determine whether the queue is empty
049     Input Parameter:
050         self (implicit parameter)- object of type LinkedQueue
051     Return Value: True if the queue is empty, otherwise False
052     """
053     return self.front is None
054
055     def getFront(self):
056         """
057             Objective: To return value at the front of Linkedqueue
058             Input Parameter:
059                 self (implicit parameter)- object of type Queue
```

```

060     Return Value: Value of the data attribute of the front node
061     of the queue if the queue is not empty, None otherwise
062     """
063     if not(self.isEmpty()):
064         return self.front.data
065     else:
066         print('Queue Empty')
067         return None
068
069     def __str__(self):
070         """
071             Objective: To return string representation of object of
072             type Queue
073             Input Parameter: self (implicit parameter)- object of type
074             LinkedQueue
075             Return Value: string
076             """
077         temp = self.front
078         result = ''
079         if temp != None:
080             while temp.next != None:
081                 result += str(temp.data)+ '->'
082                 temp = temp.next
083                 result+= str(temp.data)
084             else:
085                 result = 'Empty Queue'
086         return result

```

Fig. 14.20 Class LinkedQueue (linkedQueue.py)

if queue is not empty, return the value of the data attribute of the front node of the queue, otherwise, return None

dequeue: delete from a queue

The methods `isEmpty`, `getFront`, and `__str__` are defined in a manner similar to the corresponding methods described for the class `LinkedStack`.

In the script `linkedQueueMain` (Fig. 14.21), we present `main` function to demonstrate various operations of the class `LinkedQueue`.

```
01 from linkedQueue import LinkedQueue
02 def main():
03     """
04     Objective: To provide queue functionality
05     Input Parameter: None
06     Return Value: None
07     """
08     while 1:
09         print('Choose an option \n')
10         print('1: Create queue')
11         print('2: Delete queue')
12         print('3: Enqueue')
13         print('4: Dequeue')
14         print('5: Print Queue Data')
15         print('6: Front value')
16         choice = int(input('Enter Choice: '))
17         if choice == 1:
18             q = LinkedQueue()
19         elif choice == 2:
20             del q
21             print('Queue Deleted')
22         elif choice == 3:
23             value = int(input('Enter value to be inserted: '))
24
```

```

24         q.enqueue(value)
25         print('Node Inserted!!!')
26     elif choice == 4:
27         value = q.dequeue()
28         print('Value dequeued:', value)
29     elif choice == 5:

```

```

30         print(q)
31     elif choice == 6:
32         value = q.getFront()
33         print('Front value', value)
34     proceed = input('Enter y if you wish to continue: ')
35     if proceed != 'y' and proceed != 'Y':
36         break
37
38 if __name__=='__main__':
39     main()

```

Fig. 14.21 main function using operations of the class
LinkedQueue (linkedQueueMain.py)

SUMMARY

1. A linked list comprises objects, called nodes, each of which contains a link to the next object (node). Thus, next link defines the relationship of a node to the *next* node.
2. In Python, all names reference objects, i.e. instances of classes.
3. A linked list may require the following functionality:
 1. Creating an empty linked list
 2. Inserting a node at the beginning of the linked list
 3. Deleting a node from the beginning of the linked list
 4. Deleting a node with particular value from the linked list
 5. Inserting a node in sorted linked list

4. In the stack implementation of linked list, push and pop operations are analogous to the following operations in a linked list: insertion at the beginning and deletion from the beginning.
5. In queue implementation using linked list, the method enqueue inserts at the rear end of the linked queue and the method dequeue deletes from the front end of the linked queue.

EXERCISES

1. Write a method `insertEnd` for the class `LinkedList` that takes a value as an input and adds a node having that value at the end of the linked list.
2. Write a method `delEnd` for the class `LinkedList` that deletes the last node of the linked list and returns the value of the `data` attribute of the last node.
3. Write a function that takes two sorted linked lists as input parameters and returns the merged, sorted list.
4. Write a method `findVal` for the class `LinkedList` that takes `n` as an input parameter and returns value of the `data` attribute of the `n`th node starting from the beginning.
5. Write an iterative method `reverse` for the class `LinkedList` that reverses the given linked list.
6. Write a recursive method `reverDisplay` for the class `LinkedList` that displays the data values of the linked list from the right end.
7. Write a method `divideList` for the class `LinkedList`. The method should create and return a tuple comprising two linked lists, one comprising nodes at even positions and another comprising nodes at odd positions.
8. A variant of the linked list is known as doubly linked list or two-way linked list (Fig. 14.22) comprising the following node structure having two links (`next` and `prev`) for traversing in both directions.

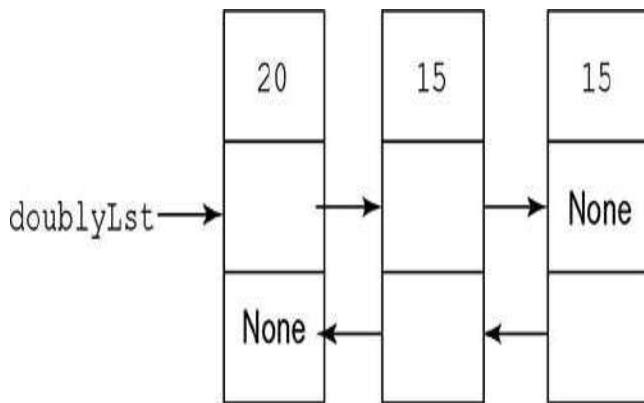


Fig. 14.22 Doubly linked list

```

class Node:
    def __init__(self, value):
        ...
        Objective: To initialize an object of
        class Node
  
```

```

Input Parameter:
    self (implicit parameter) - object of
type Node
    Return Value: None
    ''
    self.data = value
    self.next = None
    self.prev = None

```

Define a class `DoublyLinkedList` which supports methods `insertBegin`, `insertEnd`, `delBegin`, `delEnd`, `delVal`, and `traverse`.

9. A variant of linked list is known as a circular list (Fig. 14.23) for traversing the nodes of the linked list in a circular manner. Define a class `CircularLinkedList` which supports the methods `insertBegin`, `insertEnd`, `delBegin`, `delEnd`, `delVal`, and `traverse`.

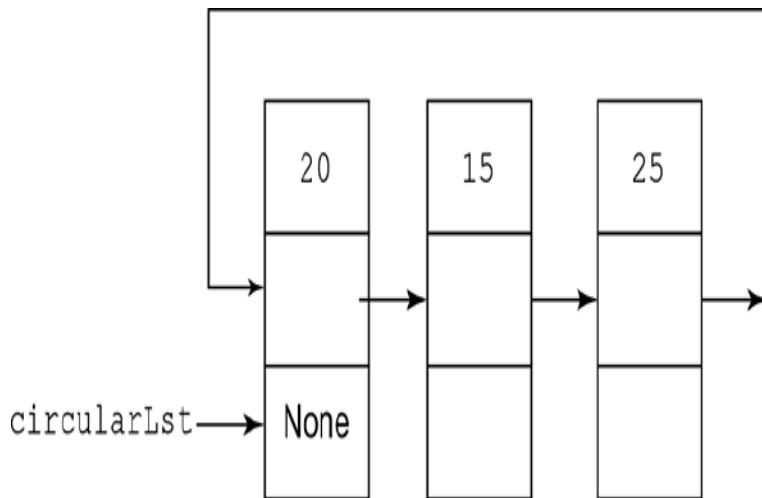


Fig. 14.23 Circular linked list

CHAPTER 15

DATA STRUCTURES III: BINARY SEARCH TREES

CHAPTER OUTLINE

[15.1 Definitions and Notations](#)

[15.2 Binary Search Tree](#)

[15.3 Traversal of Binary Search Trees](#)

[15.4 Building Binary Search Tree](#)

In the previous chapters, we discussed linear data structures stacks, queues, and linked lists. However, there are situations that necessitate the use of the non-linear arrangement of data; for example, a directory is a hierarchical structure like a tree. In this chapter, we will briefly talk about trees and move on to study in detail binary search trees—a particular form of trees, used for efficiency in searching.

15.1 DEFINITIONS AND NOTATIONS

A tree is a hierarchical structure. It is a collection of elements called nodes along with a relation called parenthood that defines a hierarchical structure on the nodes. The node not having any parent node is called the root node. Every node other than the root has a unique parent node. If node A is the parent of node B, we also say that node B is a child of the node A. Multiple nodes may have the same parent node. In a diagram, we usually represent a node by encircling the node label. A node label is also called value of the node. In Fig. 15.1, we see a tree comprising eight nodes including the root A. The arrows are used to express parent–child relationships. The node A has three children, namely, B, C, and D. Alternatively, we may say that A is the parent of each of the nodes B, C, and D. Similarly, the node D is the parent node for nodes F and G. Note that a node may

or may not have any children, for example, none of the nodes E, C, H, and G has any child node.

tree: a hierarchical arrangement of nodes

root node: node without any parent node

no two nodes may have the same child node, however,
multiple nodes may have the same parent node

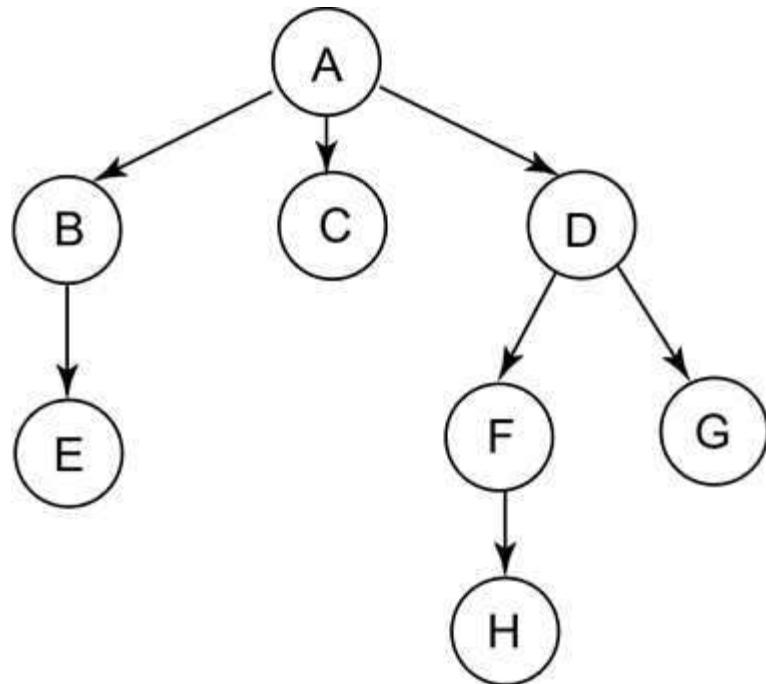


Fig. 15.1 A tree

root: A

internal nodes: A, B, D, F

leaves: E, C, H, G

Now, we describe some terminology in the context of the tree shown in Fig. 15.1:

1. **Leaf:** A leaf node is a node which has no children. In the figure, the nodes E, C, H, and G are leaves.
2. **Internal node:** A node which is not a leaf node is called internal node or a non-leaf node. In the figure, the nodes A, B, D, and F are internal nodes.
3. **Path and Path Length:** If n_1, n_2, \dots, n_k is a sequence of nodes in a tree such that n_i is the parent of n_{i+1} for $1 \leq i \leq k-1$, the sequence is called a path of length $k-1$ from n_1 to n_k . In the figure, A, D, G is a path of length 2.

tree terminology

4. **Height:** The length of the longest path from the root to a leaf in a tree defines the height of the tree. For example, in Fig. 15.1, A, D, F, H is the longest path, and thus the height of the tree is three. Also, we define height of an empty tree to be zero.
5. **Level:** Level of a node is the number of edges on the path from the root to the node. For example, nodes A, D, F, and H are at levels 0, 1, 2, and 3, respectively.
6. **Ancestor/Descendant:** If there is a path from node n_1 to node n_2 , we say that n_1 is an ancestor of node n_2 and n_2 is a descendant of node n_1 . Thus, if node n_1 is an ancestor of node n_2 , then n_1 is either parent of n_2 or parent of an ancestor of node n_2 . Also, by definition, a node is an ancestor as well as descendant of itself. For example, the node D is an ancestor of each of the nodes D, F, G, and H. Note that node D being parent of each of the nodes F and G, is an ancestor of each of the nodes F and G. Similarly, the node F being parent of the node H is an ancestor of node H. Also, the node D being an ancestor of node F, which in turn is an ancestor of node H, is an ancestor of node H. Similarly, the nodes D, F, G, and H are descendants of the node D.
7. **Siblings:** Children of the same node are siblings to each other. For example, B, C, and D being children of A, are siblings. Similarly, F and G are siblings.
8. **Subtree:** A subtree of a tree is a tree that comprises a node of the tree together with all its descendants, for example in Fig. 15.1, the tree comprising the nodes D, F, G, and H is subtree of the tree having root A.

15.2 BINARY SEARCH TREE

A binary tree is a particular type of tree whose nodes have two or fewer children. In a binary tree, the child nodes of a node are called the left-child and right-child. A binary tree may be an empty tree, comprising no nodes at all, or may comprise a root node along with other nodes which may be organized as a left binary

subtree and a right binary subtree of the root node. The left and right binary subtrees may also be empty (as per the definition of binary tree). For example, in Fig. 15.2(a), binary tree having the node labelled 6 as the root (also called binary tree rooted at node 6) has two subtrees—a binary tree rooted at 10 and another binary tree rooted at 23. The binary tree rooted at 10 has a left-child—the binary tree rooted at 15. We say that the binary tree rooted at 10 does not have a right-child, or that the right-child is an empty tree. The node labeled 15 is a leaf node as its left-child as well as the right-child is an empty tree. Similarly, the nodes labelled 30 and 20 are leaves.

binary tree: empty tree or a tree with root node having a left binary subtree and a right binary subtree

strictly binary tree: binary tree in which every non-leaf node has non-empty left and right subtree

There are some special types of binary tree like strictly binary tree (Fig. 15.2(b)) and complete binary tree (Fig. 15.2(c)). A strictly binary tree is a binary tree in which every non-leaf node has non-empty left and right subtree. Thus, each node in a strictly binary tree has either two children or none. A complete binary tree is a binary tree that is full on all levels except the lowest level which is filled in from left to right.

complete binary tree: a binary tree that is full on all levels except the lowest level which is filled in from left to right

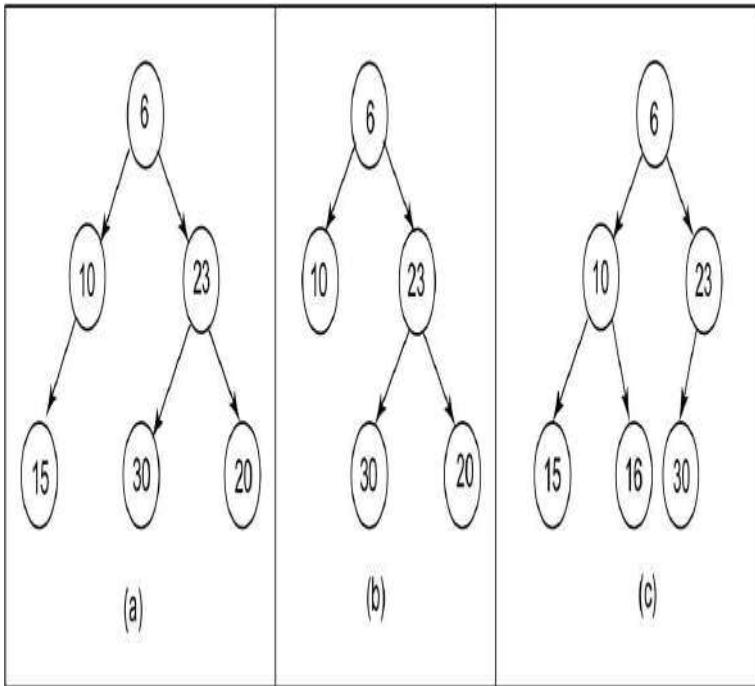


Fig. 15.2 Binary Tree

A binary tree is called a binary search tree if for any node n of the binary tree, every node in its left subtree has value smaller or equal to that of the node n , and every node in its right subtree has value larger than that of node n . For example, in the following binary search tree (Fig. 15.3), the root node has value 15. Note that values 10 and 6 in the left subtree of the root are smaller than 15, and values 23, 20, and 30 in the right subtree of the root are larger than 15. Similarly, value 6 in the left subtree of binary search tree rooted at 10 is less than 10. Further, values 20 and 30 in the left and right subtree, respectively of the binary search tree rooted at 23 are smaller and larger than 23, respectively. In the rest of this chapter, we will focus only on the binary search trees.

binary search tree: binary tree in which, for every node n , every node in its left subtree has value smaller or equal to it and every node in its right subtree have value larger than it

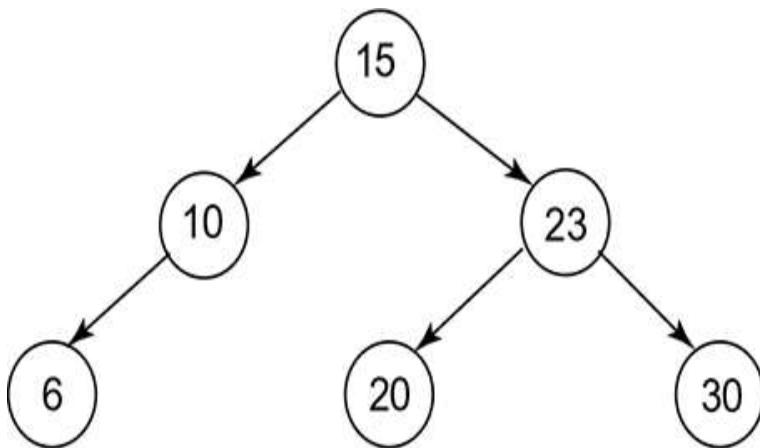


Fig. 15.3 Binary search tree

As a node in a binary tree defines *left-child* and *right-child* relationship, it may be thought of as an object having three members, namely, `left` (denoting left-child), `data`, and `right` (denoting right-child). In the class `Node` (Fig. 15.4), we define method `__init__` for initializing nodes of a binary tree. Note that when we create a node object, `left` and `right` children are assigned value `None`.

data members of `Node` class: `left`, `data`, and `right`

Now let us examine the script `BSTree` (Fig. 15.5) that creates the binary search tree `bst` shown in Fig. 15.3. Execution of line 8 (Fig. 15.5) creates a node instance comprising `data` (int object 15), `left` link (object `None`) and `right` link (object `None`) (Fig. 15.6(a)). This node instance is named as `bst`. Execution of line 9 creates a node instance comprising `data` (int object 23), `left` link (object `None`) and `right` link (object `None`) (Fig. 15.6(b)). Now, the name `bst.right` which referred to `None` on the execution of line 8, refers to the new node just created. Thus, execution of line 9 defines the right-child of the root node. Similarly, execution of lines 10 and 11 defines the right child and the left child of the binary search tree having the root with `data` 23 (Figs. 15.6(c) and 15.6(d)) by creating node instances

with `int` objects 30 and 20, respectively. Further, execution of line 12 defines left-child of the root node (Fig. 15.6(e)). It creates a node instance comprising `data` (`int` object 10), `left` link (object `None`), and `right` link (object `None`). Now, `bst.left` which referred to `None` on execution of line 8, refers to the new node just created. Finally, execution of line 13 defines the left-child of this node having `data` (`int` object 6) (Fig. 15.6(f)).

```
01 class Node:  
02     def __init__(self, value):  
03         '''  
04         Objective: To initialize object of class Node  
05         Input Parameter:  
06             self (implicit parameter) - object of type Node  
07             Return Value: None  
08         '''  
09         self.left = None  
10         self.data = value  
11         self.right = None
```

Fig. 15.4 Class Node of binary search tree (bNode.py)

```
01 from bNode import Node  
02 def main():  
03     '''  
04     Objective: To build a binary search tree  
05     Input Parameter: None  
06     Return Value: None  
07     '''  
08     bst = Node(15)  
09     bst.right = Node(23)  
10     bst.right.right = Node(30)  
11     bst.right.left = Node(20)  
12     bst.left = Node(10)
```

```

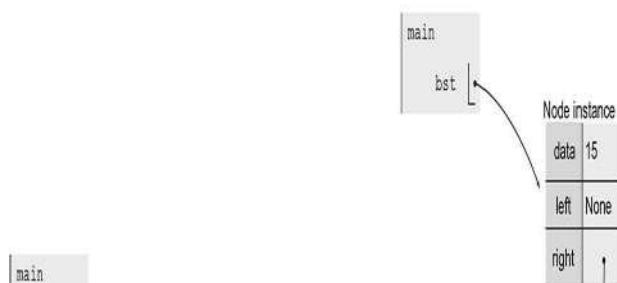
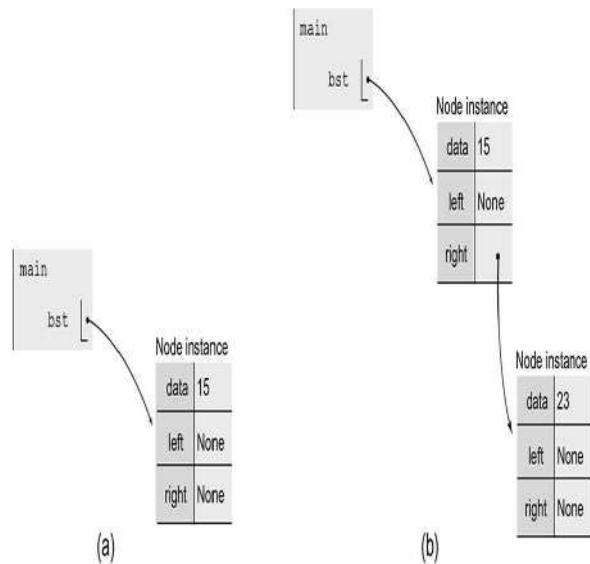
12     bst.root = Node(10)
13     bst.left.left = Node(6)
14
15 if __name__ == '__main__':
16     main()

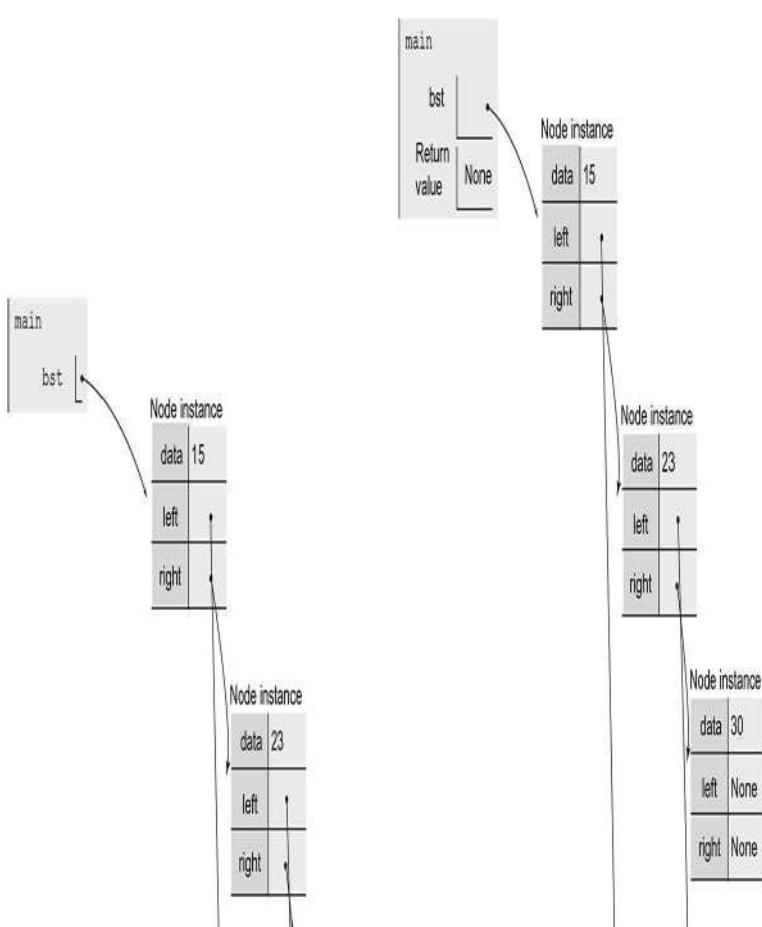
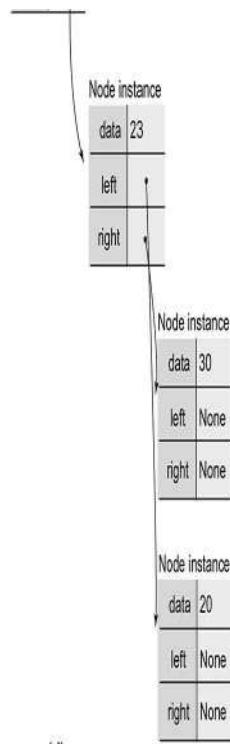
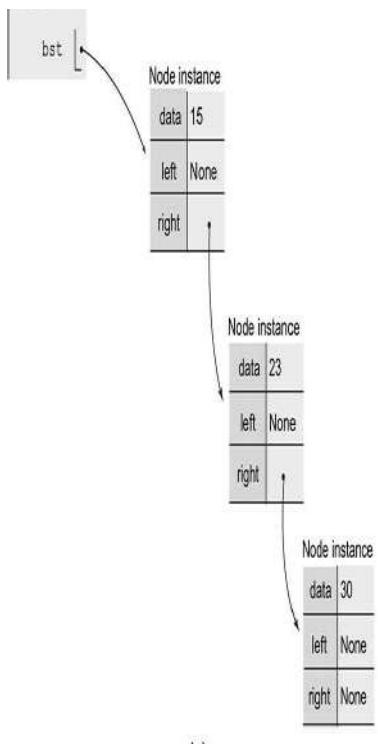
```

Fig. 15.5 Creating a binary search tree (BSTree.py)

empty binary tree can be created by assigning `None` to `bst`

Note that an empty binary tree can be constructed by assigning `None` to `bst`.





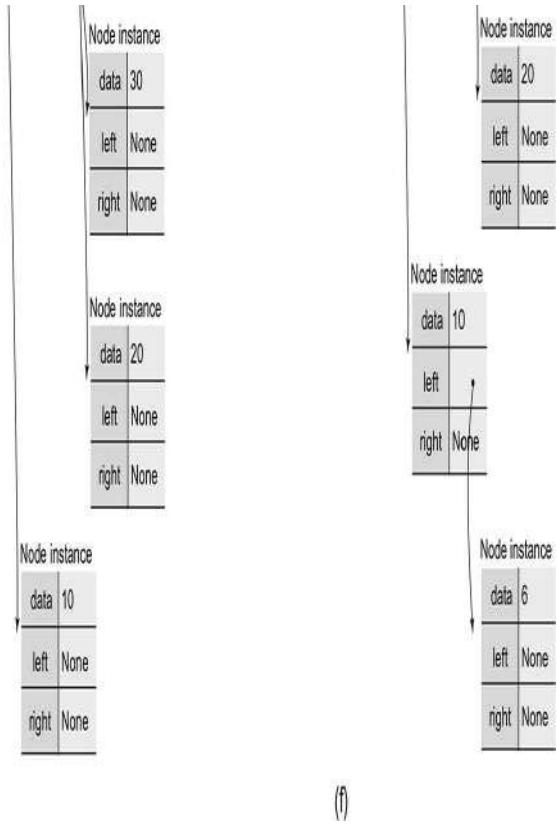


Fig. 15.6 Visualization of building a binary search tree

15.3 TRAVERSAL OF BINARY SEARCH TREES

Often we need to traverse all nodes of a tree, visiting each node exactly once. There are several ways in which the nodes of a binary tree may be visited in an entire scan of the binary tree. We shall describe some commonly used binary tree traversal methods, namely inorder, preorder, and postorder.

15.3.1 Inorder Traversal

Inorder traversal of a non-empty binary tree comprises the following steps:

1. Inorder traversal of the left subtree.
 2. Visit the root.
 3. Inorder traversal of the right subtree.

steps in inorder traversal: left, root, right

In Fig. 15.7, we describe the function `inorder` for inorder traversal of the binary tree. On invoking the function `inorder` with the argument `bst`—binary search tree instance constructed above, the nodes will be visited in the following order of data values:

```
>>> inorder(bst)
```

```
6 10 15 20 23 30
```

```
1 def inorder(root):  
2     if root != None:  
3         inorder(root.left)  
4         print(root.data, end=' ')  
5         inorder(root.right)
```

Fig. 15.7 Function `inorder` (`bTree.py`)

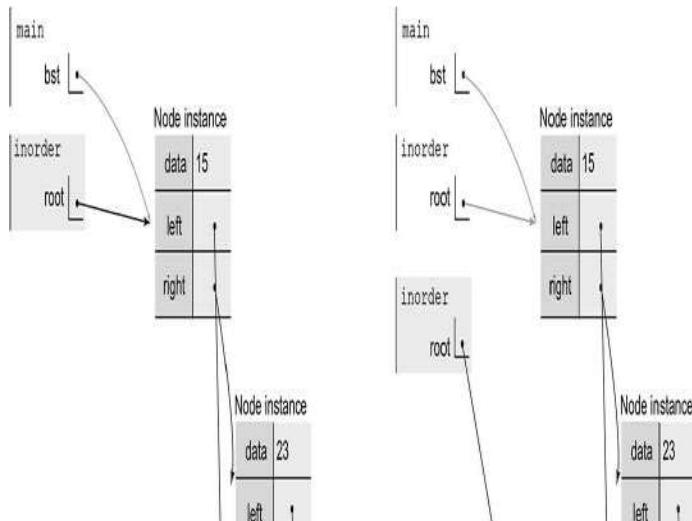
Note that when we traverse the nodes of a binary search tree in inorder, the data values appear in ascending order.

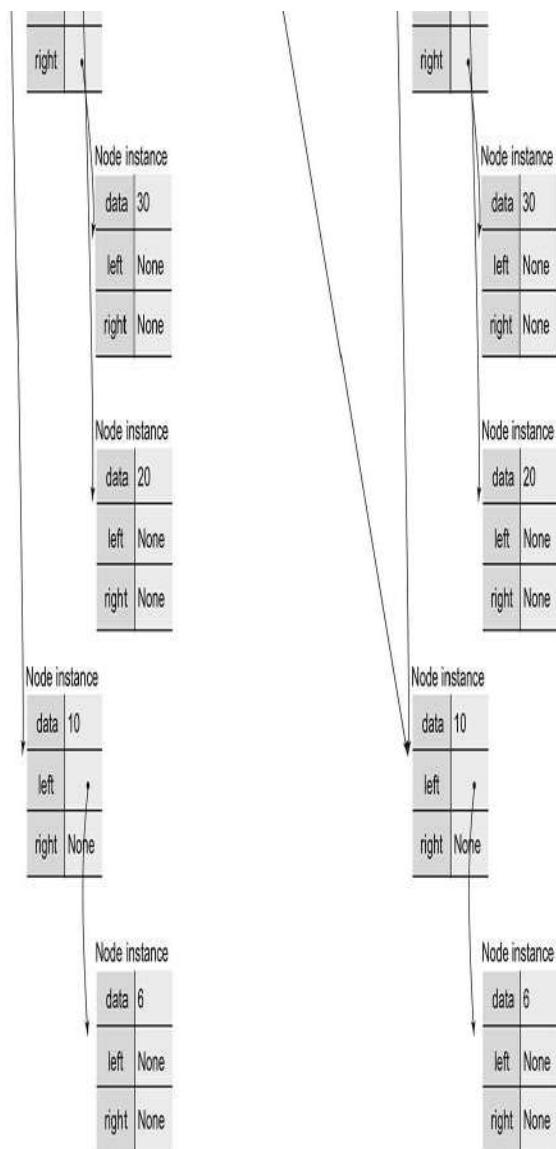
traversing binary search tree in inorder yields nodes in ascending order of their values

Given the `root` node object, the function first traverses the left subtree, then the root, and then the right subtree. We illustrate the execution of function `inorder` using Python Tutor (Fig. 15.8). On invoking the function with argument `bst`, `root` points to the node having `data` (=15) as shown in Fig. 15.8(a). Since `root` is not `None`, on execution of line 3, the function `inorder` is invoked with the left-child of the root node, i.e. node having `data` (=10), which now becomes new root node for this new function call (Fig. 15.8(b)). It

further invokes the function recursively (line 3) with left-child node having `data` (=6). Being a leaf node, the root is not `None` (Fig. 15.8(c)). So the function `inorder` is invoked with the argument `root.left` (= `None`). Now that `root` is `None` (Fig. 15.8(d)), the function terminates, and the control is returned to line 4 of the previous function call (Fig. 15.8(e)), and value 6 is printed. On execution of line 5, the function `inorder` is invoked with the argument `root.right` (= `None`). As the `root` is `None` (Fig. 15.8(f)), the function terminates, and the control returns to the previous call. The traversal of the binary tree having root label 6 is now complete (Fig. 15.8(g)). So, the control returns to the previous call (root labeled 10, line 4). Now the left subtree of this subtree (root labeled 10) has already been traversed (Fig. 15.8(h)). On execution of line 4, `root.data` (=10) is printed. On execution of line 5, the function `inorder` is invoked with the right-child (`None`) as the argument (Fig. 15.8(i)). The right-child of the `root` (labeled 10) being an empty tree, the control immediately returns to the previous call of the function `inorder` (root labeled 10, see Fig. 15.8(j)). As this call to function `inorder` is now complete, the control is transferred to the previous call of the function `inorder` (root labeled 15, Fig. 15.8(k)). Line 4 is now executed and the root label (= 15) is printed. Next, on execution of line 5, the function `inorder` is invoked with the right-child (`root` labeled 23) as the argument.

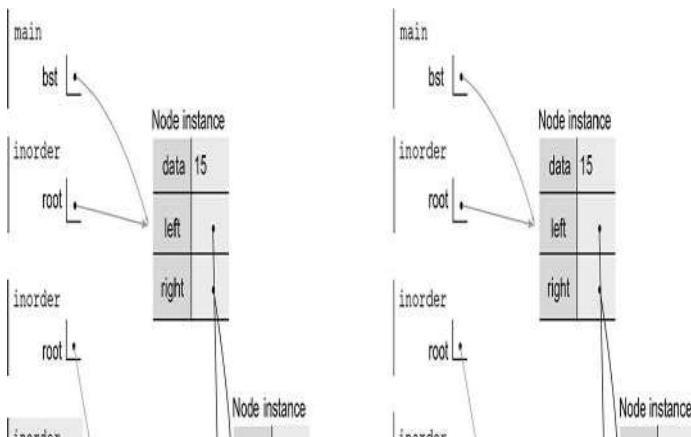
Continuing in this manner, 20, 23, and 30 get printed.

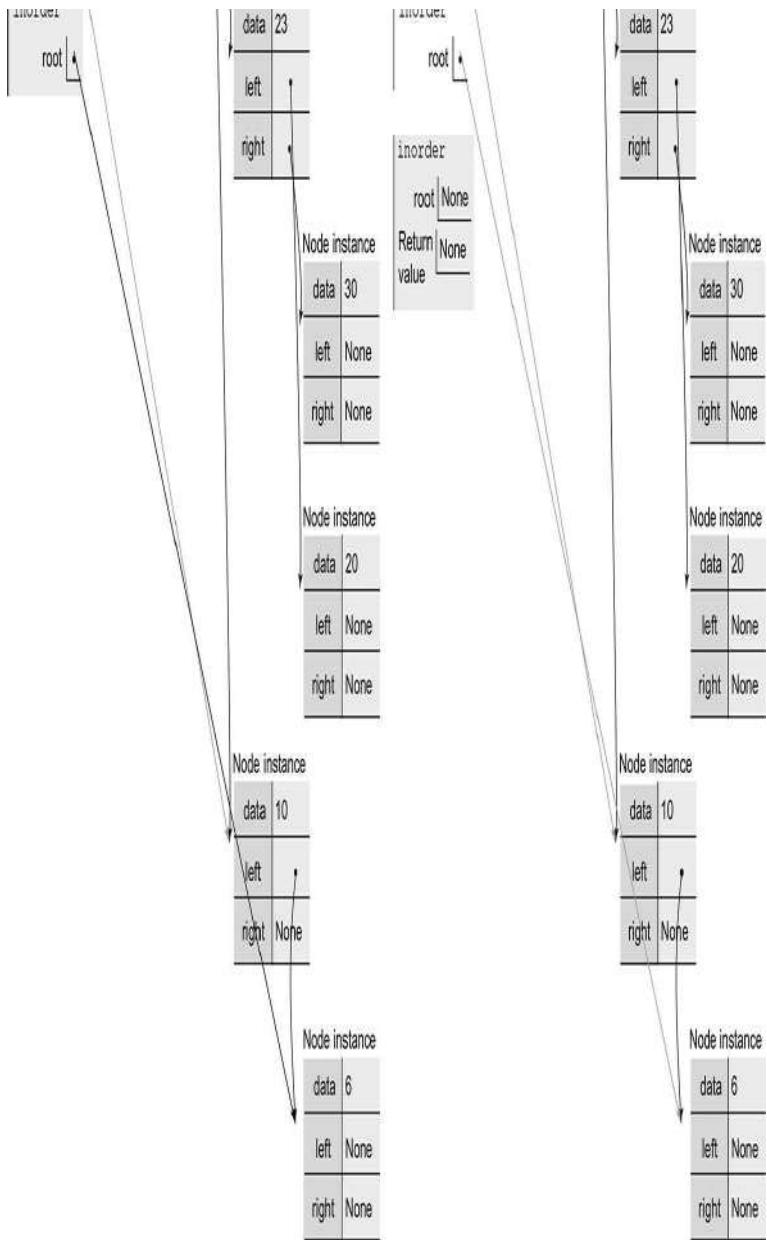




(a)

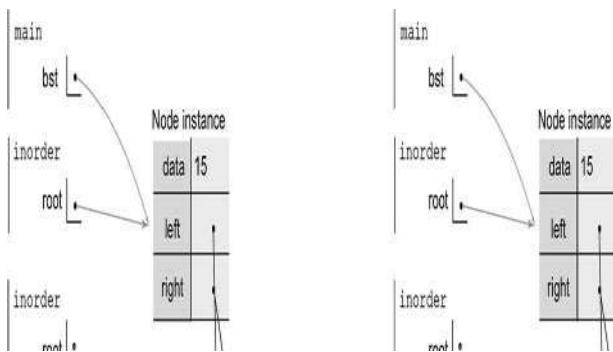
(b)

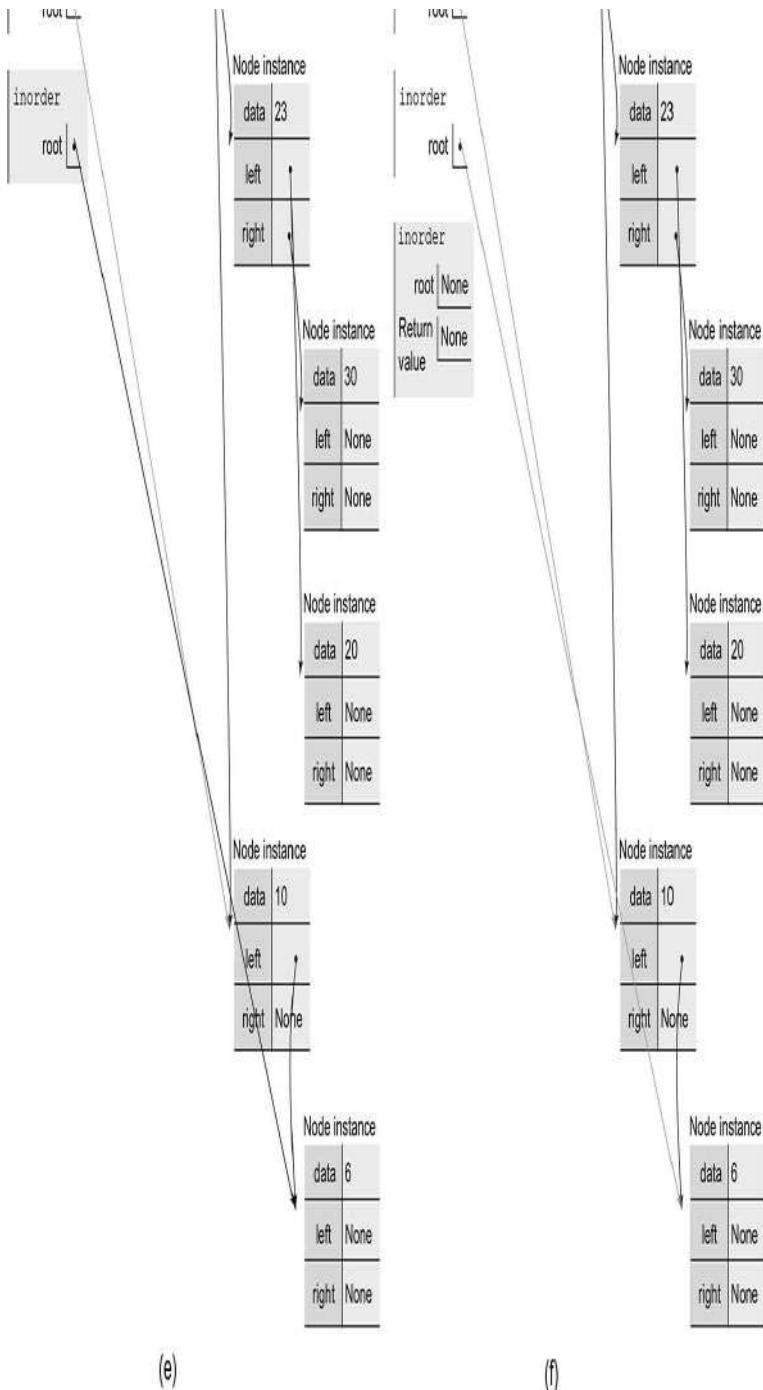




(c)

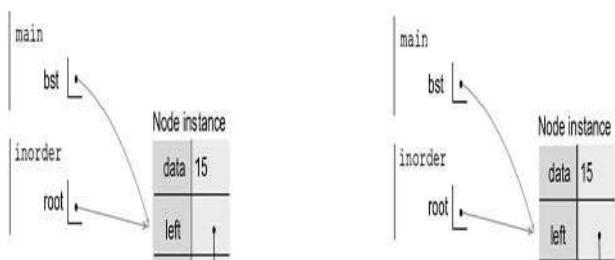
(d)

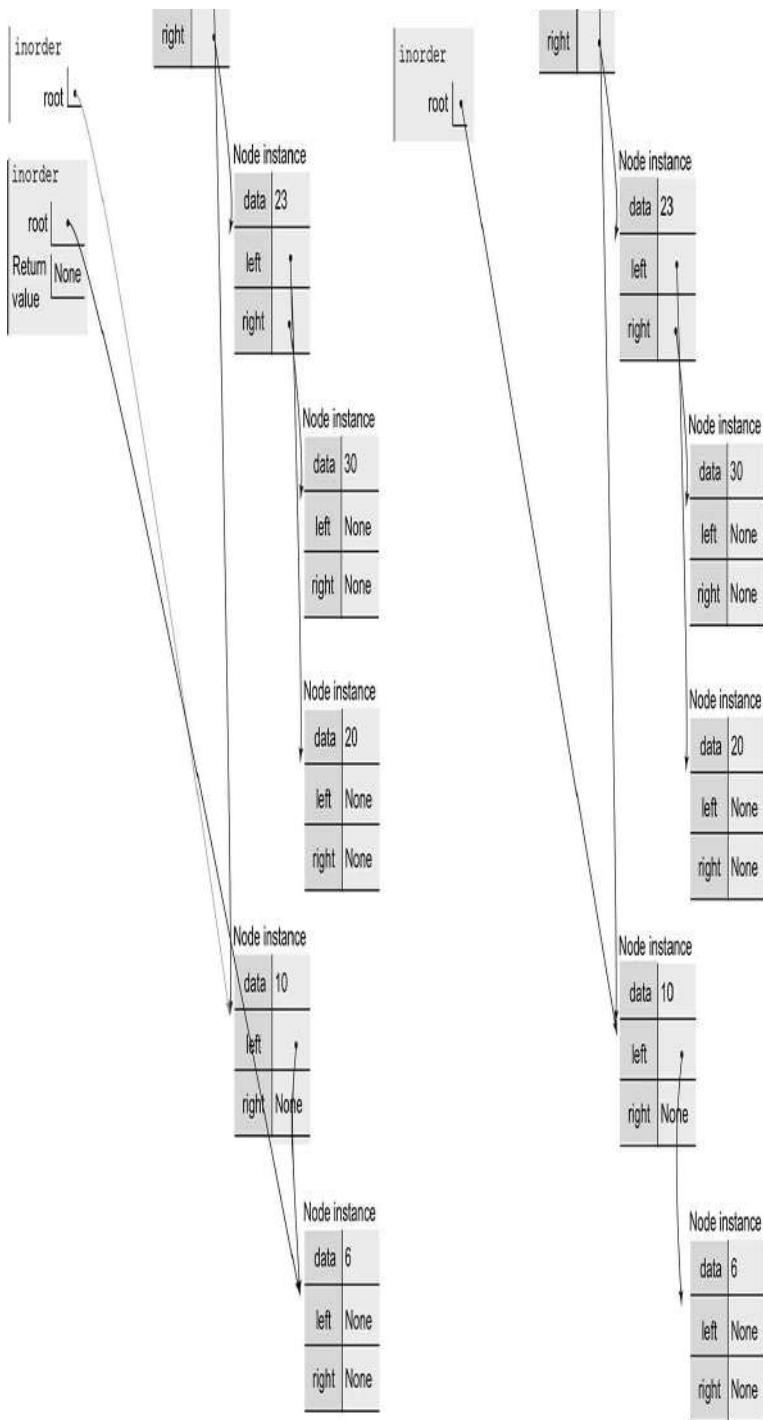




(e)

(f)

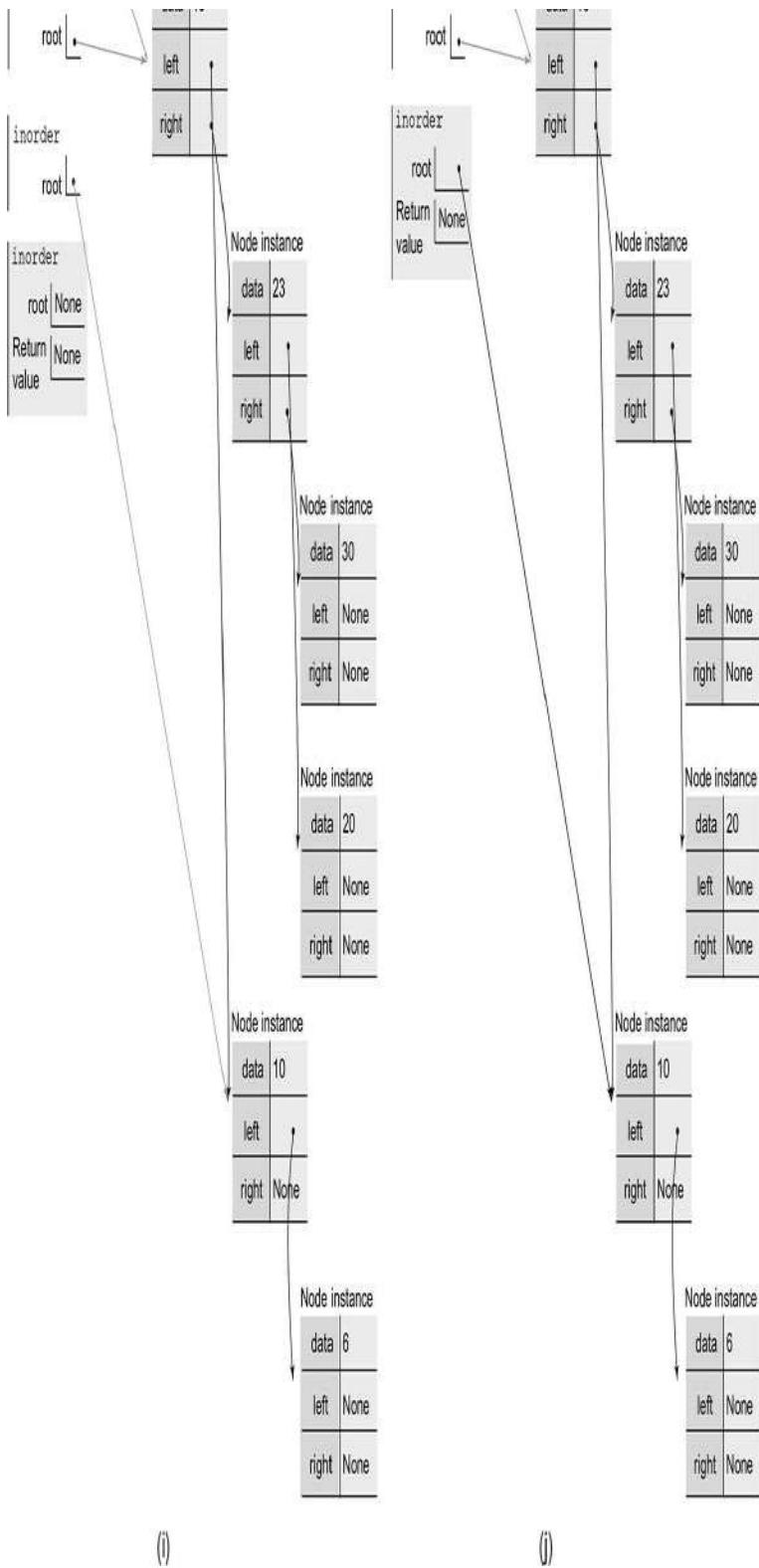


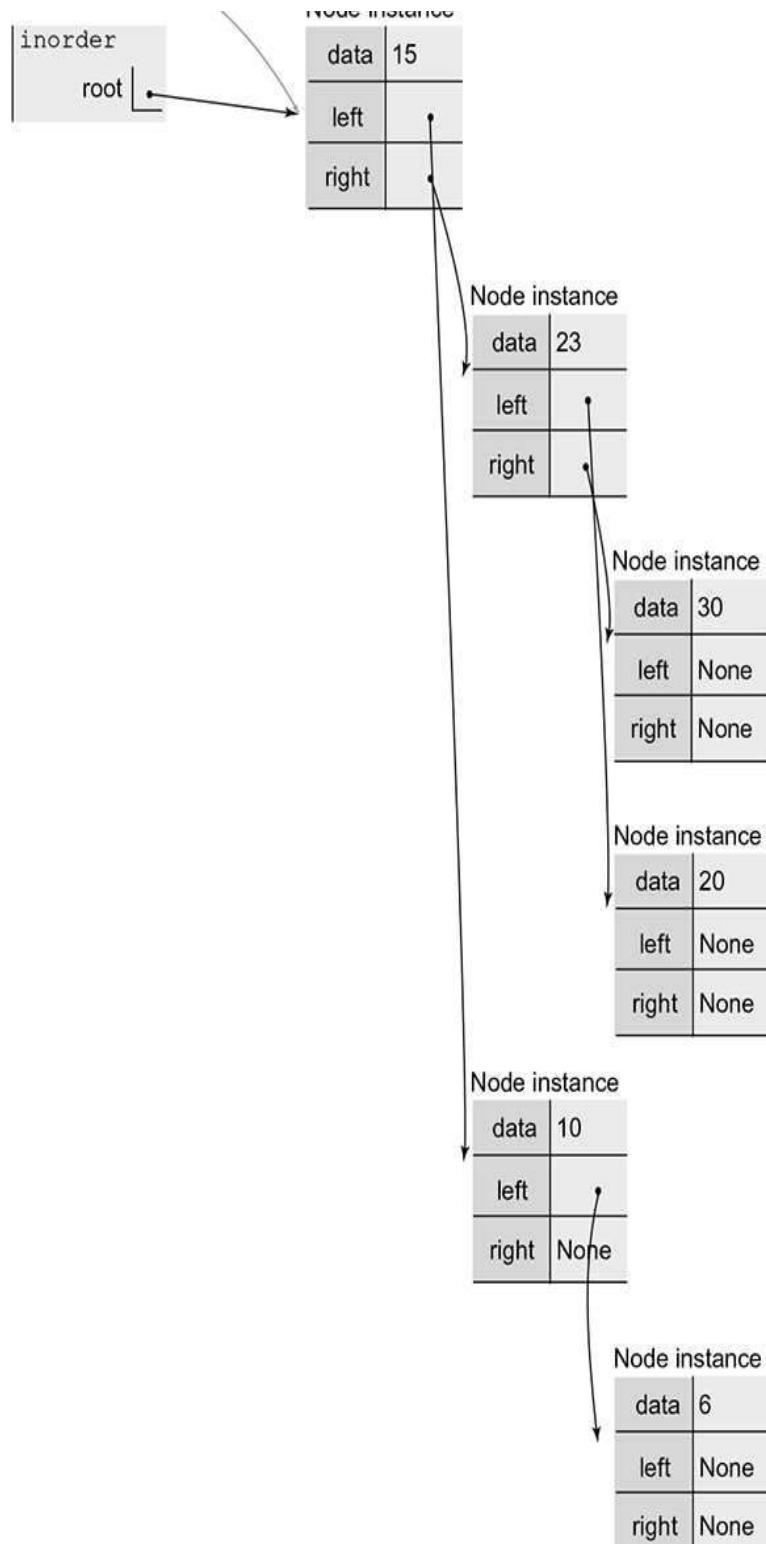


(g)

(h)







(k)

Fig. 15.8 Inorder Traversal

15.3.2 Preorder Traversal

Preorder traversal of a non-empty binary tree comprises the following steps:

1. Visit the root.
2. Preorder traversal of the left subtree.
3. Preorder traversal of the right subtree.

steps in preorder traversal: root, left, right

In Fig. 15.9, we present the function `preorder` for preorder traversal of the binary tree. On invoking the function `preorder` with `bst` as the argument, we get the data values in preorder traversal:

```
1 def preorder(root):
2     if root != None:
3         print(root.data, end=' ')
4         preorder(root.left)
5         preorder(root.right)
```

Fig. 15.9 Function `preorder` (`bTree.py`)

```
>>> preorder(bst)
```

```
15 10 6 23 20 30
```

15.3.3 Postorder Traversal

Postorder traversal of a non-empty binary tree comprises the following steps:

1. Postorder traversal of the left subtree.
2. Postorder traversal of the right subtree.
3. Visit the root.

steps in postorder traversal: left, right, root

The function `postorder` for postorder traversal of the binary tree appears in Fig. 15.10. On invoking the function `postorder` with `bst` as the argument, we get the data values in postorder traversal:

```
>>> postorder(bst)
```

```
6 10 20 30 23 15
```

```
1 def postorder(root):  
2     if root != None:  
3         postorder(root.left)  
4         postorder(root.right)  
5         print(root.data, end=' ')
```

Fig. 15.10 Function `postorder` (`bTree.py`)

15.3.4 Height of a Binary Tree

As mentioned before, the number of edges on the longest path from root to a leaf in the tree defines the height of a tree. To find the height of a binary tree, we note that the height of an empty tree as well that of a binary tree rooted at a leaf node (`root.left == None` and `root.right == None`) is zero, otherwise, the height of a binary tree is computed as one plus the maximum of the heights of the left and the right subtrees. This idea is implemented in the function `height` (Fig. 15.11). Next, we compute the height of the binary tree `bst`:

height of the tree is one more than the maximum of the heights of left and right subtree

height of empty tree or a tree comprising only one node is zero

```
>>> height(bst)
```

```
2
```

```
1 def height(root):  
2     if root == None or (root.left == None and root.right == None):  
3         return 0  
4     else:  
5         return 1 + max(height(root.left), height(root.right))
```

Fig. 15.11 Function `height` (`bTree.py`)

Note that each of the longest paths (15->10->6 or 15->23->20, or 15->23->30) has length 2.

15.4 BUILDING BINARY SEARCH TREE

To build a binary search tree, we must maintain the insertion order. We define a class `binSearchTree` (Fig. 15.12), which includes the method `insertVal` for inserting new nodes in a binary search tree.

a new node should be carefully inserted in a binary search tree, so that it still remains a binary search tree

To build a binary search tree from given data, we proceed as follows: We insert the nodes with new data values one by one taking care that the resulting tree remains a binary search tree. To begin with, let us insert a node with value 15. Initially, the tree is empty (i.e., `self.root == None`). So, execution of line 21 (Fig. 15.12) invokes the constructor for the `Node` class with the argument `value` (=15) to create a node having

data value 15 and None as the value of the left and right links. Now, the name root refers to the tree having the node just created (Fig. 15.13(a)). Fig 15.13(a)–15.13(g) have been generated using the script binarySearchTree (Fig. 15.15) which includes the methods __init__ and insertVal defined in Fig. 15.12.

