CONTENTS

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.
- **Animations** simulate the execution of a program by stepping through the code. They help students comprehend the code. More importantly, the visual illustration in Animations help students understand programming concepts.
- **VideoNotes** simulate the "office hours experience" through narrated video tutorials that show how to solve problems completely, from design through coding.

New Features

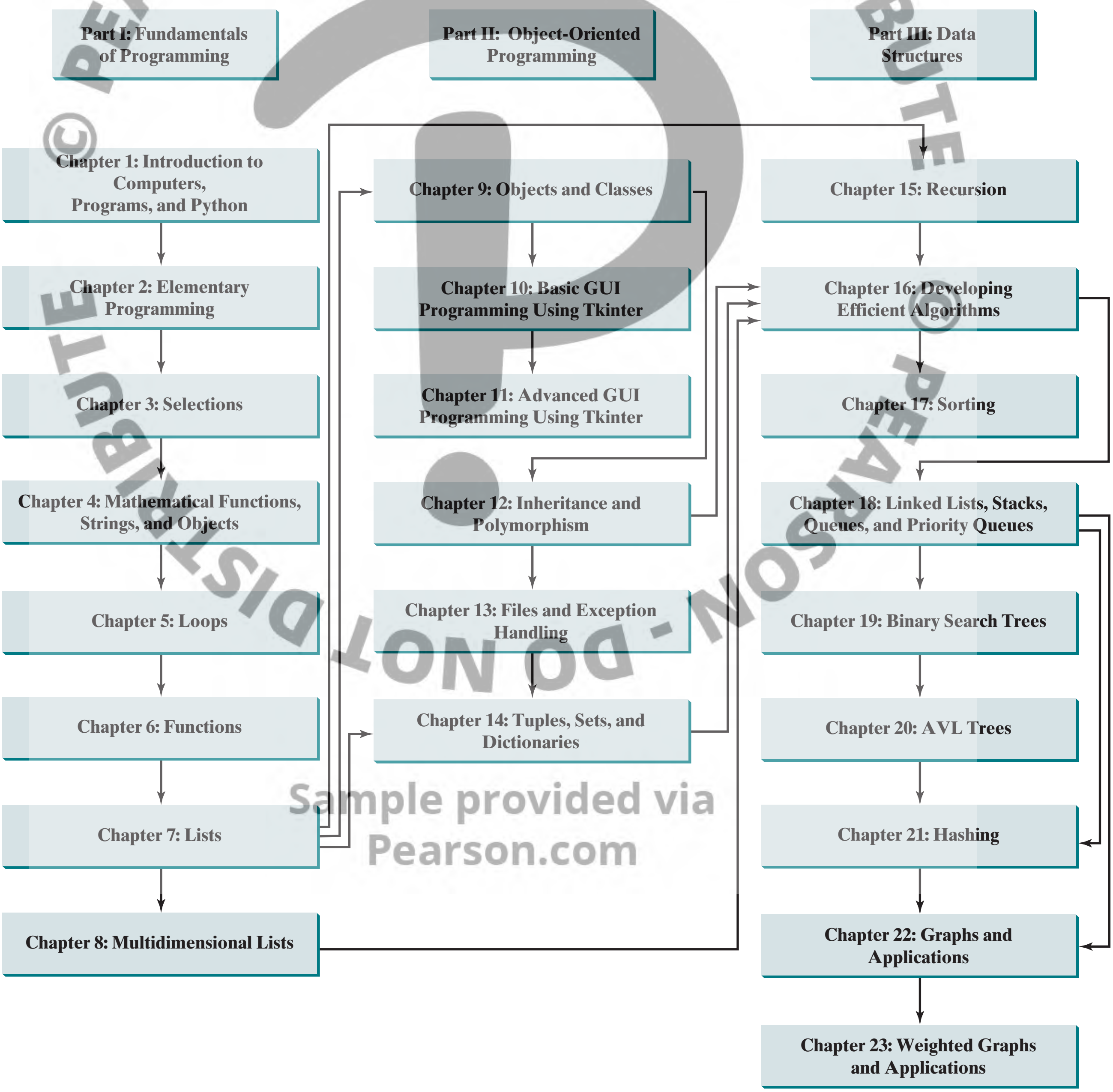
This new edition is completely revised in every detail to enhance clarity, presentation, content, examples, and exercises. The major improvements are as follows:

- **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.
- Provided additional exercises not printed in the book. These exercises are available for instructors only.

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.