```
01 from bNode import Node
02 class binSearchTree:
03     def __init__(self):
04         """
05             Objective: To initialize object of class binSearchTree
06             Input Parameter:
07                 self (implicit parameter) - object of type binSearchTree
08             Return Value: None
09         """
10         self.root = None
11
12     def insertVal(self, value):
13         """
14             Objective: To insert a value in the binSearchTree
15             Input Parameters:
16                 self (implicit parameter) - object of type binSearchTree
17                 value - data for the node to be inserted
18             Return Value: None
19         """
20         if self.root == None:
21             self.root = Node(value)
```

```
21     child = None
22 else:
23     child = self.root # Although root has no parent, we
24         call it child
25     parent = None # Root node has no parent node
```

```
25     while child != None:
26         parent = child
27         if value<=child.data:
28             child = child.left
29         else:
30             child = child.right
31         if value<=parent.data:
32             parent.left = Node(value)
33         else:
34             parent.right = Node(value)
35     print("Value Inserted!!")
```

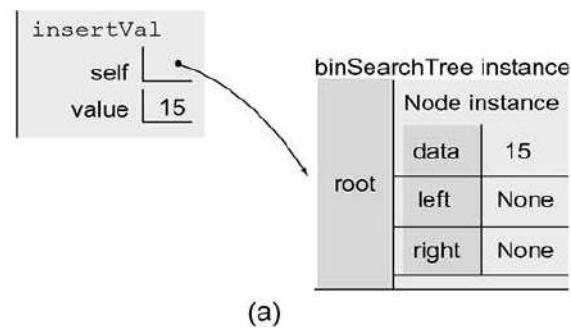
Fig. 15.12 Class binSearchTree (binarySearchTree.py)

To insert a node in a non-empty tree, we have to determine the position of insertion. If `value <= root.data`, then new node will be inserted in the left-subtree of the `root` node, otherwise it will be inserted in the right-subtree of the `root` node. The search for the place where the new node is to be inserted, will proceed until we reach an empty tree (`None`). Thus, the new node is always inserted as a leaf node.

in a binary search tree, the new node is always inserted as a leaf node

To illustrate the process of inserting a node into a non-empty binary search tree, having a given value of the data attribute, we consider the binary search tree shown in Fig. 15.13(b). Suppose, we wish to insert a new node (having data = 12) in the existing binary search tree. The variable child is used to keep track of the node we are currently dealing with. Initially, it is set to point to the root node (line 23, Fig. 15.12). Another variable parent is used to keep track of parent of the child node. Initially, we assign the value None to parent as the root node does not have any parent node (Fig. 15.13(c)). Execution of while loop (lines 25–30, Fig. 15.12) causes traversal of the binary search tree until an empty tree is reached (i.e., child == None) is found. In this process, execution of line 26 (Fig. 15.12) updates parent (= child), and execution of lines 27–30 updates child (= child.left or = child.right), depending on the value of data in the new node to be inserted.

parent and child node: to keep track of nodes while traversing the binary search tree for locating position of insertion for new node



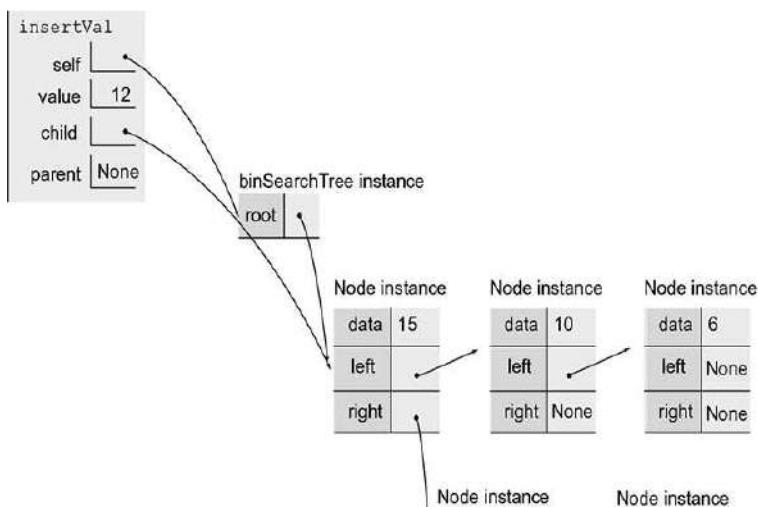
binSearchTree instance

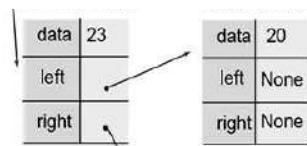
Node instance	
	data 15
left	Node instance
	data 10
	left
	Node instance
	data 6
	left
	None
	right
	None
	right
	None

Node instance	
	data 23
left	Node instance
	data 20
	left
	None
	right
	None

Node instance	
	data 30
right	Node instance
	data None
	left
	None
	right
	None

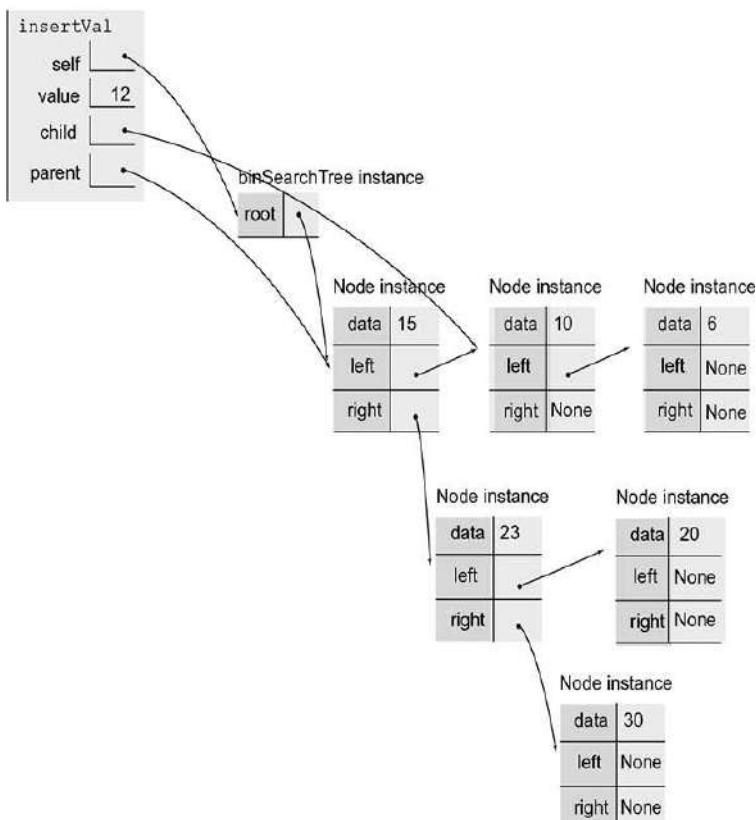
(b)



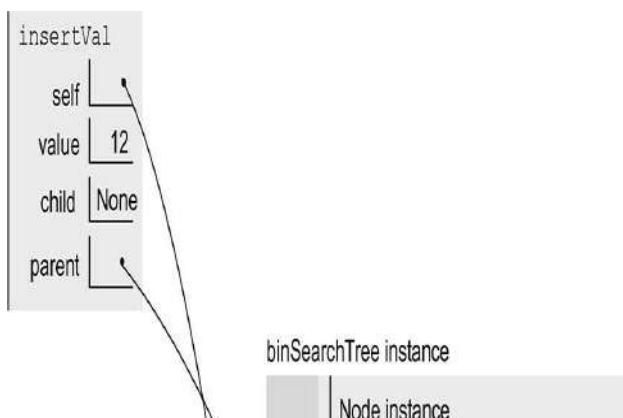


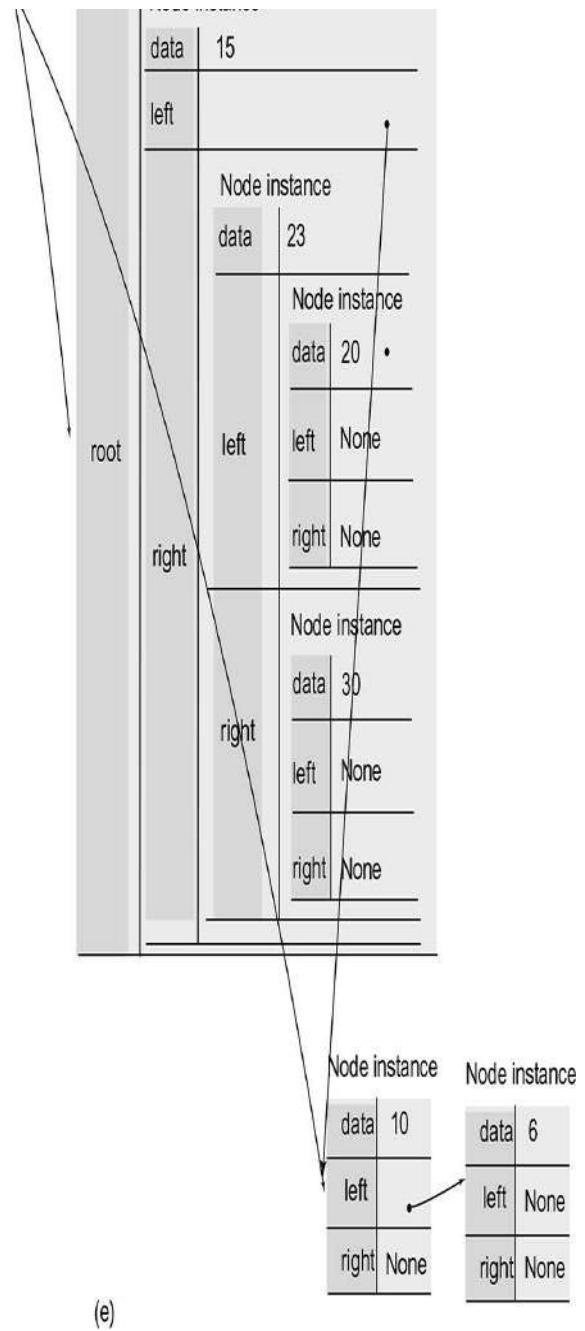
Node instance

(c)

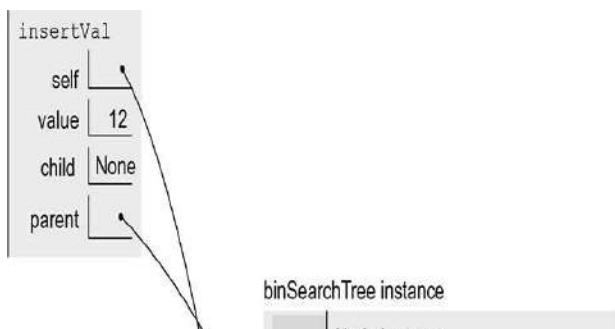


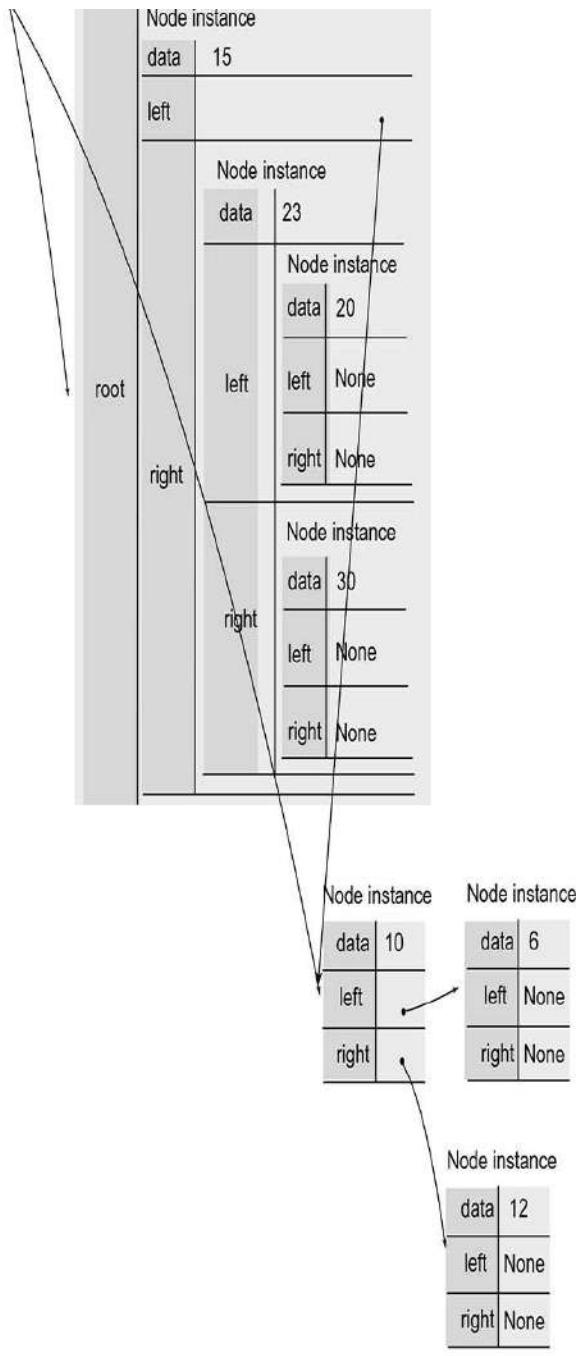
(d)



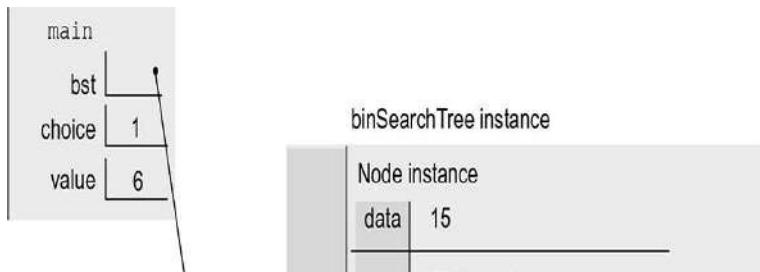


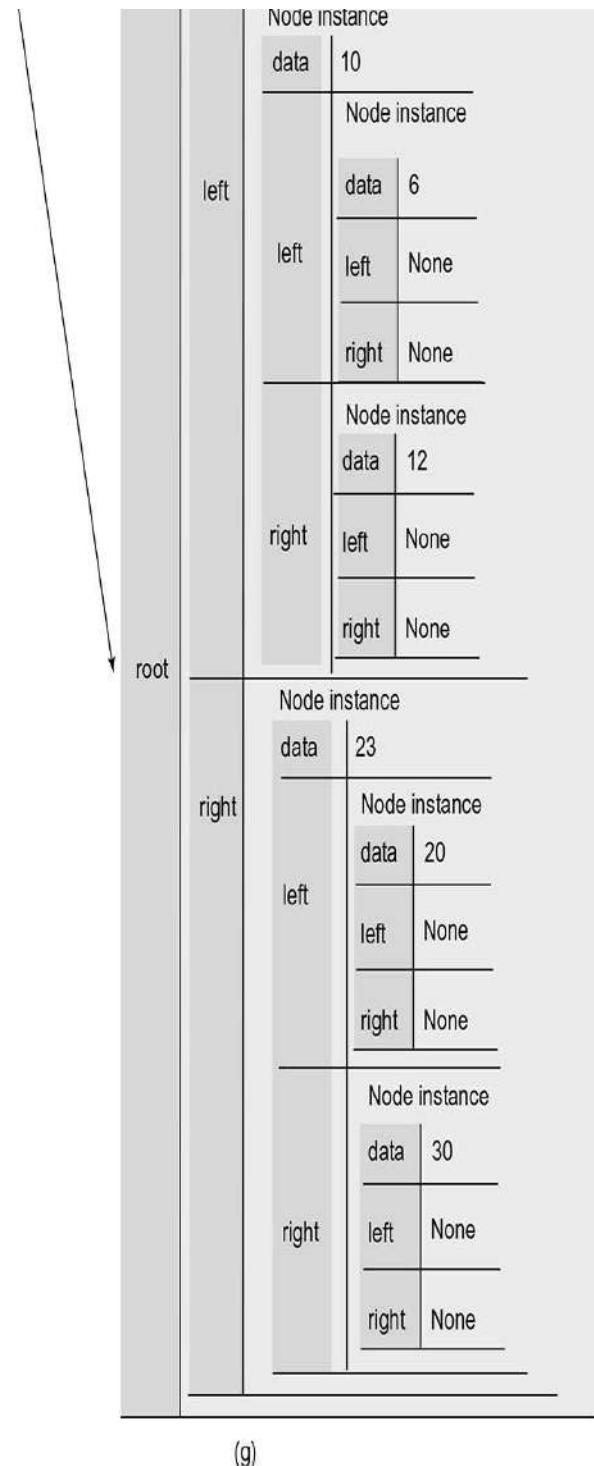
(e)





(f)





(g)

Fig. 15.13 Insertion of value 12 in binary search tree

Since value 12 (data value for the new node to be inserted) is less than the value of child node, i.e. value ($=12 \leq \text{child}.\text{data}$ ($=15$)), child is updated to point to the left node of child node (i.e., `child =`

`child.left`) on execution of line 28 (Fig. 15.12, Fig. 15.13(d)). Since `child` is not `None` (`child.data = 10`), another iteration of the `while` loop begins and execution of line 26 updates `parent` (`parent = child`). Again, value 12 is compared with the data value (10) at `child` node, and since $12 > 10$, `child` is updated to point to the right-child of the `child` node (i.e., `child = child.right`) on execution of line 30 (Fig. 15.12, Fig. 15.13(e)). As `child.right` is `None`, `child` becomes `None` on execution of line 30 (Fig. 15.12). This leads to termination of the `while` loop.

Thus, we have found the place where the new node is to be inserted. Now, the condition `value (=12) \leq parent.data (=10)` is `False`. So, execution of line 34 (Fig. 15.12) creates a node instance by invoking the constructor for the `Node` class with the argument `value (=12)`. As usual, the constructor assigns the value `None` to `left` and `right` links of the node being created. The parent–child relationship is established by assigning the new node (`data =12`) to `parent.right` (line 34 (Fig. 15.12), Fig. 15.13(f)). Finally, in Fig. 15.13(g), we see the compact representation of binary search tree after insertion of the node with value 12.

compare the value of new node to be inserted as child node with parent node and assign it as left or right child accordingly

Alternatively, we may define the method `insertVal` using recursion (Fig. 15.14). Since the recursive function would require passing `root` as an argument at the point of call, it leads to violation of abstraction principle. So, we use the wrapper function `insert` which calls nested function `insertVal` with the argument `root` (line 24, Fig. 15.14).

use of a wrapper function

In the script `binarySearchTree` ([Fig. 15.15](#)), we present complete code for the binary search tree. Note that `inorderTraversal`, `postorderTraversal`, `preorderTraversal`, and `treeHeight` are wrapper methods used for nested recursive functions `inorder`, `postorder`, `preorder`, and `height`.

```
01 def insert (self,value):  
02     '''  
03         Objective: To serve as wrapper function for insertVal  
04         Input Parameter: self (implicit parameter) - object of type  
05                         binSearchTree,  
06                         value - value to be inserted  
07         Return Value: None  
08     '''  
09     def insertVal(root, value):
```

```
10         '''  
11         Objective: To insert a value in the binSearchTree  
12         Input Parameters:  
13             root - object of type Node  
14             value - Value to be inserted  
15         Return Value: object of type Node  
16         '''  
17         if root == None:  
18             root = Node(value)  
19         elif value <= root.data:  
20             root.left = insertVal(root.left, value)  
21         else:  
22             root.right = insertVal(root.right, value)  
23         return root  
24         self.root = insertVal(self.root, value)
```

Fig. 15.14 Recursive method recurInsertVal

```
001 from bNode import Node
002 class binSearchTree:
003     def __init__(self):
004         """
005             Objective: To initialize object of class binSearchTree
006             Input Parameter:
007                 self (implicit parameter) - object of type binSearchTree
008             Return Value: None
009         """
010         self.root = None
011
012     def insertVal(self, value):
013         """
014             Objective: To insert a value in the binSearchTree
015             Input Parameters:
016                 self (implicit parameter) - object of type binSearchTree
017                 value - data for the node to be inserted
018             Return Value: None
019         """
020         if self.root == None:
021             self.root = Node(value)
022         else:
023             child = self.root
024             parent = None
```

025 while child != None:

```
026               parent = child
027                if value<=child.data:
028                child = child.left
029                else:
030                child = child.right
031                if value<=parent.data:
032                parent.left = Node(value)
033                else:
034                parent.right = Node(value)
035                print('Value Inserted!!!')
036
037       def inorderTraversal(self):
038               """
039               Objective: To serve as wrapper function for the method
040               inorder
041               Input Parameter: self (implicit parameter) - object of
042               type binSearchTree
043               Return Value: None
044               """
045       def inorder(root):
046               """
047               Objective: To carry out inorder traversal of
048               binSearchTree
049               Input Parameter: root - object of type Node
050               Return Value:None
051               """
052        if root != None:
053                inorder(root.left)
054                print(root.data, end=' ')
055                inorder(root.right)
056        inorder(self.root)
057
058       def preorderTraversal(self):
059               """
060               Objective: To serve as wrapper function for the method
061               preorder
```

```
```
062 Input Parameter: self (implicit parameter) - object of
063 type binSearchTree
064 Return Value: None
```

```
065 """
066 def preorder(root):
067 """
068 Objective: To carry out preorder traversal of
069 binSearchTree
070 Input Parameter: root - object of type Node
071 Return Value:None
072 """
073 if root != None:
074 print(root.data, end=' ')
075 preorder(root.left)
076 preorder(root.right)
077 preorder(self.root)
078
079 def postorderTraversal(self):
080 """
081 Objective: To serve as wrapper function for the method
082 postorder
083 Input Parameter: self (implicit parameter) - object of
084 type binSearchTree
085 Return Value: None
086 """
087 def postorder(root):
088 """
089 Objective: To carry out postorder traversal of
090 binSearchTree
091 Input Parameter: root - object of type Node
092 Return Value: None
093 """
094 if root != None:
095 postorder(root.left)
096 postorder(root.right)
097 print(root.data, end=' ')
098 postorder(self.root)
099
100 def treeHeight(self):
101 """
```

```
101
102 Objective: To serve as wrapper function for height
103 Input Parameter: self (implicit parameter) - object of
104 type binSearchTree
105 Return Value: numeric
106 """
```

```
107 def height(root):
108 """
109 Objective: To find height of the tree
110 Input Parameter: root - object of type Node
111 Return Value: numeric
112 """
113 if root == None or (root.left == None and root.right
114 == None):
115 return 0
116 else:
117 return max(height(root.left), height(root.right))
118 + 1
119 return height(self.root)
120
121 def main():
122 """
123 Objective: To provide binary search tree functionality
124 Input Parameter: None
125 Return Value: None
126 """
127 bst = binSearchTree()
128 while 1:
129 choice = int(input('\n1: Insert a value \n2: Inorder\
130 Traversal\n3: Preorder Traversal\n4: Postorder\
131 Traversal\n5: Height of the tree\n6:Quit\n'))
132 if choice == 1:
133 value = eval(input('Enter value to be inserted: '))
134 bst.insertVal(value)
135 elif choice == 2:
136 bst.inorderTraversal()
137 elif choice == 3:
138 bst.preorderTraversal()
```

```

139 elif choice == 5:
140 print('Height of the tree:', bst.treeHeight())
141 elif choice == 6:
142 break
143
144 if __name__ == '__main__':
145 main()

```

**Fig. 15.15** Class BinSearchTree (binarySearchTree.py)

#### SUMMARY

1. A tree is a hierarchical structure. It is a collection of elements called nodes along with a relation called parenthood that defines a hierarchical structure on nodes.
2. The root node is a node which does not have any parent node.
3. A leaf node is a node which has no children.
4. A node which is not a leaf node is called internal node or a non-leaf node.
5. If  $n_1, n_2, \dots, n_k$  is a sequence of nodes in a tree such that  $n_i$  is the parent of  $n_{i+1}$  for  $1 \leq i \leq k-1$ , the sequence is called a path of length  $k-1$  from  $n_1$  to  $n_k$ .
6. The number of edges on the longest path from the root to a leaf in a tree defines the height of the tree.
7. The level of a node is the number of edges on the path from the root to the node.
8.  $n_1$  is an ancestor of node  $n_2$  if  $n_1$  is either parent of  $n_2$  or parent of an ancestor of node  $n_2$ .
9.  $n_1$  is a descendant of node  $n_2$  if  $n_1$  is either child of  $n_2$  or child of a descendant of node  $n_2$ .
10. By definition, a node is an ancestor as well as descendant of itself.
11. Children of the same node are called siblings.
12. A binary tree is a particular type of tree whose nodes have two or fewer children. In a binary tree, the child nodes of a node are called left-child and right-child. A binary tree may be an empty tree, comprising no nodes at all, or may comprise a root node along with other nodes which may be organized as a left binary subtree and a right binary subtree of the root node.
13. Strictly binary tree is a binary tree in which every non-leaf node has non-empty left and right subtree. Thus, each node in a strictly binary tree has either two children or zero.
14. A complete binary tree is a binary tree that is full on all levels except the lowest level which is filled in from left to right.
15. A binary tree is called a search tree if for any node  $n$  of the binary tree, every node in its left subtree has smaller or equal value as that of the value of node  $n$ , and every node in its right subtree will have value larger than the value of node  $n$ .

16. Inorder traversal of non-empty binary tree comprises the following steps:
1. Inorder traversal of the left subtree.
  2. Visit the root.
  3. Inorder traversal of the right subtree
17. Preorder traversal of non-empty binary tree comprises the following steps:
1. Visit the root.
  2. Preorder traversal of the left subtree.
  3. Preorder traversal of the right subtree.
18. Postorder traversal of non-empty binary tree comprises the following steps:
1. Postorder traversal of the left subtree.
  2. Postorder traversal of the right subtree.
  3. Visit the root.
19. The height of an empty tree as well that of a binary tree rooted at a leaf node (`root.left == None` and `root.right == None`), otherwise, the height of a binary tree is computed as one plus the maximum of the heights of the left and the right subtrees.

#### EXERCISES

1. Given inorder and preorder traversal, write a method that constructs a binary tree.
2. Write a method `delVal` for class `binSearchTree` that deletes a node having a given value from the binary search tree.
3. Write a method `levelOrderTraversal` for the class `binSearchTree` that traverses binary search tree level by level.
4. Write a method `count` for the class `binSearchTree` that counts and returns the number of nodes in a binary search tree.
5. Write a method `countLeaves` for the class `binSearchTree` that returns the number of leaf nodes in a binary search tree.
6. Write a method `countNonLeaves` for the class `binSearchTree` that returns the number of non-leaf nodes in a binary search tree.
7. Write a method `delLeaves` for the class `binSearchTree` that deletes all the nodes that are currently the leaf nodes of a binary search tree.
8. Write a method `mirrorBST` for the class `binSearchTree` that creates and returns mirror image of a binary search tree.
9. Write a method `copyBST` for the class `binSearchTree` that creates and returns a copy of a binary search tree.
10. Write a method `minVal` for the class `binSearchTree` that finds and returns the minimum value in a binary search tree.
11. Write a method `ancestor` for the class `binSearchTree` that finds and returns the first common ancestor of a pair of nodes in the binary search tree.

# CHAPTER 16

## MORE ON RECURSION

### CHAPTER OUTLINE

- [16.1 Pattern Within a Pattern](#)
- [16.2 Generalized Eight Queens Problem](#)
- [16.3 Knight's Tour Problem](#)
- [16.4 Stable Marriage Problem](#)
- [16.5 Fractal \(Hilbert Curve and Sierpinski Triangle\)](#)
- [16.6 Suduko](#)
- [16.7 Guidelines on Using Recursion](#)

In Chapter 8, we discussed recursion and its applications in detail. In this chapter, we focus on more advanced problems that can be solved using recursion.

### 16.1 PATTERN WITHIN A PATTERN

In this section, we will use recursion to print a picture comprising several squares within each other. For this purpose, we need to import a graphical package, and `matplotlib` is one such choice (discussed in detail in the next chapter). It contains methods for 2D graphics that would be required in this section. Let us first plot a square on the graph. The `plot` function takes as arguments two lists, say, `x` and `y`, comprising `x` and `y` coordinates, respectively of a sequence of points and connects them using line segments. Thus, a square of side `size` may be constructed by joining the points `(0, 0), (size, 0), (size, size), (0, size), (0, 0)` in sequence using line segments. For this purpose, the square may be defined using two lists, say, `x` and `y` as follows:

`x`: list of x-coordinates of corners of a square in a sequence

```
y: list of y-coordinates of corners of a square in a sequence
```

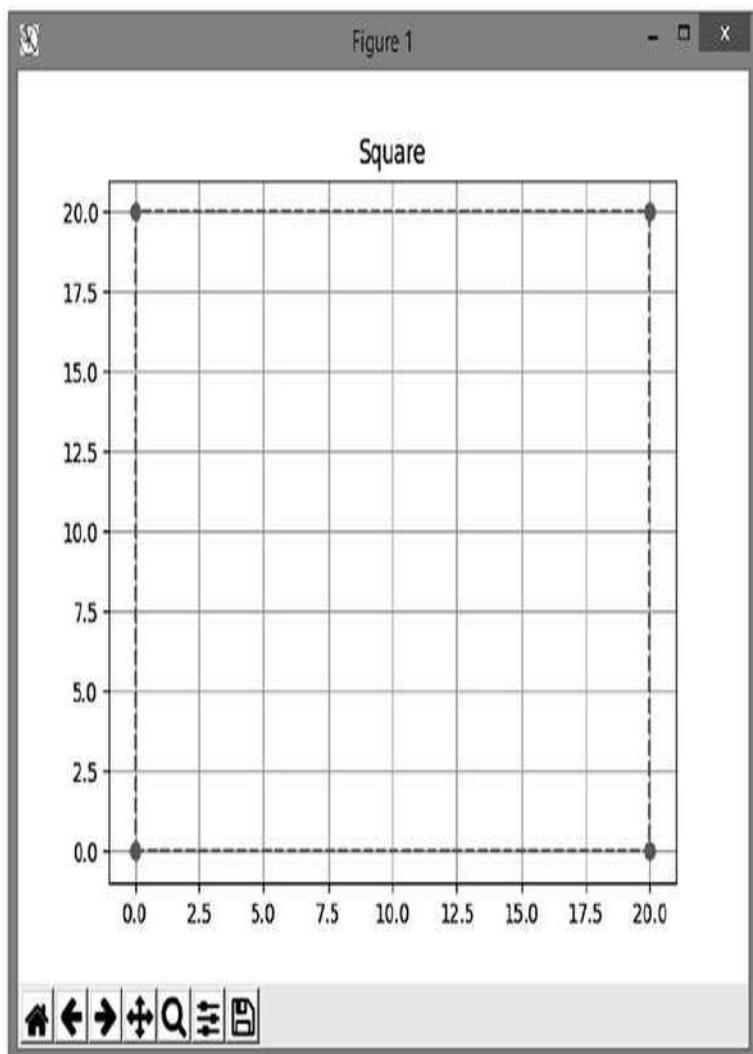
```
x = [0, size, size, 0, 0]
y = [0, 0, size, size, 0]
```

The script `square` (Fig. 16.1) creates a square of the desired size. When we enter 20 as the size of the square, the square would appear as shown in Fig. 16.2.

```
01 import matplotlib.pyplot as plt
02
03 def square(x, y):
04 """
05 Objective: To plot a square
06 Input Parameters: x, y - lists of x coordinates and y
07 coordinates respectively
08 Return Value: None
09 """
10 plt.plot(x, y, 'ro--')
11
12 def main():
13 """
14 Objective: To plot a square based on user input
15 Input Parameter: None
16 Return Value: None
17 """
18 size = int(input('Enter size of the square: '))
19 x = [0, size, size, 0, 0]
20 y = [0, 0, size, size, 0]
21
```

```
1 square(x, y)
2 plt.title('Square')
3 plt.axis([min(x)-1, max(x)+1, min(y)-1, max(y)+1])
4 plt.grid()
5 plt.show()
6
7 if __name__ == '__main__':
8 main()
```

**Fig. 16.1** Program to plot a square (`square.py`)



**Fig. 16.2** Square

Next, let us print squares within squares. Evidently, recursion would be our natural choice to design such a function. However, before we develop the recursive function, we need to do some housekeeping. In the script `squareRecur`, we develop a separate function `squareWrapper` to do the housekeeping work related to the main function `square` (Fig. 16.3). Such a function is called a wrapper function. It accepts the size of the square as a parameter, initializes the vectors `x` and `y`, invokes the recursive function `square`, prints the title of the figure, fixes the axes, invokes the function `grid()` to plot the grid, and finally invokes the method `show()` to show the result. For every call to the function `square`, we decrease the size of the square by two, thus, causing the square to shrink inwards. Finally, we terminate the function `square` when the difference between adjacent coordinates is less than one. When we enter 20 as the size of the square, the square would appear as shown in Fig. 16.4.

`squareWrapper`: wrapper function for housekeeping work

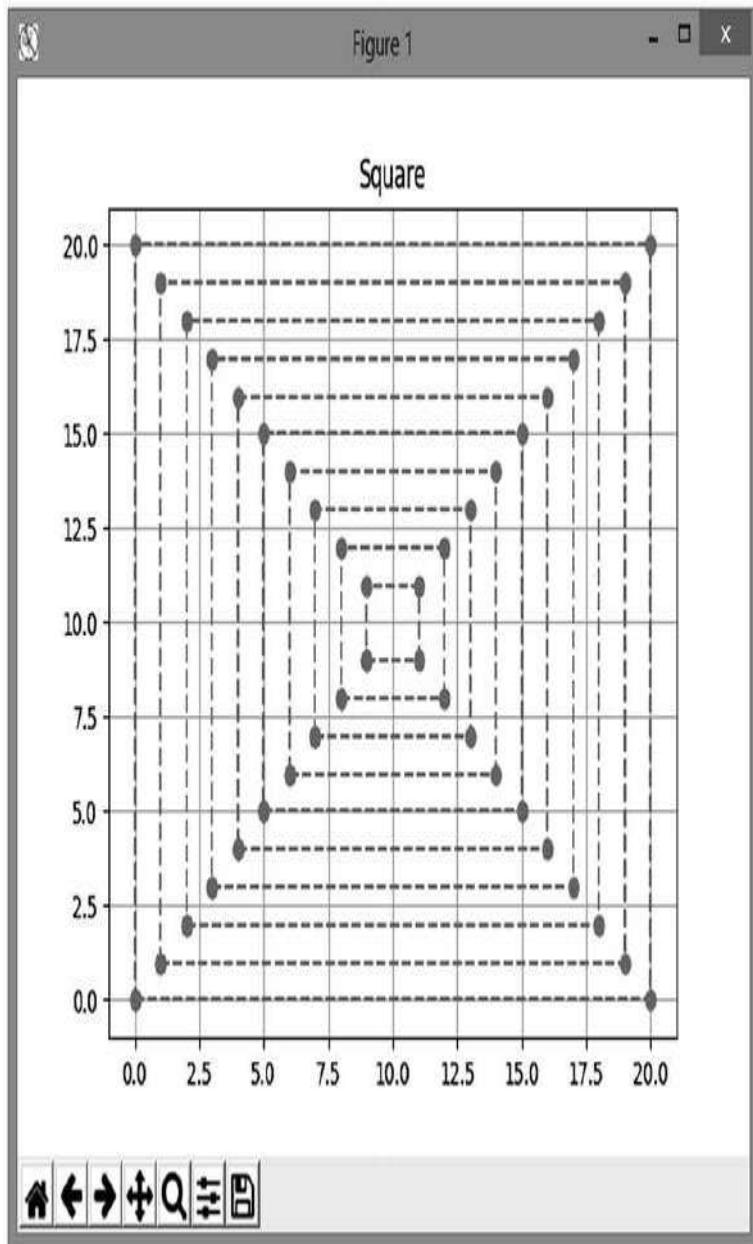
in every recursive call to the function `square`, decrease square size by two



```
01 import matplotlib.pyplot as plt
02
03 def square(x, y):
04 """
05 Objective: To plot a square
06 Input Parameters: x, y - list of coordinates
07 Return Value: None
08 """
09 if (x[1]-x[0]>=1):
10 plt.plot(x, y, 'ro--')
11 x = [x[0]+1, x[1]-1, x[2]-1, x[3]+1, x[0]+1]
12 y = [y[0]+1, y[1]+1, y[2]-1, y[3]-1, y[0]+1]
13 square(x, y)
14
15 def squareWrapper(size):
16 """
17 Objective: To define wrapper function for function square
18 Input Parameter: size - numeric value
19 Return Value: None
20 """
21 x = [0, size, size, 0, 0]
22 y = [0, 0, size, size, 0]
23 square(x, y)
24 plt.title('Square')
25 plt.axis([min(x)-1, max(x)+1, min(y)-1, max(y)+1])
26 plt.grid()
27 plt.show()
28
29
```

```
30 def main():
31 """
32 Objective: To print squares within squares based on user
33 input
34 Input Parameter: None
35 Return Value: None
36 """
37 size = int(input('Enter size of outer square: '))
38 squareWrapper(size)
39
40 if __name__ == '__main__':
41 main()
```

**Fig. 16.3** Program to recursively plot square within a square  
(squareRecur.py)



**Fig. 16.4** Squares within squares

#### 16.2 GENERALIZED EIGHT QUEENS PROBLEM

The eight queens problem is the problem of placing eight queens on the chessboard so that none can attack each other. A queen is attackable if another queen lies in the same row, same column, or same diagonal (either direction). In this problem, we need to find a solution in the form of a placement of queens on the  $8 \times 8$  chessboard so that each of the eight queens is non-

attackable (i.e. safe). We may begin as follows: initially, place a queen at any place in the first row, and then successively choose the position of the next queen on the subsequent rows in such a manner that chosen square is unguarded, i.e. not attackable by any other queen. As we follow the procedure just described, it is possible that we may get stuck on the way, as there may be a row for which no position is a safe position. If this happens, we backtrack one or more steps as required, to try another alternative.

place eight queens on the chessboard so that no one can attack any one

unguarded chessboard square: not attackable by any other queen

In this section, we develop a general solution to the problem mentioned above that works not just for an  $8 \times 8$  chessboard, but for an arbitrary board of size  $n \times n$ . In the script `nQueens` (Fig. 16.5), we define the class `Queens` that contains several data members and methods for keeping track of the current chessboard configuration, and for adding or removing a queen. The data member `board` defines a chessboard of size `boardSize` (lines 20 and 21). All entries of the board are initialized as `False`, to indicate that initially all positions are unguarded. The recursive function `solveFrom` finds solutions to the  $n$ -queens problem. The function receives an object `configuration` of class `Queens`, and approaches the problem in the following manner:

class `Queens`: used to modify and keep track of the current chessboard configuration

```
n = boardSize
```

```

if Queens configuration already contains n queens
 print configuration
else
 for every chessboard square p that is unguarded
 update the configuration by adding a queen on
 square p
 solveFrom(configuration)
 restore the configuration to its initial state
 by removing the queen from square p

```

exploring solutions to n-queens problem

The function `solveFrom` extends the current configuration to an n-queens configuration (of course, only if the present configuration is extendable to an n-queens configuration). As mentioned before, class `Queens` supports the following methods for solving the *n*-queens problem:

methods of class `Queens`

1. `__init__`

This method creates an instance of class `Queens` and initializes the data members `boardSize`, `count` (number of queens), and `board`.

2. `getBoardSize`

This method returns the size of the chess board.

3. `__str__`

This method returns string representation of object of type `Queens`.

4. `unguarded`

This method returns `True`, if the current chessboard square is unguarded, and `False` otherwise. It checks whether a queen is guarding the current square. As all the rows of the board having row number greater than the row number of the current square are vacant (not yet occupied by a queen), we only need to check whether a queen in the rows above the current square is guarding the current square. This is accomplished by checking upper parts of the column, left diagonal, and

the right diagonal for the current square. This approach is summarized below:

```
check: is the current chessboard square unguarded?

if the current square is unguarded in the
upper part of the column, upper-left
diagonal, and upper-right diagonal
 return True
else
 return False
```

5. add

This method adds the queen at the square position `row = count` and `column = col`. It also updates the current number of queens placed on the chessboard by incrementing the `count` by 1.

6. remove

This method removes the queen from the square position `row = count` and `column = col`. It also updates the current number of queens placed on the chessboard by decrementing the `count` by 1.

7. `isSolved`

This method returns `True`, if the number of queens already placed on the board equals `boardSize`, and `False` otherwise.

The complete program is given in Fig. 16.5.

```
001 class Queens:
002 ''''
003 Objective: To solve n-queens problem
004 ''''
005
006 def __init__(self, size):
007 ''''
008 Objective: To initialize object of class Queens
009 Input Parameters:
010 self (implicit parameter) - object of type Queens,
011 size - numeric
012 Return Value: None
013 ''''
014 #Dimension of board = Max. no. of Queens
015 self.boardSize = size
016 # the count for queen as well row no. on the chessboard
017 self.count = 0
018 # The Queens object is set up as an empty configuration
```

```
019 # on a chessboard with boardSize x boardSize squares.
020 self.board= [[False for i in range(self.boardSize)] \
021 for j in range(self.boardSize)]
022
023 def getBoardSize(self):
024 ''''
025 Objective: To determine size of chess board
026 Input Parameter:
027 self (implicit parameter) - object of type Queens
028 Return Value: boardSize - numeric
```

```

029 """
030 return self.boardSize
031
032 def __str__(self):
033 """
034 Objective: To return string representation of object of
035 type Queens
036 Input Parameter: self (implicit parameter)- object of
037 type Queens
038 Return Value: string
039 """
040 result = ''
041 for i in self.board:
042 for j in i:
043 if j == False:
044 result += '-+' + ' '
045 else:
046 result += 'Q' + '+' + ' '
047 result += '\n'
048 return result
049
050 def unguarded(self, col):
051 """
052 Objective: To determine whether the current chessboard
053 square is safe i.e. not guarded by any other queen
054 Input Parameters:
055 self (implicit parameter) - object of type Queens
056 col - numeric
057 Return Value: True or False - boolean
058 """
059 ok = True # turns False if we find a queen in column
060 or diagonal
060 i=0

```

```

061 # Check upper part of column
062 while(ok and i<self.count):
063 ok = not(self.board[i][col])
064 i+=1
065 i = 1
066 # Check upper-left Diagonal
067 while (ok and self.count-i >=0 and col-i >=0):

```

```

068 ok = not(self.board[self.count-i][col-i])
069 i += 1
070 i = 1
071 # Check upper-right diagonal
072 while(ok and self.count-i>=0 and col +i <self.boardSize):
073 ok = not(self.board[self.count-i][col+i])
074 i+=1
075 return ok
076
077
078 def add(self, col):
079 """
080 Objective: To add the queen at the given column no.
081 Input Parameters:
082 self (implicit parameter) - object of type Queens
083 col - numeric
084 Return Value: None
085 """
086 """
087 PreCondition - The square in the first unoccupied row (row
088 count) and column col is not guarded by any queen
089 """
090 self.board[self.count][col] = True
091 self.count += 1
092
093 def remove(self, col):
094 """
095 Objective: To remove the queen from the given column no.
096 Input Parameters:
097 self (implicit parameter) - object of type Queens
098 col - numeric
099 Return Value: None
100 """
101 """
102 PreCondition -There is a queen in the square in row count

```

```

103 - 1 and column col
104 """
105 self.board[self.count-1][col] = False
106 self.count -= 1
107

```

```
108 def isSolved(self):
109 """
110 Objective: To determine whether the current chessboard
111 configuration is fully or partially completed
112 Input Parameter:
113 self (implicit parameter) - object of type Queens
114 Return Value: True or False - boolean
115 """
116 if self.count == self.boardSize:
117 return True
118 else:
119 return False
120
121 solNum = 0
122 def solveFrom(configuration):
123 """
124 Objective: To find solutions to n-Queens problem
125 Input Parameter:
126 configuration - object of type Queens
127 Return Value: None
128 """
129 """
130 PostCondition: All n-queens solutions that extend the given
131 configuration are printed.
132 The configuration is restored to its initial state.
133 """
134 global solNum
135 if (configuration.isSolved()):
136 solNum += 1
137 print('Solution No. ', solNum)
138 print(configuration)
139 else:
140 for col in range(0, configuration.getBoardSize()):
141 if configuration.unguarded(col):
142 configuration.add(col)
```

```
143 solveFrom(configuration) #Recursively continue to
144 add queens
144 configuration.remove(col)
```

```
145
146
147
148 def nQueens():
149 """
150 Objective: To find solutions to n-Queens problem
151 Input Parameter: None
152 Return Value: None
153 """
154 boardSize = int(input('Enter board size: '))
155 print(boardSize, '-Queens Problem')
156 print('The board has', boardSize, 'rows and', boardSize,
157 'columns.')
158 configuration = Queens(boardSize)
159 solveFrom(configuration)
160
161
162 def main():
163 """
164 Objective: To find solutions to n-Queens problem
165 Input Parameter: None
166 Return Value: None
167 """
168 nQueens()
169
170 if __name__ == '__main__':
171 main()
```

**Fig. 16.5** Program to solve n-Queens problem (`nQueens.py`)

Execution of the script nQueens (Fig. 16.5) for 4-queens problem yields the following chessboard configurations resulted as the output:

```
Enter board size: 4
```

```
4 -Queens Problem
```

```
The board has 4 rows and 4 columns.
```

```
Solution No. 1
```

```
- Q - -
```

```
- - - Q
```

```
Q - - -
```

```
- - Q -
```

```
Solution No. 2
```

```
- - Q -
```

```
Q - - -
```

```
- - - Q
```

```
- Q - -
```

```
solutions to 4 queens problem
```

#### 16.3 KNIGHT'S TOUR PROBLEM

In this section, we will develop a recursive solution for the well-known problem of Knight's Tour. Given an  $n \times n$  chessboard of  $n^2$  squares; in the tour, the entire board is traversed by a knight so that each chessboard square is visited exactly once. We wish to find all such tours. Any of the  $n^2$  positions on the chessboard can act as an initial starting candidate for the tour. While traversing the

board, at any step, we need to determine whether the tour has been completed, i.e. whether all  $n^2$  squares have been traversed exactly once so that the configuration is complete. If the current configuration is not yet complete, we determine the next sequence of possible moves. For every possible move from the current position, we need to explore all possible tours. As we follow the procedure just described, it is possible that we may get stuck on the way, as there may be a position on board from where no further move is possible. If this happens, we backtrack one or more steps as required, to try another alternative.

Knight's tour: traversing the entire chessboard, without stepping over any square more than once

Given a particular square position, a knight may move:

- two steps vertically upwards or downwards, followed by one step right or left
- two steps horizontally right or left, followed by one step vertically upwards, or downwards

possible moves for a knight

Thus, the knight may, possibly, jump to one of the eight square positions described above. For example, in a  $5 \times 5$  chessboard, if the knight is placed at the centre, there are eight potential moves (Fig. 16.6). Obviously, the number of choices for making a move is limited by a horse position on the chessboard, for example, a horse placed on any of the four corners of the chessboard has only two choices.

Note that the class `Knight` contains several data members and methods for keeping track of the current chessboard configuration and carrying out moves of the knight (Fig. 16.7). The data member `board` represents the chessboard configuration of size `boardSize`. It keeps track of the history of successive moves. It is

initialized with value 0 to mark all chessboard squares as untraversed.

|        |   |  |   |   |
|--------|---|--|---|---|
|        | 2 |  | 3 |   |
| 1      |   |  |   | 4 |
| KNIGHT |   |  |   |   |
| 5      |   |  |   | 8 |
|        | 6 |  | 7 |   |

**Fig. 16.6** Possible moves of a knight

possible moves for a knight placed in the center square in  $5 \times 5$  chessboard

```
if board[x,y] == 0
=>> square x,y has not been visited

else if board[x,y] = i
=>> square x,y has been visited in ith
move
```

data member board: keeps track of history of moves

Function `solveFrom` finds solutions to Knight's Tour problem. The function receives an object

configuration of the class Knight. The overall approach can be summarized as follows:

for every chessboard square p denoting starting position on the chessboard

```
place the knight on square p of current configuration
determine all possible next sequences of moves from the current square p
remove the knight from square p of the current configuration
```

discovering Knight's tour from every possible chessboard position

In summary, the class Knight supports the following methods for solving Knight's Tour problem.

methods of the class Knight

1. `__init__`

This method creates an instance of class Knight and initializes the data members boardSize, moveNum (number of moves taken by the knight), solNum (keeps track of the number of solutions explored), moves (stores all possible moves possible from the current position), and board.

2. `getBoardSize`

This method returns the size of the chessboard.

3. `__str__`

This method returns a string representation of the object of type Knight.

4. `possible`

This method returns True or False depending on whether the move to a given position in the chessboard is possible. It works in the following manner:

if the position is a valid position on chessboard i.e. x and y lie in the valid index range [0, boardSize-1], and the current chessboard square is not traversed before

Return True to indicate that this move is possible

```
 else
 Return False to reject the move
```

is the move to a position possible?

#### 5. add

The method assumes that the current chessboard square has not yet been traversed by the knight. It places the knight at the given chessboard position by marking it *traversed*, i.e. assigning it the current move number and incrementing the `moveNum` by 1 to denote the number of moves taken so far.

traverse a chessboard square

#### 6. remove

The method assumes that the knight has traversed the chessboard position  $(x,y)$ . It removes the knight from given chessboard position by marking it *untraversed* (value zero), and decrementing the `moveNum` by 1 to update the number of moves taken so far.

undo the traversal of a chessboard square

#### 7. moveFurther

This method determines next sequence of moves if the current configuration is not yet complete. It works in the following manner:

```
if a solution to Knight's tour is found
 print configuration
else
 for every possible candidate move p from the
 current position(x,y)
 place the knight on square p of the current
 configuration.
 try the next sequence of moves.
 remove the knight from square p of current
 configuration.
```

discover the next sequence of moves for the knight

#### 8. isSolved

This method determines whether the current chessboard configuration is complete, i.e. all the squares have been traversed. It works in the following manner:

```
Determine the number of total moves in the
current chessboard configuration.
If the total number of moves equals
boardSize*boardSize
 return True
else
 return False
```

is the current chessboard configuration complete?

In the script `knightTour` (Fig. 16.7), we have developed a generic solution to the Knight's Tour problem, which takes chess board size as input from the user and prints the possible solutions.

```
001 class Knight:
002 '''
003 Objective: To find solutions to Knight's Tour problem
004 '''
005
006 def __init__(self, size):
007 '''
008 Objective: To initialize object of class Knight
009 Input Parameters:
010 self (implicit parameter) - object of type Knight
011 size - numeric
012 Return Value: None
013 '''
014 #Dimension of board
015 self.boardSize = size
016 #Specifies the number of moves taken by the Knight
017 self.moveNum = 0
018 #Total number of Knight's tour explored
019 self.solNum = 0
020 #Initialize all the squares on the chessboard with value 0
021 #to mark them as untraversed.
022 #Keeps track of history of successive moves
023 # if board[x,y] = 0 square x,y has not been visited
024 # if board[x,y] = i square x,y has been visited in
 ith move
025 #if board[x,y] > 0 square x,y has been visited more than once
 #so ignore it
```

```

025 self.moves= [(0 1)(1 0)(1 1) in range(size)] for j in \
026 range(size)]
027 #Relative moves possible from the current position
028 self.moves = [(2,1), (1,2), (-1,2), (-2,1), (-2,-1) \

```

```

029 ,(-1,-2), (1,-2), (2,-1)]
030
031 def getBoardSize(self):
032 """
033 Objective: To determine size of chessboard
034 Input Parameter:
035 self (implicit parameter) - object of type Knight
036 Return Value: boardSize - numeric
037 """
038
039 return self.boardSize
040
040 def __str__(self):
041 """
042 Objective: To return string representation of object of
043 type Knight
044 Input Parameter: self (implicit parameter)- object of
045 type Knight
046 Return Value: string
047 """
048
048 result = ''
049 for i in range(self.boardSize):
050 for j in range(self.boardSize):
051 result += str(self.board[i][j]) + '\t'
052 result += '\n'
053
054
055 def possible(self, xyPos):
056 """
057 Objective: To determine whether the move to position
058 xyPos is possible
059 Input Parameters:
060 self (implicit parameter) - object of type Knight
061 xyPos - tuple of numeric data
062 Return Value: True or False - Boolean
063 """
064
064 x, y = xyPos[0], xyPos[1]
065 if x >=0 and x <self.boardSize and y>=0 and \

```

```
066 if y < self.boardSize and self.board[x][y] == 0:
067 return True
068 else:
069 return False
070
071 def add(self, xyPos):
072 '''
```

```
073 Objective: To place the knight to given chessboard
074 position
075 Input Parameters:
076 self (implicit parameter) - object of type Knight
077 xyPos - tuple of numeric data
078 Return Value: None
079 '''
080
081 x, y = xyPos[0], xyPos[1]
082 self.moveNum += 1
083 self.board[x][y] = self.moveNum
084
085 def remove(self, xyPos):
086 '''
087 Objective: To remove the knight from given chessboard
088 position.
089 Input Parameters:
090 self (implicit parameter) - object of type Knight
091 xyPos - tuple of numeric data
092 Return Value: None
093 '''
094
095 x, y = xyPos[0], xyPos[1]
096 self.board[x][y] = 0
097 self.moveNum -= 1
098
099 def moveFurther(self, xyPos):
100 '''
101 Objective: To determine next sequence of moves if the
102 current configuration is not completed.
103 Input Parameters:
104 self (implicit parameter) - object of type Knight
105 xyPos - tuple of numeric data
106 Return Value: None
107 '''
108
109 PostCondition: All knight's tour solutions for the given
```

```
108 configuration are printed.
109 The configuration is restored to its initial state.
110 ''
111 x, y = xyPos[0], xyPos[1]
112 if (self.isSolved()):
113 self.solNum += 1
114 print('Solution Number: ', self.solNum)
115 print(self)
116 return
117 for (dx,dy) in self.moves:
```



```
118 #Select candidate for the next sequence of moves
119 newMove = (x+dx, y+dy)
120 if self.possible(newMove):
121 self.add(newMove)
122 self.moveFurther(newMove)
123 self.remove(newMove)
124
125 def isSolved(self):
126 """
127 Objective: To determine whether the current chessboard
128 configuration is fully or partially completed
129 Input Parameter:
130 self (implicit parameter) - object of type Knight
131 Return Value: True or False - boolean
132 """
133 if self.moveNum == self.boardSize * self.boardSize:
134 return True
135 else:
136 return False
137
138
139 def solveFrom(configuration):
140 """
141 Objective: To find solutions to Knight's Tour problem
142 Input Parameter:
143 configuration - object of type Knight
144 Return Value: None
145 """
146 for i in range(configuration.getBoardSize()):
147 for j in range(configuration.getBoardSize()):
148 configuration.add((i,j))
149 configuration.moveFurther((i,j))
150 configuration.remove((i,j))
151
152 def knightTour():
153 """
154 Objective: To find solutions to Knight's Tour problem
155 Input Parameter: None
156 Return Value: None
157 """
158 boardSize = int(input("Enter board size: "))
```

```
150 boardSize = int(input("Enter board size: "))
```

```
159 print('The board has ', boardSize, ' rows and ', boardSize,
160 ' columns.')
160 configuration = Knight(boardSize)
161 solveFrom(configuration)
162
163 def main():
164 """
165 Objective: To find solutions to Knight's Tour problem
166 Input Parameter: None
167 Return Value: None
168 """
169 knightTour()
170
171
172 if __name__ == '__main__':
173 main()
```

**Fig. 16.7** Program to solve Knight's Tour problem  
(knightTour.py)

On executing the script knightTour (Fig. 16.7) for a chessboard of size 5, it generated 1728 solutions. We give below a few chessboard configurations generated by the script knightTour:

```
Enter board size: 5
```

```
The board has 5 rows and 5 columns.
```

```
Solution Number: 1
```

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 6  | 15 | 10 | 21 |
| 14 | 9  | 20 | 5  | 16 |
| 19 | 2  | 7  | 22 | 11 |
| 8  | 13 | 24 | 17 | 4  |
| 25 | 18 | 3  | 12 | 23 |

Solution Number: 2

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 6  | 11 | 18 | 21 |
| 12 | 17 | 20 | 5  | 10 |
| 7  | 2  | 15 | 22 | 19 |
| 16 | 13 | 24 | 9  | 4  |
| 25 | 8  | 3  | 14 | 23 |

Solution Number: 3

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 6  | 11 | 16 | 21 |
| 12 | 15 | 20 | 5  | 10 |
| 7  | 2  | 13 | 22 | 17 |
| 14 | 19 | 24 | 9  | 4  |
| 25 | 8  | 3  | 18 | 23 |

Solution Number: 4

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 6  | 17 | 12 | 21 |
| 16 | 11 | 20 | 5  | 18 |
| 7  | 2  | 9  | 22 | 13 |
| 10 | 15 | 24 | 19 | 4  |
| 25 | 8  | 3  | 14 | 23 |

Solution Number: 5

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 12 | 17 | 6  | 21 |
| 18 | 5  | 20 | 11 | 16 |
| 13 | 2  | 9  | 22 | 7  |
| 4  | 19 | 24 | 15 | 10 |
| 25 | 14 | 3  | 8  | 23 |

solutions to knight's tour problem

Solution Number: 6

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 16 | 11 | 6  | 21 |
| 10 | 5  | 20 | 15 | 12 |
| 17 | 2  | 13 | 22 | 7  |
| 4  | 9  | 24 | 19 | 14 |
| 25 | 18 | 3  | 8  | 23 |

Solution Number: 7

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 18 | 11 | 6  | 21 |
| 10 | 5  | 20 | 17 | 12 |
| 19 | 2  | 15 | 22 | 7  |
| 4  | 9  | 24 | 13 | 16 |
| 25 | 14 | 3  | 8  | 23 |

Solution Number: 8

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 10 | 15 | 6  | 21 |
| 16 | 5  | 20 | 9  | 14 |
| 11 | 2  | 7  | 22 | 19 |
| 4  | 17 | 24 | 13 | 8  |
| 25 | 12 | 3  | 18 | 23 |

Solution Number: 9

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 16 | 5  | 10 | 21 |
| 6  | 11 | 20 | 15 | 4  |
| 19 | 2  | 17 | 22 | 9  |
| 12 | 7  | 24 | 3  | 14 |
| 25 | 18 | 13 | 8  | 23 |

Solution Number: 10

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 12 | 5  | 18 | 21 |
| 6  | 17 | 20 | 13 | 4  |
| 11 | 2  | 9  | 22 | 19 |
| 16 | 7  | 24 | 3  | 14 |
| 25 | 10 | 15 | 8  | 23 |

#### 16.4 STABLE MARRIAGE PROBLEM

In this problem, we deal with two groups of equal number of persons, one of the men and the other of women. Each man gives his relative preference for every woman from best to worst, and each woman gives her relative preference for every man. Based on their preferences, we are required to find a pairing between men and women. Further, the pairing should be stable in the following sense: there are no two people of the opposite sex who would both prefer to have each other than their current partners. For example, let  $M = \{m_1, m_2\}$  and  $W = \{w_1, w_2\}$  be two sets of men and women, respectively. Let us suppose  $m_1$  and  $m_2$  have an identical list of ordered preferences of women  $[w_1, w_2]$ , i.e. each of them prefers  $w_1$  over  $w_2$ . Further, let us suppose  $w_1$  and  $w_2$  also have an identical list of ordered preferences of men  $[m_1, m_2]$ , i.e. each of them prefers  $m_1$  over  $m_2$ . Next, suppose we do the following matching  $\{\{m_1, w_2\}, \{m_2, w_1\}\}$ . This matching is not stable because  $m_1$  and  $w_1$  prefer each other over the partners assigned to them. However, the matching  $\{\{m_1, w_1\}, \{m_2, w_2\}\}$  is stable because there are no two persons of opposite sex that would prefer each other over the partners assigned to them, for example, although  $m_2$  prefers  $w_1$  over  $w_2$ , but  $w_1$  does not prefer  $m_2$  over  $m_1$ .

notion of a stable marriage

The goal is to determine  $n$  pairs of  $\langle m, w \rangle$  so that each pair represents a stable marriage, i.e. there is no pair  $\langle m, w \rangle$  of persons who would prefer each other to

their assigned partners. In other words, a pair  $\langle m, w \rangle$  is considered to be stable if the following two conditions hold:

1. Every women candidate who is preferred by man  $m$  to his current assignment  $w$  prefers her current partner over the man  $m$ .
2. Every man candidate who is preferred by woman  $w$  to her current assignment  $m$  prefers his current partner over the woman  $w$ .

conditions for a pair  $\langle m, w \rangle$ , to be stable

A straightforward approach is to enumerate all possible permutations of women for the given sequential list 1 to  $n$  of men. This simplifies the original problem to the subproblem of determining whether each of the  $n!$  permutations represents a stable marriage. However, examining all the  $n!$  permutations is a computationally expensive task.

`menPref`: dictionary for storing men's preferences for women

In the following discussion, we will examine another approach in which we will search for a match  $w$  for a man  $m$  on the basis of his preferences. For this purpose, we define a class `StableMarriage` that contains several data members and methods for discovering the stable pairings for the given set of  $n$  men and  $n$  women. The data members `menPref` and `womenPref` are dictionaries that store men's preferences for women and women's preferences for men, respectively. For example, for a set of three men, the dictionary `menPref` may be `{1:[2,1,3], 2:[3,2,1], 3:[1,3,2]}`. Another pair of dictionaries `engagedMen` and `engagedWomen` keep a record of engaged men and engaged women respectively at any point in time. The list `freeWomen` includes women who are not yet engaged. The recursive function `findMatching` discovers solutions to the

stable marriage problem. The function receives the following arguments: an object configuration of the class `StableMarriage` and the man for whom a stable match is to be searched. The approach used in the function may be summarized as follows:

`womenPref`: dictionary for storing women's preferences for men

`engagedMen` and `engagedWomen`: used to keep track of engaged men and women at any point in time

`freeWomen`: list of women who are not yet engaged

if a solution to stable marriage problem is found

(Pairing is determined for all the men)

print the pairing

else

for every woman `w` in the ordered preference

list of the man `m` in consideration

if the woman `w` is not engaged and pairing is stable

engage the man `m` and woman `w`

find stable matching for the man `m+1`

Set free the engaged pair of man `m` and woman `w`

discovering all solutions to stable marriage problem

Note that if for a given man  $m$ , if none of the women not yet engaged yields a stable pairing, the control returns to the previous call and the matching for the man  $m-1$  is undone to try another alternative. This backtracking may proceed up to man  $m = 1$ . Finally, the man  $m$  would be paired. Whenever all the men have been matched, a stable marriage solution is printed.

backtrack if no stable matching is found for the given man  $m$

As discussed earlier, the class `StableMarriage` supports the following methods for solving Stable Marriage problem.

methods of the class `StableMarriage`

1. `__init__`

This method creates an instance of class `StableMarriage` and initializes the data members `count`, `menPref`, `womenPref`, `freeWomen`, `engagedMen` and `engagedWomen`.

2. `__str__`

This method returns a string representation of the object of type `StableMarriage`.

3. `isStable`

This method returns `True` or `False` depending on whether the pairing of given man and woman under consideration indicates a stable marriage. It works in the following manner:

1. For man  $m$ , all other women candidate who are preferred by man  $m$  to his current assignment  $w$ , are already married and prefer their current partners over man  $m$ .

is the given pair  $\langle m, w \rangle$  stable?

2. For woman  $w$ , all other men candidate who are preferred by woman  $w$  to her current assignment  $m$ , are either yet not engaged or prefer their current partners over woman  $w$ . (Candidate men who are not engaged yet are not considered till now) .  
 4. free  
 This method sets free an engaged pair of man and woman. It achieves this by adding the given woman to the list `freeWomen` who are not engaged. It also updates dictionaries `engagedMen` and `engagedWomen` to reflect that they are no longer engaged.

set free an engaged pair of man and woman

5. engage  
 This method engages the given man and woman by pairing them. It achieves this by removing the given woman from the list `freeWomen`. It also updates dictionaries `engagedMen` and `engagedWomen` to reflect that they are engaged.

engage the given man and woman by pairing them

6. `findMatching`  
 This method finds stable matching for the given man.

The complete program is given in Fig. 16.8.

```

001 class StableMarriage:
002 """
003 Objective: To solve Stable Marriage problem
004 """
005
006 def __init__(self, n, menPref, womenPref):
007 """
008 Objective: To initialize object of class StableMarriage
009 Input Parameters:

```

```

010 self (implicit parameter)-object of type StableMarriage
011 n - numeric

```

```
012 menPref, womenPref - list
013 Return Value: None
014 """
015 #There are n men and n women
016 self.count = n
017 #Dictionary specifying men's preference for women
018 self.menPref = menPref
019 #Dictionary specifying women's preference for men
020 self.womenPref = womenPref
021 #List specifying women who are not engaged
022 self.freeWomen = list(self.womenPref.keys())
023 #Dictionary keeping track of engaged men and women
024 self.engagedMen = {i:None for i in self.menPref.keys()}
025 self.engagedWomen = {i:None for i in self.womenPref.
026 keys()}
027
028 def __str__(self):
029 """
030 Objective: To return string representation of object of
031 type StableMarriage
032 Input Parameter: self (implicit parameter)- object of type
033 StableMarriage
034 Return Value: string
035 """
036 return str(self.engagedMen)
037
038
039 def isStable(self, man, woman):
040 """
041 Objective: To determine whether the pairing of given man
042 and woman is a stable marriage
043 Input Parameters:
044 self (implicit parameter)-object of type StableMarriage
045 man, woman - numeric
046 Return Value: True, if stable, False otherwise.
047 """
048 # Find all candidate women who are preferred by man to
049 # their current assignment
050 rank = self.menPref[man].index(woman)
```

```

051 i = 0
052 stable = True
053 while i < rank and stable:
054 otherWoman = self.menPref[man][i]
055 # Must have already been married, otherwise man would
056 # have picked her. otherWoman prefer her current
057 # partner over the given man.
058 if otherWoman not in self.freeWomen:
059 stable = self.womenPref[otherWoman].index(man) > \
060 self.womenPref[otherWoman].index(self.
061 engagedWomen[otherWoman])
062
063
064 # Find all candidate men who are preferred by woman to
065 # their current assignment
066 rank = self.womenPref[woman].index(man)
067 i = 0
068 while i < rank and stable:
069 otherMan = self.womenPref[woman][i]
070 # For otherMan who have been already considered(if
071 # yet not considered, he will automatically be
072 # considered later on.), otherMan prefer his
073 # current partner over the given woman.
074 if otherMan < man:
075 stable = self.menPref[otherMan].index(woman) > \
076 self.menPref[otherMan].index(self.
077 engagedMen[otherMan])
078
079
080 def free(self, man, woman):
081 """
082 Objective: To set free the engaged pair of given man and
083 woman
084 Input Parameters:
085 self (implicit parameter)-object of type StableMarriage
086 man, woman - numeric
087 Return Value: None
088 """
089 PreCondition - Given man and woman are engaged

```

```
090 """
091 self.freeWomen.append(woman)
092 self.engagedMen[man] = None
093 self.engagedWomen[woman] = None
094
095 def engage(self, man, woman):
096 """
097 Objective: To engage the given man and woman by pairing
098 them
099 Input Parameters:
100 self (implicit parameter)-object of type StableMarriage
101 man, woman - numeric
102 Return Value: None
103 """
104 """
105 PreCondition - Given man and woman are not engaged
106 """
107 self.freeWomen.remove(woman)
108 self.engagedMen[man] = woman
109 self.engagedWomen[woman] = man
110
111 def findMatching(self, man = 1):
112 """
113 Objective: To find stable matching for the given man
114 Input Parameters:
115 self (implicit parameter)-object of type StableMarriage
116 man - numeric
117 Return Value: None
118 """
119 """
120 PostCondition - Stable pairing for given man is determined.
121 If all the men are considered, a stable marriage solution
122 is printed.
123 """
124 if man > self.count:
125 print(self)
126 return
127 for woman in self.menPref[man]:
128 if woman in self.freeWomen and self.isStable(man,
```

```
128 #If both are not engaged
```

```
129 self.engage(man, woman)
130 self.findMatching(man+1)
131 self.free(man, woman)
132
133
134 def findStableMarriage():
135 """
136 Objective: To determine a pairing between men and women based
137 on their preferences
138 Input Parameter: None
139 Return Value: None
140 """
141 n = int(input('Enter number of men/women: '))
142 menPref = eval(input('Specify men\'s preferences: '))
143 womenPref = eval(input('Specify women\'s preferences: '))
144 print('Stable pairings:')
145 ob = StableMarriage(n, menPref, womenPref)
146 ob.findMatching()
147
148 def main():
149 """
150 Objective: To find solution to Stable Marriage problem
151 Input Parameter: None
152 Return Value: None
153 """
154 findStableMarriage()
155
```

```
|156 if __name__ == '__main__':
157 main()
```

**Fig. 16.8** Program to solve Stable Marriage Problem  
(stableMarriage.py)

Below we give a sample execution of the script  
stableMarriage (Fig. 16.8):

```
Enter number of men/women: 3
```

```
Specify men's preferences: {1:[2,1,3], 2:
[3,2,1], 3:[1,3,2]}
```

```
Specify women's preferences: {1:[2,1,3],
2:[3,2,1], 3:[1,3,2]}
```

```
Stable pairings:
```

```
{1: 2, 2: 3, 3: 1}
```

```
{1: 1, 2: 2, 3: 3}
```

```
{1: 3, 2: 1, 3: 2}
```

```
stable pairings
```

#### 16.5 FRACTAL (HILBERT CURVE AND SIERPINSKI TRIANGLE)

Fractal is a curve or a geometric figure that repeats itself. Such a figure comprises a recursive pattern that repeats itself up to a desired level of nesting. In this section, we have considered two fractals, Hilbert Curve, and Sierpinski Triangle. Turtle graphics provided in the `turtle` module is used for drawing to draw various shapes and pictures. `turtle` methods used in this section are as follows:

**fractal:** curve or a geometric figure which comprises a recursive pattern that repeats itself

**turtle module:** provides turtle graphics for drawing various shapes and pictures

1. `forward()`: Used for moving the turtle forward by a given distance in the direction of the turtle.
2. `backward()`: Used for moving the turtle backward by a given distance in the direction of the turtle.
3. `left()`: Used for rotating the turtle in the left direction by a specified angle.
4. `right()`: Used for rotating the turtle in the right direction by a specified angle.
5. `goto(x, y)`: Used for moving the turtle to the location specified ( $x, y$  coordinates).
6. `penup()`: Used to specify no drawing while moving.
7. `pendown()`: Used to specify drawing while moving.
8. `fillcolor(r, g, b)`: Used to specify the colour to be filled in the shape. Colour specified could either be a string or a tuple ( $r, g, b$ ).
9. `begin_fill()`: Used just before drawing the shape to be filled.
10. `end_fill()`: Used to fill the shape drawn after last call to `begin_fill()`.
11. `done()`: Completes the turtle graphics work.

#### methods of `turtle` module

Let us examine the Sierpinski Triangle first. The simplest Sierpinski triangle is a triangle subdivided into nested equilateral triangles formed by bisecting sides of the triangle having the central triangle removed. This results in an outer triangle with three upward facing equilateral triangles on top, bottom left and bottom right side with one downward facing triangle that is removed from the centre. Each of the triangles facing upward may further contain three nested equilateral triangles depending upon the level of depth. Sierpinski triangles of levels 1, 2 and 3 are shown in Fig. 16.9. Next, we describe the approach used for drawing Sierpinski triangle:

Sierpinski triangle: recursive triangle subdivided into nested equilateral triangles

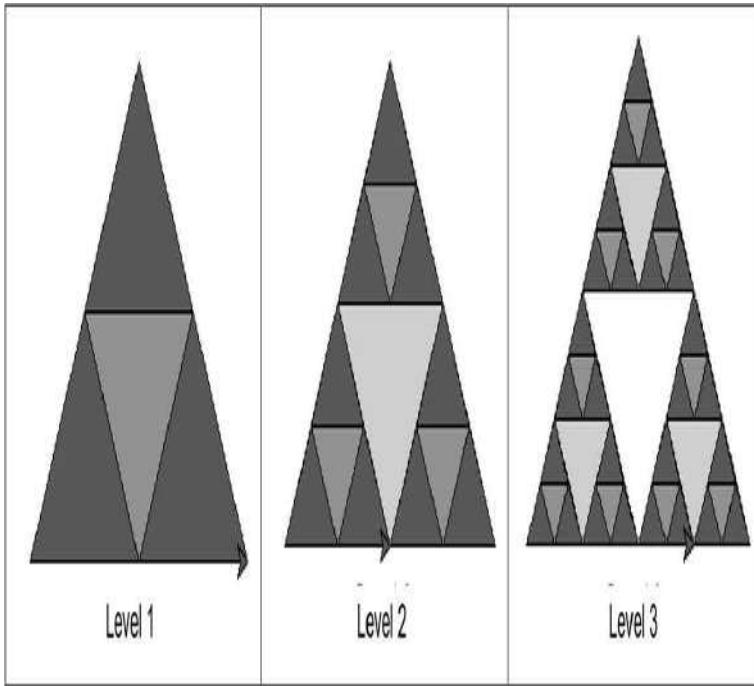
Draw an outer triangle.

If the levels of the Sierpinski curve to be created are more than zero

Call Sierpinski curve method for each of the nested three Sierpinski triangles.

approach for drawing sierpinski triangle

The nested triangles are formed by joining the midpoints of the sides of given equilateral triangle. The function mid is repeatedly used for this purpose.



**Fig. 16.9** Sierpinski Triangle of Level 1, 2 and 3

The triangle drawn as the part of the first statement in Sierpinski triangle can be created using the following steps:

1. Specify the color to be filled in the closed object to be created, using `fillcolor` method.
2. Position the turtle to one of the end points of the triangle in pen-up mode so that there is no drawing while moving.
3. Invoke method `begin_fill` before drawing the shape to be filled.
4. After successful positioning, create a triangle by traversing all its endpoints in the pen-down mode.
5. Invoke method `end_fill` after drawing the shape to be filled.

Program for drawing Sierpinski triangle is given in the script `sierpinski` (Fig. 16.10).

```
01 from turtle import *
02 def createTriangle(points, color):
03 """
04 Objective: To draw the triangle using given points filled
05 with given color
06 Input Parameters:
07 points- list of three-tuples comprising of (x,y) coordinate
08 of points
09 color - string
10 Return Value: None
11 """
12 fillcolor(color)
```



```

13 penup()
14 goto(points[0][0],points[0][1])
15 pendown()
16 begin_fill()
17 goto(points[1][0],points[1][1])
18 goto(points[2][0],points[2][1])
19 goto(points[0][0],points[0][1])
20 end_fill()
21
22 def mid(p1,p2):
23 """
24 Objective: To find midpoint for the given points
25 Input Parameters:
26 p1, p2 - tuple comprising of (x,y) coordinate of points
27 Return Value: tuple representing mid point
28 """
29 return ((p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)
30
31 def sierpinski(points,level):
32 """
33 Objective: To draw Sierpinski triangle
34 Input Parameters: level - numeric
35 points- list of three tuples comprising of (x,y) coordinate
36 of points
37 Return Value: None
38 """
39 colormap = ['blue','red','green','white','yellow',
40 'violet','orange']
41 createTriangle(points, colormap[level % len(colormap)])
42 if level > 0:
43 sierpinski([points[0],
44 mid(points[0], points[1]),
45 mid(points[0], points[2])],
46 level-1)
47 sierpinski([points[1],
48 mid(points[0], points[1]),
49 mid(points[1], points[2])],
50 level-1)
51 sierpinski([points[2],

```

```

52 mid(points[2], points[1]),
53 mid(points[0], points[2])),
54 level-1)
55
56 def main():
57 """
58 Objective: To draw the sierpinski triangle based on user input
59 Input Parameter: None
60 Return Value: None
61 """
62 endPoints = [(-100, -50), (0,100), (100, -50)]
63 level = int(input('Enter the level for Sierpinski
64 Triangle: '))
65 sierpinski(endPoints, level)
66 done()
67
68 if __name__ == '__main__':
69 main()

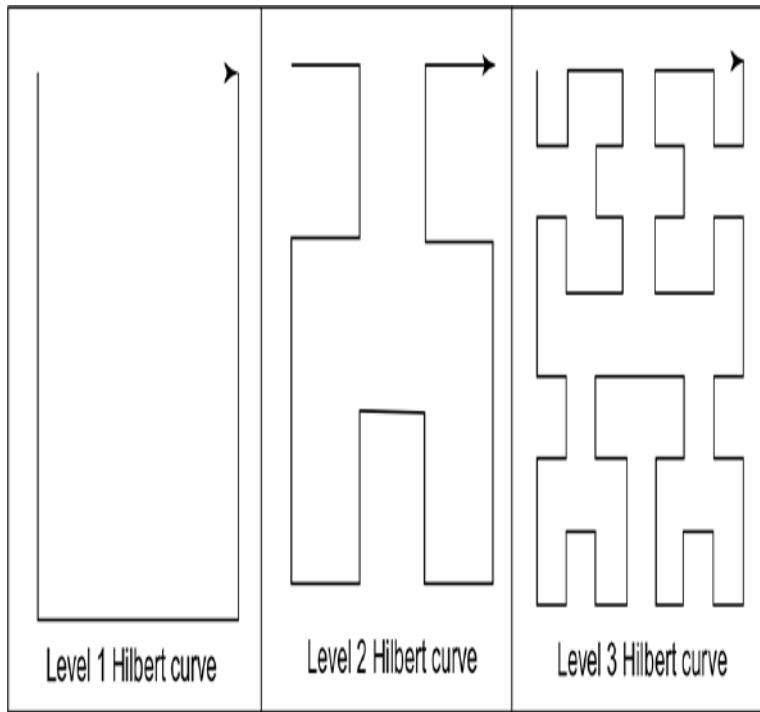
```

**Fig. 16.10** Program draw Sierpinski Triangle (*sierpinski.py*)

We may also set the speed of the turtle using function `speed`. Next, we develop a program for drawing Hilbert curve. A Hilbert curve is a curve that is formed by connecting a sequence of U-shaped curves oriented in different directions. These U-shaped curves are placed at a certain step size distance apart. Level 1 Hilbert curve is a simple U-curve that is formed by moving down step

size, then moving right step size, followed by moving up step size as shown in Fig. 16.11. Level 2 Hilbert curve is formed by orienting level 1 Hilbert curve in different directions and connecting them. Thus, in general, Hilbert curve of level  $n$  can be formed by orienting the Hilbert curve at level  $n - 1$  in different directions and connecting them.

Hilbert curve: curve formed by connecting a sequence of U-shaped curves oriented in different directions



**Fig. 16.11** Hilbert curve of Level 1, 2 and 3

Let us examine Hilbert curve at level 1. Since turtle initially points towards the right, the following steps will draw a simple U curve.

1. Move right step size.
2. Move left step size.
3. Move left step size.

steps for drawing level 1 Hilbert curve

Let  $y = 90$  degrees. The above steps may be re-written as follows:

1. Rotate  $y$  degree towards the right.
2. Move step size.
3. Rotate  $y$  degree towards left.
4. Move step size.
5. Rotate  $y$  degree towards left.
6. Move step size.
7. Rotate  $y$  degree towards the right.

Next, let us examine level 2 Hilbert curve. Again, assuming that initially turtle points towards the right, the following steps may be used to draw a level 2 Hilbert curve:

1. Rotate  $y$  degree towards the right.
2. Create a Hilbert curve at level 1 rotated by  $-y$  degree (i.e.  $y$  degree in anticlockwise direction).
3. Move step size.
4. Rotate  $y$  degree towards the left.
5. Create a Hilbert curve at level 1 rotated by  $y$  degree (i.e.  $y$  degree in clockwise direction).
6. Move step size.
7. Create a Hilbert curve at level 1 rotated by  $y$  degree (i.e.  $y$  degree in clockwise direction).
8. Rotate  $y$  degree towards the left.
9. Move step size.
10. Create a Hilbert curve at level 1 rotated by  $-y$  degree (i.e.  $y$  degree in anticlockwise direction).
11. Rotate  $y$  degree towards the right.

steps for drawing level 2 Hilbert curve

Thus, Hilbert curve of any level  $n$  greater than zero with given degree  $y$  can be drawn using the following steps:

1. Rotate  $y$  degree towards the right.
2. Create a Hilbert curve at level  $n - 1$  rotated by  $-y$  degree (i.e.  $y$  degree in anticlockwise direction).
3. Move step size.
4. Rotate  $y$  degree towards the left.
5. Create a Hilbert curve at level  $n - 1$  rotated by  $y$  degree (i.e.  $y$  degree in clockwise direction).
6. Move step size.

7. Create a Hilbert curve at level  $n - 1$  rotated by  $y$  degree  
(i.e.  $y$  degree in clockwise direction)
8. Rotate  $y$  degree towards the left.
9. Move step size.
10. Create a Hilbert curve at level  $n - 1$  rotated by  $-y$  degree  
(i.e.  $y$  degree in anticlockwise direction).
11. Rotate  $y$  degree towards the right.

steps for drawing level n Hilbert curve

Program for drawing Hilbert curve is given in Fig. 16.12.

```

01 from turtle import *
02
03 def hilbert(level, angle, step):
04 """
05 Objective: To draw Hilbert curve of given level rotated by given
06 angle
07 Input Parameters:
08 level, angle, step - numeric
09 Return Value: None
10 """
11 if level == 0:
12 return
13 right(angle)
14 hilbert(level - 1, -angle, step)
15 forward(step)
16 left(angle)
17 hilbert(level - 1, angle, step)
18 forward(step)
19 hilbert(level - 1, angle, step)
20 left(angle)
21 forward(step)
22 hilbert(level - 1, -angle, step)
23 right(angle)
24
25 def main():
...
```

```

26 """
27 Objective: To draw hilbert curve based on user input
28 Input Parameter: None
29 Return Value: None
30 """
31 level = int(input('Enter the level for Hilbert curve: '))
32 size = 200
33 penup()
34 goto(-size / 2.0, size / 2.0)
35 pendown()
36 hilbert(level, 90, size/(2**level-1)) #For positioning turtle
37 done()

```

```

38
39 if __name__ == '__main__':
40 main()

```

**Fig. 16.12** Program draw Hilbert curve (`hilbert.py`)

#### 16.6 SUDOKU

A Sudoku puzzle comprises a  $9 \times 9$  grid divided into nine blocks of size  $3 \times 3$ . Each of the blocks contains digits from 1 to 9 without repetition as shown in Fig. 16.13. The same constraint is applicable on every row and column. In a Sudoku problem, a partially filled  $9 \times 9$  grid is given, and the challenge is to find the missing numbers while ensuring that all the constraints are satisfied. Apart from the Sudoku, which we just mentioned, there are other variants of differing grid size, alphabetical Sudoku, hyper sudoku, etc. However, in this section, we will develop a recursive approach for solving the common Sudoku having grid size  $9 \times 9$ :

challenge: assign digits 1 to 9 without repetition in each of the blocks of size  $3 \times 3$  within the  $9 \times 9$  grid

no value is allowed to repeat in any row or column

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

**Fig. 16.13** Sudoku grid

The simplest way to represent a  $9 \times 9$  grid is in the form of a nested list. We will use the value 0 to represent a missing value, yet to be determined. The problem can be solved by searching for the target locations with missing values, one by one, so long as the search is not exhausted. For every target location, we will find all such solutions  $x$  that do not violate any of the constraints, i.e. a number not repeated in the row, column, or block containing the target location. For every currently determined solution, the search will proceed by calling the function for solving Sudoku recursively. Approach for determining a solution to Sudoku is mentioned below:

0 : denotes a missing value, yet to be determined

```
if the sudoku is solved, i.e. there are no
zeroes in the grid lst,
 print the solution.

else,
 find a target index (i,j) on the grid
 with value 0.

 # To find values that cannot be used in

 # target index (i,j) in current solution

 Create set excludeNums to store elements
 to be excluded.

 for every position containing non-zero
 value in the grid

 if the position lies in the same row,
 column, or the block containing target
 index

 add the value at that position to
 excludeNums set

 # For discovering a solution

 for every number in possible number list
 [1,2,3,4,5,6,7,8,9]

 if number is not in excludeNums set:

 Place number at index (i,j)

 Call sudokuSolver with current partial
 solution lst.

 Replace number at index (i,j) by 0.
```

Program for solving sudoku is given in Fig. 16.14.

```
01 rows, cols = 9, 9
02
03 def find(lst, searchEle):
04 """
05 Objective: To find index of element searchEle in list lst
06 Input Parameters:
07 lst - list
08 searchEle - numeric
09 Return Value: (i,j) - two element tuple containing index of
10 searchEle
11 """
12 (i, j) = (-1, -1) # Assume no element with value 0
13 for row in range(0, rows):
14 for col in range(0, cols):
```



```

15 if lst[row][col] == searchEle:
16 (i,j) = (row, col)
17 return (i,j)
18 return (i,j)
19
20 def printSol(lst):
21 """
22 Objective: To print the list
23 Input Parameter:
24 lst - list
25 Return Value: None
26 """
27 print('\nSolution:')
28 for row in lst:
29 print(row)
30
31
32 def sudokuSolver(lst):
33 """
34 Objective: To find solution to sudoku problem
35 Input Parameter: lst - list
36 Return Value: None
37 """
38 (i,j) = find(lst,0)
39 if (i,j) == (-1,-1):
40 printSol(lst)
41 return
42 excludedNums = set()
43 for row in range(0, rows):
44 for col in range(0, cols):
45 if lst[row][col] != 0:
46 # If same row, same column, or same block
47 if i==row or j==col or \
48 (i//3 == row//3 and j//3 ==col//3):
49 excludedNums.add(lst[row][col])
50
51 possibleNums = [1,2,3,4,5,6,7,8,9]
52 for number in possibleNums:
53 if number not in excludedNums:
54 lst[i][j] = number
55 sudokuSolver(lst)

```

```
56
57
58 def main():
59 """
60 Objective: To solve sudoku problem based on user input
61 Input Parameter: None
62 Return Value: None
63 """
64
65 Test Input -
66 lst = [[5,3,0,0,7,0,0,0,0],[6,0,0,1,9,5,0,0,0],\
67 [0,9,8,0,0,0,6,0],[8,0,0,0,6,0,0,0,3],\
68 [4,0,0,8,0,3,0,0,1],[7,0,0,0,2,0,0,0,6],\
69 [0,6,0,0,0,0,2,8,0],[0,0,0,4,1,9,0,0,5],\
70 [0,0,0,8,0,0,7,9]]
71
72 lst = eval(input('Enter the list (0 for missing values): '))
73 sudokuSolver(lst)
74
75 if __name__ == '__main__':
76 main()
```

**Fig. 16.14** Program for solving sudoku (`sudoku.py`)

On executing the script `sudoku` (Fig. 16.14) with the following input,

```
Enter the list (0 for missing values):
[[5, 3, 0, 0, 7, 0, 0, 0, 0], [6, 0, 0, 1,
9, 5, 0, 0, 0], [0, 9, 8, 0, 0, 0, 0, 6,
0], [8, 0, 0, 0, 6, 0, 0, 0, 3], [4, 0, 0,
8, 0, 3, 0, 0, 1], [7, 0, 0, 0, 2, 0, 0,
0, 6], [0, 6, 0, 0, 0, 2, 8, 0], [0, 0,
0, 4, 1, 9, 0, 0, 5], [0, 0, 0, 0, 8, 0,
0, 7, 9]]
```

Python responded with the following solution:

Solution:

```
[5, 3, 4, 6, 7, 8, 9, 1, 2]
```

```
[6, 7, 2, 1, 9, 5, 3, 4, 8]
```

```
[1, 9, 8, 3, 4, 2, 5, 6, 7]
```

```
[8, 5, 9, 7, 6, 1, 4, 2, 3]
```

```
[4, 2, 6, 8, 5, 3, 7, 9, 1]
```

```
[7, 1, 3, 9, 2, 4, 8, 5, 6]
```

```
[9, 6, 1, 5, 3, 7, 2, 8, 4]
```

```
[2, 8, 7, 4, 1, 9, 6, 3, 5]
```

```
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```

sudoku solutions for the given grid values

#### 16.7 GUIDELINES ON USING RECURSION

Recursion is an important programming tool. But the unnecessary use of recursion makes the programs inefficient. However, we would like to point out that in special cases of recursion like tail recursion that occurs at the tail of the function (no code occurring after the

recursive call), an optimizing compiler would replace the recursive call by an iterative code.

recursion may lead to inefficiency

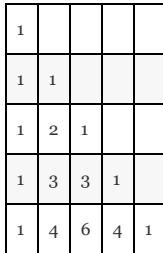
#### SUMMARY

1. Wrapper function is the function needed to do some housekeeping.
2. Eight-queens problem : place eight queens on the chessboard so that none can attack each other. A queen is attackable if another queen lies in the same row, same column, or same diagonal (either direction).
3. Knight's Tour: traverse the entire chessboard, without stepping over any square more than once.
4. Stable Marriage problem : We are given two groups of equal number of persons, one of the men and the other of women. Each man gives his relative preference for every woman from best to worst, and each woman gives her relative preference for every man. Based on their preferences, we are required to find a pairing between men and women. The pairing should be stable in the following sense: there are no two people of opposite sex who would both prefer to have each other than their current partners.
5. A fractal is a curve or a geometric figure that repeats itself. Such a figure comprises a recursive pattern that repeats itself up to a desired level of nesting.
6. The simplest Sierpinski triangle is a triangle subdivided into nested equilateral triangles formed by bisecting sides of the triangle having the triangle at the center removed. This results in an outer triangle with three upward facing equilateral triangles on top, bottom left and bottom right side with one downward facing triangle that is removed from the centre. Each of the triangles facing upward may further contain three nested equilateral triangles depending upon the level of depth.
7. A Hilbert curve is a curve that is formed by connecting a sequence of U-shaped curves oriented in different directions. These U-shaped curves are placed at a certain step size distance apart. Level 1 Hilbert curve is a simple U-curve that is formed by moving down a distance step size, then moving right a distance step size, and finally moving up a distance step size. Level 2 Hilbert curve is formed by orienting level 1 Hilbert curve in different directions and connecting them. Thus, in general, Hilbert curve of level  $n$  can be formed by orienting the Hilbert curve at level  $n - 1$  in different directions and connecting them.
8. A Sudoku puzzle comprises a  $9 \times 9$  grid divided into nine blocks of size  $3 \times 3$ . Each of the blocks should contain digits from 1 to 9 without repetition. The same constraint is applicable on every row and column.

Given a partially filled  $9 \times 9$  grid, the challenge is to find the missing numbers while ensuring that all the constraints are satisfied.

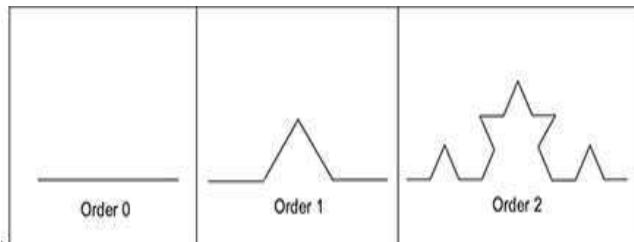
#### EXERCISES

1. Write a generalized version of Sudoku problem.
2. Write a program that takes a number  $n$  as an input and prints pascal triangle comprising first  $n$  rows. For example, for  $n = 5$ , following output should be displayed:



|   |   |   |   |   |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 1 | 1 |   |   |   |
| 1 | 2 | 1 |   |   |
| 1 | 3 | 3 | 1 |   |
| 1 | 4 | 6 | 4 | 1 |

3. Write a recursive function for Tug of war problem. The function should take a list of  $n$  integers and should divide the elements into two lists of  $n/2$  sizes so that the absolute difference between the sum of elements of the two lists is the minimum possible.
4. Write a recursive function that draws order  $n$  Koch fractal curve (Fig. 16.15).



**Fig. 16.15** Koch fractal curve

5. Write a program to solve Stable Matching Problem. Given,  $n$  (even) individuals and their ordered preference list for their room partner, the problem is to divide them into  $n/2$  stable pairs based on their preferences. Further, the pairing should be stable in the following sense: there are no two persons (not forming a pair) both of whom would prefer each other than their current partners.
6. Write a program that takes a matrix (list of lists) of 0's and 1's as an input from the user and invokes a recursive function which returns the maximum length of connected cells having value 1. The two cells are said to be connected if they are adjacent to each other, horizontally, vertically, or diagonally.



## CHAPTER 17

# GRAPHICS

### CHAPTER OUTLINE

[17.1 2D Graphics](#)

[17.2 3D Graphics](#)

[17.3 Animation – Bouncing Ball](#)

So far, we have developed programs, which produce output in the form of text content. However, there is a famous saying, ‘A picture is worth a thousand words.’ In this chapter, we shall study, how to visualize the input data and the results produced by a program, in the form of graphs, pie-charts, histograms, and 3D plots. We will also see the effectiveness of animation in communication. Python supports 2D and 3D graphics, and animations in the form of various packages and libraries like Matplotlib, PyQtGraph (Scientific Graphics and GUI Library for Python), VisPy (2D/3D visualization library), OpenGL (library for 2D and 3D graphics), turtle, plotly (2D and 3D graphics library), VisPy, Biggles (2D scientific plotting package), and VPython (3D and animation). In this chapter, we use the standard library matplotlib for 2D graphics and the visual library of VPython for 3D graphics and animation.

graphics and animation: used for visualization

### 17.1 2D GRAPHICS

Two-dimensional graphical objects include point, line, circle, rectangle, oval, polygon, and text. Python library `matplotlib` provides several methods that facilitate drawing these objects. To be more specific, the library supports graphs, histograms, bar charts, pie charts,

scatter plots, error charts, etc. A graphics library, suited to a particular hardware and operating system, may be downloaded from

<http://matplotlib.org/downloads.html>. The graphics libraries typically have a few required dependencies, which include setuptools, numpy, dateutil, pyparsing, six, and pytz

(<http://matplotlib.org/users/installing.html>). For windows, we may directly install these dependencies using the following command at command prompt:

```
pip install numpy python-dateutil pytz
pyparsing six setuptools
```

If python fails to recognize the pip command, it can be downloaded from <https://bootstrap.pypa.io/get-pip.py> and installed on the machine by pressing the keys *WINDOWS R*, and executing the following commands:

```
cmd
```

```
python get-pip.py
```

Alternatively, the downloaded file get-pip.py may directly be executed in Python IDLE. Also, the environment path (system variable) should be updated to include C:\Users\  
<user\_name>\AppData\Local\Programs\Python\Python36-32\Scripts.

The module matplotlib can be downloaded and installed by pressing the keys *WINDOWS R*, and executing the following commands:

```
cmd
```

```
pip install matplotlib
```

installing module matplotlib

In this chapter, we make use of `pyplot` module of `matplotlib`, which contains several functions for plotting figures and modifying or setting various properties such as labels for the figure and layout of the plot area. To be able to use graphics, we need to import `matplotlib.pyplot` module in the current shell environment. Another important module of `matplotlib` that serves the same purpose as `pyplot` is `pylab`.

`pyplot` module: contains functions for plotting figures and setting various properties

#### 17.1.1 Point and Line

Suppose we wish to draw a single point, say,  $(3, 2)$  on the graph. For this purpose, we use the function `plot(x, y)` that takes two arguments `x` and `y` as coordinates (on x-axis and y-axis respectively) of the point to be plotted on the graph. The following sequence of commands will display the point (Fig. 17.1):

```
import matplotlib.pyplot as plt

plt.plot(3, 2)

plt.show()
```

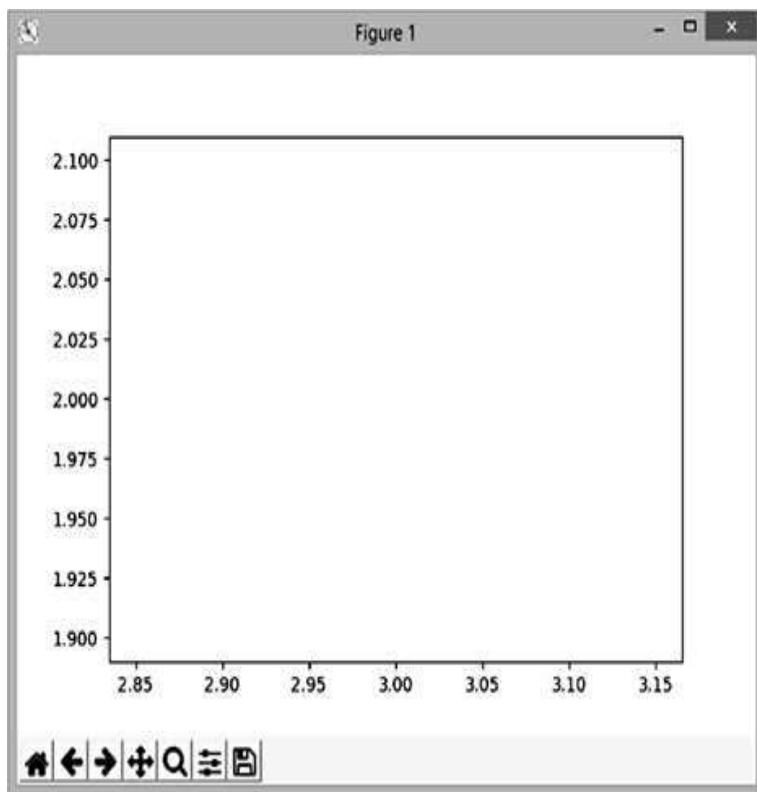
plotting a point

The function `show` is used for displaying the figure. Once the function `show` is executed, the system waits for us to have a look at the graph, and execution of other instructions is blocked until the graphical window is closed. Hence, the function `show` is called a blocking function. When we wish to continue the execution of

other statements, without closing the window manually, we specify `block = False` as an argument while invoking the function `show`. Note that the first line introduces `plt` as an alternative name of the module `matplotlib.pyplot`.

```
show(): to display a figure
```

Note that the point shown in Fig. 17.1 is hardly visible. The use of color and style comes in handy in such situations. Table 17.1 and Table 17.2 show a few choices for color and style.



**Fig. 17.1** Function `plot(3, 2)` to display point  $(3, 2)$  (see page 583 for the colour image)

**Table 17.1** Color

| Specifier | Color          |
|-----------|----------------|
| b         | Blue (default) |
| g         | Green          |
| r         | Red            |
| c         | Cyan           |
| m         | Magenta        |
| y         | Yellow         |
| k         | Black          |
| w         | White          |

choices for color

**Table 17.2** Point style

| Specifier | Point / Marker style |
|-----------|----------------------|
| .         | Point (default)      |
| o         | Circle               |
| *         | Star                 |
| +         | Plus                 |
| x         | X                    |
| v         | Down triangle        |
| ^         | Up triangle          |
| <         | Left triangle        |
| >         | Right triangle       |
| s         | Square               |
| p         | Pentagon             |
| h         | Hexagon              |

choices for marker style

Let us plot the point  $(3, 2)$  as a circular red-colored point (Fig. 17.2) as follows:

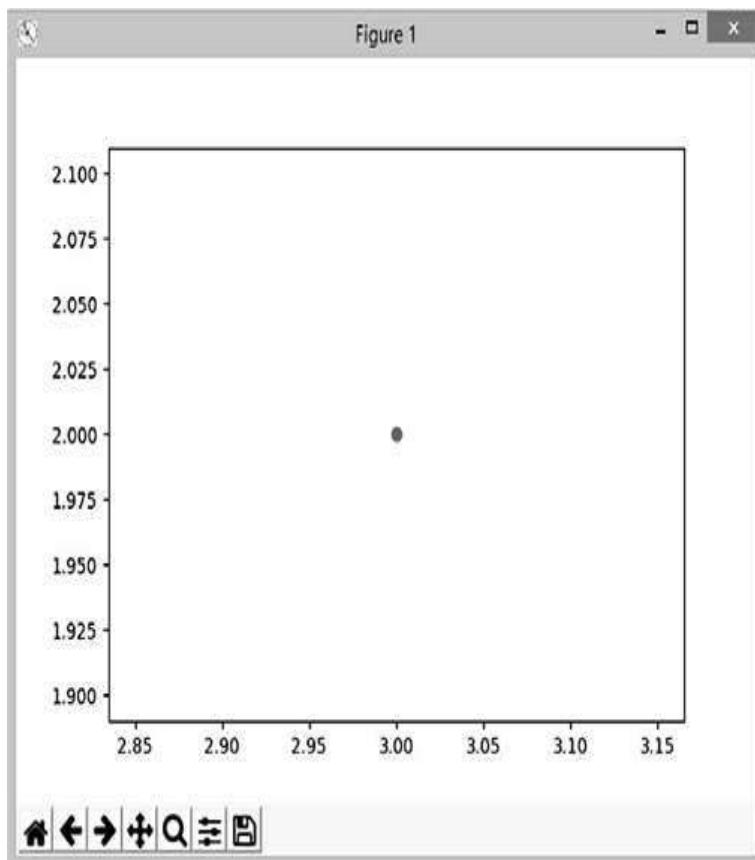
```
plt.plot(3, 2, 'ro')
```

plotting a circular redcolored point

The third argument (format string) is a string that describes a combination of point style and color, which may appear in any order. Thus, above call to function `plot` may also be written as:

```
plt.plot(3, 2, 'or')
```

interactive buttons on the bottom panel for manipulating current view of the figure



**Fig. 17.2** Point (3, 2) (see page 583 for the colour image)

Note that Fig. 17.2 has several interactive buttons on the bottom panel that help in manipulating the current view of the figure. The first button is called the home button and is used to reset original view of the figure. The fourth button is used for controlling the axis by panning the axis with the left button of mouse and zooming with

the right button of the mouse. We may also zoom in the rectangular portion of the current view of the figure using the fifth button. In the case of multiple sub-plots, we may configure them with the sixth button. At any point in time, we may revert to previous view or next view using second and third button respectively. The seventh button is used for saving the figure in the desired directory.

At times, we may be interested in plotting several points on the graph. For example, suppose we wish to plot five points:  $(2, 3)$ ,  $(4, 5)$ ,  $(6, 7)$ ,  $(8, 9)$ ,  $(10, 11)$ . We construct two lists:  $x$  and  $y$  – the list  $x$  comprising x-coordinates of all the points, and the list  $y$  comprising y-coordinates of the corresponding points. As each point comprises an x-coordinate and a y-coordinate, both lists have the same length. Now we are ready to call the `plot` function with arguments  $x$ ,  $y$ , and '`ro`'. The resulting graph is shown in Fig. 17.3.

```
import matplotlib.pyplot as plt

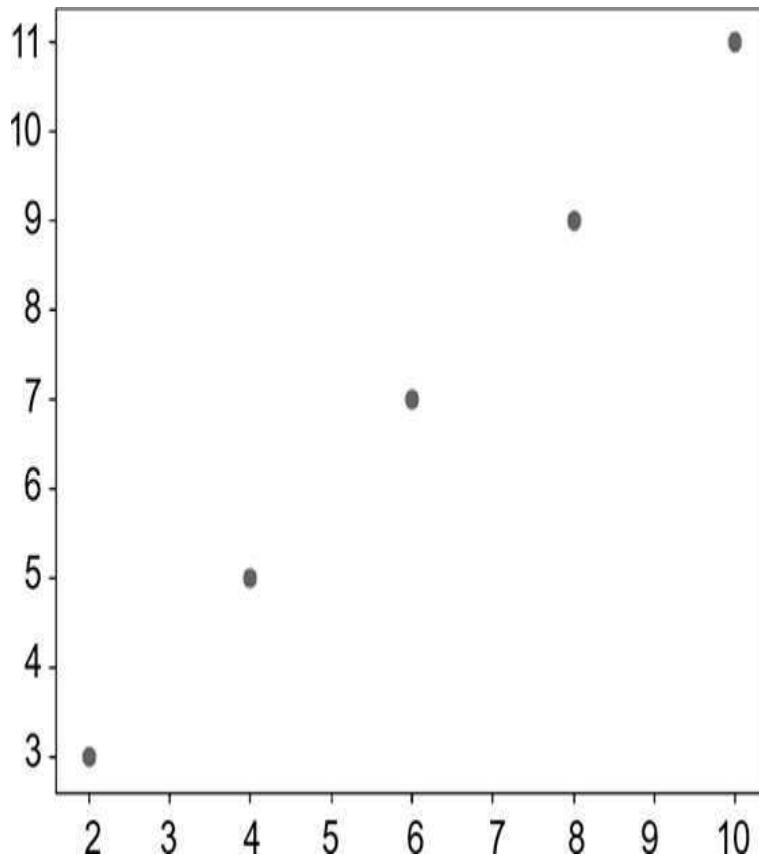
x = [2, 4, 6, 8, 10]

y = [3, 5, 7, 9, 11]

plt.plot(x,y, 'ro')

plt.show()
```

plotting several points on the graph



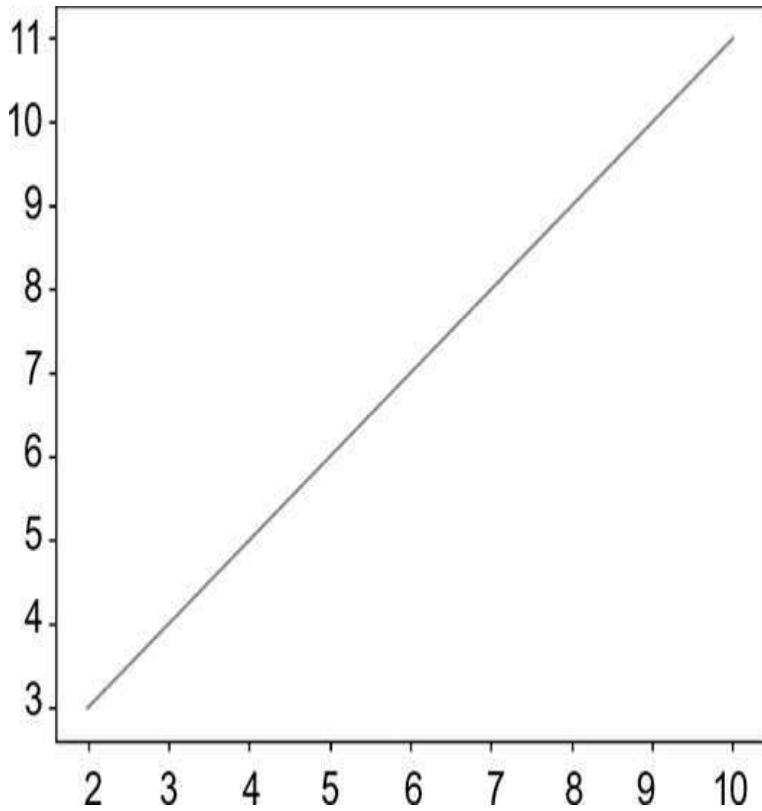
**Fig. 17.3** Multiple points (see page 583 for the colour image)

Often, we are interested in plotting a line connecting a list of points. Indeed, the `plot` function can be used again for this purpose. When we skip the third argument entirely, or just skip the shape of the points to be plotted as part of the third argument, by default, Python joins the points based on input lists `x` and `y`, by line segments, as shown below:

```
plt.plot(x, y)
```

plotting a line connecting coordinates of points specified as lists

On execution of the above instruction, a line joining the points determined by lists  $x$  and  $y$  is displayed.



**Fig. 17.4** Line joining points specified by vector  $x$  and  $y$  (see page 583 for the colour image)

Note that the system displays a solid line in blue color. These are default options for line style and color. Thus, default format string is '`b-`'. We may choose any of the four line styles that appear in Table 17.3. Color options have already been described in Table 17.1.

by default, the line is plotted as a solid line in blue color

default format string: '`b-`'

**Table 17.3** Line style

| Specifier | Line style           |
|-----------|----------------------|
| -         | Solid line (default) |
| --        | Dashed line          |
| :         | Dotted line          |
| -.        | Dash-dot line        |

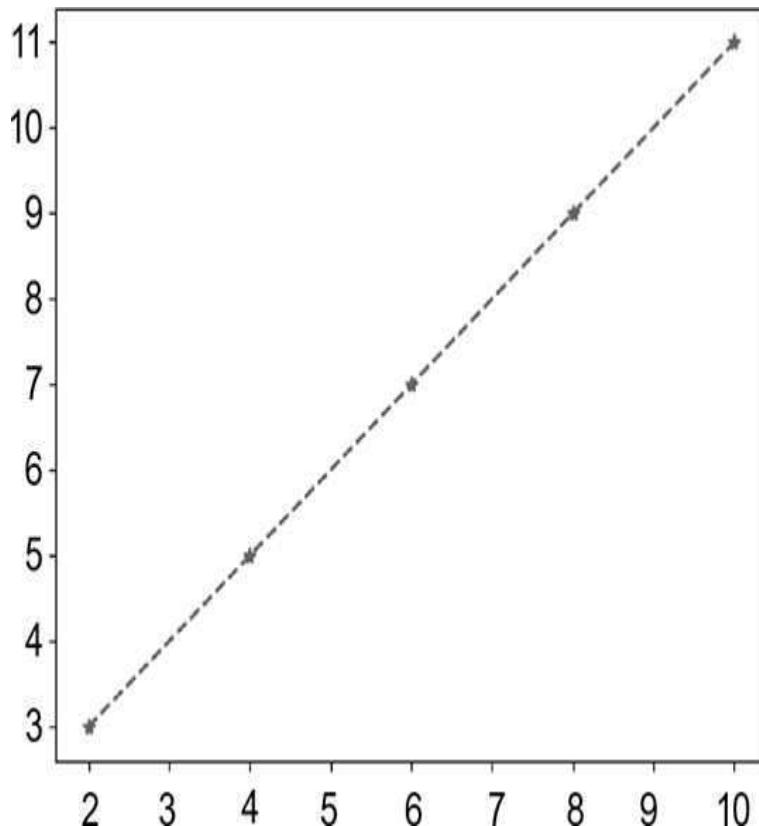
choices for line style

Next, we display a line comprising dashes and asterisks (in red color) (Fig. 17.5).