Student Resource Website

The Student Resource Website (www.pearsonglobaleditions.com) contains the following resources:

- Solutions to majority of even-numbered programming exercises
- Source code for the examples in the book
- Interactive quiz (organized by sections for each chapter)
- Supplements
- Debugging tips
- Video notes
- Algorithm animations

Supplements

The text covers the essential subjects. The supplements extend the text to introduce additional topics that might be of interest to readers. The supplements are available from the Companion Website.

Instructor Resource Website

The Instructor Resource Website, accessible from www.pearsonglobaleditions.com, contains the following resources:

- Microsoft PowerPoint slides with interactive buttons to view full-color, syntax-highlighted source code and to run programs without leaving the slides.
- Solutions to majority of odd-numbered 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.
- Web-based quiz generator. (Instructors can choose chapters to generate quizzes from a large database of more than two thousand questions.)
- Sample exams. Most exams have four parts:
  - Multiple-choice questions or short-answer questions
  - Correct programming errors
  - Trace programs
  - Write programs
- Sample exams with ABET course assessment.
- Projects. In general, each project gives a description and asks students to analyze, design, and implement the project.

Some readers have requested the materials from the Instructor Resource Website. Please note that these are for adopting instructors only. Such requests will not be answered.

Video Notes

VideoNote

We are excited about the new Video Notes feature that is found in this new edition. These videos provide additional help by presenting examples of key topics and showing how to solve problems completely from design through coding. Video Notes are available from the Companion Website.

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 Mayfield (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.

Acknowledgements for the Global Edition

The publishers would like to thank the following for their contribution to the Global Edition:

Contributors

Asif Irshad Khan – King Abdulaziz University
Markus Wolf – University of Greenwich

Reviewers

Greg Baatard – Edith Cowan University
Dmitry Konovalov – James Cook University
Francesco Tusa – University of Westminster
Lindsay Ward – James Cook University

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).

Key Point

5.1 Introduction

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.

Key Point

5.2 The `while` Loop

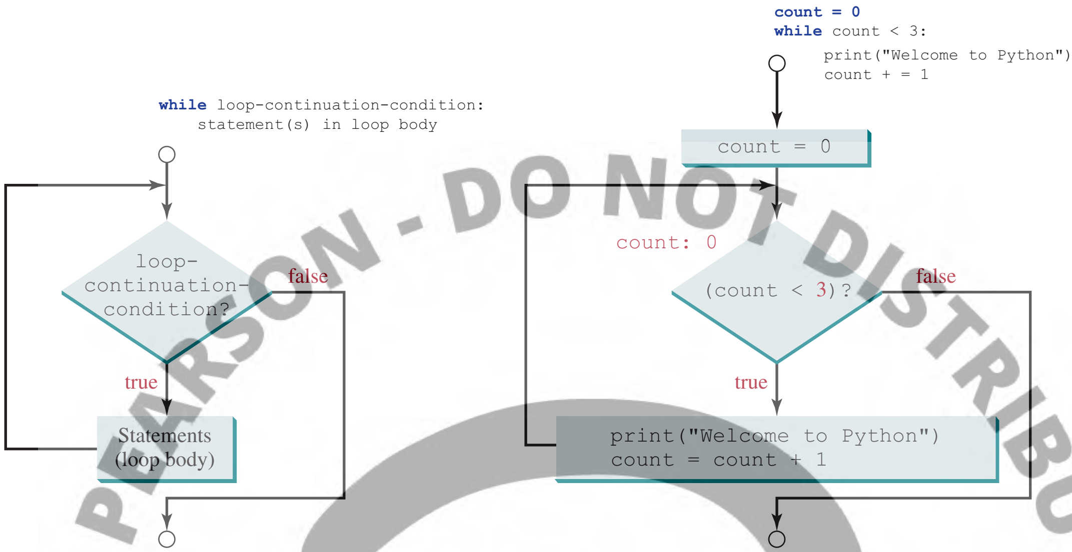
A `while` loop executes statements repeatedly as long as a condition remains true.

VideoNote
while loop

The syntax for the `while` loop is:

```
while loop-continuation-condition:
    # Loop body
    Statements
```

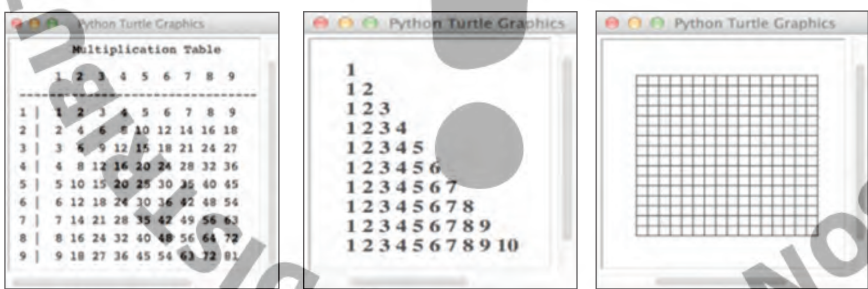
Figure 5.1a shows the `while`-loop flowchart. A single execution of a loop body is called an *iteration* (or repetition) of the loop. Each loop contains a *loop-continuation-condition*, a Boolean expression that controls the body's execution. It is evaluated each time to determine if the loop body is 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.