```
x = [2, 4, 6, 8, 10]
y = [3, 5, 7, 9, 11]

plt.plot(x, y, 'r*--')
plt.show()
```

plotting a line comprising dashes and asterisks (in red color)



**Fig. 17.5** Dashed line joining points (star marker) specified by vector  $x$  and  $y$  (see page 583 for the colour image)

In general, the plot function may be specified as follows:

```
plot(x, y, LineSpecification)
```

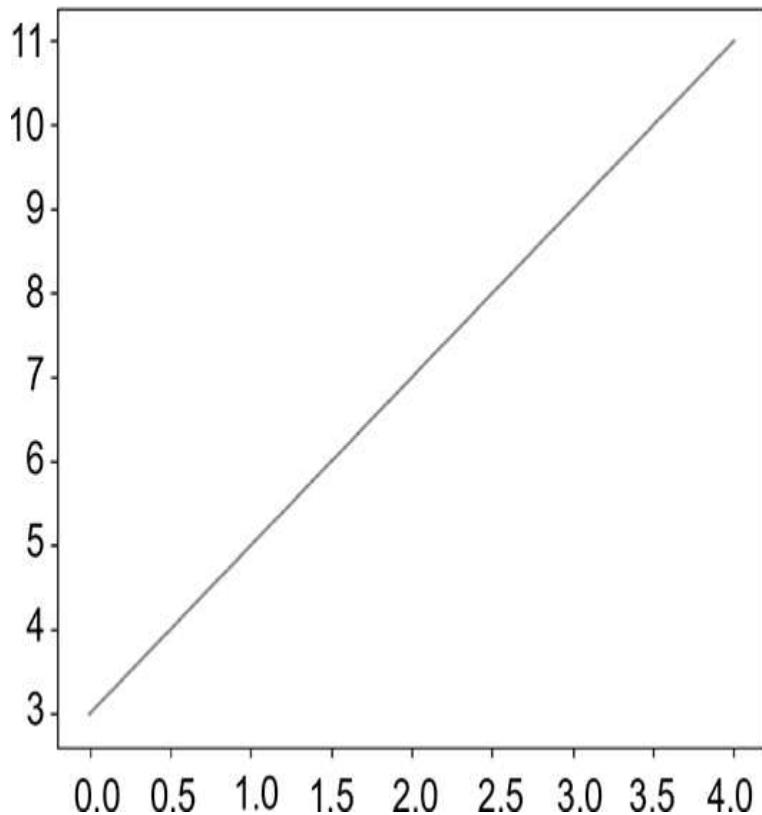
syntax of `plot` function

Note that if only one list is specified as an argument to `plot` function, it assumes x-list to be values in the range  $(0, \text{len}(y))$ . For example, in Fig. 17.6, we show the output on invoking the function `plot(y)`:

```
plt.plot(y)
```

The function `plot` supports several properties that may be set as per the user preferences such as `markerfacecolor`, `markersize` (decimal value),

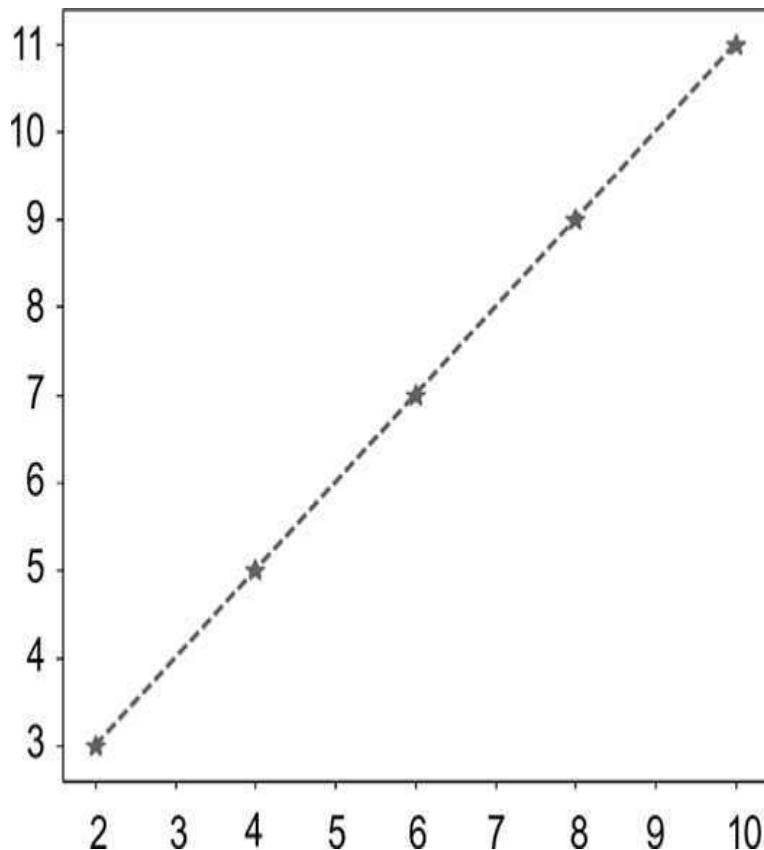
`markeredgecolor`, `markeredgewidth` (decimal value), `linestyle` and `linewidth` (decimal value). For example, to set the width of the line as 2.2 and size of the marker as 10.5, we use the following call to `plot` function (Fig. 17.7):



**Fig. 17.6** Solid line joining points defined by vectors [0, 1, 2, 3, 4] and y (see page 583 for the colour image)

```
plt.plot(x, y, 'r*--', markersize = 10.5,
 linewidth = 2.2)
```

setting `markersize` and `linewidth` of the line



**Fig. 17.7** Line joining points specified by vectors  $x$  and  $y$  (see page 584 for the colour image)

#### *Axis, Title, and Label*

In the above examples, given lists,  $x$  and  $y$ , the system determined the plot area, sufficient for the output graph. However, the use of `axis` function enables us to specify the plot area explicitly through use of the axis parameters  $x_{min}$ ,  $x_{max}$ ,  $y_{min}$ , and  $y_{max}$ :

specifying plot area axis parameters

```
axis ([xmin, xmax, ymin, ymax])
```

For example, we redraw Fig. 17.7 by setting the axis parameters (Fig. 17.8):

```

x = [2, 4, 6, 8, 10]

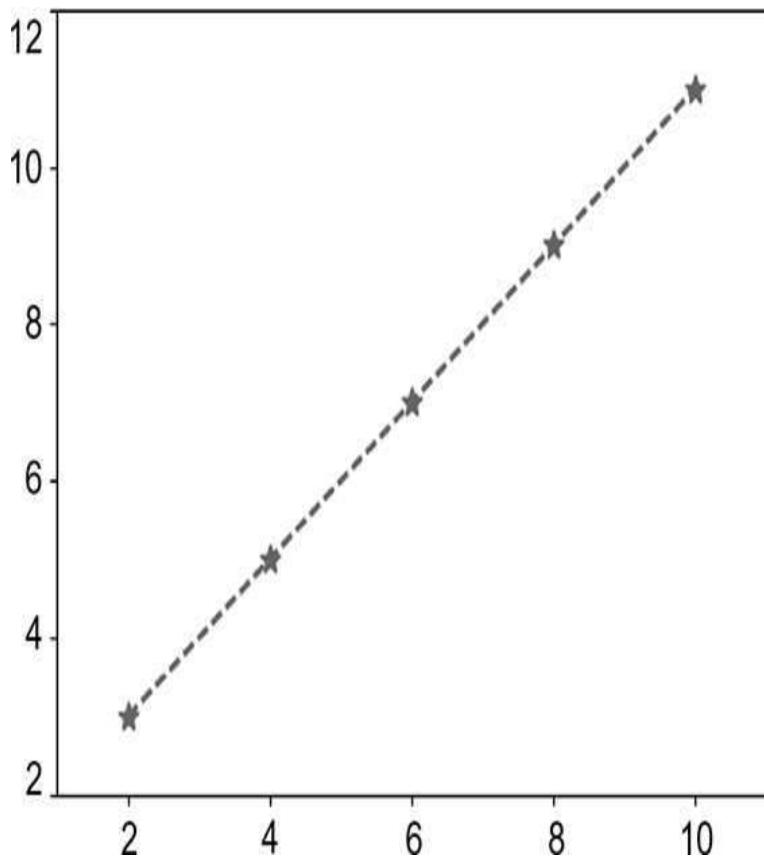
y = [3, 5, 7, 9, 11]

plt.plot(x, y, 'r*--', markersize = 10.5,
linewidth = 2.2)

plt.axis([1, 11, 2, 12])

plt.show()

```



**Fig. 17.8** Line joining points specified by vectors  $x$  and  $y$  (see page 584 for the colour image)

To make the graph easy to understand, we label the axes and assign a title to the graph. Functions `xlabel`, `ylabel`, and `title` may be used for this purpose, as illustrated below:

```
plt.xlabel('X')
plt.ylabel('X * X')
plt.title('X vs X * X')
```

specifying *x* and *y* labels

*specifying figure title* labels

The appearance of a grid in the background makes it easy to read the coordinates of the points in a graph. To display a grid, we need to invoke the function `grid` (Fig. 17.9):

```
plt.grid()
```

displaying a grid

### *Plotting Multiple Functions in the Same Figure*

Sometimes, we wish to plot graphs of two or more functions in the same figure. For example, we may like to compare two graphs. For this purpose, we may make repeated calls to `plot` function as illustrated below:

```
plot (X1, Y1, LineSpec1)
```

```
plot (X2, Y2, LineSpec2)
```

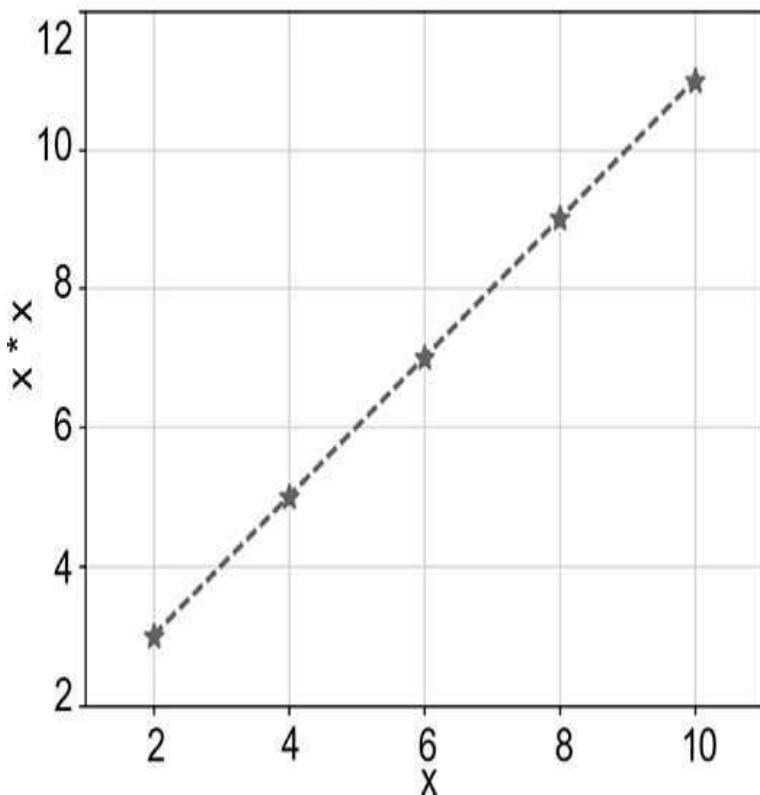
.

.

.

```
plot (Xn, Yn, LineSpecn)
```

X vs X \* X



**Fig. 17.9** Line joining points specified by vectors  $x$  and  $y$  (see page 584 for the colour image)

Alternatively, several functions may be plotted using a single call to `plot` function as shown below:

```
plot (X1, Y1, LineSpec1, X2, Y2, LineSpec2,
..., Xn, Yn, LineSpecn)
```

syntax for plotting multiple functions in the same graph

For example, let us plot functions  $f(x) = x^2$  and  $f(x) = x^3$  in the same figure in the interval  $[a, b]$  in steps of  $\text{step}$ . When we display more than one graph in the same figure, we need a mechanism to distinguish between them. For this purpose, we make use of different colors, width, and style, or a combination of these as illustrated in the function `plotFunctions`

(Fig. 17.10). In the script `plotLines1`, we have chosen colors red and blue for plotting the functions  $x^{**2}$  and  $x^{**3}$  respectively. Legends associated with different plot functions are specified using the keyword `label` while invoking the `plot` function (lines 13 and 14). Finally, we invoke function `legend` for displaying the legends for the two functions being plotted (line 15).

keyword `label` is used for specifying legend

```
01 import matplotlib.pyplot as plt
02 def plotFunctions(a,b, step):
03 """
04 Objective: To plot functions $f(x) = x^{**2}$ and $f(x) = x^{**3}$ in
05 the same figure
06 Input Parameters: a, b, step - numeric value
07 Return Value: None
08 """
09 nSteps = int((b-a)/step)
10 x = [a + step * i for i in range(nSteps+1)]
11 y1 = [t**2 for t in x]
```

```
12 y2 = [t**3 for t in x]
13 plt.plot(x, y1, 'ro--', label = 'X vs X**2')
14 plt.plot(x, y2, 'b<-.', label = 'X vs X**3')
15 plt.legend()
16 plt.xlabel('X')
```

```

17 plt.ylabel('I')
18 plt.title('X vs X**2 and X vs X**3')
19 plt.grid()
20 plt.show()
21
22 def main():
23 """
24 Objective: To plot two functions on same graph based on user
25 input
26 Input Parameter: None
27 Return Value: None
28 """
29 a = float(input('Enter first element of the range: '))
30 b = float(input('Enter last element of the range: '))
31 step = float(input('Enter step size: '))
32 plotFunctions(a, b, step)
33
34 if __name__ == '__main__':
35 main()

```

**Fig. 17.10** Program to plot functions  $f(x) = x^2$  and  $f(x) = x^3$  on the graph (plotLines1.py)

legend() : for displaying the legends

On executing the script plotLines1 (Fig. 17.10), the user is asked to enter the range parameters and the step size:

```
Enter first element of the range: 2.5
```

```
Enter last element of the range: 4.5
```

```
Enter step size: 0.1
```

On entering the range parameters 2.5 and 4.5, Python responds with [Fig. 17.11](#).

### *Multiple Plots*

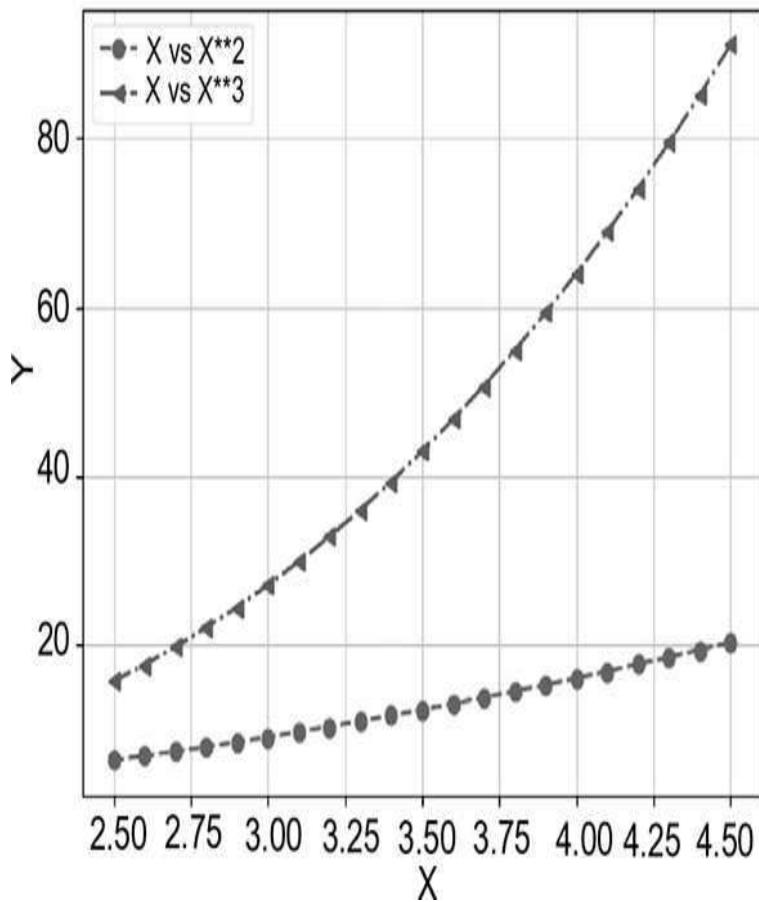
Suppose we wish to plot several functions, each in a separate graph, but in the same figure. The function `subplot` described below may be used for this purpose.

`subplot()` : plotting in sub-graphs

syntax for function `subplot`

`subplot(rowNum, colNum, FigNum)`

### X vs X\*\*2 and X vs X\*\*3



**Fig. 17.11** Functions  $f(x) = x^2$  and  $f(x) = x^3$  plotted in the interval  $[2.5, 4.5]$  in steps of 0.1 (see page 584 for the colour image)

The function takes three arguments, the number of rows, the number of columns, and the figure number. Instead of a comma-separated sequence of parameters, we may specify an ordered sequence of these values, for example, we may use either of the function calls `subplot(1, 2, 1)` and `subplot(121)`. In the script `plotLines2` (Fig. 17.12), we plot six functions  $f_1(x) = x$ ,  $f_2(x) = x^2$ ,  $f_3(x) = x^3$ ,  $f_4(x) = x^4$ ,  $f_5(x) = x^5$ , and  $f_6(x) = x^6$  as six different graphs (two rows and three columns) but in the same figure (Fig. 17.13). Note that the call to the function `plot` in line 20 plots the graph in the subplot mentioned in line 19. On executing the script `plotLines2` (Fig. 17.12), Python prompts the user to enter the list `x` to be plotted:

```
01 import matplotlib.pyplot as plt
02 def plotFunctions(x):
03 """
04 Objective: To plot functions f(x) = x,
05 f(x) = x**2, f(x) = x**3,
06 f(x) = x**4, f(x) = x**5, and
07 f(x) = x**6 in different plots on
08 same graph
09 Input Parameter: x - list
10 Return Value: None
11 """
12 y1 = [i**1 for i in x]
13 y2 = [i**2 for i in x]
14 y3 = [i**3 for i in x]
15 y4 = [i**4 for i in x]
```

```
16 y5 = [i**5 for i in x]
17 y6 = [i**6 for i in x]
18
19 plt.subplot(2,3,1)
20 plt.plot(x, y1, 'ro-')
21 plt.xlabel('X')
22 plt.ylabel('X')
23 plt.title('X vs X')
24 plt.grid()
25
26 plt.subplot(2,3,2)
27 plt.plot(x, y2, 'b<--')
28 plt.xlabel('X')
29 plt.ylabel('X**2')
30 plt.title('X vs X**2')
```

```
31 plt.grid()
32
33 plt.subplot(2,3,3)
34 plt.plot(x, y3, 'g*-.')
35 plt.xlabel('X')
36 plt.ylabel('X**3')
37 plt.title('X vs X**3')
38 plt.grid()
39
40 plt.subplot(2,3,4)
41 plt.plot(x, y4, 'kx:')
42 plt.xlabel('X')
43 plt.ylabel('X**4')
44 plt.title('X vs X**4')
45 plt.grid()
46
47 plt.subplot(2,3,5)
48 plt.plot(x, y5, 'mv-')
49 plt.xlabel('X')
50 plt.ylabel('X**5')
51 plt.title('X vs X**5')
52 plt.grid()
53
54 plt.subplot(2,3,6)
55 plt.plot(x, y6, 'cp-.')
56 plt.xlabel('X')
57 plt.ylabel('X**6')
```

```
58 plt.title('X vs X**6')
59 plt.grid()
60
61 plt.tight_layout()
62 plt.show()
63
64 def main():
65 """
66 Objective: To plot six functions in different plots on
67 same graph based on user input
68 Input Parameter: None
69 Return Value: None
70 """
71 x = eval(input('Enter list x to be plotted: '))
72 plotFunctions(x)
73
74 if __name__ == '__main__':
75 main()
```

**Fig. 17.12** Program to plot six functions on different subgraphs in the same figure (`plotLines2.py`)

```
Enter list x to be plotted: range(1,15)
```

On providing the input `range(1,15)`, Python outputs Fig. 17.13. All the list sequences that are provided to functions of `matplotlib.pyplot` are internally converted to `numpy arrays` (homogeneous multidimensional arrays). Also, to add sufficient spacing

between subplots so that they do not overlap, function `tight_layout` has been invoked in line 61.

```
tight_layout() : to add sufficient spacing between subplots
```

### *Saving Figure*

Suppose, we wish to plot a graph between  $x$  and  $x^2$ , and save it in a file for future reference, thus, avoiding the need to re-run the code every time such a figure is needed. To save a graph in the current directory, we use the function `savefig`, for example,

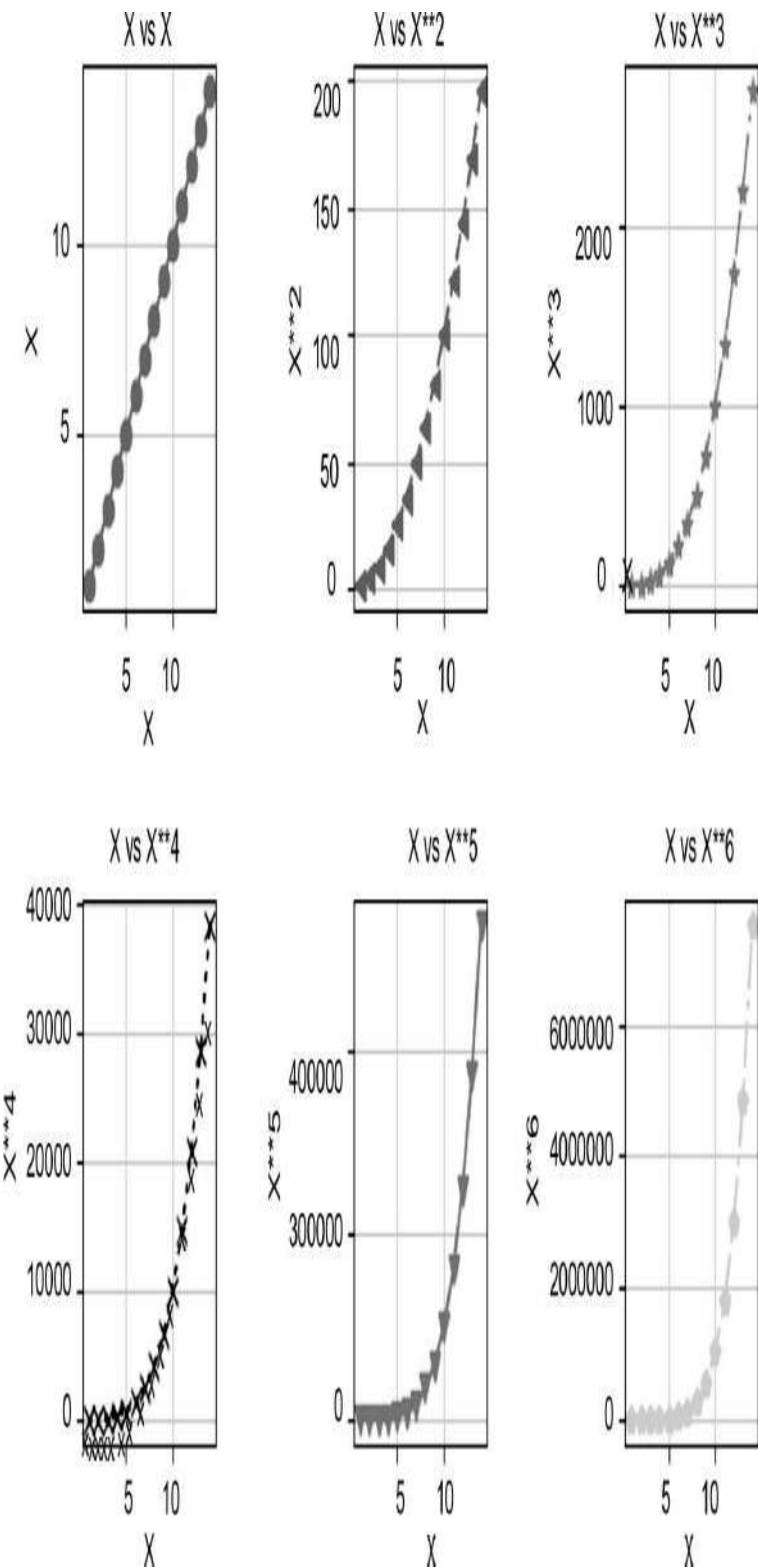
```
savefig() : to save a graph in the current directory
```

```
x = range(0, 5)

y = [i**2 for i in x]

plt.plot(x, y, label = 'X vs X**2')

plt.savefig('xSquare')
```



**Fig. 17.13** Six functions plotted on different subgraphs in same figure (see page 585 for the colour image)

### 17.1.2 Histogram and Pi Chart

A histogram is used to represent the frequency of various values in a data set using bars, also known as bins. The function `hist` is used for plotting the histogram for a given data set as illustrated in the script `histogram` (Fig. 17.14). In line 13, we define the limits of x-axis using the function `xlim`. Next, we execute the script `histogram` (Fig. 17.14) and Python outputs a histogram (Fig. 17.15) for the list [1,1,4,4,1,2,2,3,3,3,4,4,4,6] :

```
hist(): to plot histogram for a given data set
```

```
xlim(): to define the limits of x-axis
```

```
Enter data to be plotted as histogram:
[1,1,4,4,1,2, 2,3,3,3,4,4,4,6]
```

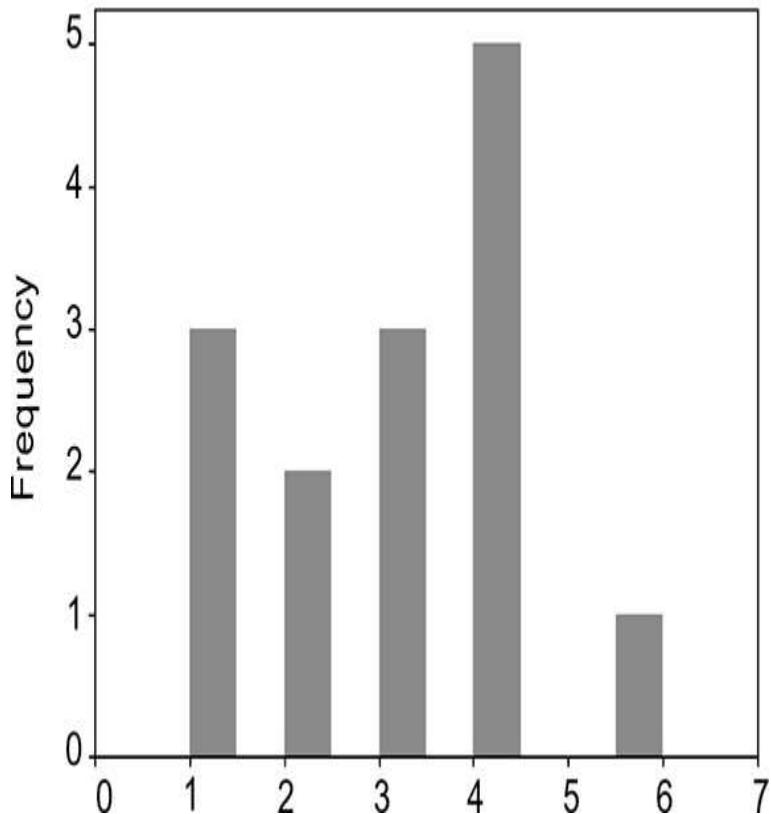
By default, the method `hist` assumes the maximum number of bins to be 10. If we wish to have more than 10 bins, we need to set the number of bins to the desired value explicitly, say `n`, using keyword argument `bins = n` along with input data. Also, as the bins of the histogram are not centred around values (Fig. 17.15), we may prefer to specify the range (boundaries) for each bin of the histogram. For example, the following statement defines the bins with boundaries 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, and 6.5 which creates six bins as [0.5,1.5), [1.5,2.5), [2.5,3.5), [3.5,4.5), [4.5,5.5), and [5.5,6.5].



```
01 import matplotlib.pyplot as plt
02 def plotHistogram(data):
03 """
04 Objective: To plot a histogram
05 Input Parameter: data - list
06 Return Value: None
07 """
08 plt.hist(data)
09 plt.xlabel('Value')
10 plt.ylabel('Frequency')
11 plt.title('Histogram')
12 #For setting limits of x axis
13 plt.xlim(min(data)-1, max(data)+1)
14 plt.show()
15
16 def main():
17 """
18 Objective: To plot a histogram based on user input
19 Input Parameter: None
20 Return Value: None
21 """
22 data = eval(input('Enter data to be plotted as histogram: '))
23 plotHistogram(data)
24
25 if __name__ == '__main__':
26 main()
```

```
histogram.py
```

**Fig. 17.14** Program to plot histogram (`histogram.py`)

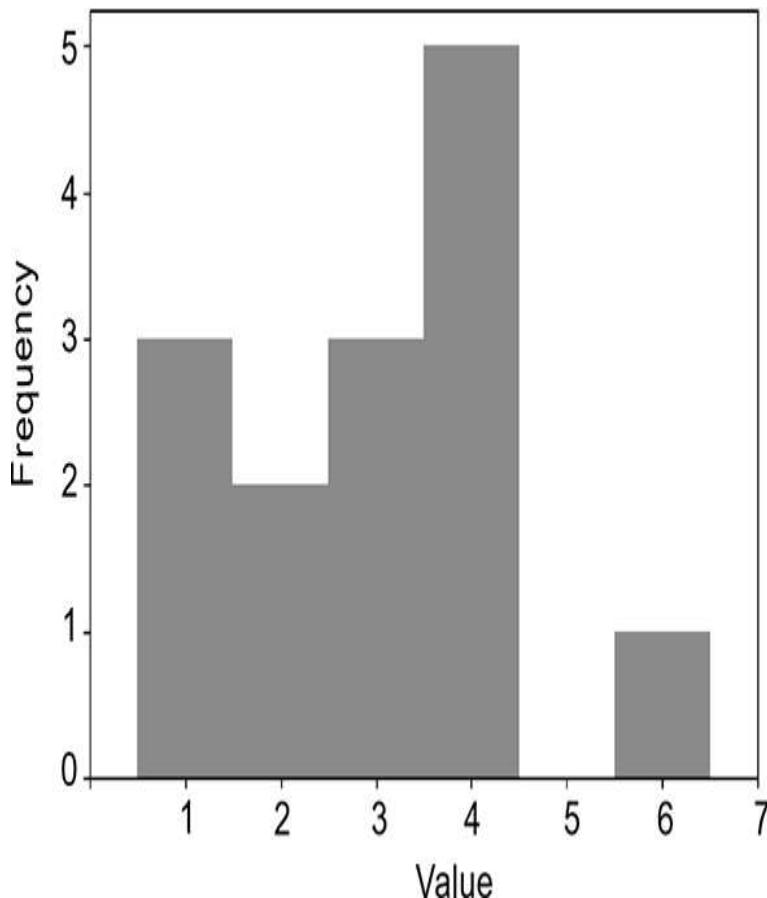


**Fig. 17.15** Histogram (see page 585 for the colour image)

```
plt.hist(data,bins = [i-0.5 for i in \
range(0,max(data)+2)])
```

use of keyword argument `bins` for specifying bin boundaries

On replacing line 8 by the above statement in the script `histogram` (Fig. 17.15), we get the desired histogram (Fig. 17.16).



**Fig. 17.16** Histogram (see page 585 for the colour image)

A *pie* chart is a circular representation of data under different categories. A circle is divided into sectors, also called wedges. A sector denotes the proportion of data that falls in the corresponding category. The function `pie` is used for creating a pie chart for the given sequence of data. The function `pie` requires two arguments: a sequence `x` of numeric values and a sequence of labels to be assigned to the wedges (Fig. 17.17). For plotting the *pie* chart; the function computes the proportion of each data value `x` of the sequence `x` as `x/sum(x)`. Sometimes, we need to display the percent area covered by each wedge. For this purpose, we use the argument `autopct`. In the script `piechart` (Fig. 17.17), we have set formatted percentage string to contain two digits after the decimal point.

`pie()`: to create a pie chart for the given sequence of data  
node

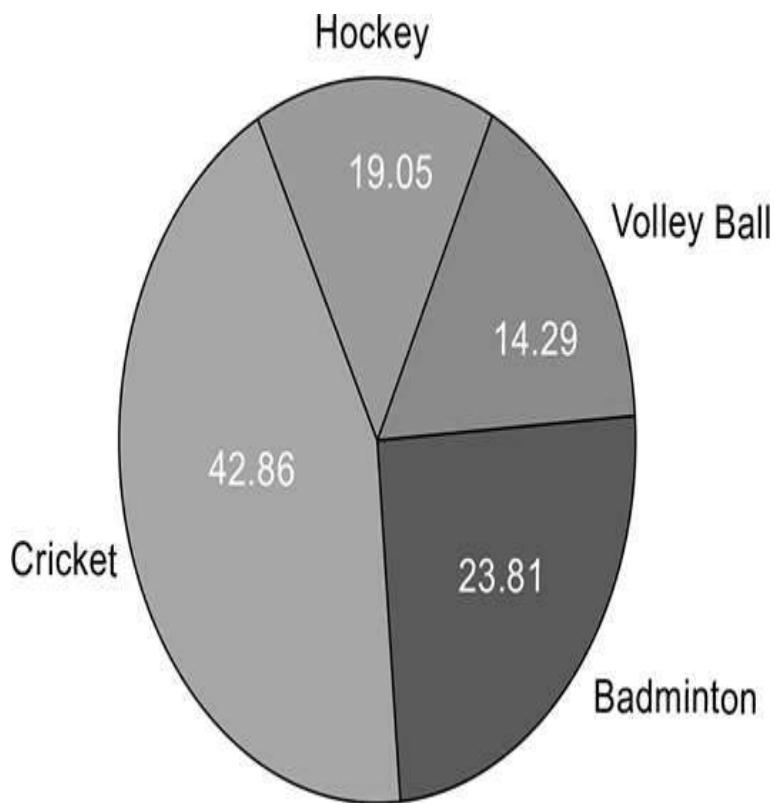
`autopct`: keyword argument to specify the percent area  
covered by each wedge

On executing the script `piechart` ([Fig. 17.17](#)),  
Python prompted with following inputs and displayed pie  
chart as shown in [Fig. 17.18](#).

```
01 import matplotlib.pyplot as plt
02 def plotPieChart(data, labels):
03 """
04 Objective: To plot a pie chart
05 Input Parameters: data, labels - list
06 Return Value: None
07 """
08 plt.pie(data, labels = labels, autopct='%.2f')
09 plt.title('Pie Chart')
10 plt.show()
11
12 def main():
13 """
14 Objective: To plot pie chart based on user input
15 Input Parameter: None
16 Return Value: None
17 """
18 data = eval(input('Enter data to be plotted \\'
```

```
19 as pie chart: '))
20 labels = eval(input('Enter the labels: '))
21 plotPieChart(data, labels)
22
23 if __name__ == '__main__':
24 main()
```

**Fig. 17.17** Program to plot pie-chart (piechart.py)



**Fig. 17.18** Pie chart (see page 586 for the colour image)

Enter data to be plotted as pie chart:  
[30, 40, 90, 50]

Enter the labels: ['VolleyBall', 'Hockey',  
'Cricket', 'Badminton']

### 17.1.3 Sine and Cosine Curves

In this section, we develop two functions `sineCurve` and `cosineCurve` to plot sine and cosine curve in the range  $0^\circ$  to  $360^\circ$  (Fig. 17.19). We determine sine values in the range  $0^\circ$  to  $360^\circ$  using sine function of `math` module. The sine values so obtained are then plotted using `plot` function (`sineCurve`). The cosine values are plotted similarly in the function `cosineCurve`. The curves obtained on executing the script `curve` are shown in Fig.17.20.

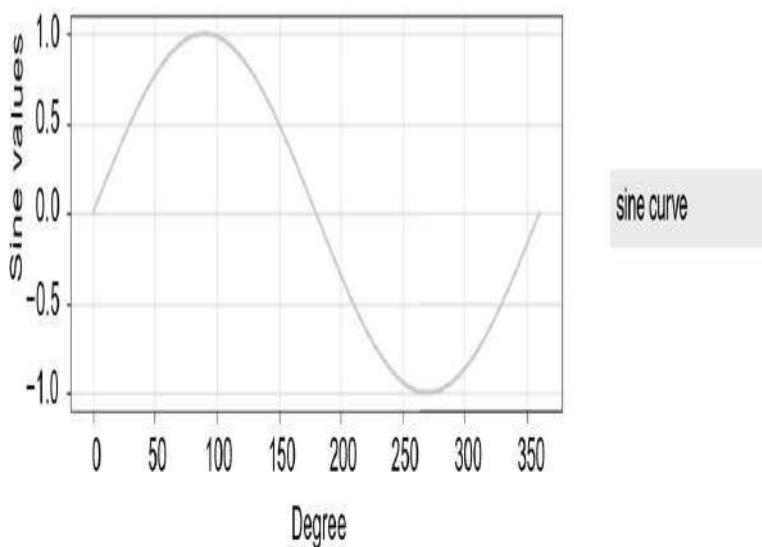
plotting sine and cosine curves

```
01 import matplotlib.pyplot as plt
02 import math
03 def sineCurve():
04 """
05 Objective: To plot sine function
06 Input Parameter: None
07 Return Value: None
08 """
09 plt.subplot(2,1,1)
10 degrees = range(0, 360 + 1)
11 sineValues = [math.sin(math.radians(i)) for i in degrees]
12 plt.plot(sineValues)
13 plt.xlabel('Degree')
14 plt.ylabel('Sine Values')
15 plt.title('Sine Curve')
16 plt.grid()
17
18 def cosineCurve():
19 """
20 Objective: To plot cosine function
21 Input Parameter: None
22 Return Value: None
```

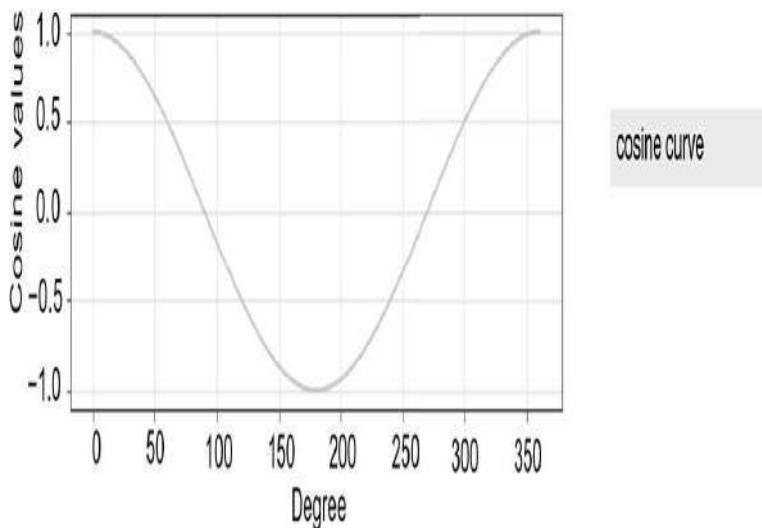
```
23 ''
24 plt.subplot(2,1,2)
25 degrees = range(0, 360 + 1)
26 cosineValues = [math.cos(math.radians(i)) for i in degrees]
27 plt.plot(cosineValues)
28 plt.xlabel('Degree')
29 plt.ylabel('Cosine Values')
30 plt.title('Cosine Curve')
31 plt.grid()
32
33 def main():
34 ''
```

```
35 Objective: To plot sine and cosine curves
36 Input Parameter: None
37 Return Value: None
38 ''
39 sineCurve()
40 cosineCurve()
41 plt.tight_layout()
42 plt.show()
43
44 if __name__ == '__main__':
45 main()
```

**Fig. 17.19** Program to plot sine and cosine curve (curve.py)



sine curve



cosine curve

**Fig. 17.20** Sine and cosine curve (see page 586 for the colour image)

#### 17.1.4 Graphical Objects: Circle, Ellipse, Rectangle, Polygon, and Arrow

To support the classes such as `Rectangle`, `Circle`, `Ellipse`, `RegularPolygon`, `Polygon`, `CirclePolygon`, `Rectangle`, `Arrow`, and `FancyArrow`, `matplotlib` library provides `patches` module. Consequently, the graphical shapes such as rectangle, circle, ellipse, regular polygon, polygon, circle polygon, rectangle, arrow, and fancy arrow are called patches. All these classes are sub-classes of the class

`Patch` which further inherits from the class `Artist` of the module `artist`.

`patches`: the module provides support for graphical objects

Drawing a graphical object requires creating the object of a particular shape and adding that patch to the current figure's axis using the function `add_patch`. We can obtain current figure's axis using the function `gca` (get current axis). When the axis is set to 'scaled', the system chooses a suitable scale for displaying the figure.

In Table 17.4, we describe some of the properties of the graphical objects that they inherit from class `Patch`. These properties can be set to suitable values for improving the visual effectiveness of a figure.

**Table 17.4** Patch class properties

| Property          | Description                                                                                                       | Default value |
|-------------------|-------------------------------------------------------------------------------------------------------------------|---------------|
| edgecolor or ec   | any matplotlib color                                                                                              | blue          |
| facecolor or fc   | any matplotlib color                                                                                              | blue          |
| color             | any matplotlib color                                                                                              | blue          |
| linewidth or lw   | decimal value                                                                                                     | 0.5           |
| angle             | rotation in anticlockwise direction in degrees                                                                    | 0             |
| linestyle         | valid values: ('solid', 'dashed', 'dashdot', 'dotted')                                                            | 'solid'       |
| antialiased or aa | valid values: (True, False)                                                                                       | True          |
| hatch             | sets a pattern for filling a graphical object<br>valid values: ('/', '\', ' ', '-', '+', 'x', 'o', 'O', '.', '*') | None          |
| fill              | valid values: (True, False)                                                                                       | True          |

properties of `Patch` class

### *Circle*

Creating an instance of the class `Circle` requires a tuple `(x, y)` denoting the centre of the circle and an optional

value for the parameter `radius` (default 5). Further, we may set different properties of the class `Patch` while instantiating the class `Circle`. In the script `circle` (Fig. 17.21), we draw a circle on the plot area. The circle is centred at  $(0, 0)$  and has the user-specified `radius`. Further, `facecolor`, `edgecolor`, `linestyle`, and `linewidth` have been set to `green`, `red`, `dotted`, and `2.2`, respectively.

```
Circle: the class used to plot a circle
```

```
default value of radius: 5
```

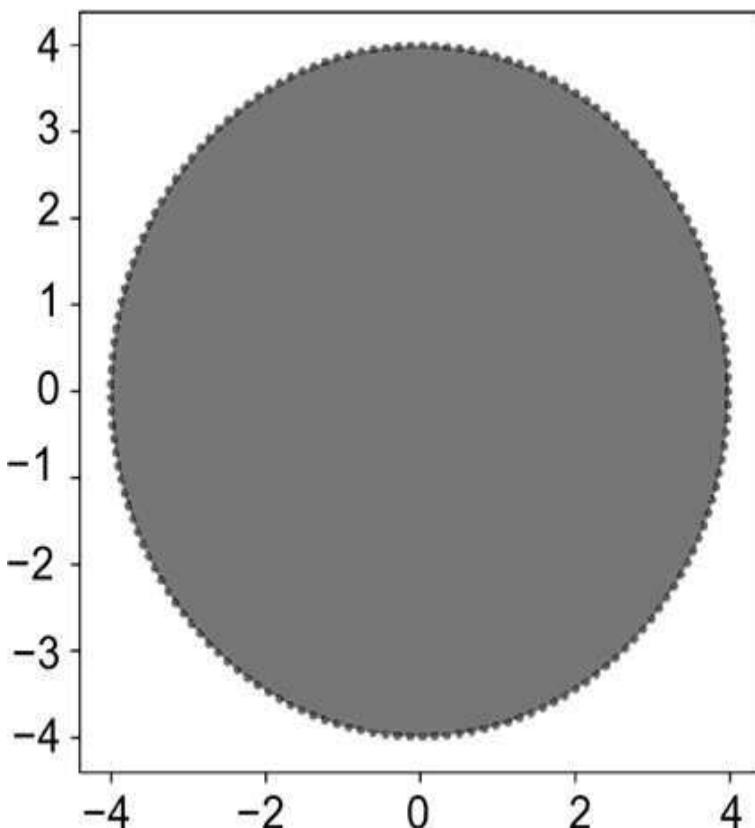
```
01 import matplotlib.pyplot as plt
02 import matplotlib.patches as patches
03 def plotCircle(radius):
04 """
05 Objective: To plot a circle of given radius
06 Input Parameter: radius - numeric
07 Return Value: None
08 """
09 circle = patches.Circle((0, 0), radius, facecolor = 'green'\
10 ,edgecolor = 'red', linestyle = 'dotted', linewidth ='2.2 ')
11 plt.gca().add_patch(circle)
12 plt.axis('scaled')
13 plt.title('Circle')
14 plt.show()
15
```

```
16 def main():
17 """
18 Objective: To plot circle based on input radius
19 Input Parameter: None
20 Return Value: None
21 """
22 radius = float(input('Enter the radius: '))
23 plotCircle(radius)
24
25 if __name__ == '__main__':
26 main()
```

**Fig. 17.21** Program to draw a circle (`circle.py`)

On executing the script `circle` (Fig. 17.21), Python prompts the user to enter the radius and displays circle (Fig. 17.22).

```
Enter the radius: 4
```



**Fig. 17.22** Circle (see page 586 for the colour image)

### *Ellipse*

Creating an instance of the `Ellipse` class takes width and height as the required attributes and an optional value for the parameter angle (default 0.0) denoting rotation in degrees (anti-clockwise). As usual, we may set different properties of the class `Patch` while instantiating the class `Ellipse`. In the script `ellipse` (Fig. 17.23), we draw an ellipse on the plot area. The ellipse is centred at (0, 0) and has user-specified width and height. Further, `fc`, `ec`, `linestyle`, and `lw` have been set to `cyan`, `red`, `dashed`, and `2.2`, respectively.

`Ellipse`: the class used to plot an ellipse

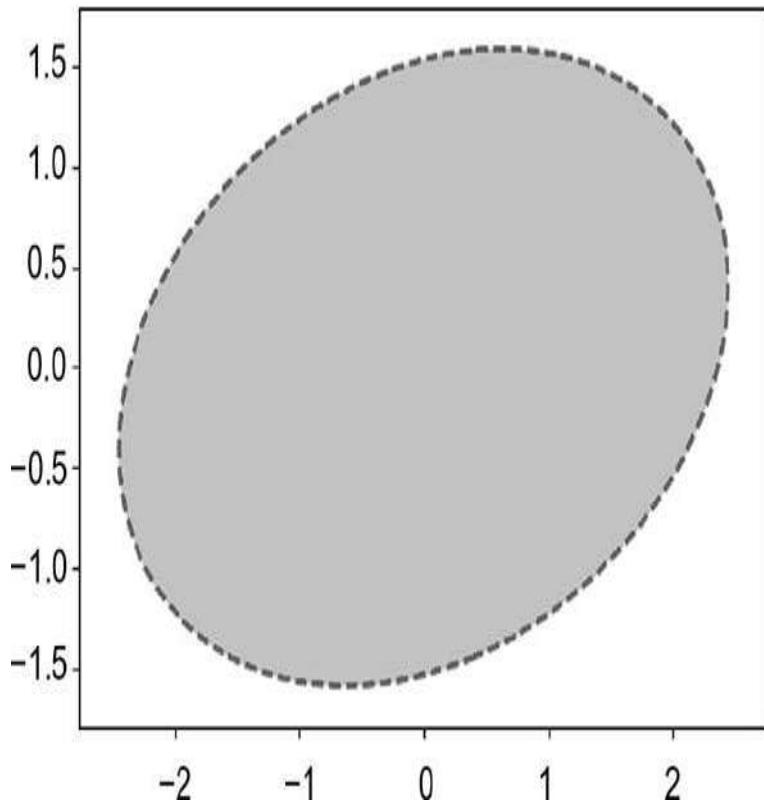
```
01 import matplotlib.pyplot as plt
02 import matplotlib.patches as patches
03 def plotEllipse(width, height):
04 """
05 Objective: To plot an ellipse of given width and height
06 Input Parameters: width, height - numeric
07 Return Value: None
08 """
09 ellipse = patches.Ellipse((0, 0), width, height, angle = 15,\n10 fc = 'cyan', ec = 'red', linestyle = 'dashed', lw = '2.2')
11 plt.gca().add_patch(ellipse)
12 plt.axis('scaled')
13 plt.title('Ellipse')
14 plt.show()
15
16 def main():
17 """
18 Objective: To plot ellipse based on user input
19 Input Parameter: None
20 Return Value: None
21 """
22 width = float(input('Enter the width: '))
23 height = float(input('Enter the height: '))
24 plotEllipse(width, height)
25
26 if __name__ == '__main__':
27 main()
```

**Fig. 17.23** Program to draw an ellipse (`ellipse.py`)

On executing the script `ellipse` (Fig. 17.23), Python prompts the user to enter the width and the height. On entering values 5 and 3 for width and height, respectively, an ellipse is displayed (Fig. 17.24).

```
Enter the width: 5
```

```
Enter the height: 3
```



**Fig. 17.24** Ellipse (see page 586 for the colour image)

### *Rectangle*

For creating an instance of `Rectangle` class, we specify width and height as the required attributes. We may also specify the optional parameter `angle` (default 0.0) denoting rotation in degrees (anti-clockwise). In the script `rectangle` (Fig. 17.25), we draw a rectangle on the plot area. The rectangle is centred at (0, 0) with user-specified `length` and `breadth`. As usual, we set some of the properties of the class `Patch` while

instantiating the class `Rectangle`. Default values are used for other properties.

`Rectangle`: the class used to plot a rectangle

```
01 import matplotlib.pyplot as plt
02 import matplotlib.patches as patches
03 def plotRectangle(length, breadth):
04 """
05 Objective: To plot a rectangle of length and breadth
06 Input Parameters: length, breadth - numeric
07 Return Value: None
08 """
09 rect = patches.Rectangle((0, 0), length, breadth)
10 plt.gca().add_patch(rect)
11 plt.axis('scaled')
12 plt.title('Rectangle')
13 plt.show()
14
15 def main():
16 """
17 Objective: To plot a rectangle based on user input
18 Input Parameter: None
19 Return Value: None
20 """
```

```
21 length = float(input('Enter the length: '))
22 breadth = float(input('Enter the breadth: '))
23 plotRectangle(length, breadth)
24
```

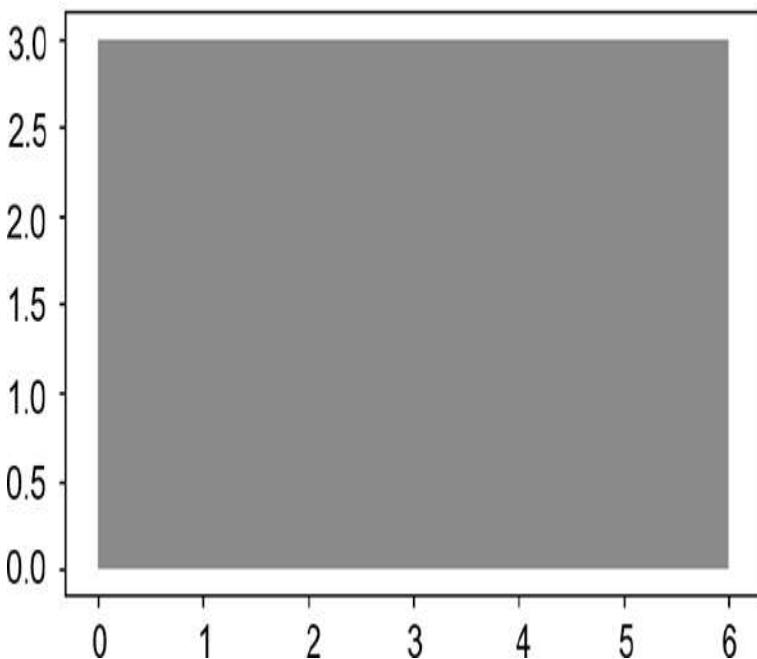
```
25 if __name__ == '__main__':
26 main()
```

**Fig. 17.25** Program to draw a rectangle (`rectangle.py`)

On executing the script `rectangle` (Fig. 17.25), and entering 6 and 3 as the values of length and breadth, respectively, a rectangle is displayed (Fig. 17.26).

Enter the length: 6

Enter the breadth: 3



**Fig. 17.26** Rectangle (see page 587 for the colour image)

### *Polygon*

Creating an instance of `Polygon` class requires a list (vector) of points. Each point in the list denotes one of the endpoints of the line segments of the polygon. As usual, we may set different properties of the class `Patch` while instantiating the class `Polygon`. In the script `polygon` (Fig. 17.27), we draw a polygon on the plot area that joins the user-specified points having `ec`,

`linestyle`, and `lw` set to `red`, `dashdot`, and `4`, respectively.

`Polygon`: the class used to plot a polygon

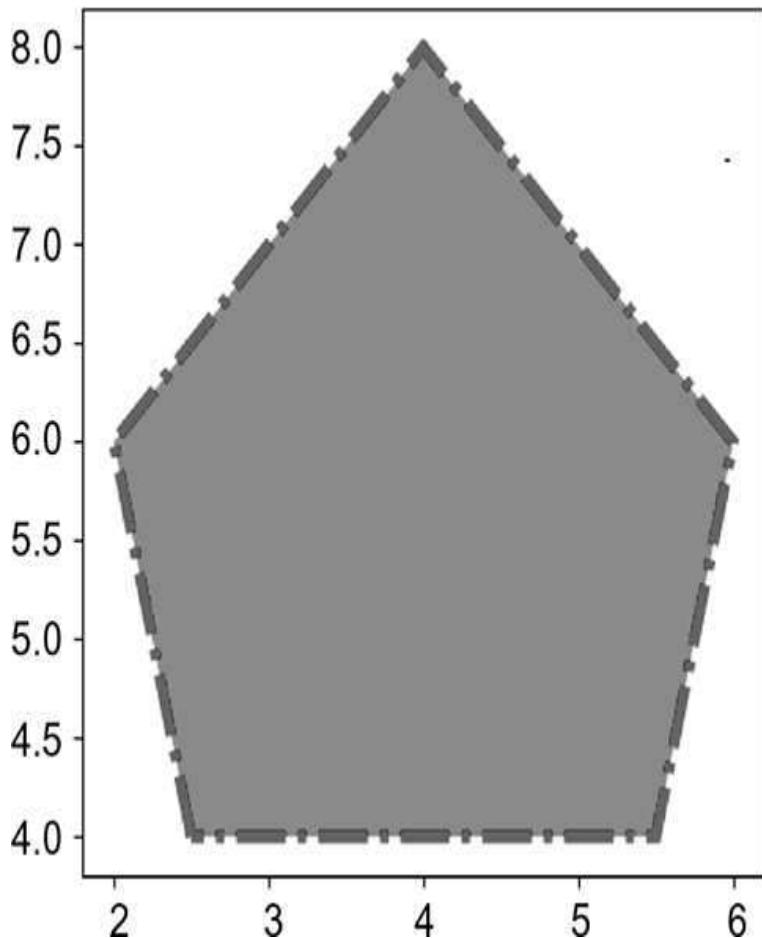
On executing the script `polygon` (Fig. 17.27), Python prompts the user to enter the list of endpoints of a polygon. On entering the input `[[2.5, 4], [2, 6], [4, 8], [6, 6], [5.5, 4]]`, a polygon is displayed (Fig. 17.28).

```
Enter the points: [[2.5,4], [2,6], [4,8],
[6,6], [5.5,4]]
```



```
01 import matplotlib.pyplot as plt
02 import matplotlib.patches as patches
03 def plotPolygon(points):
04 """
05 Objective: To plot a polygon joining given a list of points
06 Input Parameter: points - list
07 Return Value: None
08 """
09 poly = patches.Polygon(points, ec = 'red', linestyle = \
10 'dashdot', lw = '4')
11 plt.gca().add_patch(poly)
12 plt.axis('scaled')
13 plt.title('Polygon')
14 plt.show()
15
16 def main():
17 """
18 Objective: To plot a polygon based on user input
19 Input Parameter: None
20 Return Value: None
21 """
22 points = eval(input('Enter the points: '))
23 plotPolygon(points)
24
25 if __name__ == '__main__':
26 main()
```

**Fig. 17.27** Program to draw a polygon (`polygon.py`)



**Fig. 17.28** Polygon (see page 587 for the colour image)

### Arrow

Creating an instance of `Arrow` class requires attributes `x`, `y`, `dx`, and `dy`. Whereas  $(x, y)$  denotes the starting point (base) of the arrow, and  $(dx, dy)$  defines the position of the head of the arrow relative to its base. As usual, we may set different properties of the class `Patch` while instantiating the class `Arrow`. In the script `arrow` (Fig. 17.29), we draw an arrow that begins from a user specified position  $(x, y)$  having coordinates of head of the arrow as  $(x + dx, y + dy)$ . We set the `fill` property to `False` so that the arrow is not filled with any color. Also, we set the `hatch` property to value '`o`' so that the arrow is filled with `os`.

Arrow: the class used to plot an arrow

```
01 import matplotlib.pyplot as plt
02 import matplotlib.patches as patches
03 def plotArrow(pos, dx, dy):
04 """
05 Objective: To plot an arrow beginning base position pos (x,y)
06 and ending the head position defined by
07 dx and dy relative to the base of the arrow
08 Input Parameters: pos, dx, dy - numeric
09 Return Value: None
10 """
11 x,y = pos[0], pos[1]
12 arrow = patches.Arrow(x, y, dx, dy, fill = False,\n hatch = '0')
13 plt.gca().add_patch(arrow)
14 plt.axis('scaled')
15 plt.title('Arrow')
16 plt.show()
17
18
19
20 def main():
21 """
22 Objective: To plot an arrow based on user input
23 Input Parameter: None
24 Return Value: None
25 """
26 pos = eval(input('Enter the starting position (x,y): '))
27 dx = float(input('Enter the dx: '))
28 dy = float(input('Enter the dy: '))
```

```
29 plotArrow(pos, dx, dy)
30
31 if __name__ == '__main__':
32 main()
```

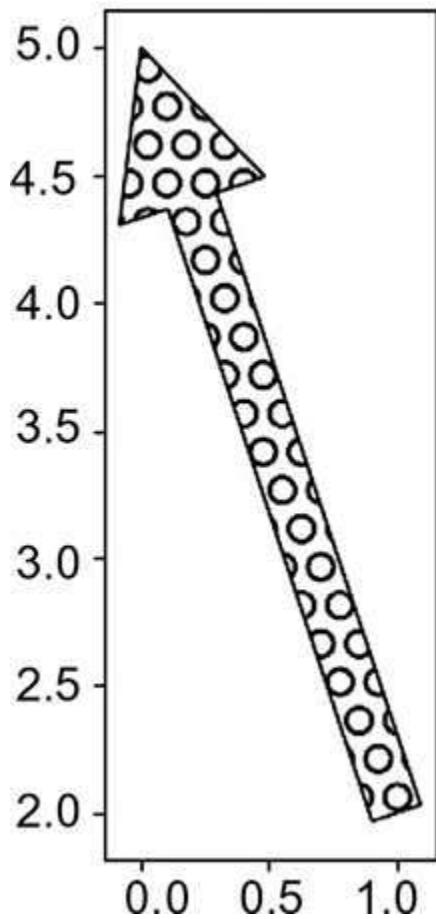
**Fig. 17.29** Program to draw an arrow (arrow.py)

On executing the script `arrow` (Fig. 17.29), Python prompts the user to enter the specifications for a polygon and displays it (Fig. 17.30).

```
Enter the starting position (x,y): (1,2)
```

```
Enter the dx: -1
```

```
Enter the dy: 3
```



**Fig. 17.30** Arrow

#### 17.2 3D OBJECTS

The `visual` module of Visual Python supports 3D graphics and animation. It may be downloaded from [vpython.org](http://vpython.org). Since Python 3 does not support VPython6 (latest version till date), we will be using older version 5.74 of VPython. Also, note that Vpython version 5.74 is supported with Python 3.2.2, thus, if you wish to use 3D objects, you may need to install Python 3.2.2 before installing VPython version 5.74.

When we click on the icon VIDLE for VPython, 3D graphics and animation functionality becomes available to us. Module `visual` of VPython contains several classes such `box`, `sphere`, `cone`, `cylinder`, `arrow`, `ring`, `pyramid`, `ellipsoid`, and `helix` which facilitate us to draw various 3D graphical objects. All these classes

inherit from the class `py_renderable`. In Table 17.5, we list some important attributes used in 3D graphics. We may set these attributes of the `py_renderable` class while instantiating the classes `box`, `sphere`, `cone`, `cylinder`, `arrow`, `ring`, `pyramid`, `ellipsoid`, and `helix`.

`visual`: the module used to create 3D graphics and animation

classes corresponding to 3D objects inherit from `py_renderable`

**Table 17.5** `py_renderable` class attributes

| Property                | Description                                                                                                     |
|-------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>visible</code>    | Controls visibility of object<br>Valid values: (True, False)<br>Default value – True                            |
| <code>frame</code>      | Creates a frame around the object using <code>frame()</code>                                                    |
| <code>display</code>    | Controls which display window contains the objects that we create<br>Default display window: <code>scene</code> |
| <code>make_trail</code> | Leaves a trail behind moving object if set to True                                                              |

### *Box*

A box may be drawn by creating an instance of class `box`. While creating an instance of `box` class, we may

specify attributes listed in [Table 17.6](#).

`box`: the class used to create a box

**Table 17.6** box class attributes

| Attribute | Description                                                                                                                                                                               |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pos       | A tuple (x, y, z) denoting centre of the box                                                                                                                                              |
| length    | Length of the box                                                                                                                                                                         |
| height    | Height of the box                                                                                                                                                                         |
| width     | Width of the box                                                                                                                                                                          |
| color     | Color of the object.<br>The value of color can either be a tuple comprising of RGB values or a color such as color.red.                                                                   |
| opacity   | Denotes transparency of the object.<br>Takes value in the range [0, 1] where 0 means total transparency                                                                                   |
| material  | Denotes material of the object. Example: materials.wood, materials.rough, materials.marble, materials.plastic, materials.earth, materials.diffuse, materials.emissive, materials.unshaded |
| axis      | A tuple (a, b, c) denotes direction of the length of the box                                                                                                                              |
| up        | A tuple (a, b, c) is used for rotating the box around its axis by changing which way is up for the box                                                                                    |
| size      | A tuple (l, h, w) is used for setting length, height, and width                                                                                                                           |

attributes of class box

A box, green in color, having length 7, height 4, breadth 5, and opacity 0.25 may be drawn on the plot area by invoking class `box` as follows:

```
from visual import *
box(size = (7,4,5), color = color.green,
opacity = 0.25)
```

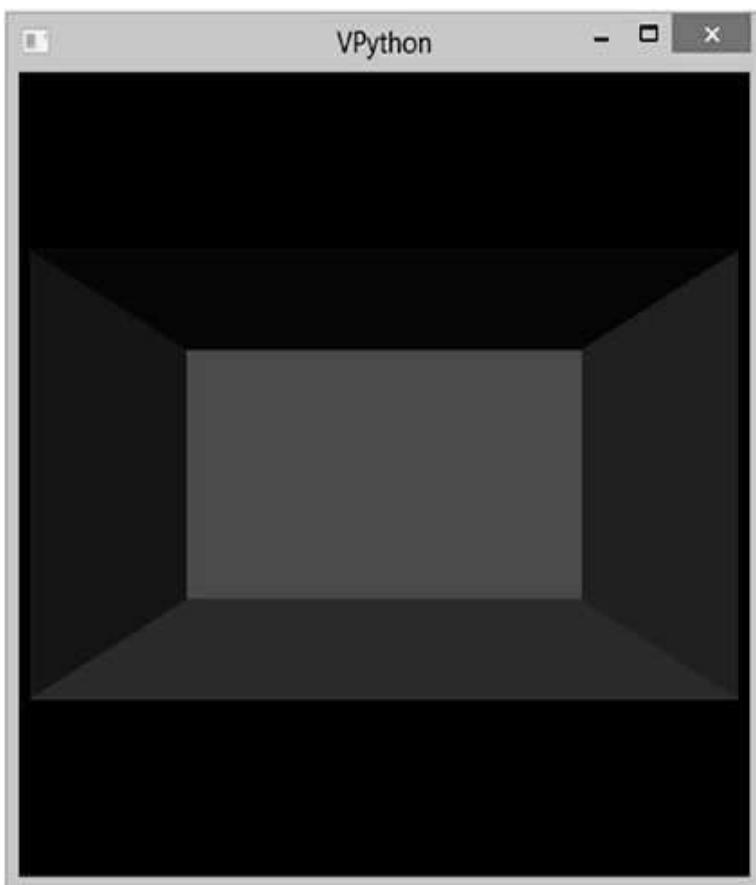
The box is shown in Fig. 17.31.

### *Sphere*

A sphere may be drawn by creating an instance of the class `sphere` as shown below:

```
sphere()
```

`sphere`: the class used to create a sphere



**Fig. 17.31** Box (see page 587 for the colour image)





**Fig. 17.32** Sphere (see page 587 for the colour image)

While creating an instance of the `sphere` class, we may specify attributes listed in [Table 17.7](#).

**Table 17.7** `sphere` class attributes

| Attribute | Description                                                                                                                                                                                                                                                                                                |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pos       | A tuple (x, y, z) denoting center of the sphere                                                                                                                                                                                                                                                            |
| radius    | Radius of the sphere                                                                                                                                                                                                                                                                                       |
| color     | Color of the object.<br>The value of <code>color</code> can either be a tuple comprising of RGB values or a color such as <code>color.red</code>                                                                                                                                                           |
| opacity   | Denotes transparency of the object.<br>Takes value in the range [0, 1] where 0 means total transparency                                                                                                                                                                                                    |
| material  | Denotes material of the object. Example: <code>materials.wood</code> , <code>materials.rough</code> , <code>materials.marble</code> , <code>materials.plastic</code> , <code>materials.earth</code> , <code>materials.diffuse</code> , <code>materials.emissive</code> , <code>materials.unshaded</code> . |
| axis      | A tuple (a, b, c) denotes orientation of the sphere                                                                                                                                                                                                                                                        |
| up        | A tuple (a, b, c) can be used for rotating the sphere around its axis by changing which way is up for the sphere                                                                                                                                                                                           |

```
attributes of class sphere
```

We may set various attributes of the `py_renderable` class while instantiating the class `sphere`. The following sequence of statements draws a sphere on the plot area (Fig. 17.32):

```
from visual import *
sphere(pos=(0,0,0), radius = 0.6, color =
\
color.cyan, opacity = 0.75, material = \
materials.rough)
```

### *Ring*

A ring can be drawn by creating an instance of the class `ring`. While creating an instance of `ring` class, we may specify attributes listed in Table 17.8.

`ring`: the class used to draw a ring

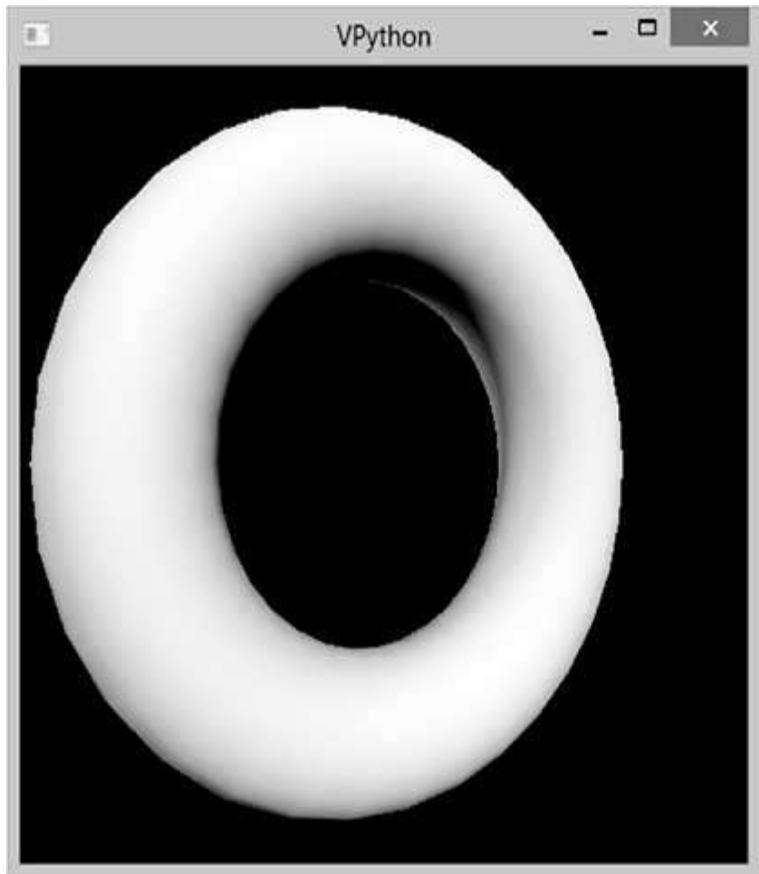
**Table 17.8** ring class attributes

| Attribute | Description                                                                                                                                                                               |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pos       | A tuple (x, y, z) denoting centre of the ring                                                                                                                                             |
| radius    | Radius of the ring. The default value is 1                                                                                                                                                |
| thickness | Thickness of the ring. The default value is 1/10th of the radius.                                                                                                                         |
| color     | Color of the object<br>The value of color can either be a tuple comprising of RGB values or a color such as color.red                                                                     |
| opacity   | Denotes transparency of the object<br>Takes value in the range [0, 1] where 0 means total transparency                                                                                    |
| material  | Denotes material of the object. Example: materials.wood, materials.rough, materials.marble, materials.plastic, materials.earth, materials.diffuse, materials.emissive, materials.unshaded |
| axis      | A tuple (a, b, c) denotes orientation of the ring                                                                                                                                         |
| up        | A tuple (a, b, c) can be used for rotating the ring around its axis by changing which way is up for the ring                                                                              |

attributes of the class ring

We may also set various attributes of the py\_renderable class while instantiating the class ring. The following sequence of statements draws a ring on the plot area (Fig. 17.33).

```
from visual import *\n\nring(axis=(0.5,0,0.9), radius=0.5,\n thickness=0.15)
```



**Fig. 17.33** Ring (see page 587 for the colour image)

### *Cylinder*

Graphical object cylinder can be drawn by creating an instance of the class `cylinder`. While creating an instance of `cylinder` class, we may specify attributes listed in [Table 17.9](#).

`cylinder`: the class used to draw a cylinder

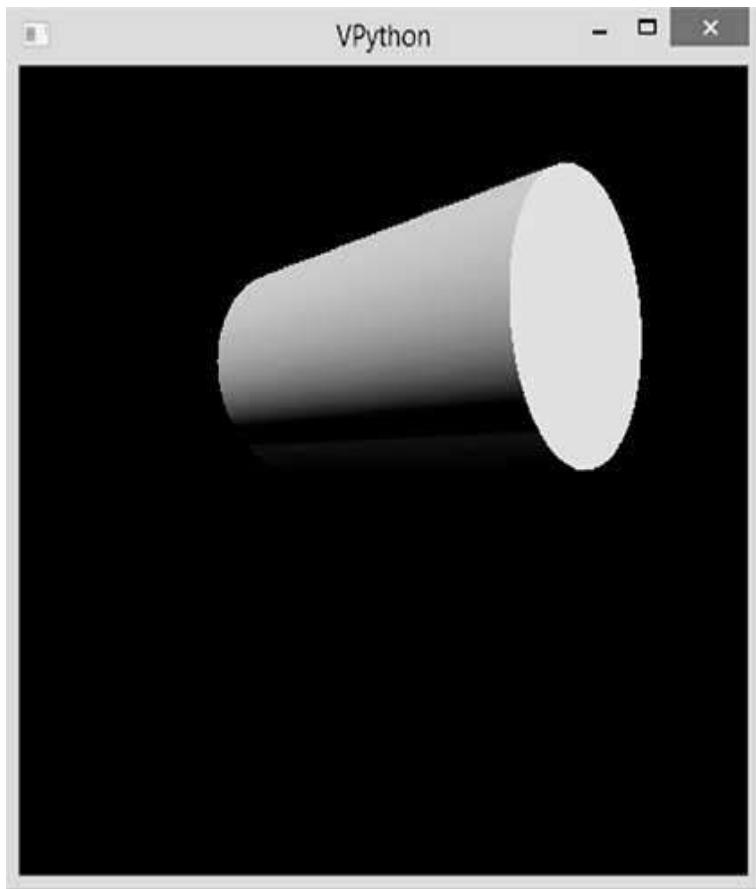
**Table 17.9** cylinder class attributes

| Attribute | Description                                                                                                                                                                               |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pos       | A tuple (x, y, z) denoting centre of one end of the cylinder                                                                                                                              |
| radius    | Radius of end circles of the cylinder                                                                                                                                                     |
| color     | Color of the object<br>The value of color can either be a tuple comprising RGB values or a color such as color.red                                                                        |
| opacity   | Denotes transparency of the object<br>Takes value in the range [0, 1] where 0 means total transparency                                                                                    |
| material  | Denotes material of the object. Example: materials.wood, materials.rough, materials.marble, materials.plastic, materials.earth, materials.diffuse, materials.emissive, materials.unshaded |
| axis      | A tuple (a, b, c) denotes direction of the length of cylinder                                                                                                                             |
| up        | A tuple (a, b, c) can be used for rotating the cylinder around its axis by changing which way is up for the cylinder                                                                      |

attributes of the class cylinder

We may also set various attributes of the py\_renderable class while instantiating the class cylinder. The following instantiation of class cylinder yields a cylinder on the plot area (Fig. 17.34).

```
from visual import *\n\n cylinder(pos=(-2,2,1), axis=(5,0,5),\n radius=2)
```



**Fig. 17.34** Cylinder (see page 587 for the colour image)

### *Arrow*

An arrow can be drawn by creating an instance of the class `arrow`. While creating an instance of `arrow` class, we may specify attributes listed in [Table 17.10](#).

`arrow`: the class used to draw an arrow

**Table 17.10** arrow class attributes

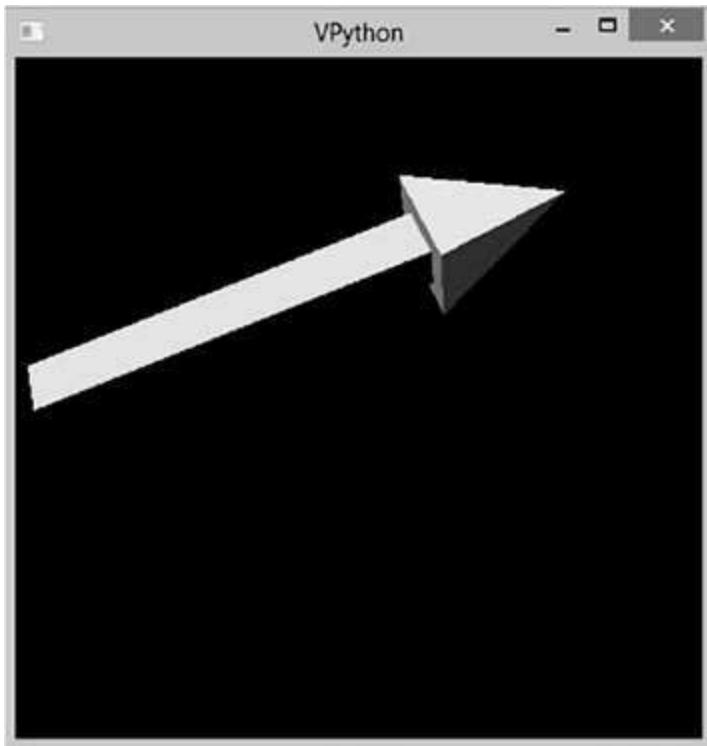
| Attribute  | Description                                                                                                                                                                               |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pos        | A tuple (x, y, z) denoting starting point of arrow                                                                                                                                        |
| shaftwidth | Width of the shaft. The default value is 0.1 times the length of the arrow                                                                                                                |
| headwidth  | Width of the head. The default value is 2 times the shaftwidth                                                                                                                            |
| headlength | Length of the head. Default value is 3 times the shaftwidth                                                                                                                               |
| color      | Color of the object<br>The value of color can either be a tuple comprising of RGB values or a color such as color.red                                                                     |
| opacity    | Denotes transparency of the object<br>Takes value in the range [0, 1] where 0 means total transparency                                                                                    |
| material   | Denotes material of the object. Example: materials.wood, materials.rough, materials.marble, materials.plastic, materials.earth, materials.diffuse, materials.emissive, materials.unshaded |
| axis       | A tuple (a, b, c) denotes direction of the length of arrow                                                                                                                                |
| up         | A tuple (a, b, c) can be used for rotating the arrow around its axis by changing which way is up for the arrow                                                                            |

attributes of the class arrow

We may also set various attributes of the `py_renderable` class while instantiating the class

arrow. The following sequence of statement draws an arrow on the plot area (Fig. 17.35):

```
from visual import *\n\narrow(pos = (-3,0,0), axis = (5,2,0),\\\nshaftwidth=0.5, color = color.yellow,\\\nup = (0,10,20))
```



**Fig. 17.35** Arrow (see page 588 for the colour image)

### Cone

A cone can be drawn by creating an instance of the class `cone`. While creating an instance of `cone` class, we may specify attributes listed in Table 17.11.

`cone`: the class used to draw a cone

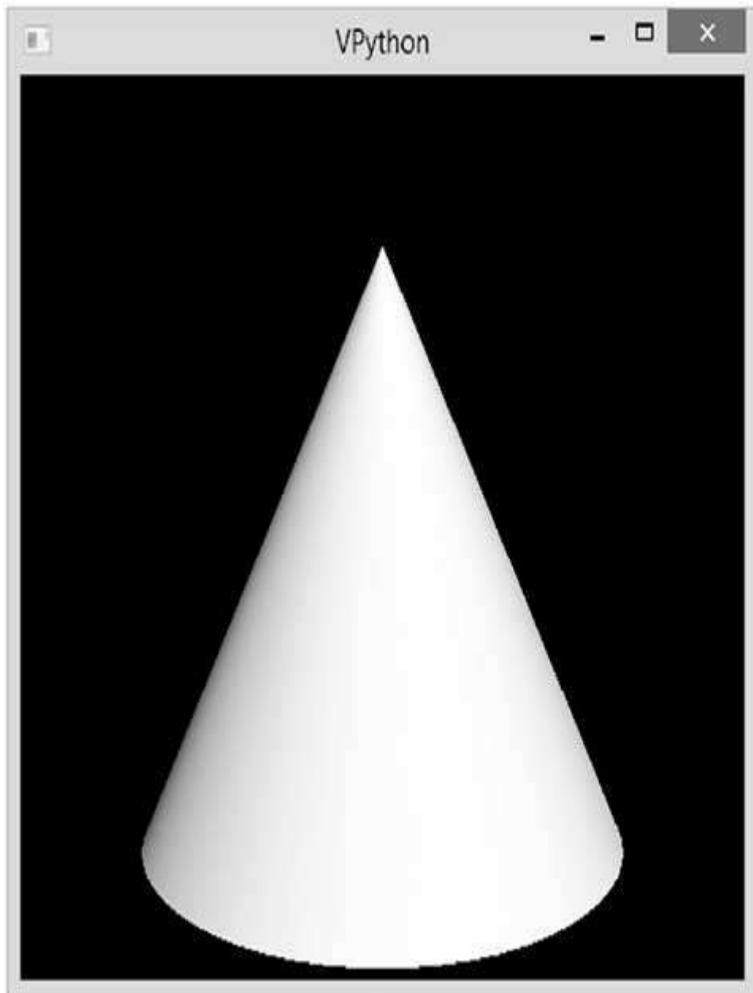
**Table 17.11** cone class attributes

| Attribute | Description                                                                                                                                                                               |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pos       | A tuple (x, y, z) denoting the center of the circular base                                                                                                                                |
| color     | Color of the object<br>The value of color can either be a tuple comprising of RGB values or a color such as color.red                                                                     |
| opacity   | Denotes transparency of the object<br>Takes value in the range [0, 1] where 0 means total transparency                                                                                    |
| material  | Denotes material of the object. Example: materials.wood, materials.rough, materials.marble, materials.plastic, materials.earth, materials.diffuse, materials.emissive, materials.unshaded |
| axis      | A tuple (a, b, c) denotes direction of the length of cone                                                                                                                                 |
| up        | A tuple (a, b, c) can be used for rotating the cone around its axis by changing which way is up for the cone                                                                              |

attributes of the class cone

We may also set various attributes of the `py_renderable` class while instantiating the class `cone`. The following sequence of statement draws a cone on the plot area (Fig. 17.36):

```
from visual import *
cone(pos=(0, -2, 0), axis=(0, 4, 0), radius=2)
```



**Fig. 17.36** Cone (see page 588 for the colour image)

### *Curve*

An object curve can be drawn by creating an instance of the class `curve`. While creating an instance of `curve` class, we may specify attributes listed in [Table 17.12](#).

`curve`: the class used to draw a curve

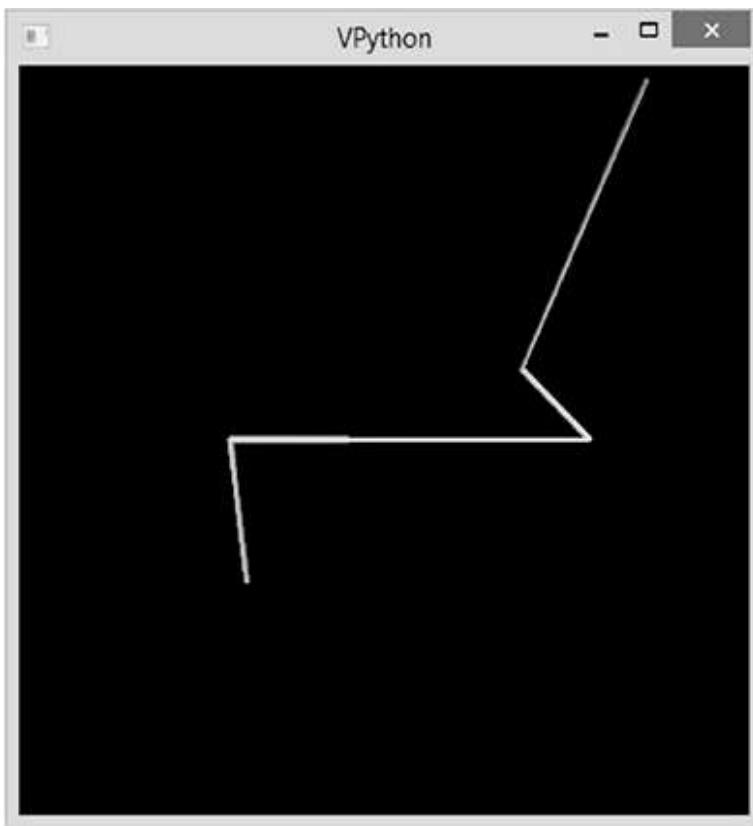
**Table 17.12** curve class attributes

| Attribute | Description                                                                                                                                                                               |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pos       | A list of points to be joined in sequence by line segments to yield a curve                                                                                                               |
| radius    | Radius of the cross-section of the curve                                                                                                                                                  |
| color     | Color of the object<br>The value of color can either be a tuple comprising of RGB values or a color such as color.red                                                                     |
| material  | Denotes material of the object. Example: materials.wood, materials.rough, materials.marble, materials.plastic, materials.earth, materials.diffuse, materials.emissive, materials.unshaded |

attributes of the class `curve`

We may also set various attributes of the `py_renderable` class while instantiating the class `curve`. The following sequence of statements yields a curve on the plot area (Fig. 17.37):

```
from visual import *
curve(pos=[(-2,-2,0), (-2,0,1), (0,0,0),
(3,0,0), \
(2,1,0), (3,4,2)], radius=0.05)
```



**Fig. 17.37** Curve (see page 588 for the colour image)

#### 17.3 ANIMATION – BOUNCING BALL

The animation is a continuous sequence of graphics, often termed frames, that are played back to back with respect to time. In the script `bouncingBall` (Fig. 17.38), we create an animation of a bouncing ball as shown in snapshots (Fig. 17.39).

animation: continuous sequence of graphics

In lines 2 and 4 (Fig. 17.38), we create a base ground using a `box` object and a ball using `sphere` object and define their attributes. The center point of the visual display window is positioned at  $(0, 5, 5)$  (lines 6). The ball is projected with an initial velocity 10 unit/sec in the direction of  $y$ -axis, and the coefficient of restitution is 0.9 (line 7 and 8). Change in time `dt` is set to 0.01 sec (line

9). The while loop produces a bouncing ball animation. Function `rate` defines a maximum number of loops per second. The position of the bouncing ball in each iteration is updated using current velocity and time. If the current position of the ball is less than the radius of the ball, it touches the floor. In this case, the velocity of the ball is negated changing the direction of motion and its position is set to coordinates (0,1,0). Also, velocity is updated using the formula  $v = u + a*t$ . Note that bouncing ball animation lasts until the position of the ball becomes static i.e. there is no change in the position of the ball in successive iterations (line 17). Also, we may set the visual display window to full-screen by using the following command immediately after importing the module `visual`:

```
rate() : maximum number of loops per second
```

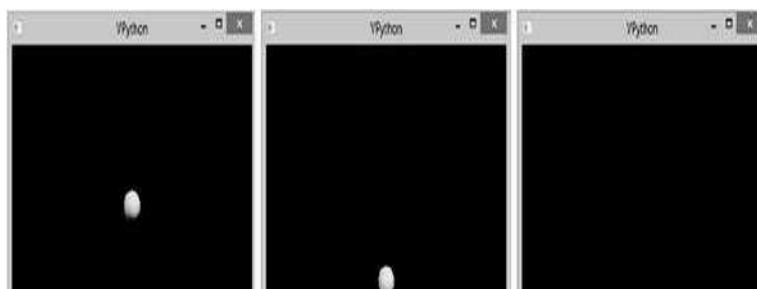
```
update velocity whenever ball changes its direction
```

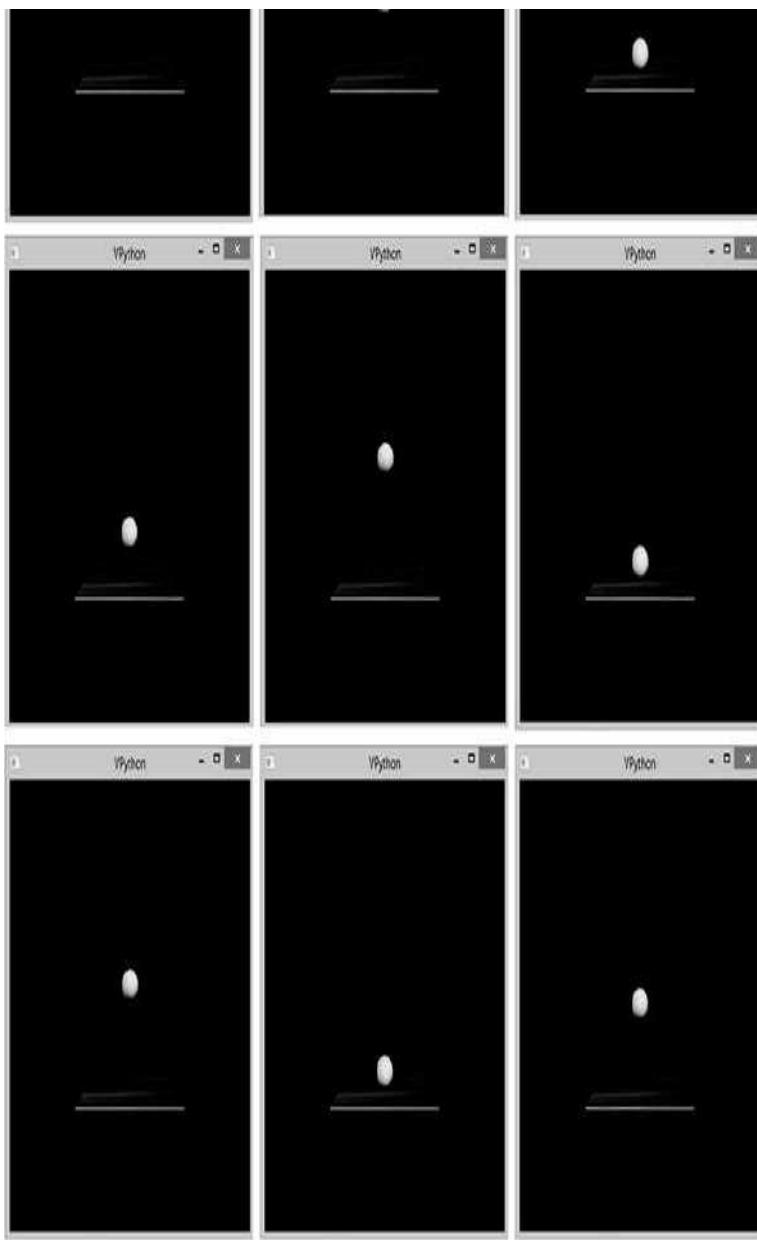
```
scene.fullscreen = True
```



```
01 from visual import *
02 ground = box(size = (10,0.1,10), color = color.white, material =
03 = materials.wood)
04 ball = sphere(pos=(0,4,0), radius = 1, color = color.yellow)
05
06 scene.center = vector(0, 5, 5)
07 velocity = 10
08 e = 0.9 #coefficient of restitution
09 dt = 0.01
10 while True:
11 rate(300) #Indicates number of loop per sec
12 prevPos = ball.y
13 ball.y += velocity * dt
14 if ball.y < ball.radius:
15 velocity = -velocity * e
16 ball.y = ball.radius
17 if (prevPos - ball.y) == 0:
18 break
19 velocity = velocity - 9.81*dt
```

**Fig. 17.38** Program to create a bouncing ball  
(bouncingBall.py)





**Fig. 17.39** Snapshots of bouncing ball (see page 588 for the colour image)

#### SUMMARY

1. Python library `matplotlib` provides several methods that facilitate us to draw graphical objects including point, line, circle, rectangle, oval, and polygon. The library also supports objects like graphs, histograms, bar charts, pie charts, scatter plots, and error charts.
2. `pyplot` module of `matplotlib` contains several functions for plotting, and modifying or setting various properties such as the layout of the plot area, labels, and attributes of the figure.

3. The function `plot` of `pyplot` module may be used for plotting as follows:  
`plot (x, y, LineSpecification)`
4. The function `show` of `pyplot` module is used for displaying the figure.
5. The function `axis` of `pyplot` module is used for setting axis parameters  $x_{min}$ ,  $x_{max}$ ,  $y_{min}$  and  $y_{max}$ .
6. The functions `title`, `xlabel`, and `ylabel` of `pyplot` module are used for setting title and labelling axes.
7. The function `subplot` of `pyplot` module is used for plotting several functions, each in a separate graph, but in the same figure. The function takes three arguments, the number of rows, the number of columns, and the figure number.
8. The function `savefig` of `pyplot` module is used to save a graph.
9. The function `hist` of `pyplot` module is used for plotting a histogram for the given data set.
10. The function `pie` of `pyplot` module is used for creating a pie chart for a given sequence of data.
11. In `matplotlib` terminology, the graphical shapes such as a circle, rectangle, polygon, and ellipse are known as patches. The module `patch` supports classes such as `Rectangle`, `Circle`, `Ellipse`, `RegularPolygon`, `Polygon`, `CirclePolygon`, `Rectangle`, `Arrow`, and `FancyArrow`.
12. Visual Python has `visual` module that supports 3D graphics and animation. It contains several classes such as `box`, `sphere`, `cone`, `cylinder`, `arrow`, `ring`, `pyramid`, `ellipsoid`, and `helix` that facilitate us to draw various 3D graphical objects.

#### EXERCISES

1. Write a program that plots graphs corresponding to the following equations by taking suitable lists as an input from the user:
  1.  $y = 3x^2 + 4x + 2$  ( $y$  vs.  $x$ )
  2.  $v = u + at$  ( $v$  vs.  $t$ )
  3.  $F = ma$  ( $F$  vs.  $m$ )
2. Write a program that plots the following functions in the range  $0^\circ$  to  $360^\circ$  in the same figure:
  1.  $\tan$
  2.  $\cot$
  3.  $\sec$
  4.  $\cosec$
3. Write a program that illustrates superposition of two waves of the same amplitude.
4. Write a program that shows the moving car.
5. Write a program that shows the motion of a rocket.

# CHAPTER 18

## APPLICATIONS OF PYTHON

### CHAPTER OUTLINE

- [18.1 Collecting Information from Twitter](#)
- [18.2 Sharing Data Using Sockets](#)
- [18.3 Managing Database Using Structured Query Language](#)
- [18.4 Developing Mobile Application for Android](#)
- [18.5 Integrating Java with Python](#)
- [18.6 Python Chat Application Using Kivy and Socket Programming](#)

Python is rapidly evolving as one of the most powerful and preferred programming choice amongst the developers. The extensive third party support has made Python a viable option in almost every application area such as handling twitter data, web access, database management, and mobile app development. In this chapter, we briefly cover the above-mentioned application areas. As a lot of application code is developed in Java, we also discuss, how Python code can be integrated with Java code seamlessly.

Python has emerged as a preferred choice for programming in almost every application area

### 18.1 COLLECTING INFORMATION FROM TWITTER

Often we are interested to know the current trend, i.e., to find the ongoing activity about an event, place, or person. Twitter is one such source of information about almost everything happening in the world. Indeed, Twitter has emerged as a popular social networking site, and people use it to air their views and increase their awareness. Because of the millions of active users

sharing the latest updates, a lot of interesting data can be collected from Twitter. Twitter can also be used to retrieve information about the tweets, connections, or followers of a user, or topics trending on Twitter.

Twitter: a rich source of up-to date information

Information is shared on Twitter in the form of tweets. A tweet may contain photos, videos, links, and up to 140 characters of text. A tweet may be published on Twitter by registered users. However, all the users, whether registered or unregistered can read the tweets. Tweets are of two types, namely, public tweets and private tweets. Public tweets are the tweets visible to anyone accessing Twitter, whereas private tweets can be seen by permitted Twitter followers. Twitter provides several APIs (Application Programming Interfaces) to access Twitter data such as user's profile information or tweets posted. These APIs are mainly categorized as Streaming and REST APIs. While streaming APIs provide a continuous stream of data in real time from Twitter server, REST APIs are used for retrieving data using a query to the Twitter server.

public tweets: can be accessed by any one

private tweets: can be accessed only by authorized followers

Twitter APIs are used for accessing twitter data in a script

In this section, we will build a simple crawler that searches and collects Twitter data in real time. So we will be using streaming APIs. An API request on Twitter is first authenticated using *Open Authentication* (OAuth). OAuth allows the users to connect to Twitter and send authorized Twitter API request. Every registered application (consumer) of Twitter is assigned a *Consumer Key* (API Key) and *Consumer Secret* ( API Secret) using which the applications can authenticate themselves. Twitter verifies the user's identity and assigns PIN (OAuth verifier) that will be required by the application to request *Access Token* and *Access Secret* to be used in an application for invoking API calls. The *Access Token* and *Access Secret* pair uniquely identify a user.

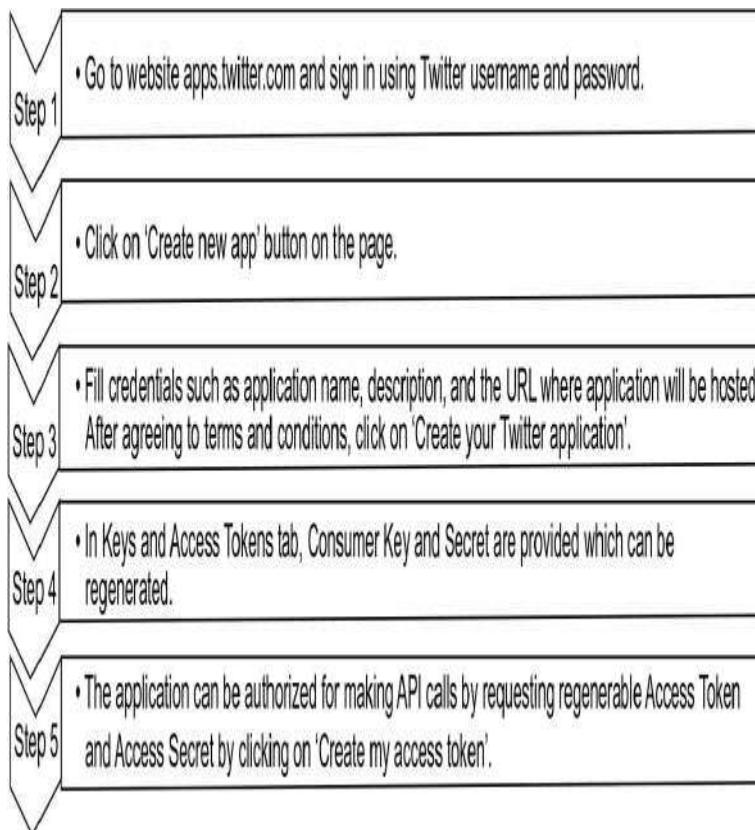
OAuth: used to authenticate API request

For Open Authentication, we need to follow the steps given in Fig. 18.1.

For performing analysis on Twitter data, we need the package `tweepy`. It may be downloaded using the following cmd command on windows:

Twitter data analysis requires importing `tweepy` package

```
pip install tweepy
```



**Fig. 18.1** Open authentication sequence

If system fails to recognize pip command, it can be downloaded from <https://bootstrap.pypa.io/get-pip.py>, and installed using the following commands from command line interface:

```
cd <pathname of the directory containing
```

```
get-pip.py>
```

```
python get-pip.py
```

installing pip

Alternatively, we may open and execute the script `get-pip.py` from Python IDLE. Also, the environment path

for system variable should be updated to include  
C:\Python27\Scripts.

### *Open Authentication*

Accessing Twitter data requires open authentication of the application. In the script `TwitterOAuth` (Fig. 18.2), we aim at authorizing a Python application to access Twitter. The function `OAuthVerifier` is used for doing so. It first initializes the variables `consumerKey`, `consumerSecret`, `accessToken`, and `accessSecret` (lines 9, 10, 12, and 13). Execution of line 11 creates an `OAuthHandler` object for using API key and API secret. The `accessToken` and `accessSecret` are set for this object using the method `set_access_token`. The authentication handler, that we just obtained, can be passed as a parameter to class `API` of the `tweepy` module (a wrapper for `API` as provided by Twitter). An object of this class (`api`) can be used for retrieving user information such as his/her followers, friends, and tweets.

an object of class `OAuthHandler` is used to instantiate the class `API`

```
01 import tweepy
02
03 def OAuthVerifier():
04 """
05 Objective: To authorize the application to access Twitter
06 Input Parameter: None
07 Return Value: API object
08 """
09 consumerKey = 'iym0XRG0SOgyP0jmlPQgB4XrC'
10 consumerSecret = 'I6gBj8RpcXJvN6xaKUUGeTkPEDpHzirXBmT9d9y
G8k5Ik0H0bF'
11 authorization = tweepy.OAuthHandler(consumerKey,
12 consumerSecret)
13 accessToken = '727494459118653446-XWf9MmBJmCUOM7Ic9xvoHl
ZCBQAWWA1'
14 accessSecret = 'jl4UJPXYSC8ygV8EsyNz6fMOW2NEs9NFvJc2Inyf
XNvve'
15 authorization.set_access_token(accessToken, accessSecret)
16 api = tweepy.API(authorization)
17 return api
18
19 def main():
20 """
21 Objective: To provide open authentication of application
22 """
```

```
21 Input Parameter: None
22 Return Value: None
23 ''
24 api = OAuthVerifier()
25 print('Application authorized')
26
27 if __name__ == '__main__':
28 main()
```

**Fig. 18.2** OpenAuth (TwitterOAuth.py)

Note that the keys used in the program are only for illustration purpose; however, the user is expected to generate the keys himself and replace them appropriately in the given code.

#### *An Example – Collecting User Information*

Often, we are interested in retrieving user statistics such as name, id, location, description, friends and followers count, and posted tweets. To retrieve such user statistics, we require an associated user object. In the script TwitterUserInfo1 (Fig. 18.3), we retrieve information about an authenticated user and a user whose id or screen name (@handle) is known. For example, the user Sachin Tendulkar has @sachin\_rt as screen name with associated id 135421739.

retrieving user statistics

```
01 import tweepy
02 from TwitterOAuth import OAuthVerifier
03
04 def getUserStatistics(user):
05 """
06 Objective: To get user statistics using various
07 variables of the api
08 Input Parameter: user - string
09 Return Value: None
10 """
11 print('\nName: ', user.name)
12 print('Screen Name: ', user.screen_name)
13 print('ID: ', user.id)
14 print('Account creation date and time: ', user.created_at)
15 print('Location: ', user.location)
16 print('Description: ', user.description)
17 print('No. of followers: ', user.followers_count)
18 print('No. of friends: ', user.friends_count)
19 print('No. of favourite tweets: ', user.favourites_count)
20 print('No. of posted tweets: ', user.statuses_count)
```

```
21 print('Associated URL: ', user.url)
22
23 def main():
24 """
25 Objective: To collect user information
26 Input Parameter: None
27 Return Value: None
28 """
29 # To print user's information
30 api = OAuthVerifier()
31 # Authenticated User
32 user = api.me()
33 getUserStatistics(user)
34 # Different User
35 name = input('\n\nEnter the user identification: ')
36 user = api.get_user(name)
37 getUserStatistics(user)
38
39 if __name__ == '__main__':
40 main()
```

**Fig. 18.3** User information (`TwitterUserInfo1.py`)

The user method `me` (line 32) of the `api` returns authenticated user information. Another user method `get_user` returns information about the particular user, whose id/screen name is taken as input in line 35. We define a function `getUserStatistics` for accessing and printing the user information. Table 18.1

lists variables associated with user object that may be accessed for information.

**Table 18.1** Variables associated with user object

| Variable         | Description                                                                 |
|------------------|-----------------------------------------------------------------------------|
| name             | Name of the user.                                                           |
| screen_name      | Handler/alias name of the user with which user uniquely identifies himself. |
| id               | Unique id associated with every user's account.                             |
| created_at       | UTC of the creation of user's account on Twitter.                           |
| location         | User-defined location of his profile.                                       |
| description      | User's account description.                                                 |
| followers_count  | Number of followers of the user.                                            |
| friends_count    | Number of users being followed.                                             |
| favourites_count | Number of favourite tweets of a user.                                       |
| statuses_count   | Number of tweets of a user.                                                 |
| url              | URL mentioned by the user in his profile.                                   |

On executing the script `TwitterUserInfo1` (Fig. 18.3), Python responds with the following output:

```
>>>

Name: Suzzane Mathew

Screen Name: SuzzaneMathew

ID: 727494459118653446
```

Account creation date and time: 2016-05-03  
13:46:10

Location: India

Description: A Python Programmer

No. of followers: 3

No. of friends: 34

No. of favorite tweets: 0

No. of posted tweets: 3

Associated URL: None

statistics of an authenticated user

Enter the user identification: @sachin\_rt

Name: sachin tendulkar

Screen Name: sachin\_rt

ID: 135421739

Account creation date and time: 2010-04-21  
07:42:23

Location: UT: 18.986431, 72.823769

Description: Proud Indian

No. of followers: 15690592

No. of friends: 79

No. of favourite tweets: 39

No. of posted tweets: 1519

Associated url: http://t.co/TCNcUDBwuE

```
statistics of the user with screen name @ sachin_rt
```

### *Collecting Followers, Friends, and Tweets of a User*

Now we develop a program that discovers followers, friends, and tweets of a user. For a given user, the methods `followers`, `friends`, and `user_timeline` return information about followers, friends, and public tweets, respectively (Fig. 18.4).

```
01 import tweepy
02 from TwitterOAuth import OAuthVerifier
03
04 def getUserFollowers(api):
```

```
05 """
06 Objective: To get followers of the user
07 Input Parameter: api - object of class API
08 Return Value: None
09 """
10 print('\nFollowers:')
11 for follower in api.followers():
12 print(follower.screen_name)
13
14 def getUserFriends(api):
15 """
16 Objective: To get friends of the user
17 Input Parameter: api - object of class API
18 Return Value: None
19 """
20 print('\nFriends:')
21 for friend in api.friends():
22 print(friend.screen_name)
23
24
```

```

25 def getUserTweets(api):
26 """
27 Objective: To get tweets of the user
28 Input Parameter: api - object of class API
29 Return Value: None
30 """
31 print('\nTweets: ')
32 for tweet in api.user_timeline():
33 print(tweet.text)
34
35 def main():
36 """
37 Objective: To print user information
38 Input Parameter: None
39 Return Value: None
40 """
41 api = OAuthVerifier()
42 getUserFollowers(api)
43 getUserFriends(api)
44 getUserTweets(api)
45
46 if __name__ == '__main__':
47 main()

```

**Fig. 18.4** Friends, followers, and tweets of a user  
(TwitterUserInfo2.py)

On executing the script TwitterUserInfo2 (Fig. 18.4), Python responds with the following output:

```

>>>

Followers:

PeaceAlwaysPARI

ikindlebook

gauravparashari

followers

```

Friends:

gvanrossum

iamsrk\_sharukh

DataSciFact

CompSciFact

Delhi\_U

htTweets

ndtv

timesofindia

EconomicTimes

msdhoni

BeingSalmanKhan

SrBachchan

imVkohli

sachin\_rt

PythonHub

PythonStack

pycoders

pythoncoders

djangoproject

ThePSF

```
friends
```

Tweets:

```
RT @gvanrossum: The very first
https://t.co/2j12YXhlJD home page, from
May 1997: https://t.co/KwtE7rjX9P
```

```
RT @gvanrossum: MicroPython 1.7 released!
Now with cross-compiler.
https://t.co/gWq50gVEwk Congrats Damien
and contributors!
```

```
Hello Twitter! #myfirstTweet
```

```
tweets
```

If the number of friends, followers, or tweets exceeds 20, only first 20 results are displayed. These 20 results correspond to the first page returned by these methods. However, if all possible results are required to be shown, the method `items` of the object of type `Cursor` of the module `tweepy` can be used (Fig. 18.5).

`items()` : returns an iterable object comprising desired number of search results

```
01 import tweepy
02 from TwitterOAuth import OAuthVerifier
03
04 def getUserFollowers(api):
05 '''
06 Objective: To get followers of the user
07 Input Parameter: api - object of class API
08 Return Value: None
09 '''
10 ...
```

```

10 print("\nFollowers:")
11 for follower in tweepy.Cursor(api.followers).items():
12 print(follower.screen_name)
13
14 def getUserFriends(api):
15 """
16 Objective: To get friends of the user
17 Input Parameter: api - object of class API
18 Return Value: None
19 """
20 print('\nFriends:')
21 for friend in tweepy.Cursor(api.friends).items():
22 print(friend.screen_name)
23
24
25 def getUserTweets(api):
26 """
27 Objective: To get tweets of the user
28 Input Parameter: api - object of class API
29 Return Value: None
30 """
31 print('\nTweets: ')
32 for tweet in tweepy.Cursor(api.user_timeline).items():
33 print(tweet.text)
34
35 def main():
36 """
37 Objective: To print user information
38 Input Parameter: None
39 Return Value: None
40 """
41 api = OAuthVerifier()
42 getUserFollowers(api)
43 getUserFriends(api)
44 getUserTweets(api)
45
46 if __name__ == '__main__':
47 main()

```

**Fig. 18.5** Friends, followers, and tweets of a user  
(TwitterUserInfo3.py)

While invoking the method `items()`, integer argument limit can be used to restrict the number of results to be displayed.

*Collecting Tweets Having Specific Words*

Often, we wish to collect some specific tweets streaming in real time. The data collected may be further used for analysis. We use Twitter Streaming API for downloading Twitter messages in high volume in real time. Python class `StreamListener` is used for collecting streaming tweets. In this section, we define a class `MyStreamListener` that inherits the class `StreamListener` of the `tweepy` module. In the class `MyStreamListener`, we define two methods: `on_status` and `on_error`. The method `on_status` tells what to do when a status (input parameter) known as tweet update is received. The overridden method `on_status` receives the data from the method `on_data` of the class `StreamListener`. The method `on_error` handles the error and gets automatically invoked on occurrence of an error. As shown in the script `KeyTweets` (Fig. 18.6), the method handles error 420 and disconnects the stream by returning `False` when error 420 occurs. This error occurs if the number of user attempts to connect to streaming API exceeds a particular limit.

`StreamListener`: a class for collecting streaming tweets

Suppose, we are interested in tweets containing the word `Python`. The execution of the script `KeyTweets` (Fig. 18.6) outputs streaming data containing the keyword `Python`. It first authorizes the application to access Twitter using the function `OAuthVerifier` (line 37). It then creates a stream listener instance `listenerOb` of class `MyStreamListener` (line 39). The class `Stream` of module `tweepy` is then used for creating a `Stream` object `myStream`, which takes two parameters: `auth` – an attribute of `api` and `listenerOb` – a stream listener object (line 41). It establishes a streaming session and routes all the tweets to the object `listenerOb`. For streaming all tweets containing the word `Python`, we make use of the stream object method `filter`, which takes a list parameter

track containing the search keywords for the stream  
(line 44).

Stream: a class used to create a stream object

```
01 import tweepy
02 from TwitterOAuth import OAuthVerifier
03
04 class MyStreamListener(tweepy.StreamListener):
05 # Class inheriting StreamListener of tweepy module
06
07 def on_status(self, status):
08 """
09 Objective: To print text stream of tweets
10 Input Parameters:
```

```
11 self (implicit parameter) - object of type
12 MyStreamListener
13 status - string value representing tweet
14 Return Value: None
15 """
16 print(status.text)
17
18 def on_error(self, status):
19 """
20 Objective: To disconnect the stream by returning False
21 if error 420 occurs
22 Input Parameters:
23 self (implicit parameter) - object of type
```

```

24 MyStreamListener
25 status - int value representing error code
26 Return Value: result - int
27 """
28 if status==420:
29 return False
30
31 def main():
32 """
33 Objective: To print streaming data containing given keywords
34 Input Parameter: None
35 Return Value: None
36 """
37 api = OAuthVerifier()
38 # Creates a stream listener object listenerOb
39 listenerOb = MyStreamListener()
40 # Create a Stream object
41 myStream = tweepy.Stream(api.auth, listenerOb)
42 # Starts streaming by specifying search keywords
43 searchList = eval(input('Enter search keywords list: '))
44 myStream.filter(track = searchList)
45
46 if __name__ == '__main__':
47 main()

```

**Fig. 18.6** Tweets having specific words (`KeyTweets.py`)

On executing the script `KeyTweets` (Fig. 18.6), Python prompts the user for input and on providing the input `['Python']`, responds with the following output:

```

>>>

Enter search keywords list: ['Python']

I liked a @YouTube video
https://t.co/cLNOpZTD3b Monty Python -
Execution in Russia (funny sketch!)

```

RT @rogerhoward: Sublime still feels near perfect for Python; VSCode definitely better for JS. Hate the thought of using both.

RT @dbader\_org: Writing a little GUI helper tool for macOS with Python—Nice "scratch your own itch with Python" article: <https://t.co/yKTI1...>

If you're looking for work in MO, check out this #job: <https://t.co/iuSMypbtOc> #Python #Relocate #NYC...  
<https://t.co/EMLMnedNgU>

RT @fmasanori: "Dev-ia C em Python". Nome de equipe na maratona de programação. Como os alunos são criativos.

1-2-6 are my top three  
<https://t.co/zLmfaaPI3s>

RT @KirkDBorne: #DeepLearning Cheat Sheet (using Keras + #Python Libraries)  
<https://t.co/QIivKejlNt> #abdsc #BigData #DataScience....

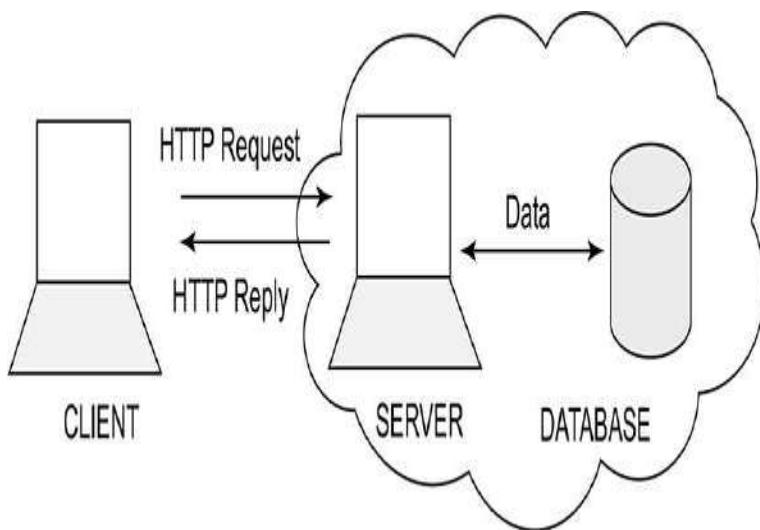
•  
•  
•  
tweets for the search keyword: Python

## 18.2 SHARING DATA USING SOCKETS

Often two processes need to communicate with each other for sharing data. These processes may reside either on the same machine or on different machines connected to a network. The two processes involved in this interprocess communication are typically the client and the server applications. The client application

requests data and the server application responds by fulfilling the request. The client-server architecture allows two remote entities to communicate. For this purpose, HTTP is often used in Internet applications as shown in Fig. 18.7.

sharing data across applications: processes need to communicate with each other



**Fig. 18.7** Web application architecture

client - server architecture

client: requests data server: responds by fulfilling the request

Interprocess communication can be achieved using one or more of the several available options such as file, signal, socket, shared memory, pipe, and message. In this section, we will learn interprocess communication via use of socket programming using Python. For the

purpose of smooth communication, the communicating processes need a protocol that defines the rules for communication between the client and the server. The network architecture mainly uses TCP/IP (Transmission Control Protocol/Internet Protocol) for communication.

socket: used for interprocess communication

Suppose, process A (say, client) needs to communicate with process B (say, server). In order to communicate, the client and the server need to make a connection between themselves. A socket defines the client and the server as the end points of such a connection. A socket works much like communication over the telephone. The two parties that wish to communicate with each other have telephone handsets at the end points. For communication to take place, one party needs to dial up a connection, and the other party must be available to attend the phone call. In the client-server environment, each of the client and the server would require a socket. A socket is uniquely identified by a pair of IP address and port number. IP address is the machine's network address, which uniquely identifies every machine on the network. Port number determines the service/application (e.g., FTP, email service) on the server that should respond to the request. For example, a server with an IP address may have several applications, each with a unique port number for handling request related to email, news, file transfer, remote login, and so on. Since the client initiates the connection and the server responds to it, a socket pair for communication is identified by client IP address, client port number, server IP address, and server port number.

a socket is uniquely identified by a pair of IP address and port number

## *Client-Server Communication on the Same Machine – A Simple Example*

Python has a library that provides support for TCP/IP sockets. It offers two socket modules – `socket` and `socketserver`. While the former provides low-level networking interface, the latter provides support in the form of classes for developing network server applications. In this chapter, we will focus on basic networking services. So, we will use the `socket` module. The `socket` module provides the class `socket` for creating a socket object.

```
mySock = socket.socket(socket_family,
socket_type,protocol=0)
```

creating a socket object using `socket` class of `socket` module

The first two parameters specify the domain, i.e., family of the protocol, such as `AF_INET`, `AF_INET6`, and `AF_UNIX`, and the type of socket to be created such as stream socket (`SOCK_STREAM`) for connection-oriented protocol and datagram socket (`SOCK_DGRAM`) for the connectionless protocol. The third parameter is a default parameter, which specifies a particular variant of protocol within given domain and socket type. The socket object `mySock` created above is a unique socket identifier which is used for communicating with other process.

A stream socket is a connection in which data is sent as a stream of bytes in an ordered manner. In this section, we create a socket using `SOCK_STREAM`. For simplicity, we create a socket object of the class `socket` without specifying any parameters, thus using the default parameters `AF_INET` as domain and `SOCK_STREAM` as the type of socket.

`SOCK_STREAM`: used to create a stream socket

Before we proceed further, it is important to understand the methods associated with a socket object. These methods are categorized as server, client, and general socket methods. We describe these methods in Tables 18.2, 18.3, and 18.4 respectively.

**Table 18.2** Server socket methods associated with socket object  
mySock

| Method                 | Description                                                                                                                                                                                                                                                                                                                                                            |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mySock.bind(address)   | Binds the socket to given address specified by the pair: (hostName, portNumber).                                                                                                                                                                                                                                                                                       |
| mySock.listen(backLog) | Lists for the incoming connections. Optional parameter backLog specifies maximum number of connections that can be queued.                                                                                                                                                                                                                                             |
| mySock.accept()        | Accepts the pending incoming connection, else waits until it arrives. For the accepted connection, it returns a pair (conn, address). Here, conn is a new socket identifier to be used for sending and receiving the data on established connection, and address comprises the pair: (hostName, portNumber) corresponding to the other end of the connection (client). |

**Table 18.3** Client socket methods associated with socket object  
mySock

| Method                  | Description                                                                       |
|-------------------------|-----------------------------------------------------------------------------------|
| mySock.connect(address) | Connects to the host having address (destinationHostName, destinationPortNumber). |

**Table 18.4** General socket methods

| Method               | Description                                                                                                                                             |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| socket.gethostname() | Returns the string containing hostname of the local machine on which process is executing.                                                              |
| mySock.send(string)  | Sends the <i>bytes-like object</i> to the TCP socket. It also returns the number of bytes sent.                                                         |
| mySock.recv(bufSize) | Receives the <i>bytes-like object</i> from the TCP socket. The maximum amount of data that can be read is specified by the buffer size <i>bufSize</i> . |
| mySock.close()       | Closes the socket <i>mySock</i> . An attempt to read or write on the closed socket will result in error.                                                |

Let us suppose that each of the client and the server has a socket created at its end. To communicate with each other, the client and the server must go through a sequence of steps:

1. **SERVER:** On creating the socket, the server must bind it to the particular port number on the local address, specifying the application that will listen to the incoming client requests. The method *bind* of the *socket* class takes a tuple comprising local address as the first parameter, and port number as the second parameter. Here, *hostname* and *IP address* are used as aliases. Since all ports below 1024 are reserved, we may use a port number between 1025 and 65535, both included. We may identify the ports already in use by executing the command (at command prompt):  
`netstat -a`.

`bind()` : binds a socket to a port number on local address

2. **SERVER:** Once the server is bound to a socket, it must be ready to service requests from the client. The server listens for incoming connections from the clients using the method *listen*. This method takes the *maximum number of connections allowed to be queued* as the input parameter.

`listen()` : listens for incoming connections from the client on the socket

**3. SERVER:** The server waits in a loop indefinitely until the client connects to the port to which server has bound itself. While listening on the socket, if a connection request from a client arrives, the server needs to accept it. The method `accept` gets the pending connection from the connection queue, and returns (i) the socket object for the client to be used for data transfer (ii) the address of the client (destination IP address and port number).

`accept()` : gets pending connections from the connection queue

**4. CLIENT:** Once the socket is created at the client end, we may use it to establish the connection between the client (our Python program) and the application running on the server by specifying the host name and the port number. The method `connect` of the `socket` module is used for this purpose. It requires a tuple comprising the name of the host and the destination port number listening to a client request.

`connect()` : used to connect to the host

**5.** Now the client and the server may exchange the data using `send` and `recv` methods. Since the client and server can understand a *bytes-like object*, we use `encode` function to convert a string into `bytes` object. By default, `encode` function uses utf-8 encoding scheme. Subsequently, the encoded `bytes` object is converted to a string using `decode` function.

`send()` and `recv()` : to send and receive bytes-like object to and from the socket

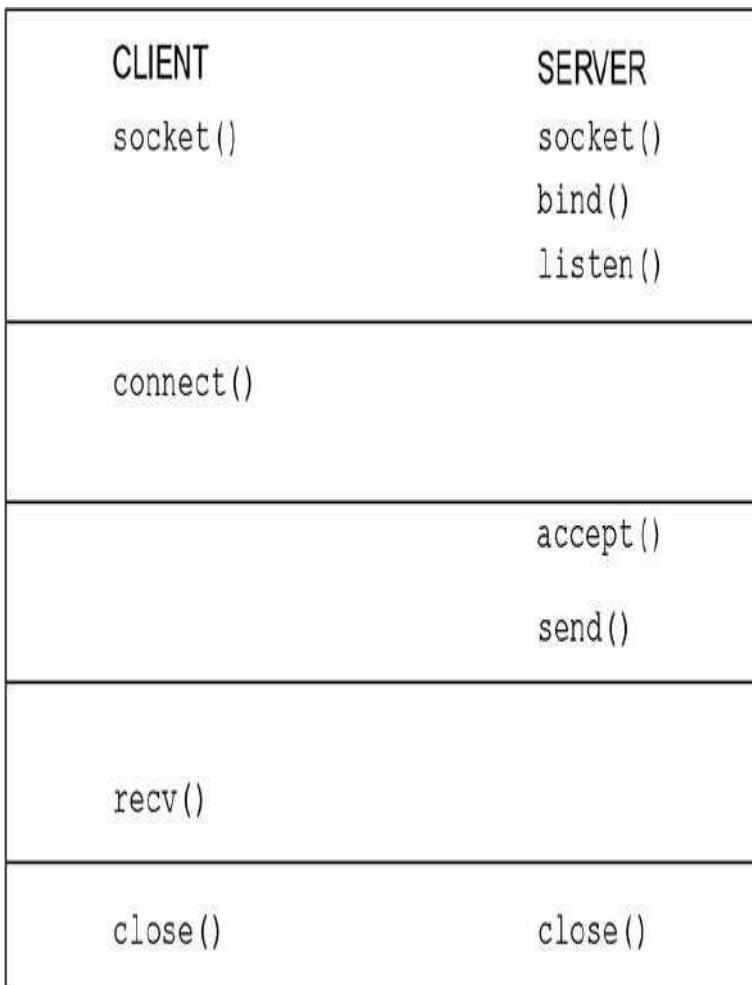
**6.** Finally, the client and the server close the socket using the `close` method.

`close()` : closes the socket

The order in which these socket methods should be invoked by client and server is illustrated in Fig. 18.8. In Figs. 18.9 and 18.10, we give the scripts for the server and the client respectively. Before the client can request the server, the server must be listening for the incoming request by the client. The server and client both create a socket object. The server binds local address, and unused port 65535 to it using the method `bind`, and waits for single incoming connection by executing the method `listen`. It then accepts the pending connection using the method `accept` and responds with the encoded version of the string 'Thank you for connecting client' on that connection. The client, on the other hand, connects to the destination host and the port using `connect` method. Since, in our case, we are running the

two processes on the same machine, the destination host will be same as the source host. It receives the data on the connected socket using `recv` method. Both the client and the server close the socket using the `close` method.

order of execution of socket methods



**Fig. 18.8** Order of execution of socket methods

```

01 import socket
02
03 mySock = socket.socket()
04 host = socket.gethostname()
05 port = 65535
06 mySock.connect((host, port))
07 data = mySock.recv(1024)
08 print(data.decode())
09 mySock.close()

```

**Fig. 18.9** Client (client.py)

```
01 import socket
02
03 mySock = socket.socket()
04 host = socket.gethostname()
05 port = 65535 # Reserve a port number
06 mySock.bind((host, port))
07 mySock.listen(1)
```

```
08 conn, addr = mySock.accept()
09 message = 'Thank you for connecting client '+
10 str(addr)
11 conn.send(message.encode())
12 conn.close()
13 mySock.close()
```

**Fig. 18.10** Server (server.py)

On executing the server program,

```
F:\PythonCode\Ch18>python server.py
```

Python responds with the following output on the client side:

```
server must be listening to requests before a client can
connect to it
```

```
F:\PythonCode\Ch18>python client.py
```

```
Thank you for connecting client
('192.168.56.1', 64203)
```

```
F:\PythonCode\Ch18>
```

Note that an unused port is randomly chosen for the client process.

#### *An Echo Server*

Next, we develop an echo application, in which the server program (Fig. 18.12) echoes the data sent by a client application (Fig. 18.11) running on a different system. So, in this case, the client invokes `send` method to send the data to the server. Subsequently, the server receives the data using `recv` method. The data received is echoed back to the client using `send` method. The client then receives it using `recv` method. The procedure continues as long as client provides the input data to be sent to echo server. Since the client and the server reside on two different systems in the network, client and server host names will be different.

echo server: echoes the data sent by a client

```
01 import socket
02
03 mySock = socket.socket()
04 host = '192.168.1.142'
05 port = 65535
06 mySock.connect((host, port))
07 while True:
08 data = input('Enter data to be sent: ')
09 if data == '':
10 mySock.send('EOF'.encode())
11 break
```

```
12 else:
13 mySock.send(data.encode())
14 data = mySock.recv(1024)
15 print(data.decode())
16 mySock.close()
```

**Fig. 18.11** Echo client (`clientEcho.py`)

```
01 import socket
02 mySock = socket.socket()
03 host = '192.168.1.142'
04 port = 65535 # Reserve a port number
05 mySock.bind((host,port))
06 mySock.listen(1)
07 while True:
08 conn, addr = mySock.accept()
09 while True:
10 data = conn.recv(1024)
11 if data.decode() == 'EOF':
12 break
13 conn.send(data)
14 conn.close()
15 mySock.close()
```

**Fig. 18.12** Echo server (`serverEcho.py`)

In this example, we have considered two applications that reside on machines connected to the same WiFi hotspot, each having its unique IP address. The IP

address of a machine can be determined using `ipconfig` command. IP address assigned to Wireless LAN adapter of the server machine is required in each of the server and client programs. On executing the server program,

```
F:\PythonCode\Ch18>python serverEcho.py
```

Python responds with the following output on the client side:

```
F:\PythonCode\Ch18>python clientEcho.py
```

```
Enter data to be sent: Hello
```

```
Hello
```

```
Enter data to be sent: I love Python
```

```
I love Python
```

```
Enter data to be sent: I am connected to
echo server
```

```
I am connected to echo server
```

```
Enter data to be sent:
```

```
F:\PythonCode\Ch18>
```

output on executing script `clientEcho`

#### *Accessing Web Data (Downloading a Pdf File)*

Many times, we need to access documents on the web. For example, we may need to download a text or a pdf file. Using socket programming, we can write an application that connects to the server on the web and retrieves the content of the file. Communication over the Internet often uses the most common protocol – Hypertext Transfer Protocol (HTTP). The port number

used for HTTP connection is 80. For accessing data over the Internet (say, contents of a page), the user needs to send a GET request. A GET request needs to specify the following pieces of information: *url* of the page, protocol for communication and its version (*protocol/version*), and a blank line (thus, double newline).

```
GET url protocol/version \n\n
```

GET request for accessing data over the Internet

Suppose we need to download the Python manual containing python tutorial by Guido Van Rossum from website of University of Idaho. For this purpose, we need to connect to Idaho server (listening to requests) having hostname `marvin.cs.uidaho.edu` at port 80 (HTTP port). Together, the hostname and the port number define a socket. Also, we need to send GET request to the server for the pdf file with the url

```
http://marvin.cs.uidaho.edu/Teaching/CS515/pythonTutorial.pdf
```

The data received from this socket can be written to another pdf file on the system. The data will be received and written to the file as long as the entire content of the file is not transferred (Fig. 18.13). Note that we open the new file `PythonGuidoVanRossum.pdf` in '`wb`' (write, binary) mode (line 10) since pdf files are binary files.

pdf files should be opened in binary mode

```
01 import socket
02
03 mySock = socket.socket()
04 host = 'marvin.cs.uidaho.edu'
05 port = 80 # HTTP port
06 mySock.connect((host, port))
07 mySock.send('GET
http://marvin.cs.uidaho.edu/Teaching/CS515/pythonTutorial.pdf
HTTP/1.0\r\n'.encode())
08 fp = open('PythonGuidoVanRossum.pdf', 'wb')
09 while True:
10 data = mySock.recv(512)
11 if (len(data)<1):
```

```
12 break
13 fp.write(data)
14 fp.close()
15 mySock.close()
```

**Fig. 18.13** Download pdf file (pdfDownload.py)

Python library `urllib.request` simplifies the task of manually creating a socket, connecting to a server, and sending and receiving data using the `socket` library. The library treats a web page as a file. It provides the method `urlopen` for specifying the web page to be accessed. The function returns a file-like object unless there is an error in connecting to the server, in which case it returns `IOError`. The script

`pdfDownloadUrllib` (Fig. 18.14) can be used to download a pdf file using the `urllib` library. The method `urlopen` returns a file-like object which is an iterator. We have used `writelines` to write `fileContent` to file `PythonGuidoVanRossum.pdf`. Alternatively, `for` loop could have been used to iterate over `fileContent` and write the content to pdf file line by line.

`urllib`: module for accessing data over web

```
1 import urllib.request
2 fileContent = urllib.request.urlopen(' http://marvin.cs.uidaho.
3 edu/Teaching/CS515/pythonTutorial.pdf')
4 fp = open('PythonGuidoVanRossum.pdf', 'wb')
5 fp.writelines(fileContent.read())
6 fp.close()
```

**Fig. 18.14** Download pdf file (`pdfDownloadUrllib.py`)

#### 18.3 MANAGING DATABASES USING STRUCTURED QUERY LANGUAGE (SQL)

In almost every field, be it, business, railways, healthcare, education, or defense, we have been dealing with data, indeed much before computers were invented. Such data is required to be stored and managed and to be accessed. Since any decision making by an organization is highly dependent on the timely availability of information, the need was felt for a system which could store and facilitate management of all related data. Database Management System (DBMS) helps in storing the collection of related data in the database and managing it through application programs. Structured Query Language (SQL) is widely used for database creation and management. Several popular database systems such as Oracle, MySQL,

Microsoft SQL Server, and SQLite exist, however, in this section, we will be using SQLite database already integrated with Python.

DBMS: helps in storing and managing related data

SQL: language for database creation and management

### 18.3.1 Database Concepts

Suppose we wish to store information about a College. Let us name the database for the college as COLLEGE . It may need to store information about students such as roll number, name, and percentage. A real-world object such as STUDENT, about whom the data is to be stored in the database is known as an **entity**, and the characteristics of this entity such as roll number, name, and percentage are called **attributes**. The database COLLEGE may store information about several entities such as Students, Teachers, Courses, and Departments in the form of tables. A two-dimensional table of rows and columns in the database is called a **relation**.

entity : real-world object about which data is to be stored

attributes: characteristics of entity

relation: two-dimensional table of rows and columns

**Table 18.5** Relation STUDENT

| RollNum | Name   | Percentage |
|---------|--------|------------|
| 1       | Hiten  | 89         |
| 2       | Muskan | 85         |
| 3       | Nidhi  | 78         |
| 4       | Nikhil | 55         |
| 5       | Deepti | 95         |

Examine the relation STUDENT (Table 18.5). Each column of the table denotes a unique **attribute**, for example, RollNum, Name, and Percentage. Each row in the table stores information about all the attributes of an entity and is known as a **tuple**. The number of attributes in a relation is known as the **degree** of the relation. The collection of rows in a relation is called entity set of a relation. The number of rows in the entity set of a relation defines **cardinality** of the relation. Each column attribute is of a particular type such as numeric type or character type. All the values in a column must conform to a predefined data type. For example, the attributes RollNum and Percentage are numeric, and the attribute Name is of string type. Each row in a table is uniquely identified by an attribute or a combination of attributes called **key**. Therefore, no two tuples should have the same key. For example, the value of attribute RollNum of relation STUDENT is unique for each row, and thus, it is a key. If the value of an attribute is unknown for a tuple, a special value called NULL (no value indicator) should be used.

tuple: a row in a relation

degree: number of attributes in a relation

cardinality: number of tuples in a relation

key: attribute(s) that uniquely identify tuples in a relation

Structured Query Language (**SQL**) is a relational database language. It is used for creation and manipulation of the database. It is a high-level language that allows users to specify what is required to be done in the form of queries, without the need to specify the procedure: how it is to be done. **SQL** is a widely used database language and has numerous advantages.

SQL is a relational database language

Based on the functionality, **SQL** operations are organized into two categories, namely, Data Definition Language (**DDL**) and Data Manipulation Language (**DML**) as described below:

#### 1. **Data Definition Language (DDL)**

**DDL** is used for defining database structure and specifying constraints. **DDL** includes those commands of the database that deal with the defining and modifying database structures. For example, **DDL** command `CREATE` can be used to create a table:

`DDL`: used to define and modify database structure

```
CREATE TABLE table-name
```

Other DDL commands include `ALTER` and `DROP`.

#### 2. **Data Manipulation Language (DML)**

**DML** is used for retrieval and modification of data. It includes commands that help to manipulate the database. It provides functions that deal with (i) retrieving data from relations, and (ii) modifying the database by inserting, updating, or deleting it. Examples of **DML** commands include `INSERT`, `SELECT`, `DELETE`, and `UPDATE`.

`DML: used to retrieve and modify data`

### 18.3.2 Creating Database and Tables

Before we can use a database for storing or manipulating information, we need to create the database structure. This step is called metadata (data about data) creation. The following lines of code can be used for creating and connecting to the COLLEGE database.

`metadata: data about data`

```
import sqlite3

conn = sqlite3.connect('COLLEGE.db')

cur = conn.cursor()
```

`sqlite3: module for database management`

First, we import module `sqlite3` which provides the functionality required for database management. Next, we need to establish a connection to the database having the name `COLLEGE.db` (specified as an argument). The database name should have extension `db`. Thus, the second line in above segment of code connects to database `COLLEGE.db`. Thus, we have named this connection `conn`. If the database mentioned already exists in the current directory, a connection is established with the already existing database, otherwise a new

database is created. For performing operations on the data stored in the database, we need a handler. The class `cursor` associated with the connection identifier `conn` returns a handler/cursor object (`cur` in this case). To close the connection, we use the method `close` associated with the `conn` object:

```
connect() : establishes database connection
```

```
conn.close()
```

```
close() : closes the database connection
```

The method `execute` of the cursor object is used for executing **SQL** commands. However, the method can be used for executing only one **SQL** command at a time. If one wishes to execute multiple **SQL** commands, the method `executescript` may be used. The method `execute` allows the use of several parameters discussed in subsequent sections.

```
execute() : a method of the cursor object for executing SQL
commands
```

Having created the database `COLLEGE`, let us now create relations corresponding to different entities involved. `CREATE TABLE` command is used for creating relations (tables):

```
CREATE TABLE table_name (
 attribute1 data_type [constraint] [,
 attribute2 data_type [constraint]] [,
```

```
table_constraints]
```

```
) ;
```

syntax of CREATE TABLE command

In the description of **SQL** command, a pair of square brackets ( [ ] ) denotes optional part. Thus, in the CREATE TABLE command, there may be any number of attributes or constraints. The attributes and the constraints are specified within parenthesis and are separated by commas. Each attribute name is followed by the type of data it contains. The data type for a given attribute determines the domain of values it can take. A data type may include INT (integer) for a numeric value, VARCHAR (a *sequence of characters*) for a string value, and DATE for the date value. Data type may further be followed by constraints on attributes such as UNIQUE and NOT NULL. Each SQL command terminates with a semicolon. However, we may skip semicolon when a command is executed by a Python script.

domain: the values, an attribute can take

Suppose we wish to create the table STUDENT. The following **SQL** command may be used this purpose:

```
CREATE TABLE STUDENT(
 RollNum INT NOT NULL UNIQUE,
 Name VARCHAR(20) NOT NULL,
 Percentage INT NOT NULL,
 PRIMARY KEY(RollNum)
);
```

SQL command for creating STUDENT table

Since each student has a unique roll number, no two students can have the same value of the attribute RollNum. So, RollNum must be unique across all tuples of STUDENT table. As we would like to use RollNum to identify a student uniquely, this attribute cannot be NULL. So, we apply column constraints UNIQUE and NOT NULL to the attribute RollNum. The attribute RollNum has been declared primary key of the table in the last line of CREATE TABLE command. Also, RollNum being the primary key, column constraints UNIQUE and NOT NULL automatically hold for it, and thus, do not need to be mentioned explicitly.

no two tuples in a database can have the same primary key

value of attribute(s) that form the primary key must not be NULL

If the primary key comprises a single attribute, we may include the phrase PRIMARY KEY while defining

the attribute in the CREATE command, as shown below:

```
CREATE TABLE STUDENT (
 RollNum INT PRIMARY KEY,
 Name VARCHAR(20) NOT NULL,
 Percentage INT NOT NULL
);
```

SQLite does not enforce strict type requirement. To ensure strict type conformity, CHECK constraint must be associated with an attribute.

use CHECK to enforce type constraint

```
CREATE TABLE STUDENT (
 RollNum INT CHECK(TYPEOF(RollNum) =
 "integer") PRIMARY KEY ,
 Name VARCHAR(20) CHECK(TYPEOF(Name) =
 "text") NOT NULL,
 Percentage INT CHECK(TYPEOF(Percentage)
 = "integer") NOT NULL
);
```

In the script SQLCommands (Fig. 18.15), we create the database COLLEGE having a table STUDENT.

```
01 import sqlite3
02
03 def main():
04
05 # Defining and manipulating a database
06
07 # Establish database connection
08 conn = sqlite3.connect('COLLEGE.db')
09 print('Database Connected Successfully!!')
10
11 # Create a cursor
12 cur = conn.cursor()
13
14 # Create a table
15 cur.execute('CREATE TABLE STUDENT\
16 RollNum INT CHECK(TYPEOF(RollNum) = "integer") PRIMARY
17 KEY , \
18 Name VARCHAR(20) CHECK(TYPEOF(Name) = "text") NOT NULL, \
19 Percentage INT CHECK(TYPEOF(Percentage) = "integer") NOT
20 NULL\
21);')
22 print('Table Created Successfully!!')
```

```
21
22 # Close the connection
23 conn.close()
24
25 if __name__ == '__main__':
26 main()
```

**Fig. 18.15** Creating database COLLEGE and table STUDENT  
(SQLCommands.py)

### 18.3.3 Inserting Data into Table

Having created the tables, we would like to store the data in the database. The process of storing the data in the database is called loading the database or populating the database. **SQL** command **INSERT** is used for this purpose. The syntax for **INSERT** command is given below:

```
INSERT INTO table_name
VALUES (value1,value2, ..., valueN);
```

syntax of **INSERT** command

Given the name of a table and associated attribute values in the form a tuple, **INSERT** command inserts a tuple in the table. The order of values within the tuple should correspond to the order of the attributes of the table. The character and date type values should be enclosed in single quotes. When the value of an attribute is not known/missing/not applicable for a tuple, we use the **NULL** value. For example, suppose we wish to insert information about a Student having RollNum 1, Name Hiten, and Percentage 89. To do this, we may use the following **INSERT** command:

```
INSERT INTO STUDENT
VALUES (1, 'Hiten', 89);
```

In the script `SQLCommands` (Fig. 18.16), we insert five rows in the table `STUDENT`.

```
1 # Inserting data into table
2 cur.execute("INSERT INTO STUDENT VALUES (1, 'Hiten', 89);")
3 cur.execute("INSERT INTO STUDENT VALUES (2, 'Muskan', 85);")
4 cur.execute("INSERT INTO STUDENT VALUES (3, 'Nidhi', 78);")
5 cur.execute("INSERT INTO STUDENT VALUES (4, 'Nikhil', 55);")
6 cur.execute("INSERT INTO STUDENT VALUES (5, 'Deepti', 95);")
7 print('Data Inserted Successfully!!!')
8
9 conn.commit()
```

**Fig. 18.16** Inserting data in `STUDENT` table (`SQLCommands.py`)

In `sqlite3`, `commit` method associated with the object `connect` is used for forcefully saving the current state of the database, thus making the changes permanent. For example, in the script `SQLCommands` (Fig. 18.16), `commit` method is executed after inserting the five tuples in the table. This ensures that the tuples just inserted in the table `STUDENT` of the database become immediately visible to other connections to this database.

`commit()` : to forcefully save current state of the database

Next, we present another format for inserting a tuple into a table using a question mark. For example, we may add a tuple in the table `STUDENT` using the question

mark as a placeholder for the data being provided as the second argument to method `execute`:

? : question mark may be used as a placeholder in an SQL command

```
rollNum = int(input('Enter roll number:\n'))\n\nname = input('Enter name: ')\\n\npercent = int(input('Enter percentage: '))\n\ncur.execute("INSERT INTO STUDENT VALUES\\\n(?, ?, ?);", (rollNum, name, percent))
```

#### 18.3.4 Retrieving Data from Table

`SELECT` command is used to retrieve data from the database tables. The syntax of `SELECT` command is given below:

```
SELECT attribute_list\n\nFROM relation_list;
```

-

syntax of `SELECT` command

`attribute_list` is a comma-separated list of attribute names whose values are to be retrieved. For example, to retrieve roll number, name, and percentage of students, we use the following **SQL** command:

```
SELECT RollNum, Name, Percentage
```

```
FROM STUDENT;
```

In the above command, first line comprising of keyword SELECT along with a list of attributes RollNum, Name, Percentage is called SELECT clause, and the second line is called FROM clause. On executing the above command, SQL will respond with the relation given in Table 18.6.

**Table 18.6** Relation STUDENT

| RollNum | Name   | Percentage |
|---------|--------|------------|
| 1       | Hiten  | 89         |
| 2       | Muskan | 85         |

| RollNum | Name   | Percentage |
|---------|--------|------------|
| 3       | Nidhi  | 78         |
| 4       | Nikhil | 55         |
| 5       | Deepti | 95         |

When we wish to extract all the attributes of a relation, we may use the wild card character \* (asterisk) as a substitute for all the attributes occurring in the relation(s). Thus, the above SELECT command may be rewritten as follows:

```
SELECT *
```

```
FROM STUDENT;
```

**\***: wild card character, used in an SQL command as a substitute for all the attributes occurring in the relation(s)

Next, suppose that we wish to list only roll numbers and names of the students in the college. For this purpose, we need to specify in the **SELECT** clause, the relevant attributes **RollNum** and **Name** of the relation **STUDENT**, as shown below:

```
SELECT RollNum, Name
FROM STUDENT;
```

Next, suppose we wish to retrieve information about students who have secured percentage more than 80. **WHERE** clause in an **SQL** query facilitates retrieval of only those tuples which satisfy a condition such as **Percentage > 80**. Thus, the following **SQL** query would retrieve the desired information:

**WHERE** clause: used to specify condition for selection of tuples

```
SELECT *
FROM STUDENT
WHERE Percentage > 80;
```

The syntax of **SELECT** command with **WHERE** clause is given below:

```
SELECT attribute_list
FROM relation_list
WHERE condition_list;
```

syntax of SELECT command involving WHERE clause

Here, *condition\_list* may comprise a single condition, or several conditions joined using Boolean operators: AND, OR, and NOT. The output of an **SQL** query returns values of attributes mentioned in attribute list for the tuples of table that satisfy the condition(s) specified in the WHERE clause.

Instead of retrieving the information interactively, we may use a Python script such as SQLCommands (Fig. 18.17) to retrieve the information from the table STUDENT.

```

01 # Retrieving data from table
02 print('Output of Select Queries')
03 print('Retrieving roll numbers, names, and percentages of
04 students')
05 cur.execute('SELECT RollNum, Name, Percentage FROM\
06 STUDENT;')
07 print(cur.fetchall())
08
09 print('Retrieving all attribute values of students')
10 cur.execute('SELECT * FROM STUDENT;')
11 print(cur.fetchall())
12
13 print('Retrieving roll numbers and names of students')
14 cur.execute('SELECT RollNum, Name FROM STUDENT;')
15 print(cur.fetchall())
16
17 print('Retrieving all attribute values of students with\
18 percentage greater than 80')
19 cur.execute('SELECT * FROM STUDENT WHERE Percentage > 80;')
20 print(cur.fetchall())

```

**Fig. 18.17** Retrieving data from STUDENT table  
(SQLCommands.py)

The result returned by an SQL query may be accessed using the method `fetchall` that returns a list of tuples (line 6, Fig. 18.17). Alternatively, we may retrieve the data from the database row by row by iterating through the `cursor` object as shown below:

```
fetchall(): returns a list of tuples as the result of executing
an SQL query
```

```
for row in cur:
 print(row)
```

The method `execute` also allows the use of named placeholders for SQL literals. For example, the query

```
cur.execute('SELECT * FROM STUDENT WHERE
Percentage > 80;')
```

may be rewritten as follows:

```
cur.execute('SELECT * FROM STUDENT WHERE
Percentage > :value;', {'value':80})
```

use of named placeholder

In the above query, the method `execute` makes use of the name `value` as a placeholder for the actual data 80 that appears as the second argument.

#### 18.3.5 Updating Data in a Table

`UPDATE` command is used to modify the attribute value(s) in one or more tuples. Given the name of the relation to be modified and an optional `WHERE` clause to select a particular set of tuples, the `UPDATE` command updates value(s) of one or more attributes. The `UPDATE` command uses the following syntax:

syntax of `UPDATE` command

```
UPDATE relation
SET attribute-value_pairs
[WHERE condition_list1];
```

For example, to increment percentage of all students having a percentage less than 50% by 2%, we use the following **SQL** command:

```
UPDATE STUDENT
SET Percentage = Percentage * 1.02
WHERE Percentage < 50;
```

In the UPDATE command, SET clause may take any number of comma-separated attributes and their updated values. In a Python program, the following statements can be used for updating data:

```
cur.execute('UPDATE STUDENT \
SET Percentage = Percentage * 1.02 \
WHERE Percentage < 50;')
```

#### 18.3.6 Deleting Data from Table / Deleting Table

DELETE command is used to delete the tuples from a relation which satisfy the condition appearing in the optional WHERE clause. In case of absence of WHERE clause, all the tuples of the table mentioned in DELETE command will be deleted. The DELETE command has the following syntax:

syntax of DELETE command

```
DELETE FROM relation_name
[WHERE condition_list1];
```

Suppose a student with roll number 3 has left the college. We may delete the information about that student using the following **SQL** command:

```
DELETE FROM STUDENT
WHERE RollNum = 3;
```

The command

```
DELETE FROM STUDENT;
```

deleting entire data from the table

will delete the entire data from the table STUDENT. If we wish to delete the table itself along with all its tuples, we may use the **DROP TABLE** command. For example, the following **SQL** command will delete the entire table STUDENT:

```
DROP TABLE STUDENT;
```

deleting the table

#### 18.4 DEVELOPING MOBILE APPLICATION FOR ANDROID

In this section, we will study how Python can be used for creating applications for android. For illustrating this, we will develop the popular two-player game Tic-Tac-Toe, also known as the game of Xs and Os. In this game, the player who successfully places three consecutive symbols horizontally, vertically or diagonally wins the game. For developing the game, we will first create a cross-platform application of the game that can be executed on the Android device. Before developing this game, we will create a registration interface for the user.

```
use module kivy for building cross platform applications
```

Python provides an open source library `kivy` for building cross-platform applications. It is a graphical user interface toolkit which supports desktop environments such as Windows®, Linux, and Mac OS®. It also supports multi-touch devices such as Android and IOS. Next, we describe how to install `kivy` in Windows environment. For this purpose, we use a sequence of commands from Windows command line interface (<Windows key +R>, cmd). First, we need to upgrade `wheel` and `pip` libraries to the latest versions. The first command upgrades `wheel` and `pip` to latest versions along with `setuptools` and `tokenize`. Smooth execution of a software package may require some related software/files. These are called dependencies. The second and third command downloads and installs all the dependencies of `kivy`.

```
python -m pip install --upgrade pip wheel
setuptools tokenize
```

`kivy` installation

```
python -m pip install docutils pygments
pypiwin32 kivy.deps.sdl2 kivy.deps.glew
kivy.deps.gstreamer cython --extra-index-
url
https://kivy.org/downloads/packages/simple
/
```

```
python -m pip install kivy.deps.angle
```

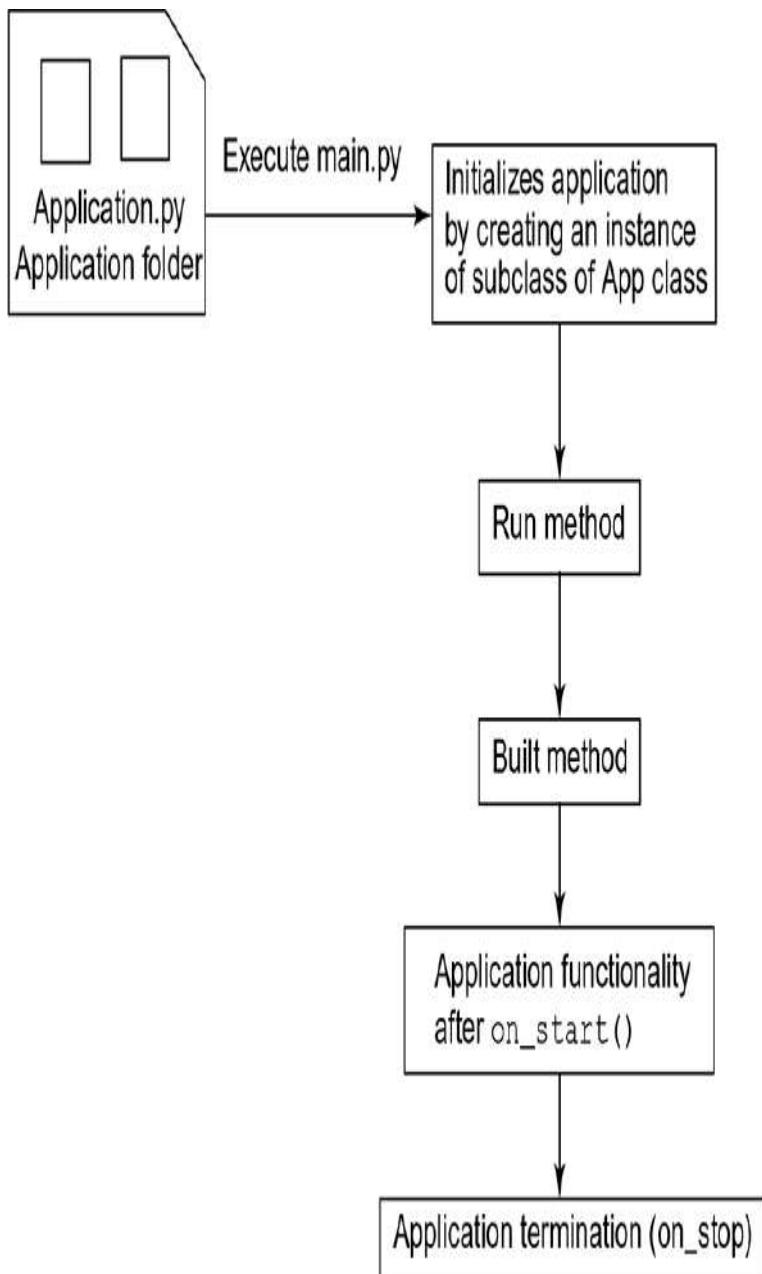
Finally, for installing `kivy` package, download the nightly wheel from the link  
<https://kivy.org/downloads/appveyor/kivy/Kivy-1.9.2.devo-cp36-cp36m-win32.whl> and install it using the following command (the file just downloaded from `kivy.org` should be available in the working directory and

needs to be moved to working directory, if downloaded in a different folder):

```
python -m pip install Kivy-1.9.2.dev0-
cp36-cp36m-win32.whl
```

To develop an application, we create a folder with a suitable name containing the application titled `main.py`. During deployment, build tools search for a file named `main.py`. The `App` class is the base class for developing `kivy` applications containing important methods for initializing, launching, building, stopping, pausing, and resuming applications. So, whenever we wish to define an implementation of an Android application as a class, it should inherit the `App` class of the `kivy.app` module. The `run` method available in the `App` class defines the entry point of a `kivy` application and is used for launching the application. Fig. 18.18 illustrates the execution sequence of a `kivy` application.

`App` class: used as a base class for developing `kivy` applications



**Fig. 18.18** Kivy Application that displays a button (`main.py`)

An application will include several layouts and widgets (graphical objects) such as buttons, labels, checkboxes, and dropdowns required for defining the user interface. Widget arguments are called properties. These user interface elements are defined in `uix` module of `kivy`. Let us first create a simple application `main` (Fig. 18.19) that displays a clickable button with text `Hello`. For this purpose, we import the class `App`, available in the

module `kivy.app`, and develop a user defined class `CreateButtonApp` as a derived class of the class `App`. The method `build` of the user defined class `CreateButtonApp` overrides the method `build` of the parent class `App`. Thus, we make use of the method `Button` to create a button. This widget should be returned by overridden `build` method of the user-defined class. When the script executes, an instance of the user-defined class `CreateButtonApp` is created (line 17), which is used for invoking the method `run` (inherited from class `App`), which in turn starts the application by invoking the method `build` of `CreateButtonApp`.

`widgets`: graphical objects  
`widget arguments`: used to set properties

`ui`: module contains user interface elements

`build`: method used to initialize application with the widget tree

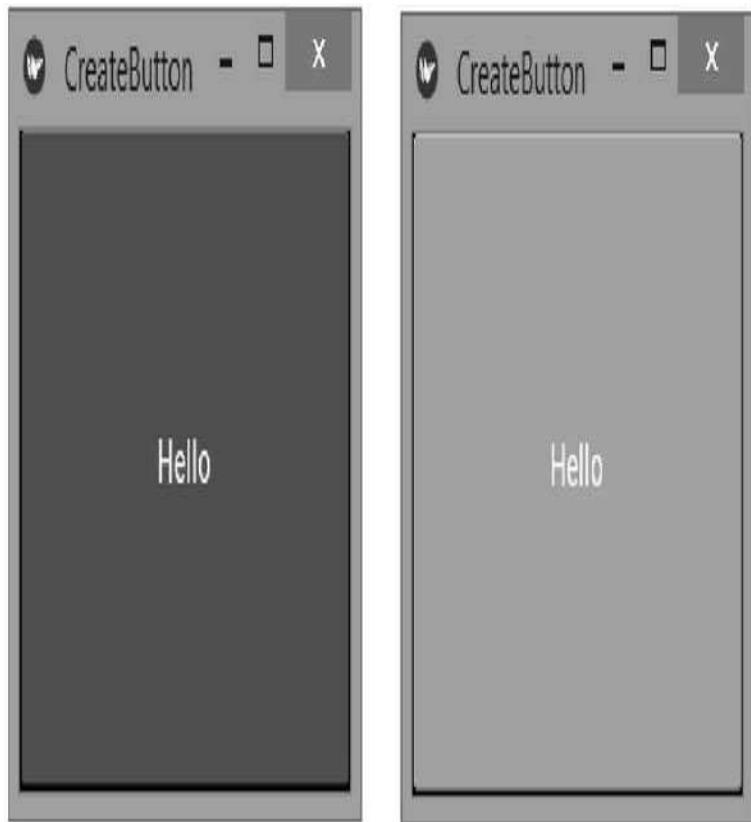
```
01 from kivy.app import App
02 from kivy.uix.button import Button
03
04 class CreateButtonApp(App):
05 # Main class instantiated for running application
06 def build(self):
07 '''
08 Objective: To initialize application with widget tree
09 Input Parameters:
10 self (implicit parameter) - object of type
11 CreateButtonApp
12 Return Value: instance of Button class (root widget)
13 '''
14 return Button(text = 'Hello')
15
16 if __name__ == '__main__':
17 CreateButtonApp().run()
```

**Fig. 18.19** Kivy application that displays a button (`main.py`)

When the method `CreateButtonApp().run()` is launched, it invokes the method `build` of the class `CreateButtonApp` that defines the user interface. The method `build` returns a root widget (button in this case) of our application. Note that the class `Button` is imported in line 2. In line 14, we have set the `text` property (argument) of the widget `Button` as '`Hello`'.

`build` method returns the root widget

On executing the script `main` (Fig. 18.19), Python responds with the following output shown in Fig. 18.20. Left window of the figure displays the button when the application starts. Right window displays the button when it is pressed.



**Fig. 18.20** Output of program 18.19

Another approach for using widgets and their definitions is to use separate files for specifying them (Fig. 18.21). Note that the widget definition is stored in a separate file having `.kv` as a suffix. The `.kv` file is named after the main app class containing `build` method. While name of a class may include uppercase letters, to name the `.kv` file, we use the same name as that of the class, but in lowercase letters and drop the suffix `App`, if included in the name of the class. The file contains the colon-separated properties and their values for each widget. Also, the root widget is enclosed in angular brackets followed by a colon.

widget definitions may be stored in a separate file having .kv  
suffix

```
1 <Button>:
2 text: 'Hello'
```

**Fig. 18.21** Widget definition (createbutton.kv)

The kivy application that displays a button appears in  
Fig. 18.22.

```
01 from kivy.app import App
02 from kivy.uix.button import Button
03
04 class CreateButtonApp(App):
05 # Main class instantiated for running application
06 def build(self):
07 """
08 Objective: To initialize application with widget tree
09 Input Parameters:
10 self (implicit parameter) - object of type
11 CreateButtonApp
12 Return Value: instance of Button class (root widget)
13 """
14 return Button()
15
16 if __name__ == '__main__':
17 CreateButtonApp().run()
```

**Fig. 18.22** Kivy application that displays a button (`main.py`)

#### 18.4.1 A Simple Application Containing Registration Interface

If you have been using a smart phone for sometime, you must have observed that several mobile apps require you to register with them. Next, we will develop a simple user interface for registration. On clicking the Register button, it prompts the user for username and password and displays a window with the message Registration Successful. However, while implementing this interface, we do not maintain a

database of valid usernames and passwords. Indeed, we not only disregard whether a valid user having the credentials entered by the user exists, but we also do not care whether the user entered the name and password information at all or just left these fields empty.

However, then you may wonder: what is the purpose of this application? Through this application, we intend to illustrate how several widgets can be added using a layout, and how an event can be invoked on clicking a button. In Fig. 18.23, we present an application program for displaying a user interface for registration. As this interface contains `button`, `label`, `textinput`, and `popup`, we need to import all these widgets in our script. For organizing the widgets in this program, we have used a grid layout comprising of one column. The class `GridLayout` of module `gridlayout` is used for defining this layout. Kivy also supports several other layouts such as `BoxLayout`, `StackLayout`, and `FloatLayout` which can be used for organizing the widgets.

adding several widgets using a layout

clicking a button invokes an event

```
01 from kivy.app import App
02 from kivy.uix.button import Button
03 from kivy.uix.label import Label
04 from kivy.uix.textinput import TextInput
05 from kivy.uix.popup import Popup
06 from kivy.uix.gridlayout import GridLayout
07
08 class Register(GridLayout):
09 # To display a registration interface
```

```
10
11 def __init__(self):
12 """
13 Objective: To initialize application with widget tree
14 Input Parameter:
15 self (implicit parameter) - object of type Register
16 Return Value: None
17 """
18 super(Register,self).__init__(cols=1,spacing=10,padding=
19 150)
20 self.add_widget(Label(text = 'User name'))
21 self.add_widget(TextInput(multiline = False))
22 self.add_widget(Label(text = 'Password'))
23 self.add_widget(TextInput(multiline = False, password =
24 True))
25 self.add_widget(Button(text = 'Register',\
26 on_press = self.displayPopup))
27
28 def displayPopup(self, btn):
29 """
30 Objective: To open a Popup window when the button is
31 clicked
32 Input Parameters:
```

```
31 self (implicit parameter) - object of type Register
32 btn - instance of Button
33
34 Return Value: None
35
36 """
37
38 myPopUp = Popup(title = 'Registration Status', \
39 content = Label(text = 'Registration Successful'), \
40 size_hint=(.5, .5))
41
42 myPopUp.open()
43
44
45 class RegistrationApp(App):
46 # Main class instantiated for running application
47
48 def build(self):
49 """
50
51 Objective: To initialize application with widget tree
52 Input Parameter:
53
54 self (implicit parameter) - object of type
55 CreateButtonApp
56
57 Return Value: instance of Register class (root widget)
58
59 """
60
61 return Register()
62
63
64 if __name__ == '__main__':
65 RegistrationApp().run()
```

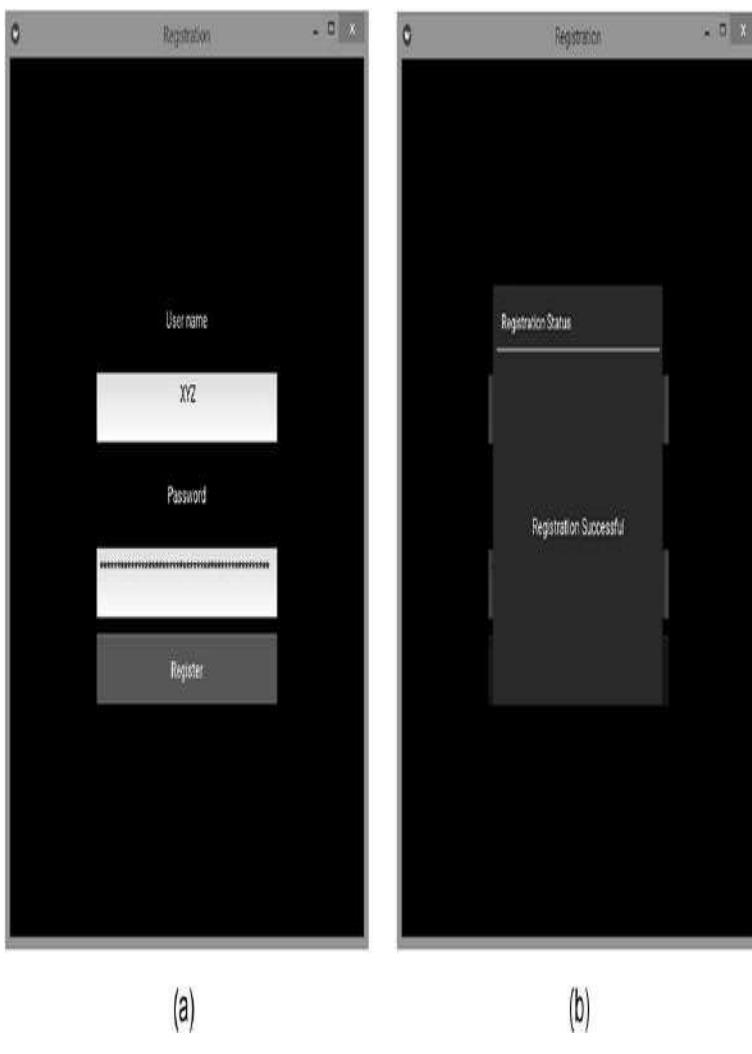
**Fig. 18.23** Kivy application that displays a register interface  
(main.py)

We have defined `RegistrationApp` as the main class containing the `build` method. The `build` method returns an instance of `Register` class as the root widget. `Register` class inherits the `GridLayout` class. In this class, we set the properties of the layout by invoking the `__init__` method (line 18). We have set the number of columns as 1, spacing between widgets as 10, and horizontal padding from the left and right side of the window as 150. In the method `__init__` of `Register` class, we add to the layout, widgets `label` and `textbox` for username as well as the password using `add_widget` method. Note that for the `TextInput` widget, the `password` property has been set to be `True` for hiding the password. In line 23, we add the `Button` widget. We have associated an event `on_press` with the button that invokes the method `displayPopup` when the button is clicked. The method `displayPopup` creates an instance of `Popup`. It has the title 'Registration Status', content `Label(text = 'Registration Successful')`, and a `size_hint` property that enables us to specify the height and width on a scale of 1. Here, we have set the `content` property to be an instance of the widget `Label`. Note that the method `displayPopup` contains two parameters – the implicit parameter `self` and the widget instance associated with it.

setting `GridLayout` properties

`displayPopup()` : method associated with event `on_press` of `Button` widget

On executing the script in Fig. 18.23, Python responds with the output given in Fig. 18.24. Left window (Fig. 18.24(a)) displays a user interface for registration, and the right window (Fig. 18.24 (b)) shows popup window which appears when Register button is clicked.



**Fig. 18.24** Output of program 18.23

Alternatively, we may specify kivy widget definitions in a separate file (Fig. 18.25), Register being the root widget.

widget definitions in a separate .kv file

```
01 <Register>:
02 cols:1
03 spacing:10
04 padding:150
05 Label:
06 text:'User name'
07 TextInput:
08 multiline:False
09 Label:
10 text:'Password'
11 TextInput:
12 multiline: False
13 password: True
14 Button:
15 text:'Register'
16 on_press:root.displayPopup()
```

**Fig. 18.25** Widget definitions (registration.kv)

The modified script `main` is shown in [Fig. 18.26](#).

```
01 from kivy.app import App
02 from kivy.uix.button import Button
03 from kivy.uix.label import Label
04 from kivy.uix.textinput import TextInput
05 from kivy.uix.popup import Popup
06 from kivy.uix.gridlayout import GridLayout
07
08 class Register(GridLayout):
09 # To display a register interface
10
11 def __init__(self, *args, **kwargs):
```

```

11 def __init__(self):
12 """
13 Objective: To initialize application with widget tree
14 Input Parameter:
15 self (implicit parameter) - object of type Register
16 Return Value: None
17 """
18 super(Register,self).__init__()
19
20
21 def displayPopup(self):
22 """
23 Objective: To open a Popup window when the button is clicked
24 Input Parameters:
25 self (implicit parameter) - object of type Register
26 Return Value: None
27 """
28 myPopUp = Popup(title = 'Registration Status',\
29 content = Label(text = 'Registration Successful'),\
30 size_hint=(.5, .5))
31 myPopUp.open()
32
33 class RegistrationApp(App):
34 # Main class instantiated for running application
35 def build(self):
36 """
37 Objective: To initialize application with widget tree
38 Input Parameters:
39 self (implicit parameter) - object of type CreateButtonApp
40 Return Value: instance of Register class (root widget)
41 """
42 return Register()
43
44 if __name__ == '__main__':
45 RegistrationApp().run()

```

**Fig. 18.26** Kivy application that displays a button (main.py)

#### 18.4.2 Tic-Tac-Toe Game

In this section, we will develop Tic-Tac-Toe game as an android application (Fig. 18.27). In this game, we have a grid of size 3X3. We have initialized a grid of this size to contain Button widgets (line 25). We have defined o and x as two symbols used by the two players. We

update the value of `count` every time a user makes a move. A user click invokes the method `action` that sets the corresponding symbol in the grid. It increments the number of moves, and checks if there is any winner at this stage. The check for the winner is performed by invoking the method `checkResult` (line 39). The method `checkResult` further invokes the method `getWinner`. In case there is a winner at this stage, it returns the symbol (O or X) corresponding to the winner, otherwise it returns `None`. Now a `closeButton` is created. If there is a winner, the `closeButton` displays the name of the winner. However, if there is no winner, but the game is over, the `closeButton` displays the message indicating that the game is finished. In either case, `checkResult` method displays a popup containing this `closeButton`. Note that in line 62 we have set `on_press` property of the button `myPopUp` to the `dismiss` method which closes the popup and `on_release` property to `restartGame` (lines 118–129) which re-initializes the game grid with blanks, and sets the number of moves for a new game to 0.

layout: GridLayout widgets: Button

a user click invokes the method `action` which sets the symbol on the grid and checks for the winner

setting `on_press` and `on_release` property of Button widget

Let us now examine how the `getWinner` method (lines 66–101) determines the winner of the game. The method checks if all cells in some row, column, forward diagonal or backward diagonal contain the same symbol

(○ or X) using the method `sameSymbol` (lines 104–116). If so, the method `getWinner` returns that symbol, otherwise, it returns `False`.

```
001 from kivy.app import App
002 from kivy.uix.button import Button
003 from kivy.uix.popup import Popup
004 from kivy.uix.gridlayout import GridLayout
005
006 class TicTacToe(GridLayout):
007 # To display a register interface
008
009 def __init__(self):
010 """
011 Objective: To initialize application with widget tree
012 Input Parameters:
013 self (implicit parameter) - object of type TicTacToe
014 Return Value: None
015 """
016 super(TicTacToe, self).__init__(cols=3, spacing=2)
017 self.count = 0 #Number of moves
```

```
018 self.symbols = ('X', 'O')
019 self.symbolNum = 0
020 self.grid = [[None for col in range(self.cols)]\
021 for row in range(self.cols)]
022 for row in range(self.cols):
023 for col in range(self.cols):
024 tile = Button(font_size=40, on_press = self.action)
025 self.grid[row][col] = tile
```

```
026 self.add_widget(tile)
027
028 def action(self, button1):
029 """
030 Objective: To set value when button is clicked
031 Input Parameters:
032 self (implicit parameter) - object of type TicTacToe
033 button1 - instance of Button
034 Return Value: None
035 """
036
037 button1.text = self.symbols[self.symbolNum]
038 self.symbolNum = (self.symbolNum+1)%2
039 self.count += 1
040 self.checkResult()
041
042 def checkResult(self):
043 """
044 Objective: To determine if someone has won
045 Input Parameter:
046 self (implicit parameter) - object of type TicTacToe
047 Return Value: None
048 """
049
050 winner = self.getWinner()
051 if winner:
052 closeButton = Button(text = ' '+winner+' won !!\
053 \nClick here to restart')
054 elif self.count == (self.cols*self.cols):
055 closeButton = Button(text = 'Game finished!!\
056 \nClick here to restart')
057 else:
058 return
059 # If there is a winner or game has finished
060 myPopUp = Popup(title = 'Result of Game', content =
061 closeButton,\n size_hint=(.5, .5))
062 myPopUp.open()
```

```
062 closeButton.bind(on_press = myPopUp.dismiss, on_release =\
063 self.restartGame)
064
065
066 def getWinner(self):
```

```
067 """
068 Objective: To determine winner of the game
069 Input Parameter:
070 self (implicit parameter) - object of type TicTacToe
071 Return Value: symbol(X or O) if someone is winning,
072 False otherwise
073 """
074 gridValues = [[self.grid[row][col].text for col in
075 range(self.cols)] for row in range(self.rows)]
076 #Check row wise
077 for row in range(self.rows):
078 result = self.sameSymbol(gridValues[row])
079 if result:
080 return result
081
082 #Check column wise
083 for col in range(self.cols):
084 column = [row[col] for row in gridValues]
085 result = self.sameSymbol(column)
086 if result:
087 return result
088
089 #Check forward diagonal
090 diag = [gridValues[i][i] for i in range(self.rows)]
091 result = self.sameSymbol(diag)
092 if result:
093 return result
094
095 #Check backward diagonal
096 diag = [gridValues[i][(self.rows-1)-i] for i in range(self.rows)]
097 result = self.sameSymbol(diag)
098 if result:
099 return result
100
101 return False
102
103
104 def sameSymbol(self, valueList):
```

```
105 """
106 Objective: To find whether all symbols in valueList are same
107 Input Parameter:
```

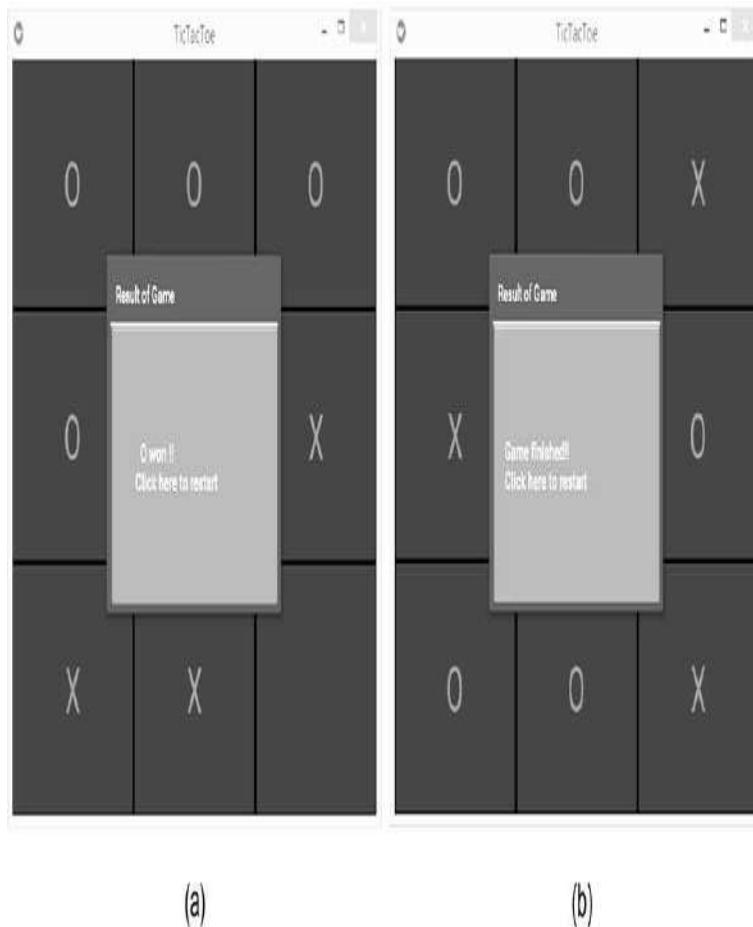
```

108 self (implicit parameter) - object of type TicTacToe
109 Return Value: symbol(X or O) if all symbols are same
110 False otherwise
111 """
112 symbol = valueList[0]
113 for element in valueList:
114 if element != symbol:
115 return False
116 return symbol
117
118 def restartGame(self, btn):
119 """
120 Objective: To restart game
121 Input Parameters:
122 self (implicit parameter) - object of type TicTacToe
123 btn - instance of Button
124 Return Value: None
125 """
126 self.count = 0 # No moves taken in new game
127 for row in range(self.cols):
128 for col in range(self.cols):
129 self.grid[row][col].text = ''
130
131
132 class TicTacToeApp(App):
133 # Main class instantiated for running application
134 def build(self):
135 """
136 Objective: To initialize application with widget tree
137 Input Parameter:
138 self (implicit parameter) - object of type TicTacToeApp
139 Return Value: instance of TicTacToe class (root widget)
140 """
141 return TicTacToe()
142
143 if __name__ == '__main__':
144 TicTacToeApp().run()

```

**Fig. 18.27** Kivy application that displays a register interface  
(main.py)

On executing the script `main` (Fig. 18.27), Python responds with the output given in Fig. 18.28. We have shown the output in each of the two scenarios: when there is a winner (Fig. 18.28(a)) and when the game results in a draw (Fig. 18.28(b)).



**Fig. 18.28** Output of program 18.27

#### 18.4.3 Running Kivy Applications on Android

Android play store has an application `Kivy Launcher` that enables us to run Python `kivy` applications on android devices. The `kivy` applications are stored in the directory `kivy` in the `/sdcard/kivy/<yourapplication>`. A `kivy` application developed on the desktop can be run on `Kivy Launcher` by creating a container folder (say, `/sdcard/kivy/TicTacToe`) and pasting the application including `main.py` in the directory

/sdcard/kivy/TicTacToe. Apart from the file main.py, the application folder should also contain a file named, android.txt that includes the title, author, and layout (orientation) of the application in the following format:

Kivy Launcher on android play store enables us to run kivy applications developed on Desktop

```
title=<Application Title>

author=<Your Name>

orientation=<portrait|landscape>
```

Next, go to Kivy Launcher and choose the project you wish to execute.

An alternative option to run a Desktop application on android is to convert it to android compatible application. Since android platform supports APK applications, the Python application must be first converted to this format. A kivy application i.e. a Python source code can be converted to APK application using buildozer tool which simplifies the task at hand. However, as of now, kivy applications for android can only be built in the Unix environment. The sequence of steps for building an APK file from buildozer is available on the website <https://kivy.org/docs/guide/packaging-android.html>. The APK can also be built using python-for-android.

converting Python application to APK

While programming in a language, a developer may feel the need to use some functionality which may have better support in another language. For example, suppose, an application has already been developed in Java, and we wish to use it in Python code. To invoke an existing Java application in Python, we need a bridge between Python and Java. Packages like `Py4j`, `Pyjnius`, `Jpype`, `javabridge`, and `JCC` help in invoking Java programs from Python. Also, since Java provides a wide variety of collections, we can directly use them in a Python program by including their java packages in the program.

for invoking Java functionality from Python, use a package like `Py4j`, `Pyjnius`, `Jpype`, `javabridge`, `JCC`

We will make use of `py4j` for invoking Java functionality from Python. It can be installed by executing the following command from command line:

use of package `py4j` for invoking Java functionality from Python

`pip install py4j`

Since we are using `Eclipse` as the development environment for writing Java programs, we need to include the `.jar` file of `py4j` in the Java project. The `.jar` file is typically available in the directory `C:\Users\<User_Name>\AppData\Local\Programs\Python\Python36-32\share\py4j`. We can add this file by right-clicking on the project in the `Package Explorer` panel, and selecting option `Add External Archives` in the `Build Path` option. It is important to point out that `py4j` supports Java version 6 and above only.

include jar file of py4j in Java project

Let us examine a simple Java program `HelloClass.java` ([Fig. 18.29](#)) that has a method `Message` which returns `Hello`.

for accessing Java class in a Python program:  
in Java program, instantiate `GatewayServer` class of library  
`py4j`

To make the `HelloClass` accessible in a Python program, two things are required. First, the Python program should be able to access the Java Virtual Machine (JVM) running the Java program. This requires an instance of the `GatewayServer` class available in the library `py4j` that makes this communication possible. Second, an entry point should be mentioned in the call to the constructor of the class `GatewayServer`. This entry point can be any object we wish to deal with in a Python program. In our case, an object of `HelloClass` serves as an entry point (line 10, [Fig. 18.29](#)). Here it is important to mention that a Python program will be able to use a Java program only if it (Java program) is ready to accept incoming requests. Therefore, in line 11, we start the gateway using the method `start`. Also, for communication to take place, the Java program must be compiled and executed before the Python program.

```
01 import py4j.GatewayServer;
02 public class HelloClass
03 {
04 public String Message()
05 {
06 return "Hello";
07 }
08 public static void main(String[] args)
09 {
10 GatewayServer gatewayServer = new GatewayServer(new
11 HelloClass());
12 gatewayServer.start();
13 System.out.println("Gateway Server Started");
14 }
15 }
```

**Fig. 18.29** Java program (`HelloClass.java`)

for communication to take place, Java program must be compiled and executed before the Python program

The script `hello.py` (Fig. 18.30) uses the Java program that we discussed above. For this purpose, we need to create an instance of the class `JavaGateway` available in the module `py4j.java_gateway`. The first two lines of the Python script `hello` achieve this. When we invoke `JavaGateway` (line 2), Python tries to connect with JVM of Java program already running and returns an instance. We can access the entry point object

using the member `entry_point` (line 3). This object now enables us to use Java functionality in the Python program (line 4). However, if no JVM is waiting for a connection, an error message `Py4JNetworkError` is displayed.

```
1 from py4j.java_gateway import JavaGateway
2 gateway = JavaGateway()
3 msgObjectFromJavaApp = gateway.entry_point
4 print(msgObjectFromJavaApp.Message())
```

**Fig. 18.30** Python applications that uses a method of java program (`hello.py`)

On running the script `hello` (Fig. 18.30), Python outputs `Hello`. If the error message address already in use appear while running the Java program, we need to explicitly specify a new port number (between 1025 and 65535) in each of the Java and Python programs. For example, in the Java program `HelloClass` (Fig. 18.29), we may replace line 10 by:

explicitly specifying new port number

```
GatewayServer gatewayServer = new
GatewayServer(new HelloClass(), 25539);
```

and in the Python script `hello` (Fig. 18.30), we replace line 2 by:

```
gateway =
JavaGateway(gateway_parameters=GatewayPara
meters(port=25539))
```

For setting the `gateway_parameters` explicitly, we need to import `GatewayParameters` along with `JavaGateway` (Fig. 18.34). Since we need to compile

and execute Java program first and then the Python program, we create a batch file (say, HelloApp.bat) comprising these two tasks one after the other. Since compiling and running an eclipse program from the command line is a more complex job, we can convert the entire project to an executable jar file which can be used anywhere. For this purpose, we right-click on the Java project (say, DemoProject) in the Package Explorer, and choose the following options in a sequence: the Export option, Runnable JAR file option. In the next window, choose the appropriate Launch configuration and export destination folder where you wish to save the jar file, say DemoProject.jar. This creates an executable jar file. We create a .bat file that has the following sequence of commands.

making a jar file makes the use of Java app easier.

```
@echo off

start cmd /k java -jar
<CompletePath>\DemoProject.jar

start cmd /k python
<CompletePath>\hello.py
```

batch file

Note that the second and third commands in the above sequence, i.e.,

java -jar <CompletePath>\DemoProject.jar

and

Python <CompletePath>\hello.py

are executed in command prompt using `start` command followed by parameter `cmd`. Further, use of the option `/k` prevents the application initiated, from involuntary closure until explicitly closed by the user. The batch file so created can be run by double-clicking it.

#### 18.5.1 Accessing Java Collections and Arrays in Python

Java supports arrays and several collections such as list, set, and hash map. Let us create a Java array of type `int` in Python. The method `new_array` associated with the object `gateway` created in `Hello.py` can be used for creating this array. The method takes Java type and size of the array as an input. We have created an `int` array of size `2` and initialized it with values `0` and `1`. Once we have created a Java array in Python, we can iterate over this array, much like a standard sequence (list, tuple, range, etc.) in Python as shown below:

```
from py4j.java_gateway import JavaGateway

gateway = JavaGateway()

intArray =
gateway.new_array(gateway.jvm.int,2)

intArray[0] = 0

intArray[1] = 1

for value in intArray:

 print(value)

using Java array in Python
```

On executing the above Python code (after starting Java Gateway Server), Python responds with the following output:

Similarly, we can also create a Java list using the gateway object. Subsequently, we may add elements to that list, either by using append method of Python list or by using add method of Java as shown below:

```
#Creating and accessing Java list
lst = gateway.jvm.java.util.ArrayList()
lst.append('red') #Calling Python method
lst.add(3) #Calling Java Method
print(lst)
```

creating a Java list in Python

Python responds with the following output on the execution of above piece of code:

```
['red', 3]
```

#### 18.5.2 Converting Python Collections to Java Collections

Suppose, we have a Python list that we need to pass to a Java method. For this purpose, we must convert Python list to Java compatible list. Module `py4j.java_collections` provides classes such as `SetConverter`, `MapConverter`, and `ListConverter`, which make this conversion possible. Let us examine the program in Fig. 18.31, where we create a Python list and convert it to a Java list by using `convert` method of the `ListConverter` class. We then use `sort` method of Java to sort the list.

use of convert method of List-Converter class for  
converting Python list to Java list

```
1 from py4j.java_gateway import JavaGateway
2 gateway = JavaGateway()
3 from py4j.java_collections import ListConverter
4 pythonList = [2,3,1]
5 javaList = ListConverter().convert(pythonList,\n6 gateway._gateway_client)
7 gateway.jvm.java.util.Collections.sort(javaList)
8 print(pythonList, javaList)
```

**Fig. 18.31** Sorting python list using Java method sort  
(SortPythonList1.py)

Note that `javaList` obtained from `pythonList` is sorted using Java method `sort` (line 7). Python responds with the following output on the execution of the program `SortPythonList1.py` (Fig. 18.31):

```
[2, 3, 1] [1, 2, 3]
```

Alternatively, we may request `py4j` to convert Python objects to compatible Java collections when invoking a Java method. This can be done by setting `auto_convert` to `True` while creating a `JavaGateway` as shown in the script `SortPythonList2` (Fig. 18.32). Since the Python list and the corresponding java list are two different objects (`py4j` makes a copy of Python list), `pythonList` does not get sorted when we invoke `Collections.sort`.

automatically converting Python objects to Java compatible collections

```
1 from py4j.java_gateway import JavaGateway, GatewayParameters
2 gateway = JavaGateway(gateway_parameters=GatewayParameters(\n3 auto_convert = True))
4 pythonList = [2,3,1]
5 gateway.jvm.java.util.Collections.sort(pythonList)
```

**Fig. 18.32** Sorting Python list using Java method  
(SortPythonList2.py)

#### 18.5.3 Invoking a Parameterized Java Method from Python

Suppose we wish to modify a list by multiplying all its elements by 2. The Java program

ListManipulation.java shown in Fig. 18.33 achieves this by defining method Update . The method takes a list as an input parameter and returns the modified list.

```
01 import py4j.GatewayServer;
02 import java.util.ArrayList;
03 public class ListManipulation
04 {
05 public ArrayList <Integer> Update(ArrayList <Integer> lst)
06 {
07
08 for(int i = 0; i < lst.size(); i++)
09 {
10 lst.set(i, lst.get(i)*2);
11 }
12 return lst;
13
14 }
15 public static void main(String[] args)
16 {
17 GatewayServer gatewayServer =
18 new GatewayServer(new ListManipulation(), 25539);
19 gatewayServer.start();
20 System.out.println("Gateway Server Started");
21 }
22 }
```

**Fig. 18.33** Java Program which modifies elements of the list received as parameter (`ListManipulation.java`)

To make methods of the class `ListManipulation` accessible in a Python program, it must be compiled and

executed before execution of Python program. Examine the Python script in Fig. 18.34 that uses this Java program. In lines 2 and 3, we have created an instance of class JavaGateway available in module py4j.java\_gateway. Entry point object is used for accessing Update method by passing pythonList as an input argument. Note that the input argument is not converted to Java compatible type explicitly. Instead, we have set gateway parameter auto\_convert which automatically achieves this.

```
1 from py4j.java_gateway import JavaGateway, GatewayParameters
2 gateway = \
3 JavaGateway(gateway_parameters=GatewayParameters(port=25539,
4 auto_convert = True))
5 ob = gateway.entry_point
6 print('Resultant list is', ob.Update(pythonList))
```

**Fig. 18.34** Python applications that uses the method Update from Java program (ListManipulation.py)

On executing the script ListManipulation (Fig. 18.34), Python responds with the following output:

```
>>>

Enter the list: [1,2,3]

Resultant list is [2, 4, 6]
```

#### 18.5.4 Invoking Parameterized Python Method from Java

Once again, we wish to modify the given list by multiplying each of its elements by 2. However, this time, we wish to develop a Python program that achieves this. As shown in Fig. 18.35, class ListManipulation has a method Update, which

takes a list as input parameter and returns the modified list.

```
1 from py4j.java_gateway import JavaGateway, GatewayParameters
2 gateway =\n3 JavaGateway(gateway_parameters=GatewayParameters(port=25539,
4 auto_convert = True))
5 pythonList = eval(input('Enter the list: '))
6 ob = gateway.entry_point
7 print('Resultant list is', ob.Update(pythonList))
```

```
17 javaClasslst = gateway.jvm.py4j.examples.List()
18
19 # Pass Python object to the Java program
20 javaClasslst.ListOperation(lst)
```

**Fig. 18.35** Python program which updates elements of the list received as parameter (ListManipulation1.py)

In order that a Java program may be able to invoke Python methods, we need to set `start_callback_server` attribute to `True` (line 14). Also, we need to pass the object `lst` of the class `ListManipulation` (line 16) to Java program so as to allow the Java method to call the Python methods. In line 17, we create an object `javaClasslst` of a Java class and pass Python class object `lst` to its method `ListOperation` (line 20).

in Python program, set `start_callback_server` attribute of Java-Gateway to `True` for enabling Java programs to invoke Python methods



```
01 package py4j.examples;
02 import java.util.ArrayList;
03 import py4j.GatewayServer;
04 public class List {
05 public void ListOperation(ListManipulationInterface op) {
06
07 ArrayList<Integer> numbers = new ArrayList<Integer>();
08 numbers.add(1);
09 numbers.add(2);
10 numbers.add(3);
11 ArrayList<Integer> result = op.Update(numbers);
12 System.out.println("Updated List: ");
13 for(int i = 0; i < result.size(); i++) {
14 System.out.print(result.get(i));
15 System.out.print(" ");
16 }
17 }
18 public static void main(String[] args) {
19 GatewayServer server = new GatewayServer(new List());
20 server.start();
21 System.out.println("Gateway Server Started");
22 }
23 }
```

**Fig. 18.36** Java applications that uses a method Update of Python program (List.java)

As shown in the program `List` (Fig. 18.36), object `lst` passed via Python program is received as object `op` in Java method `ListOperation` of class `List` (line 5). Note that, for every Python method, an interface method should be defined in Java. In Fig. 18.37, we have defined an interface `ListManipulationInterface` which declares the method `Update`.

On running the Java program followed by Python script, Java responds with the following output:

Updated List:

2 4 6

```
1 package py4j.examples;
2 import java.util.ArrayList;
3 public interface ListManipulationInterface {
4 public ArrayList<Integer> Update(ArrayList<Integer> numbers);
5 }
```

**Fig. 18.37** Interface `ListManipulationInterface`  
(`ListManipulationInterface.java`)

#### 18.6 PYTHON CHAT APPLICATION USING KIVY AND SOCKET PROGRAMMING

In this section, we will build a chat application that allows two parties to communicate with each other via *type and send* chat interface. We have used `kivy` library to create a chat interface, one for each communicating party. The interface comprises a textbox named `ChatBox` where chat logs are displayed, an entry text box named `EntryBox` to type a message to be sent, and a button named `SendButton` to send the message. Note that each widget can be assigned a name identifier using `id` property and referenced in the program using dictionary `ids` maintained by the root widget that comprises all widgets having an `id` attribute. We present

the widget definitions in the file `chatappinterface.kv` file (Fig. 18.38).

widgets used in chat interface: `ChatBox`, `EntryBox`, and `SendButton`

```
01 <Chat>:
02 orientation: 'vertical'
03 TextInput:
04 id: ChatBox
05 BoxLayout:
06 height: 90
07 orientation: 'horizontal'
08 padding: 0
09 size_hint: (1, None)
10
11 TextInput:
12 id: EntryBox
13 Button:
14 id: SendButton
15 text: 'Send'
16 size_hint: (0.3, 1)
17 on_press:root.clickAction()
```

**Fig. 18.38** Widget definitions (`chatappinterface.kv`)

In the scripts `host` (Fig. 18.39) and `client` (Fig. 18.40), we have defined the respective chat application.

The host program will initiate the communication by listening to request, and the client will connect to the host waiting for a connection and communicate with it. A message typed by either party will be displayed in its own ChatBox as well as the ChatBox of another communicating party. Thus, we need to perform the two tasks for each application: the first task takes the user input through EntryBox and displays it in his/her own ChatBox, and the second task sends the input to the connected party. Further, the chat application running at each end should be ready to receive a message sent by another party for displaying it in the ChatBox. Former functionality is performed by the clickAction method associated with the widget SendButton. The latter functionality is performed by getHostConnected and getClientConnected methods of the host and the client respectively. For this purpose, we create a separate thread by using the method start\_new\_thread of the module \_thread which takes the method to be executed as the first parameter and a list of arguments to be sent to the method as the second parameter.

the chat applications at the two ends send and receive messages at their own pace

```
01 import _thread
02 from kivy.app import App
03 from kivy.uix.button import Button
04 from kivy.uix.textinput import TextInput
05 from kivy.uix.boxlayout import BoxLayout
06 from kivy.core.window import Window
07 from socket import *
08 conn = ''
09 host = '192.168.0.8'
```

```
10 port = 25611
11 s = socket()
12 s.bind((host, port))
13 chatOb=None
14
15 class Chat(BoxLayout):
16
17 def __init__(self):
18 """
19 Objective: To initialize application with widget tree
20 Input Parameter:
21 self (implicit parameter) - object of type Chat
22 Return Value: None
23 """
24 super(Chat,self).__init__()
25
```

```
26 def clickAction(self):
27 """
28 Objective: To display the user input from EntryBox in ChatBox
29 Input Parameter:
30 self (implicit parameter) - object of type Chat
31 Return Value: None
32 """
33 global conn
34 textMsg = self.ids['EntryBox'].text
35 if textMsg != '':
36 self.ids['ChatBox'].text += '\nYou: '+textMsg
37 self.ids['EntryBox'].text = ''
38 conn.sendall(textMsg.encode())
39
```

```
40 class ChatAppInterface(App):
41 # Main class instantiated for running application
42 def build(self):
43 """
44 Objective: To initialize application with widget tree
45 Input Parameter:
46 self (implicit parameter) - object of type ChatAppInterface
47 Return Value: instance of Chat class (root widget)
48 """
49 global chatOb
50 chatOb = Chat()
51 Window.size = (400, 500)
52 return chatOb
53
54 def getHostConnected():
55 """
56 Objective: To connect host for chat
57 Input Parameter: None
58 Return Value: None
59 """
60 global conn, chatOb
61 import time
62 time.sleep(1)
63 s.listen(1)
64 chatOb.ids['ChatBox'].text = 'Waiting for Connection'
65 conn, addr = s.accept()
66 chatOb.ids['ChatBox'].text = 'Connected with: ' + str(addr)
67 while 1:
68 try:
69 data = conn.recv(1024)
```

```

70 chatOb.ids['ChatBox'].text += '\nOther: '+ data.decode()
71 except:
72 getHostConnected()
73 conn.close()
74
75 if __name__ == '__main__':
76 _thread.start_new_thread(getHostConnected, ())
77 ChatAppInterface().run()

```

**Fig. 18.39** Host application (`host.py`)

```

01 import _thread
02 from kivy.app import App
03 from kivy.uix.button import Button
04 from kivy.uix.textinput import TextInput
05 from kivy.uix.boxlayout import BoxLayout
06 from kivy.core.window import Window
07 from socket import *
08 conn = ''
09 host = '192.168.0.8'
10 port = 25611
11 s = socket()
12 chatOb=None
13
14 class Chat(BoxLayout):
15
16 def __init__(self):
17 """
18 Objective: To initialize application with widget tree
19 Input Parameter:
20 self (implicit parameter) - object of type Chat

```

```
21 Return Value: None
22 """
23 super(Chat,self).__init__()
24
25 def clickAction(self):
26 """
27 Objective: To display the user input from EntryBox in ChatBox
28 Input Parameter:
29 self (implicit parameter) - object of type Chat
30 Return Value: None
31 """
32
33 global s
34 textMsg = self.ids['EntryBox'].text
```

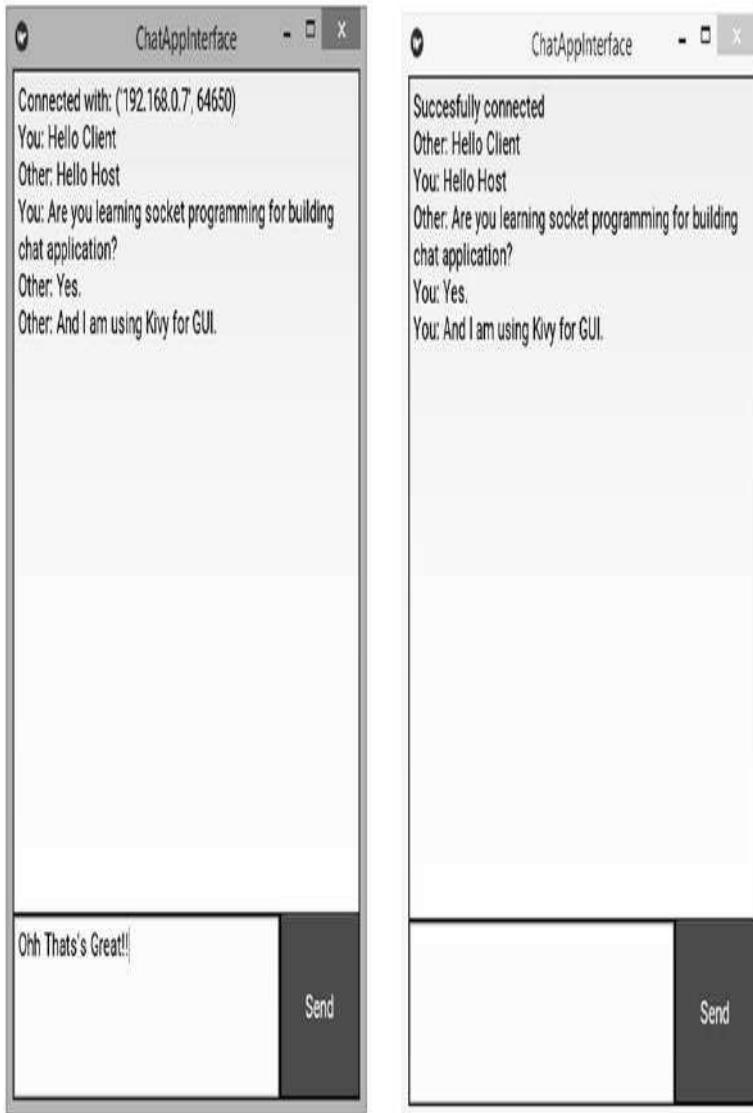
```
34 if textMsg != '':
35 self.ids['ChatBox'].text += '\nYou: '+textMsg
36 self.ids['EntryBox'].text = ''
37 s.sendall(textMsg.encode())
38
39 class ChatAppInterface(App):
40 # Main class instantiated for running application
41 def build(self):
42 """
43 Objective: To initialize application with widget tree
44 Input Parameter:
45 self (implicit parameter) - object of type ChatAppInterface
46 Return Value: instance of Chat class (root widget)
47 """
48
49 global chatOb
50 chatOb = Chat()
51 Window.size = (400, 500)
52 return chatOb
53
54 def getClientConnected():
55 """
56 Objective: To connect client for chat
57 Input Parameter: None
58 Return Value: None
```

```
58 """
59
60 global chatOb
61 import time
62 time.sleep(1)
63 try:
64 s.connect((host, port))
65 chatOb.ids['ChatBox'].text = 'Successfully connected'
66 except:
67 chatOb.ids['ChatBox'].text = 'Unable to connect'
68 return
69 while 1:
70 try:
71 data = s.recv(1024)
72 chatOb.ids['ChatBox'].text += '\nOther: ' + data.decode()
73 except:
74 chatOb.ids['ChatBox'].text = 'Peer has disconnected'
75 break
76 s.close()
```

```
77 if __name__ == '__main__':
78 _thread.start_new_thread(getClientConnected, ())
79 ChatAppInterface().run()
```

**Fig. 18.40** Client application (`client.py`)

In Fig. 18.41, we show the output of a sample chat session when the host application is run, followed by the client application.



**Fig. 18.41** Chat application

#### SUMMARY

1. `tweepy` package of Python is used for handling Twitter data .
2. Accessing Twitter data requires open authentication of an application. The authentication handler object of class `OAuthHandler` is passed as a parameter to class `API` of the `tweepy` module (a wrapper for API as provided by Twitter). An object of this class (`api`) can be used for retrieving user information such as his/her followers, friends, and tweets.
3. Python provides two socket modules - `socket` and `socketserver`. While the former provides low-level networking interface, the latter provides support in the form of classes for developing the network server application.

4. The `sqlite3` module provides the functionality required for database management. The class `connect` is used for establishing a connection with the database. The method `close` is used for closing the connection. The method `execute` of the cursor object of class `Cursor` is used for executing **SQL** commands.
5. Python provides an open source library Kivy for building cross-platform applications. It is a graphical user interface toolkit which supports desktop environments such as Windows®, Linux, and Mac OS®. It also supports multi-touch devices that support android/IOS.
6. Packages like `Py4j`, `Pyjnius`, `Jpype`, `javabridge`, and `JCC` are used to Java programs from Python.

#### EXERCISES

1. Write a program that takes a user handler as an input and extracts all tweets posted by him.
2. Write a program that extracts all tweets tweeted in the past five days regarding India's economy. Use the appropriate list of search words.
3. Write a socket programming application for transferring files across the networks.
4. Write a program that provides functionality for managing information about the college in the database COLLEGE. The database should contain relations for storing information about entities such as students, teachers, courses, and departments.
5. Develop a kivy application for Tic-Tac-Toe game with a grid of size  $4 \times 4$ .
6. Develop a kivy paint application that provides a paint palette containing basic 12 colours along with the paint area and a brush for painting.
7. Develop a kivy application for the pong game. The pong game is a table tennis game where the player who scores 11 points first by hitting the ball wins the game.

# Index

## A

abc module, 303  
ABCMeta, 303  
Abstract Base Classes (ABCs), 303  
Abstract class, 303  
Abstract data type, 259  
Abstract methods, 303  
Abstraction, 279  
Accessing Web Data, 539  
Accessor methods, 280  
Accessors, 280  
Activation record, 191  
Actual parameters, 33  
Adding methods dynamically, 282  
Aliasing, 151  
Alphabet, 137  
Animation: bouncing ball, 517  
Append, 359  
Application programming interfaces (API), 522  
Argument, 20, 33  
Arithmetic operators, 5  
ASCII values, 6  
Assert statement, 39  
Assignment statements, 9, 10  
Association, 10  
Attribute resolution order, 307  
Attributes, 541

## B

Base class, 285  
Binary search tree, 419  
    building, 431  
    traversal of, 422  
Binary search, 331  
Binary tree, 418  
    height of a, 430  
Binding, 10  
Bits, 8  
Bitwise operators, 8  
Blocking function, 484  
Boolean (bool), 250  
Boolean values, 6  
Break point, 87  
break statement, 72  
Bubble sort, 320  
Build, 551  
Built-in classes, 250  
    boolean (bool), 250  
    dictionary (dict), 250  
    floating point (float), 250

integer (int), 250  
list, 250  
string (str), 250  
tuple, 250  
Built-in function, 20, 27  
Built-in functions for classes, 309  
Built-in functions on strings, 127

## C

Called function, 30  
Callee function, 30  
Caller function, 30  
Calling the function, 20  
Cardinality, 541  
Chat applications, 571  
Child class, 285  
Class attributes, 250  
Class constructor, 253  
Class initializer, 253  
Class node, 385  
Classes, 250  
Command line arguments, 41–42  
Complete binary tree, 418  
Composition, 20, 284  
Conditional expression, 51  
Continue statement, 72  
Control structures, 46–80  
Control structures, 78  
Converting python collections to Java collections, 566  
Create table, 543  
Creating a linked list of cubes, 388

## D

Data, 279, 357  
Data definition language (DDL), 541, 542  
Data hiding, 279  
Data manipulation language (DML), 542  
Data structures, 357  
Database  
    concepts, 541  
    creating, 542  
    loading the, 545  
    populating the, 545  
Database management system (DBMS), 540  
Date class, 265  
Debugging, 85–95  
    commands, 88–89  
Deep copy, 210  
Default values, 37  
Degree, 541  
Deletion, 177  
Dependencies, 550  
Dequeue, 369, 412  
Derived class, 285  
Destructor, 257  
Developing mobile application for android, 550  
Dict, 250  
Dictionary, 175  
    inverted, 179  
    operations, 177

Docstring, 4

Domain, 543

## E

Echo server, 537  
else statement, 77  
Empty except clause, 232  
Empty language, 138  
Encapsulation, 279  
Enqueue, 369, 409  
Entity, 541  
Errors, 227  
    exceptions, 228  
    indentation error, 228  
    indentation, 228  
    IndexError, 230  
    NameError, 228  
    OSError, 230  
    syntax, 227  
    TypeError, 229  
    ValueError, 229  
    ZeroDivisionError, 229  
Escape sequences, 22  
eval function, 20  
Exceptions, 228  
Extracting comments, 144

## F

File, 221  
    handling, 221  
Finding common factors, 171  
Floating point (float), 250  
Floating point numbers, 2  
for loop, 58  
Formal parameters, 33  
Fractal, 471  
Front, 372  
Front end, 369  
Fruitful function, 36  
Function parameters, 33  
Function(s), 19–42  
    append, 154  
    axis, 489  
    built-in, 20, 27  
    called, 30  
    callee, 30  
    caller, 30  
    calling the, 20  
    composition, 20  
    comprehension, 157  
    copy, 170, 179  
    count, 127, 154, 175  
    decode, 132  
    definition and call, 27–38  
    dump, 226  
    encode, 132  
    endswith, 132  
    eval, 20  
    extend, 154  
    filter, 164

```
find, 128
from math module, 25
fruitful, 36
functions, 157
get, 178
help, 37
index, 154, 175
input, 20
insert, 154, 156
invoking the, 20
isalnum, 131
isalpha, 131
isdigit, 131
islower, 129
isspace, 131
istitle, 129
isupper, 129
join, 131
list, 154
load, 227
lstrip, 130
map, 164
max, 24
min, 24
nested, 56
overloading, 277
partition, 130
pop, 154
pow, 24
print, 21
readline, 224
recursive, 188
reduce, 164
remove, 154,
replace, 130
reverse, 156
rfind, 128
round, 22
rstrip, 130
seek, 225
show, 484
sort, 156
split, 130
startswith, 132
strip, 130
superset test
tuple, 174
type, 22
update, 178
wrapper, 438
writelines, 225
zip, 174
```

## G

Game of Xs and Os, 550  
Global frame, 29  
Global names, 114  
Graphical package, 445  
Graphics, 483–519  
    2D, 483  
    3D, 509

animation, 517–519  
arrow (2D), 508  
arrow (3D), 514  
box, 510  
circle, 502  
cone, 515  
cosine curve, 500  
curve, 516  
cylinder, 513  
ellipse, 504  
histogram, 496  
pie chart, 498  
polygon, 506  
rectangle, 505  
ring, 512  
sine curve, 500  
sphere, 510

## H

Handler, 235  
Hard-coding, 205  
hasattr, 309  
Hierarchical inheritance, 295  
Hilbert curve, 471  
Home button, 486

## I

Idle, 1  
If conditional statement, 46  
If-elif-else conditional statement, 55  
If-else conditional statement, 52  
Importing user-defined module, 38–39  
Indentation error, 228  
Indentation, 29  
IndexError, 230  
Infinite loops, 64  
Infix expression, 363  
Infix form, 363  
Inheritance, 285  
    hierarchical, 295  
    multilevel, 295  
    multiple, 299  
    single, 286  
inner function, 57  
Inorder traversal, 423  
input function, 20  
Insertion sort, 326  
Instance method, 280  
Integer (int), 250  
Intersection operation on list, 171  
Inverted dictionary, 179  
Invoking the function, 20  
isEmpty, 361  
isSubclass, 309  
Iteration, 57  
Iteration of the loop, 58

## J

Javabridge, 563  
Jupyter, 563

## K

Key, 316, 541  
Keyword arguments, 38  
Keywords, 13  
Kivy, 550  
    application, 562  
    installation, 550  
    python chat application using, 570

## L

Language, 137  
    empty, 138  
    null, 138  
    regular, 137  
Last in first out (LIFO) arrangement, 358  
Left binary subtree, 418  
Left-child relationship, 419  
LEG Rule, 114  
Linear search, 330  
Linked list, 384  
    insertion and deletion at the beginning of, 388  
    maintaining sorted, 400  
    operations, 392  
    stack implementation using, 405–409  
    string representation of a, 399  
    traversing a, 399  
list, 250  
List manipulation, 315–354  
List objects; copying, 159  
Lists, 150–152  
    as argument, 158  
    intersection operation on, 171  
    two-dimensional, 152  
    union operation on, 171  
Loading the database, 545  
Logical operators, 6  
Loop, 58  
    for, 58  
    infinite, 64  
    iteration of the, 58  
    nested, 69  
    while, 63

## M

Master data, 240  
Matplotlib, 445, 484  
max function, 24  
Membership, 127  
Memory map, 34  
Metadata, 542  
Method overriding, 289  
min function, 24  
Modifier, 280  
Modular approach, 19  
Module, 14  
Modulus, 2  
Multilevel inheritance, 295  
Multiple inheritance, 299  
Multiple plots, 492

## N

Name mangling, 279  
NameError, 228  
Names, 10  
Namespaces, 114  
Nested function, 56  
Nested if-elif-else conditional statement, 55  
Nested loops, 69  
Nesting, 55  
New-style classes, 306  
Node(s), 384  
    deleting, 396  
    internal, 417  
    keep track of, 397  
    root, 416  
    value of the, 416  
Null language, 138  
Null string, 137

## O

Object attributes, 255  
Object composition, 284  
Object-oriented programming (OOP), 272  
Objects, 2, 250  
Open authentication (OAuth), 522  
Operands, 2  
Operator overloading, 272, 273  
Operator, 2  
    arithmetic, 5  
    bitwise, 8  
    logical, 6  
    relational, 5  
OSError, 230  
Outer function, 57

## P

Parameters, 33  
    actual, 33  
    formal, 33  
    function, 33  
Parent class, 285  
Pass statement, 72  
Patches, 502  
Pattern matching, 137  
Pattern within a pattern, 445  
Pickling, 226  
Pivot element, 351  
Plotting a point, 484  
Polymorphism, 272  
Pop, 358, 361  
Populating the database, 545  
Postfix form, 363  
Postorder traversal, 430  
pow function, 24  
Preorder traversal, 429  
print function, 21  
Problem of Tower of Hanoi, 214–218  
Problems  
    eight queens, 448  
    Knight's Tour, 455

stable marriage, 464  
Program, 14  
Properties, 551  
Pseudocode, 32  
push, 358, 361  
Py4j, 563  
Pyjnius, 563  
Pyplot module, 484  
Python, 1  
    applications of, 521–576  
    arrays in, 565  
    code, 1  
    editor, 1  
    errors, 227  
    integrating java with, 563  
    shell, 2  
    strings, 4, 125  
    style, 10  
    tutor, 101  
    visual, 509  
PythonCopy, 225  
Python tutor interface, 101  
Pythonic style, 10  
Pythonic way, 10

## Q

Queue, 369  
Queue operations, 372

## R

raise keyword, 236  
Random number generation, 24–25  
Rear end, 369  
Recursion, 188–218  
Recursive function, 188  
Recursive solutions for problems on lists, 204  
    copying a list, 209  
    deep copy, 210  
    flatten a list, 204  
    permutation, 211  
Recursive solutions for problems on numeric data, 188–197  
    factorial, 188  
    fibonacci sequence, 194  
Recursive solutions for problems on strings, 197–204  
    length of a string, 197  
    palindrome, 201  
    reversing a string, 199  
Recusion, 481  
Regular languages, 137  
Relation, 541  
Relational operators, 5  
Retrieving all matching patterns, 144  
Reversing a string, 136  
Right binary subtree, 418  
Right-child relationship, 419  
Root node, 416  
Round function, 22

## S

Saving figure, 495

Scalar data types, 123, 150  
Scope rule, 290  
Scope, 100–120  
Script mode, 14  
Searching, 315, 329  
Selection sort, 316  
set functions, 167, 171  
    add, 167  
    clear, 167  
    difference, 168  
    intersection, 168  
    pop, 167  
    remove, 167  
    symmetric\_difference, 168  
    union, 168  
    update, 167  
Sets, 166  
Shallow copy, 210  
Shared session, 102  
Sharing data across applications, 532  
Sharing data using sockets, 532  
Short-circuit evaluation, 7  
Shorthand notation, 11  
Sierpinski Triangle, 471  
Single inheritance, 286  
Singleton tuple, 173  
Size, 361, 373  
Slice, 125  
Slicing, 125–126  
socket, 533  
Socket programming, 570  
Sort  
    bubble, 320  
    insertion, 326  
    merge, 349  
    quick, 351  
Sorting, 315  
Source code, 14  
Source file script, 14  
SQLite database, 540  
Stack frame, 191  
Stack, 357  
    deletion of an element from a, 358  
    insertion of an element in a, 358  
Static method, 280  
Stepwise refinement method, 19  
Stepwise refinement process, 32  
Straight line functions, 46  
Stream, 530  
streamListener, 530  
Strictly binary tree, 418  
String (str), 250  
String, 137  
    null, 137  
Strings, 123–147  
    reversing, 136  
Structured query language (SQL), 541  
Sub, 285  
Sub-programs, 19  
Subset, 170  
Sudoku puzzle, 477

Super class, 285  
Superset test, 170  
Symbols used in regular expressions, 139–140  
Syntax error, 227

## T

Table(s)  
    creating, 542  
    deleting, 549  
    inserting data into, 545  
    retrieving data from, 546  
    updating data in a, 548  
Temporary variable, 13  
Testing, 85  
Tic-Tac-Toe game, 558  
Top, 361  
Transaction data, 240  
Transmission Control Protocol/Internet Protocol (TCP/IP), 533  
Tree, 416  
    ancestor/descendant, 417  
    height, 417  
    internal node, 417  
    leaf, 417  
    level, 417  
    path and path length, 417  
    siblings, 418  
    subtree, 418  
try...except clause, 230  
Tuple, 172, 250, 541  
Tweet update, 530  
Twitter, 521  
    collecting followers, friends, and tweets of a user, 526  
    collecting tweets having specific words, 530  
    collecting information from, 521  
    data analysis, 522  
    open authentication, 523  
Two-dimensional list, 152  
Type conversion, 23  
type function, 22  
TypeError, 229

## U

uix module, 551  
Underflow condition, 358  
Underflow, 370  
Unhandled exceptions, 230  
Union operation on list, 171  
Union operator, 138  
Unpickling, 226  
Update command, 548  
User-defined classes, 250

## V

Value of the node, 416  
ValueError, 229  
Values, 2  
Variable current, 402  
Variables, 9  
Visualize execution, 102  
Void function, 36

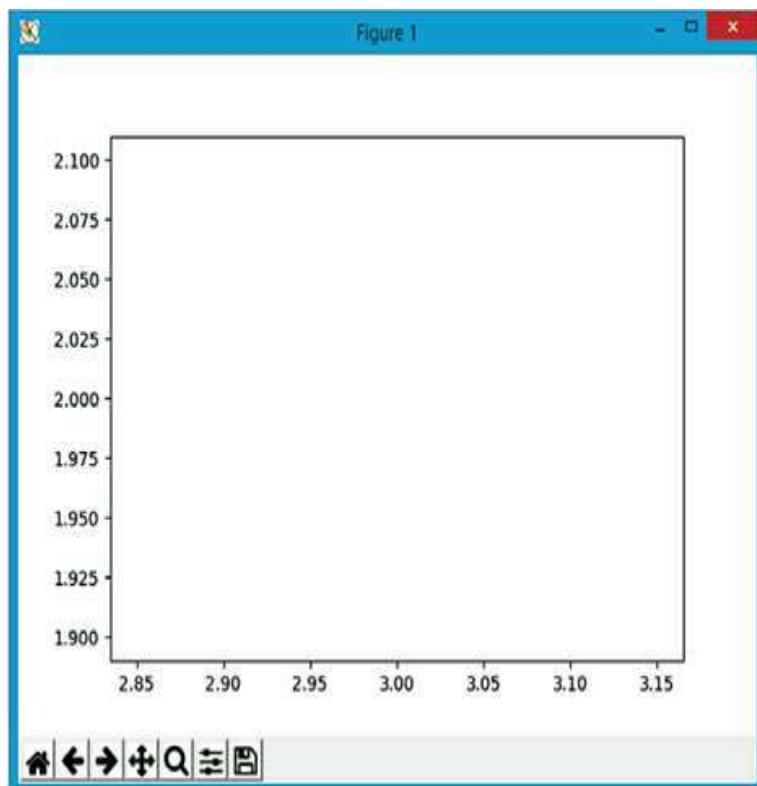
## **W**

Web Data; Accessing, [539](#)  
Wedges, [498](#)  
Where clause, [547](#)  
while loop, [63](#)  
Widgets, [551](#)  
Wrapper function, [438](#), [447](#)  
Writing structures to a file, [226](#)

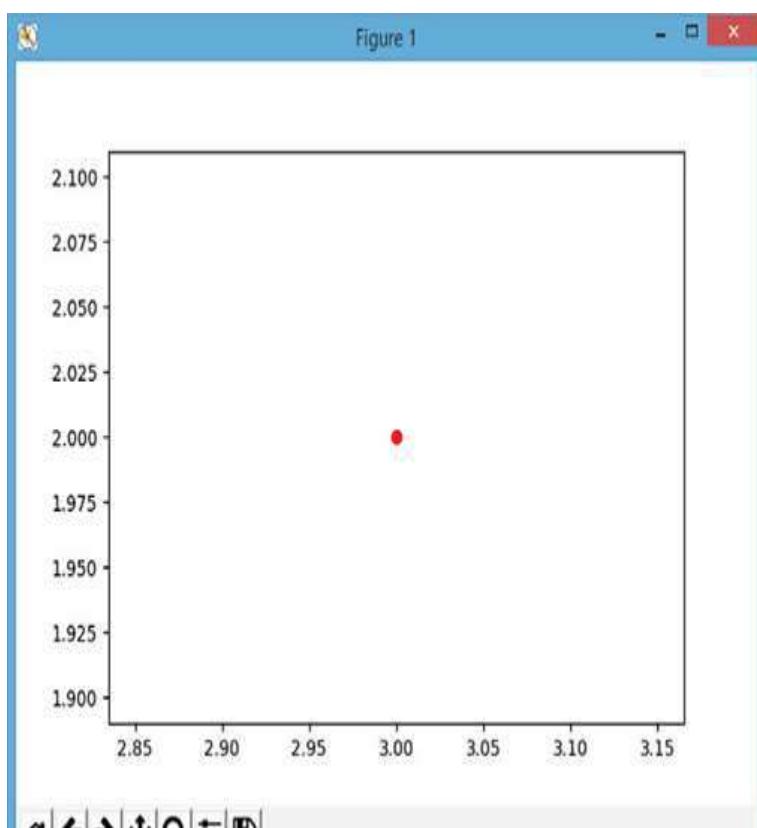
## **Z**

ZeroDivisionError, [229](#)

## Colour Illustrations

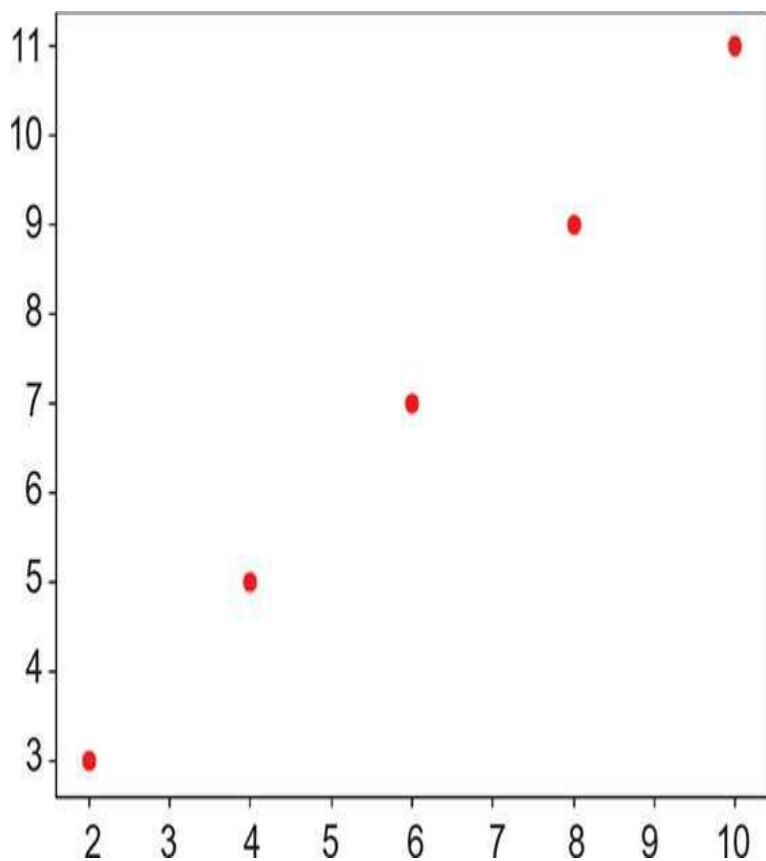


**Plate 1** Function plot(3, 2)to display point (3, 2)(see page 485)

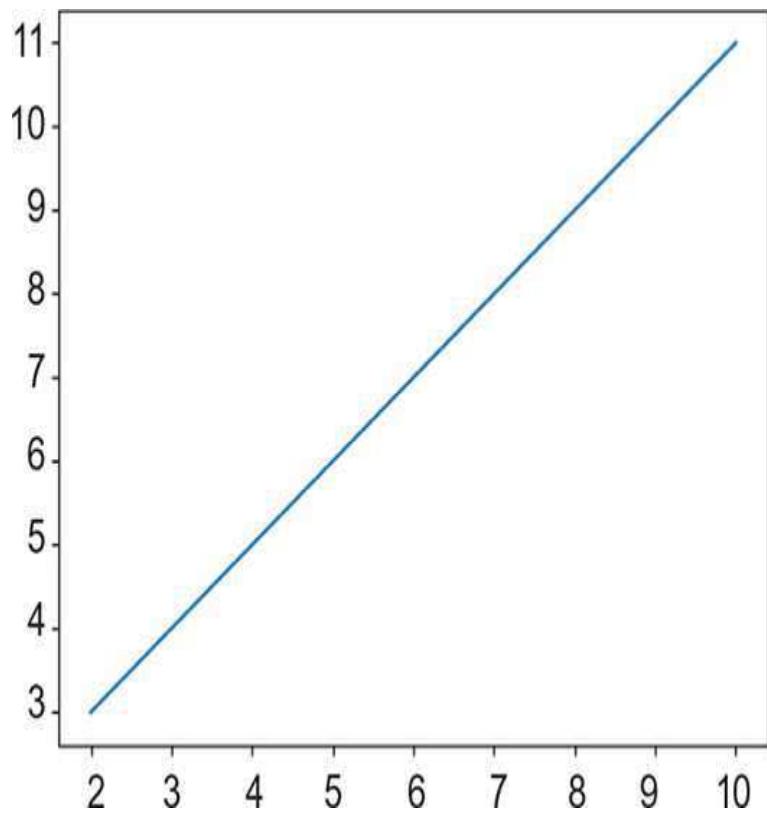




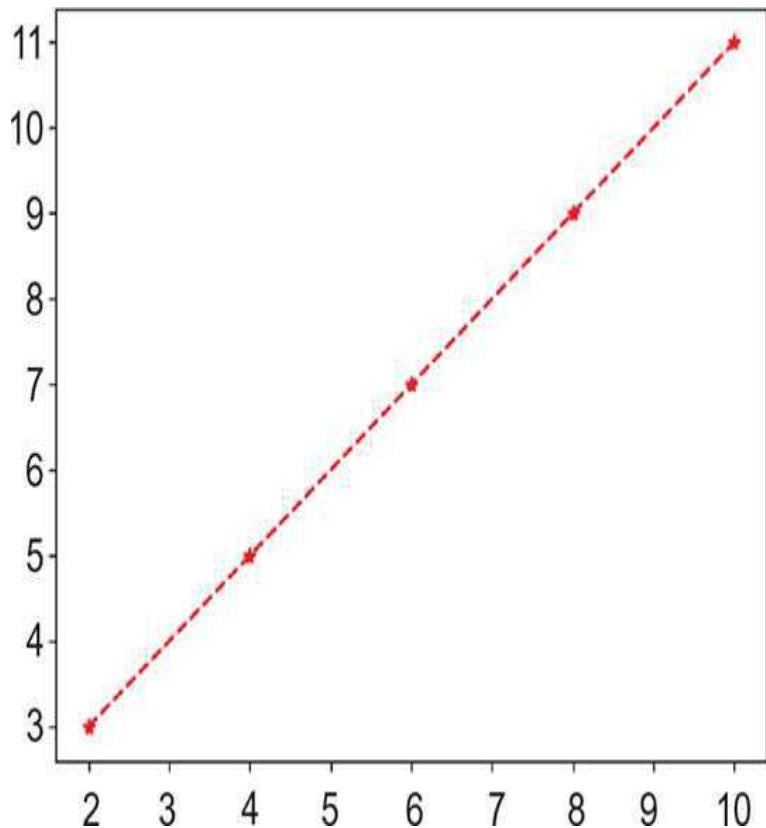
**Plate 2** Point (3, 2) (see page 486)



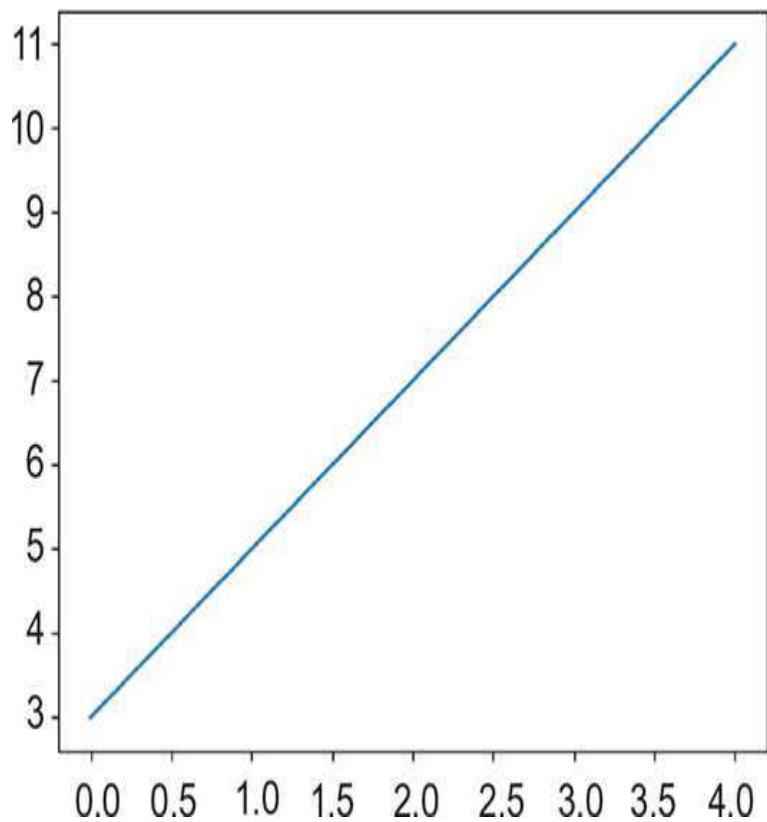
**Plate 3** Multiple points (see page 487)



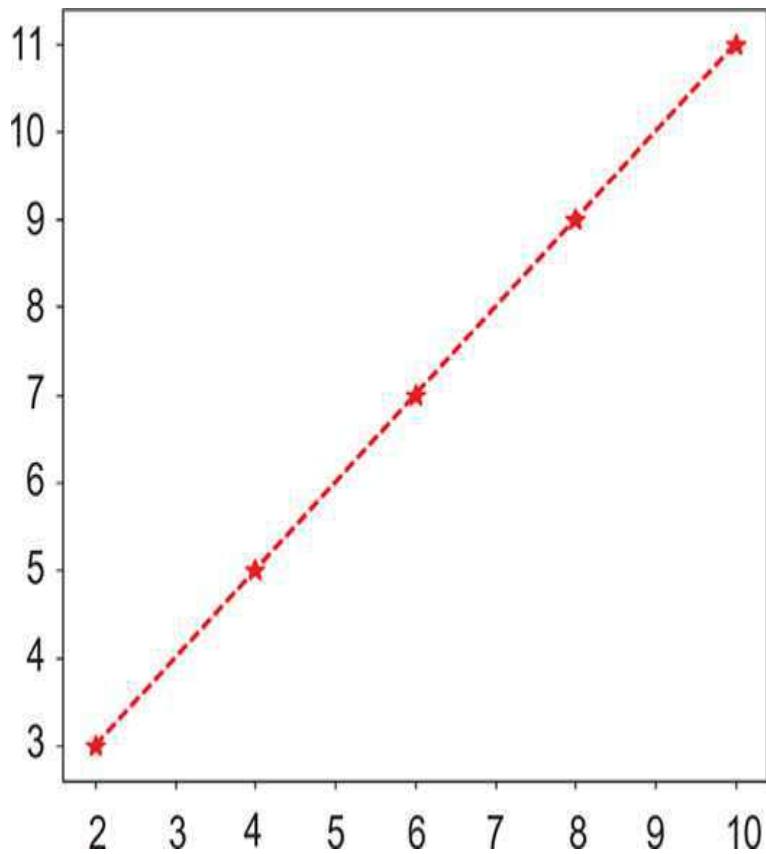
**Plate 4** Line joining points specified by vector  $x$  and  $y$  (see page 487)



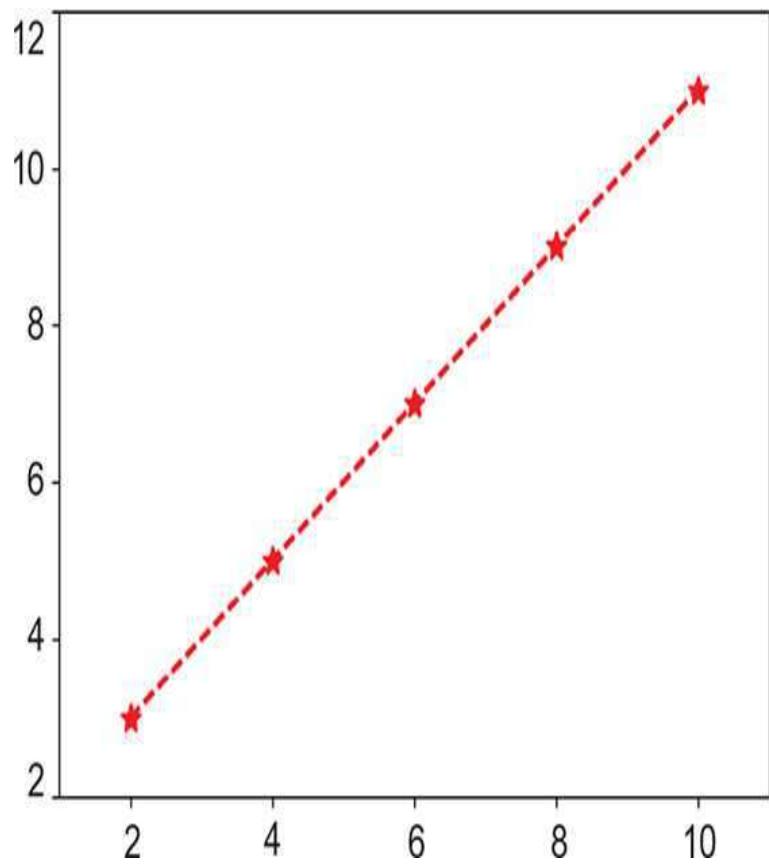
**Plate 5** Dashed line joining points (star marker) specified by vector  $x$  and  $y$  (see page 488)



**Plate 6** Solid line joining points defined by vectors [0, 1, 2, 3, 4] and y (see page 489)

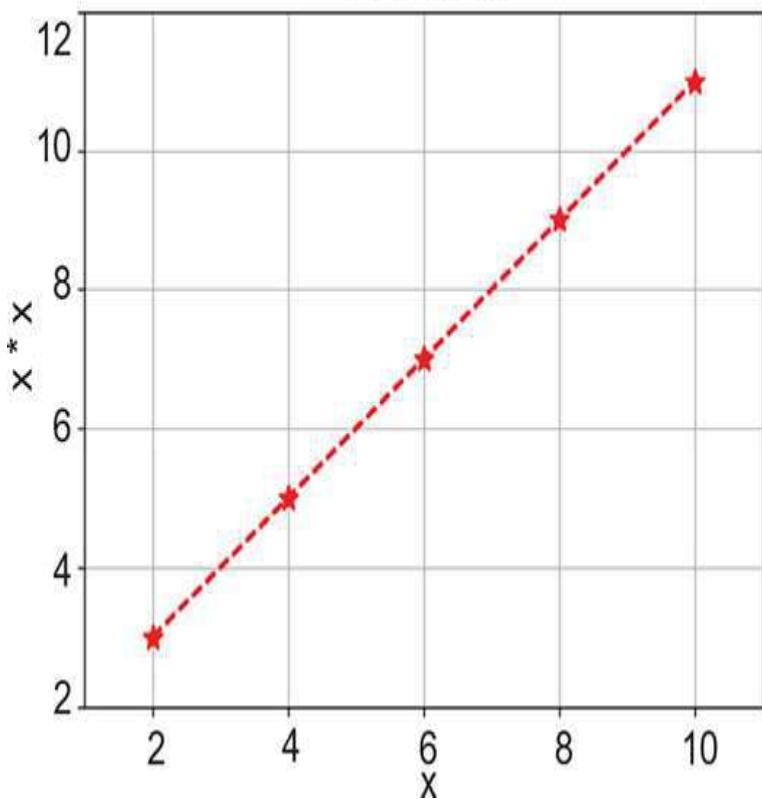


**Plate 7** Line joining points specified by vectors  $x$  and  $y$  (see page 489)



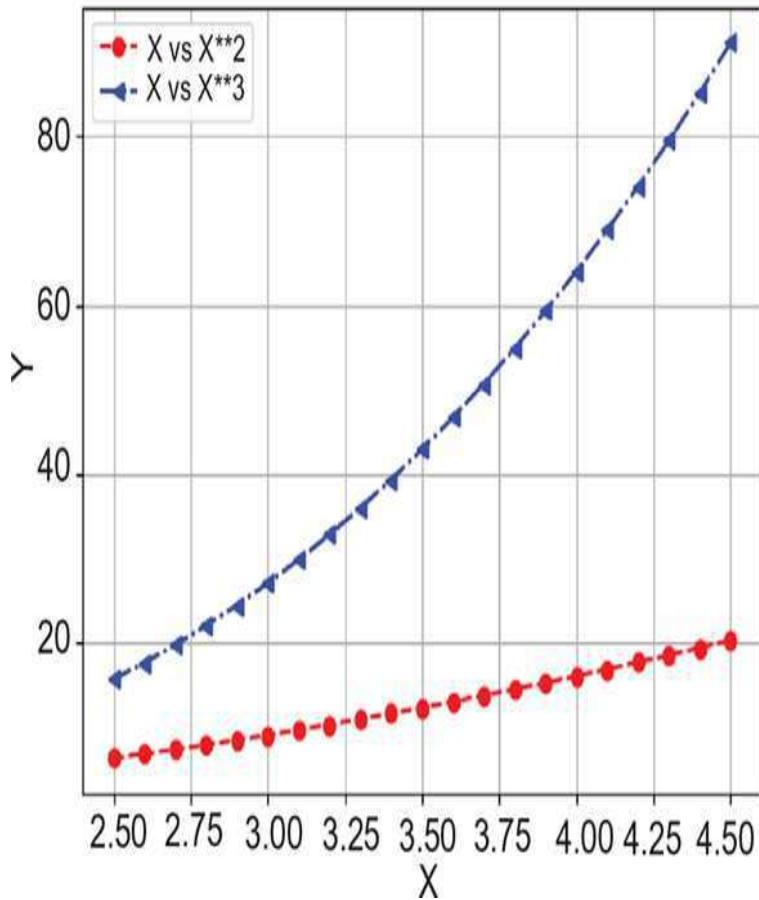
**Plate 8** Line joining points specified by vectors  $x$  and  $y$  (see page 490)

X vs X \* X

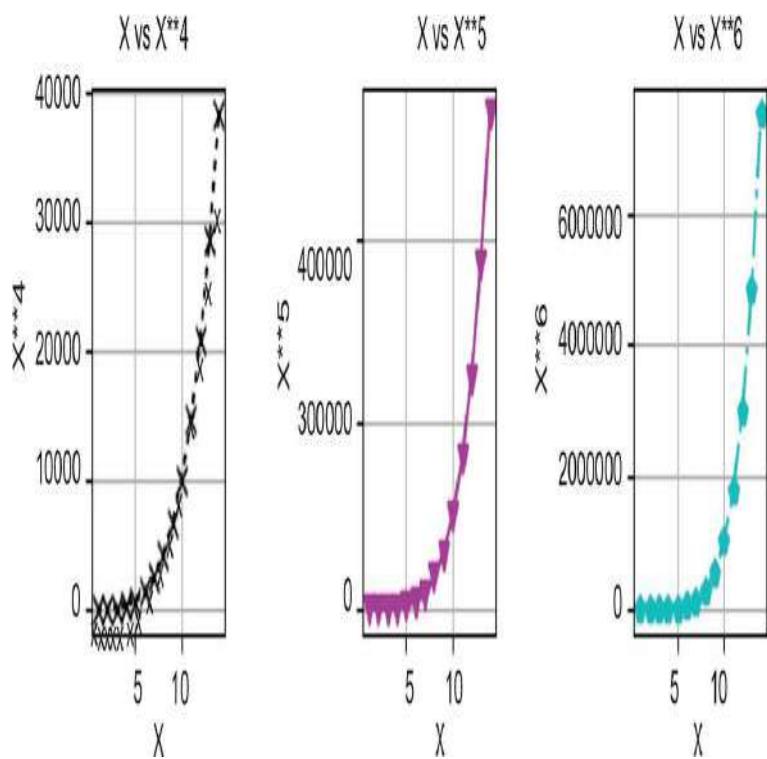
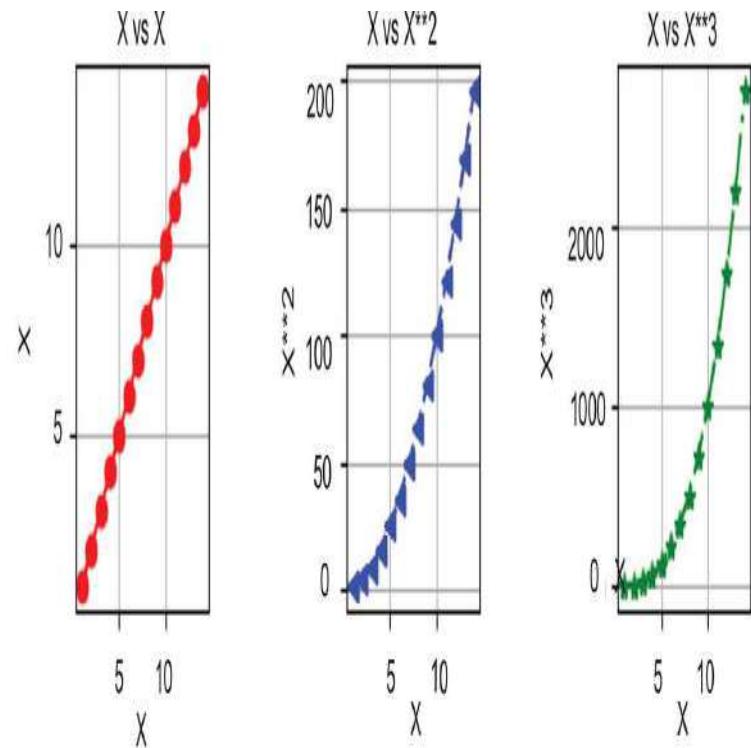


**Plate 9** Line joining points specified by vectors x and y (see page 491)

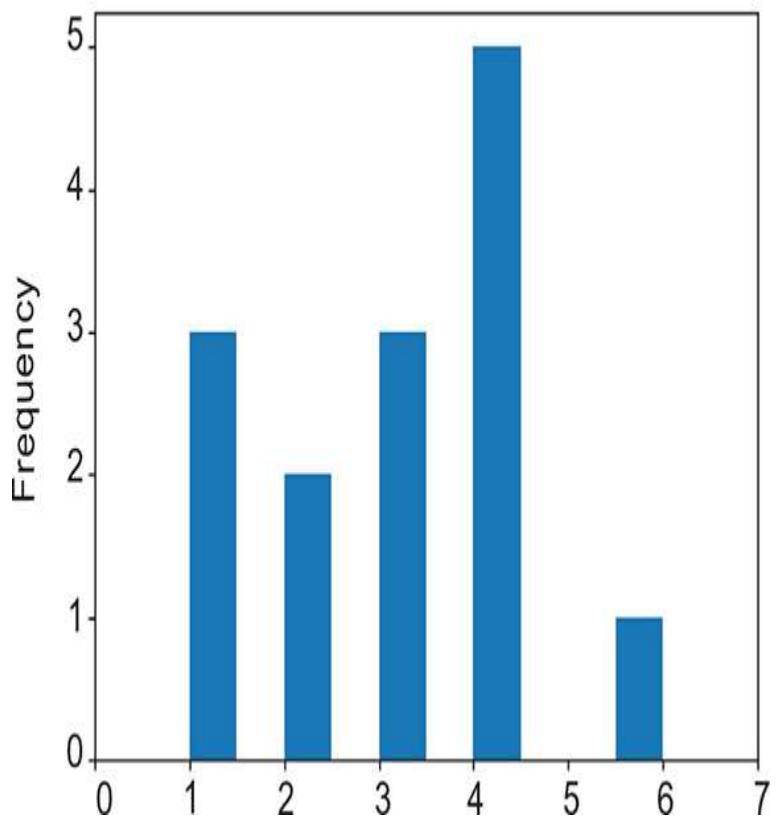
### X vs X\*\*2 and X vs X\*\*3



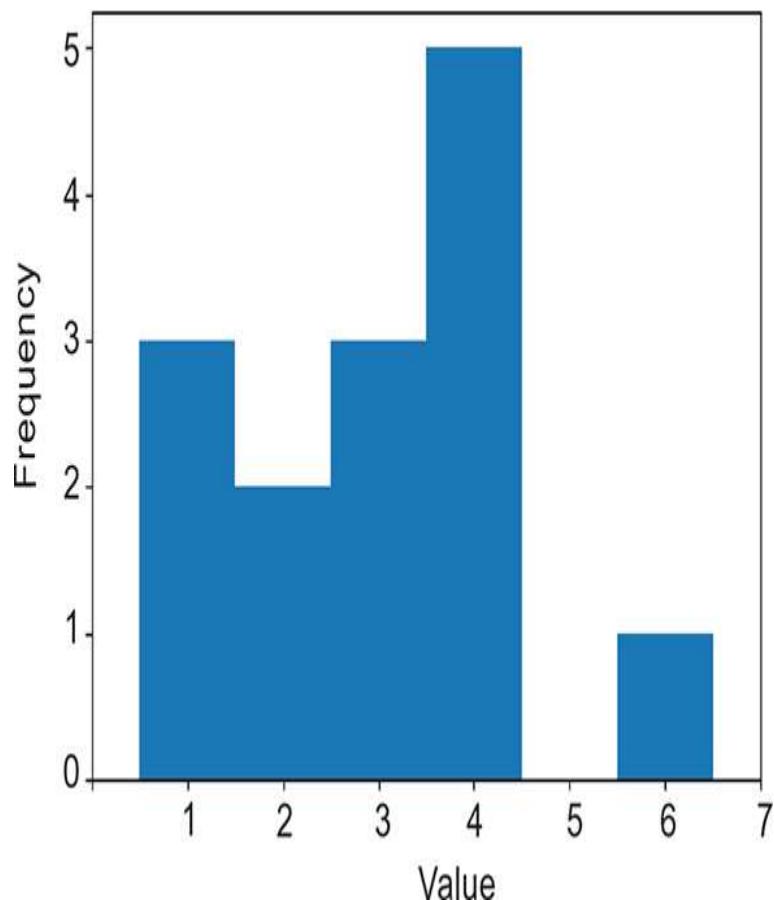
**Plate 10** Functions  $f(x) = x^2$  and  $f(x) = x^3$  plotted in the interval  $[2.5, 4.5]$  in steps of 0.1 (see page 493)



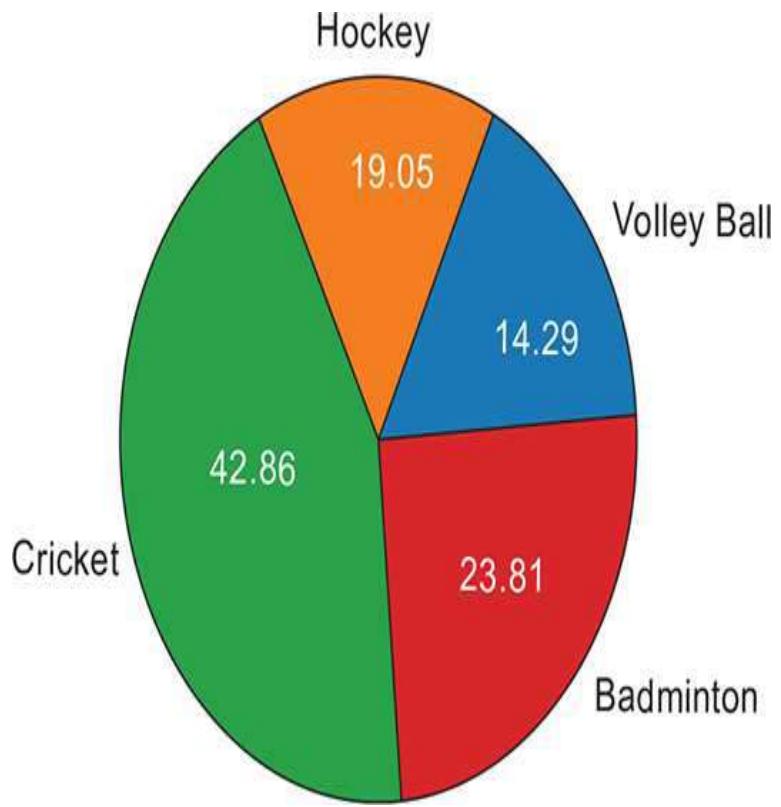
**Plate 11** Six functions plotted on different subgraphs in same figure (see page 496)



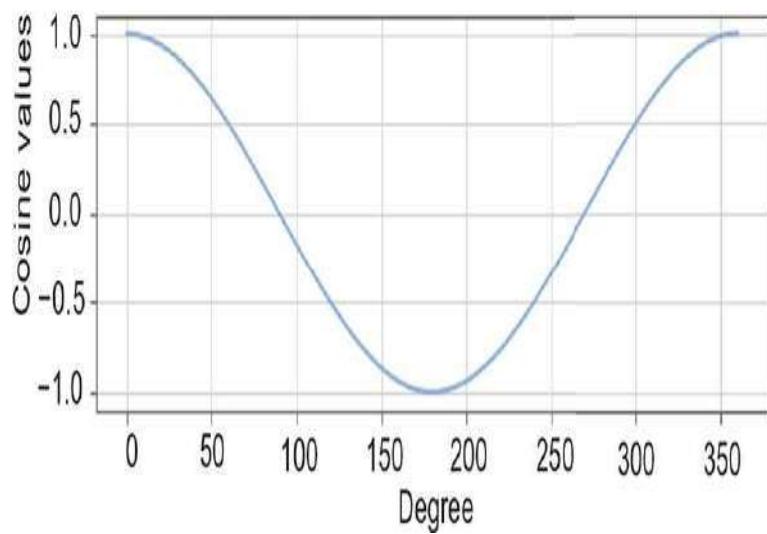
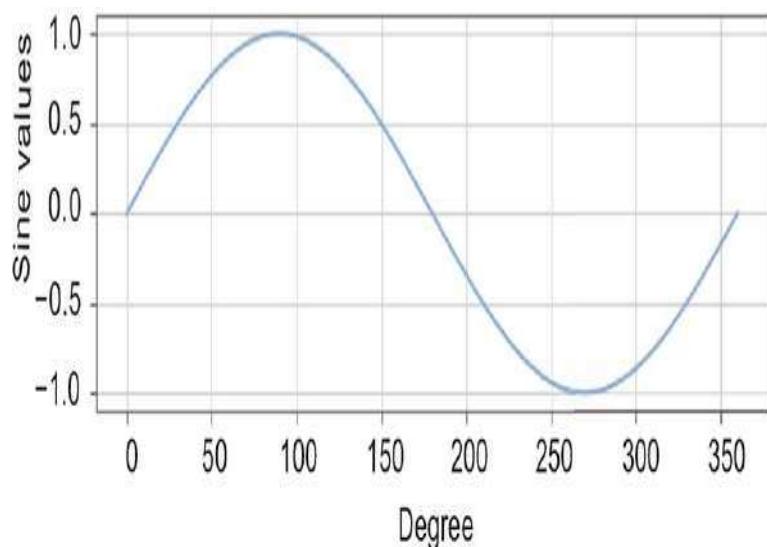
**Plate 12** Histogram (see page 497)



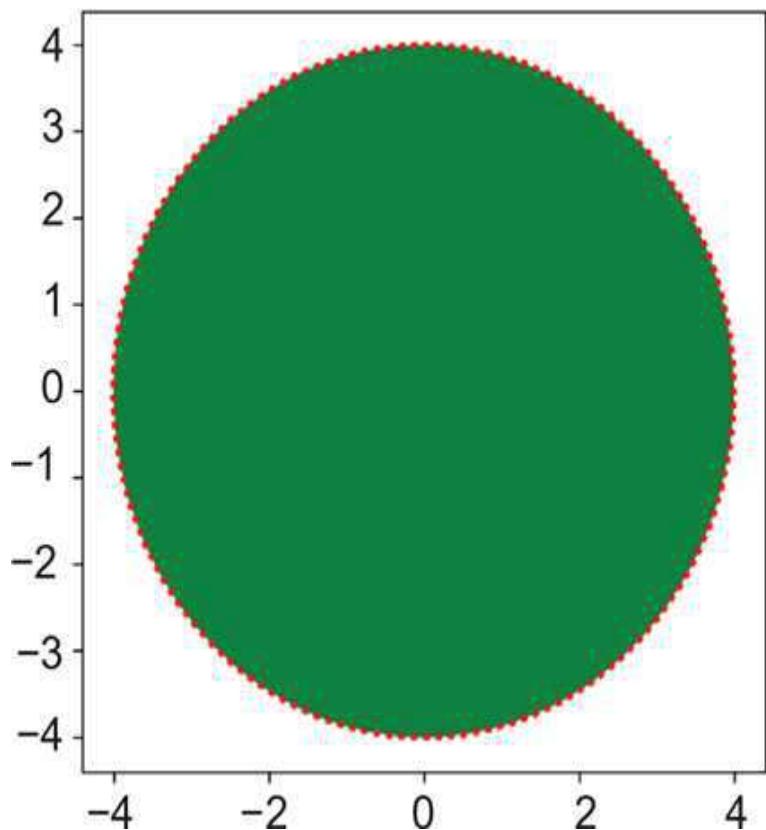
**Plate 13** Histogram (see page 498)



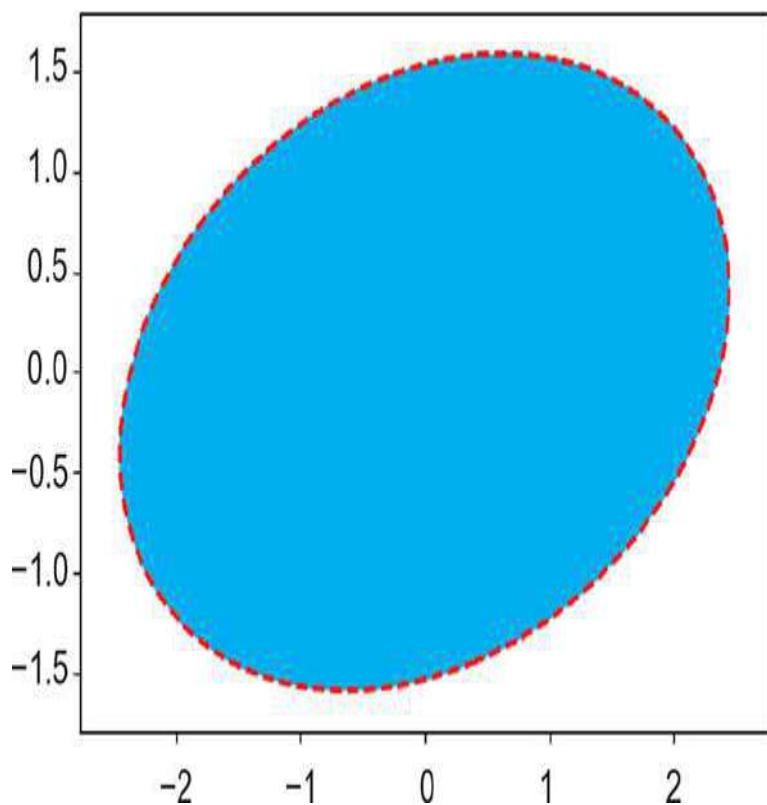
**Plate 14** Pie chart (see page 499)



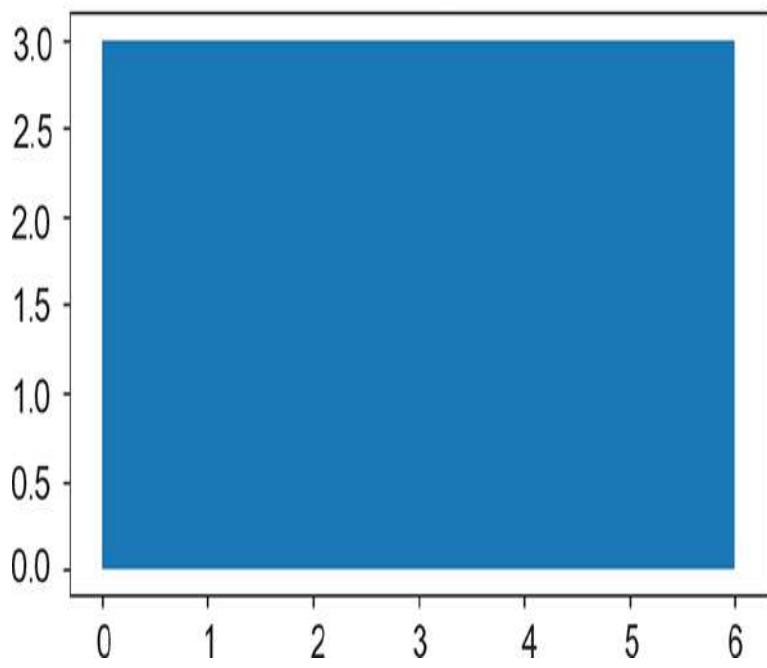
**Plate 15** Sine and cosine curve (see page 501)



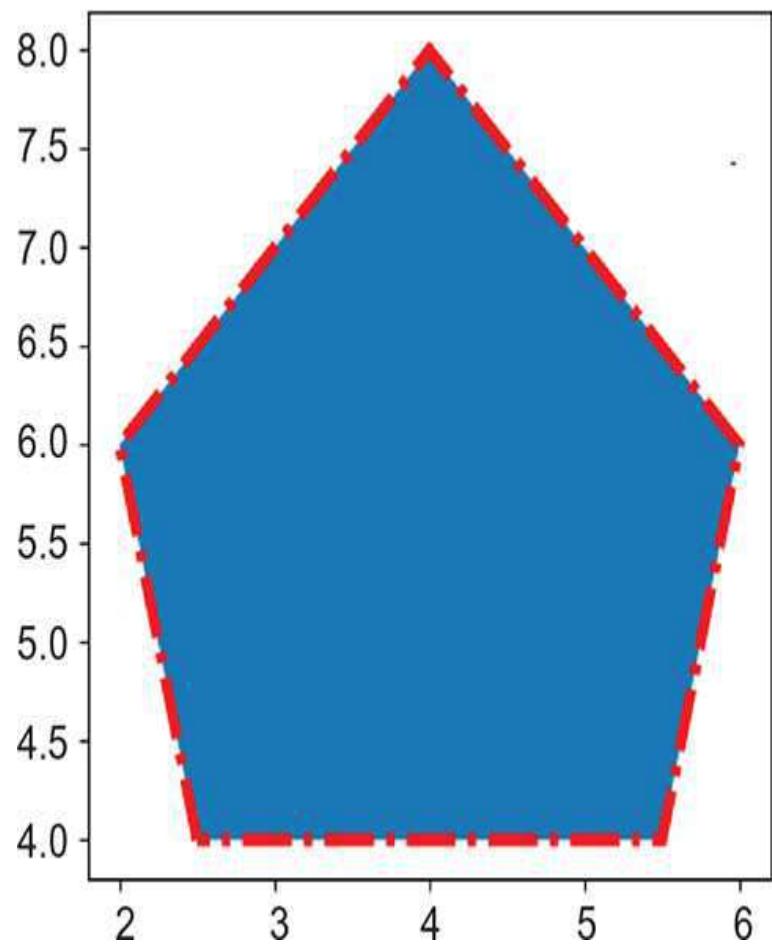
**Plate 16** Circle (see page 503)



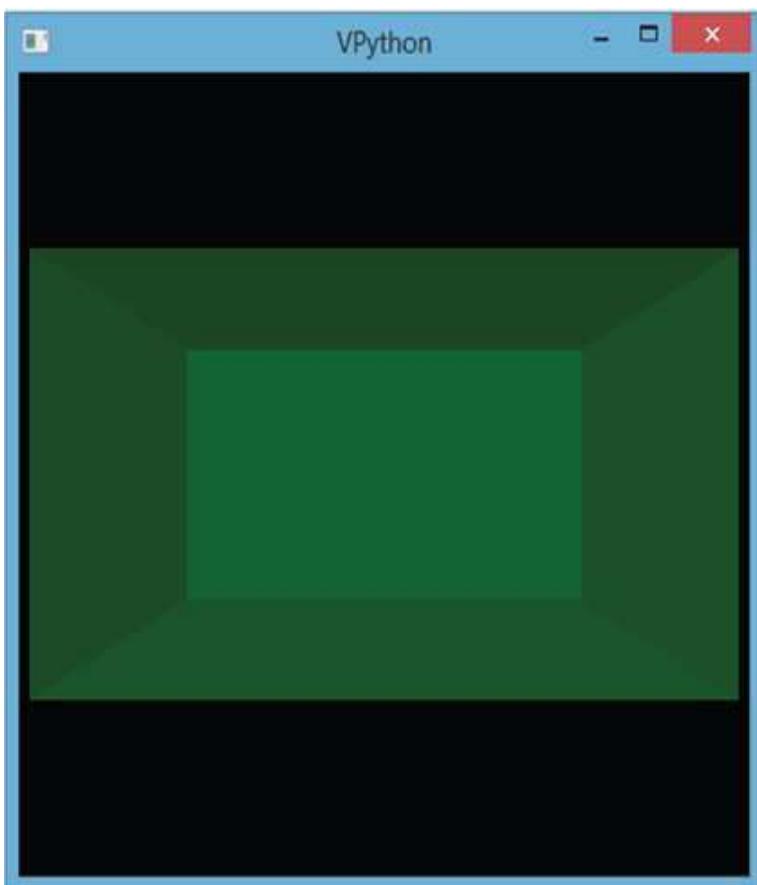
**Plate 17** Ellipse (see page 505)



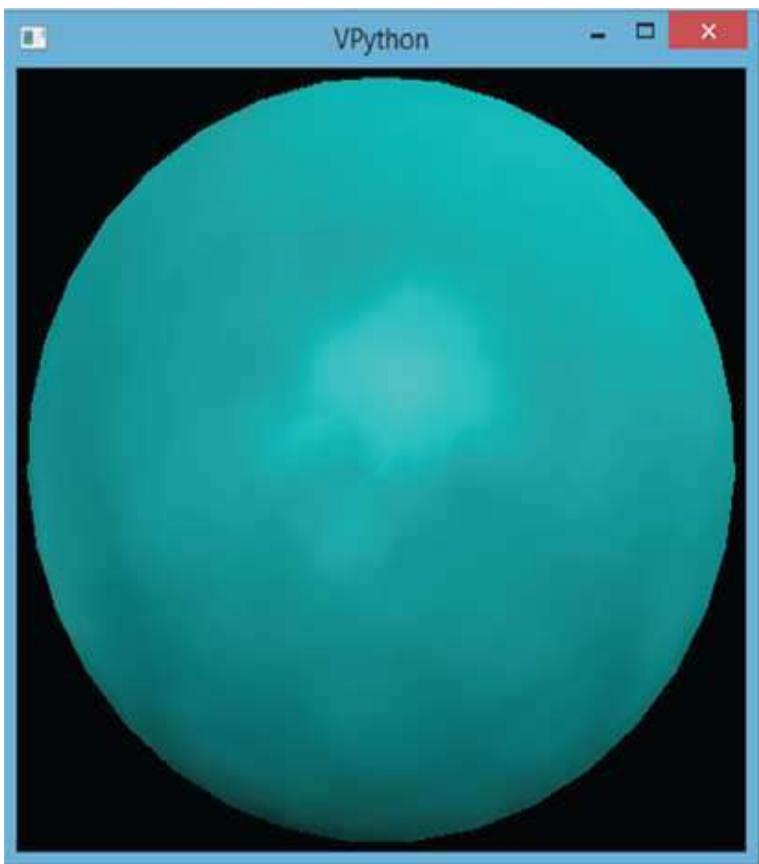
**Plate 18** Rectangle (see page 506)



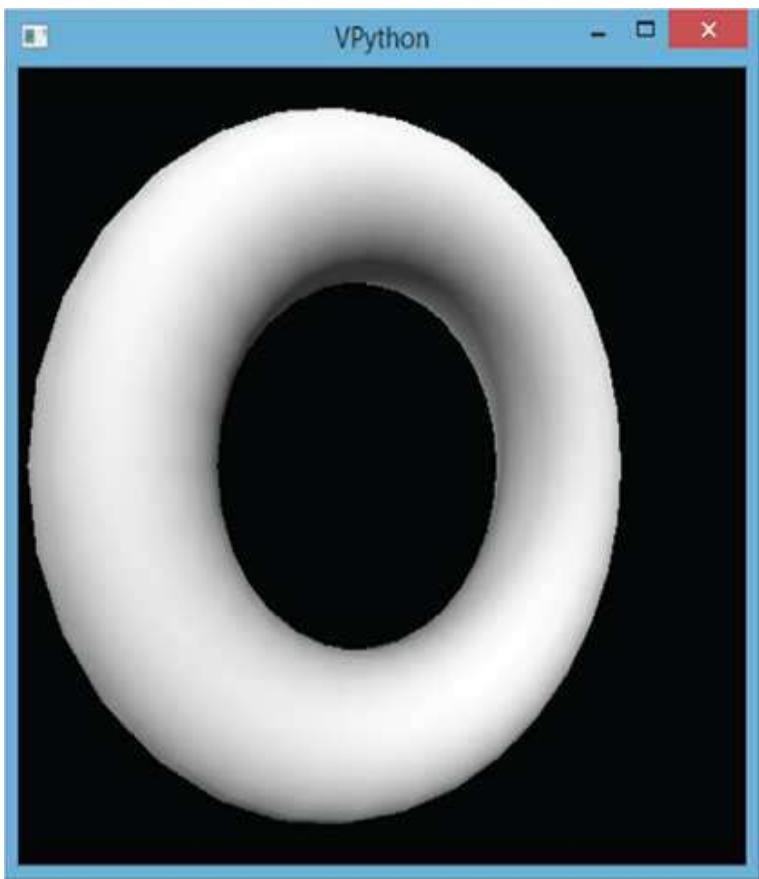
**Plate 19** Polygon (see page 507)



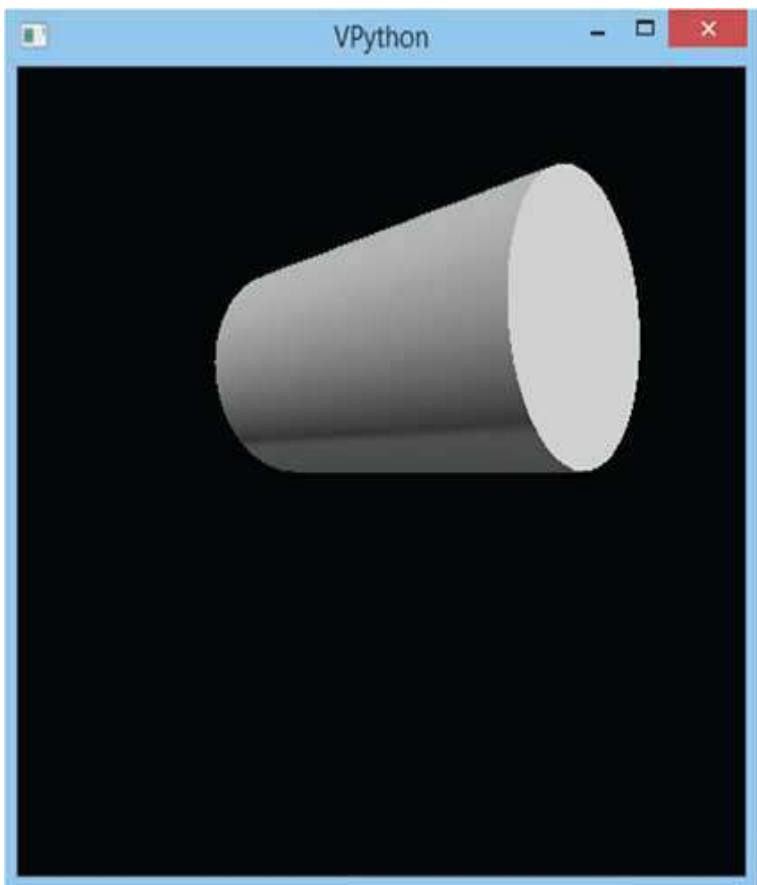
**Plate 20** Box (see page 511)



**Plate 21** Sphere (see page 511)



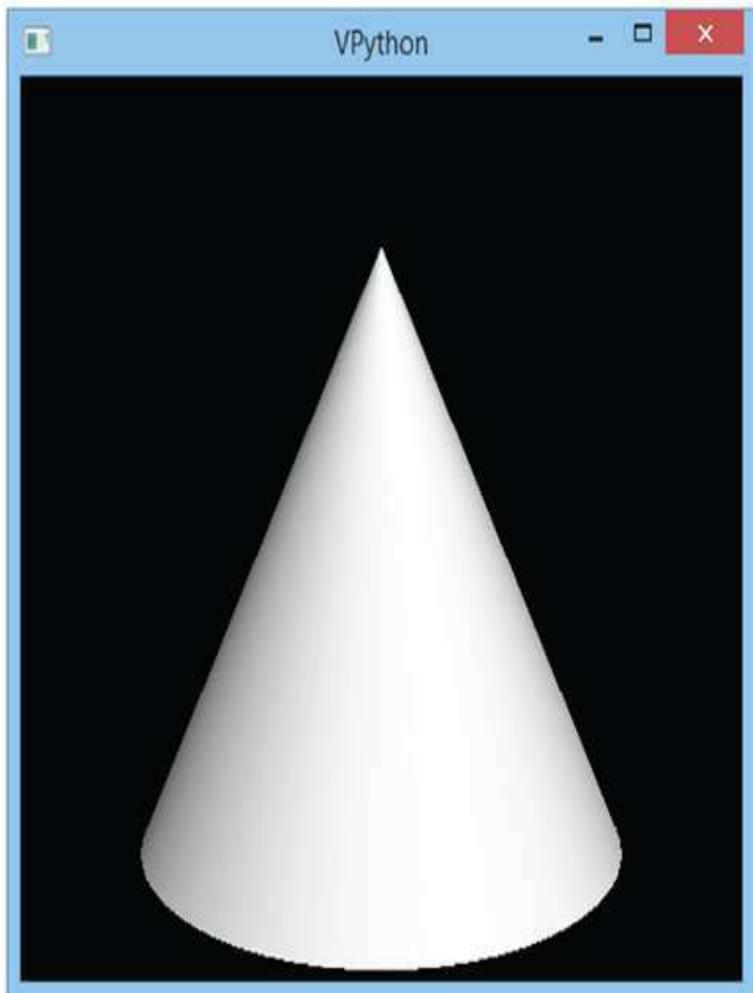
**Plate 22** Ring (see page 513)



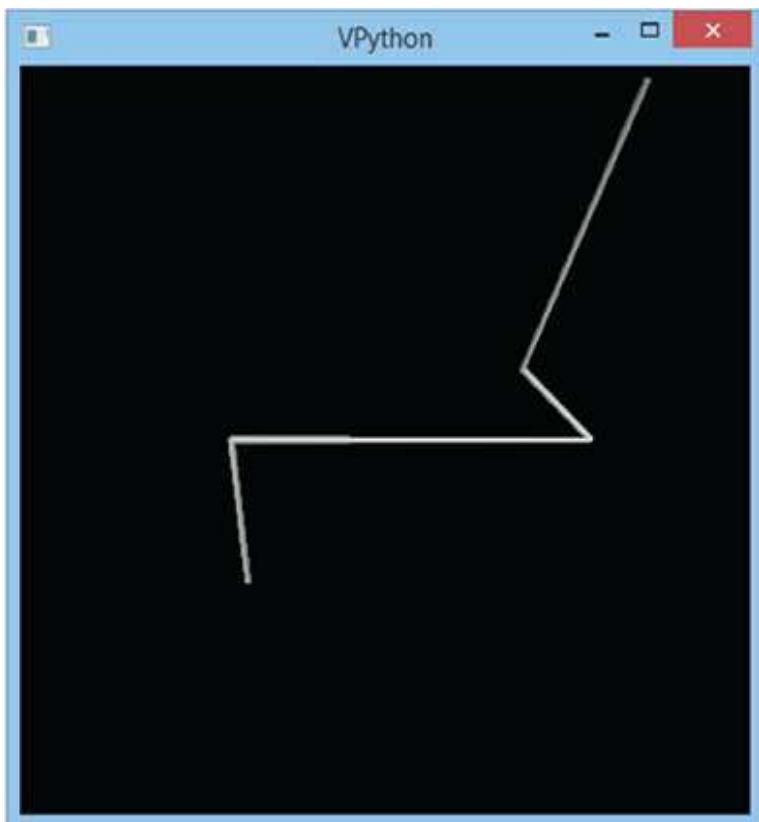
**Plate 23** Cylinder (see page 514)



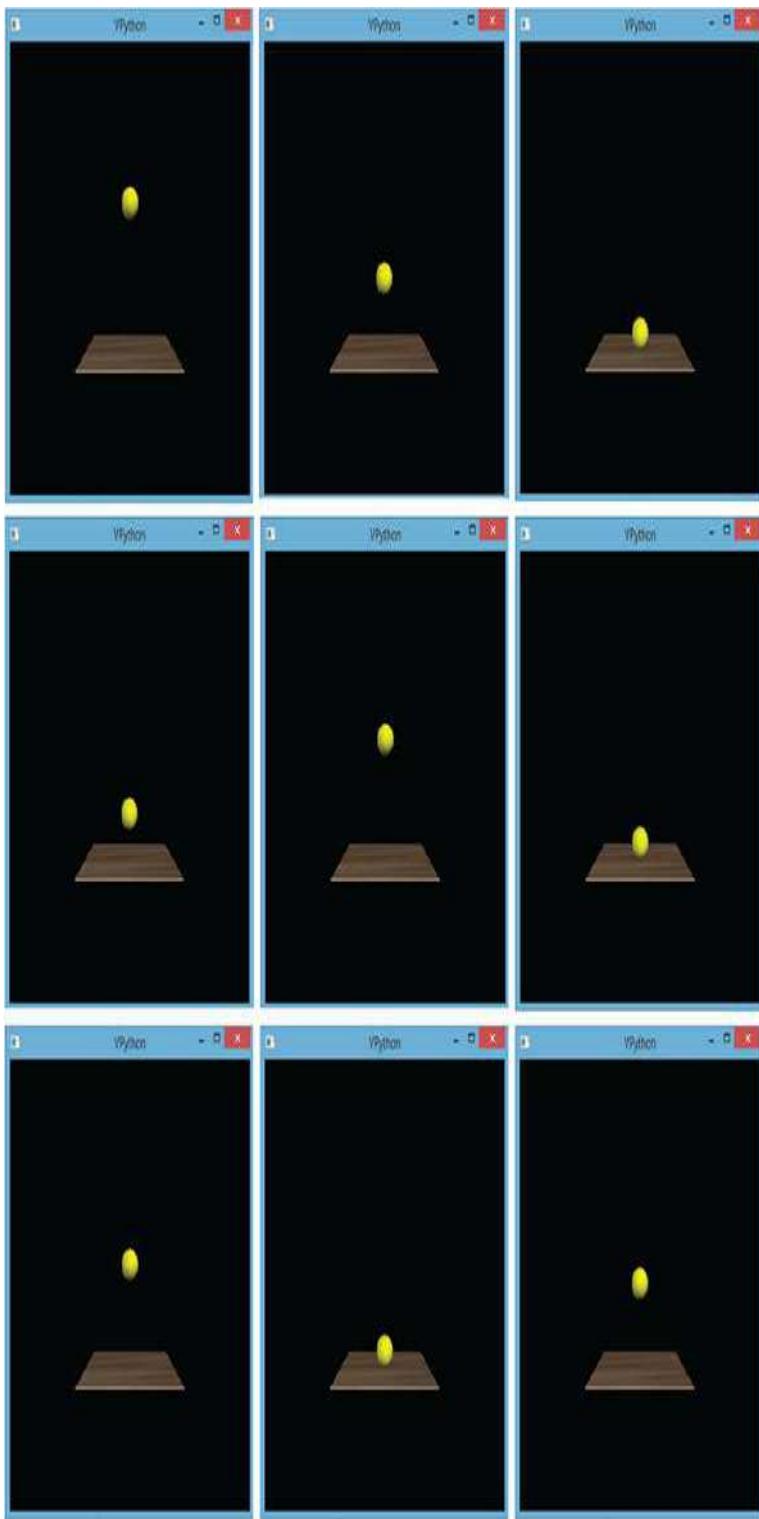
**Plate 24** Arrow (see page 515)



**Plate 25** Cone (see page 516)



**Plate 26** Curve (see page 517)



**Plate 27** Snapshots of bouncing ball (see page 518)

*Editor—Acquisitions:* Neha Goomer  
*Editor—Production:* M. Balakrishnan

**Copyright © 2018 Pearson India Education Services Pvt. Ltd**

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of both the copyright owner and the publisher of this book.

ISBN 978-93-325-8534-8  
eISBN 978-93-528-6603-8

**First Impression**

Published by Pearson India Education Services Pvt. Ltd, CIN: U72200TN2005PTC057128, formerly known as TutorVista Global Pvt. Ltd, licensee of Pearson Education in South Asia.

Head Office: 15th Floor, Tower-B, World Trade Tower, Plot No. 1, Block-C, Sector 16, Noida 201 301, Uttar Pradesh, India.

Registered Office: 4th Floor, Software Block, Elnet Software City, TS 140, Block 2 & 9, Rajiv Gandhi Salai, Taramani, Chennai 600 113, Tamil Nadu, India.

Fax: 080-30461003, Phone: 080-30461060

Website: [in.pearson.com](http://in.pearson.com). Email:  
[companysecretary.india@pearson.com](mailto:companysecretary.india@pearson.com)

*Compositor:* MacroTex Solutions, Chennai.

*Printed in India.*