(a) A while loop flowchart. (b) A while loop flowchart animation.

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:



(a) (b) (c)

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

```
import random

# 1. Generate two random single-digit integers
number1 = random.randint(0, 9)
number2 = random.randint(0, 9)

# 2. If number1 < number2, swap number1 with number2
if number1 < number2:
    number1, number2 = number2, number1

# 3. Prompt the student to answer What is number1 – number2?
answer = int(input("What is " + str(number1) + " - "
    + str(number2) + "? "))

# 5. Repeatedly ask the user the question until it is correct
while number1 - number2 != answer:
    answer = int(input("Wrong answer. Try again. What is "
        + str(number1) + " - " + str(number2) + "? "))

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

```python
import random

# Generate a random number to be guessed
number = random.randint(0, 100)

print("Guess a magic number between 0 and 100")

# Prompt the user to guess the number
guess = int(input("Enter your guess: "))

if guess == number:
    print("Yes, the number is " + str(number))
elif guess > number:
    print("Your guess is too high")
else:
    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:

```python
while True:
    # Prompt the user to guess the number
    guess = int(input("Enter your guess: "))

    if guess == number:
        print("Yes, the number is", number)
    elif guess > number:
        print("Your guess is too high")
    else:
        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:

```python
while guess != number:
    # Prompt the user to guess the number
    guess = int(input("Enter your guess: "))

    if guess == number:
        print("Yes, the number is", number)
    elif guess > number:
        print("Your guess is too high")
    else:
        print("Your guess is too low")
```

The complete code is given in Listing 5.3.

LISTING 5.3 GuessNumber.py

```python
import random

# Generate a random number to be guessed
number = random.randint(0, 100)

print("Guess a magic number between 0 and 100")

guess = -1
while guess != number:
    # Prompt the user to guess the number
    guess = int(input("Enter your guess: "))

    if guess == number:
        print("Yes, the number is", number)
    elif guess > number:
        print("Your guess is too high")
    else:
        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

```
import random
import time

correctCount = 0 # Count the number of correct answers
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.*

VideoNote
for loop

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

Using floating-point numbers in the loop-continuation-condition may cause numeric errors.

Key Point

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

```
# Initialize sum
sum = 0

# Add 0.01, 0.02, ..., 0.99, 1 to sum
i = 0.01
while i <= 1.0:
    sum += i
    i = i + 0.01

# Display result
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

```
#Prompt the user to enter two integers
n1 = int(input("Enter first integer: "))
n2 = int(input("Enter second integer: "))

gcd = 1
k = 2
while k <= n1 and k <= n2:
    if n1 % k == 0 and n2 % k == 0:
        gcd = k
```

```
    k += 1

print("The greatest common divisor for",
    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`.

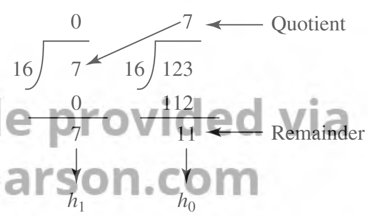
5.9.3 Problem: Converting Decimals to Hexadecimals

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}, ..., h_2, h_1, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots$$
$$+ h2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

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, ..., 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.*

Key Point

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

```
sum = 0
number = 0

while number < 20:
    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

Key Point

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")
```

```
Enter a string: mom
mom is 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

```python
NUMBER_OF_PRIMES = 50  # Number of primes to display
NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
count = 0 # Count the number of prime numbers
number = 2 # A number to be tested for primeness

print("The first 50 prime numbers are")

# Repeatedly find prime numbers
while count < NUMBER_OF_PRIMES:
    # Assume the number is prime
    isPrime = True # Is the current number prime?

    # Test if number is prime
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            # If true, the number is not prime
            isPrime = False # Set isPrime to false
            break  # Exit the for loop
        divisor += 1

    # Display the prime number and increase the count
    if isPrime:
        count += 1 # Increase the count

        print(f"{number:5d}", end = '')
        if count % NUMBER_OF_PRIMES_PER_LINE == 0:
            # Display the number and advance to the new line
            print() # Jump to the new line

```

```
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
```

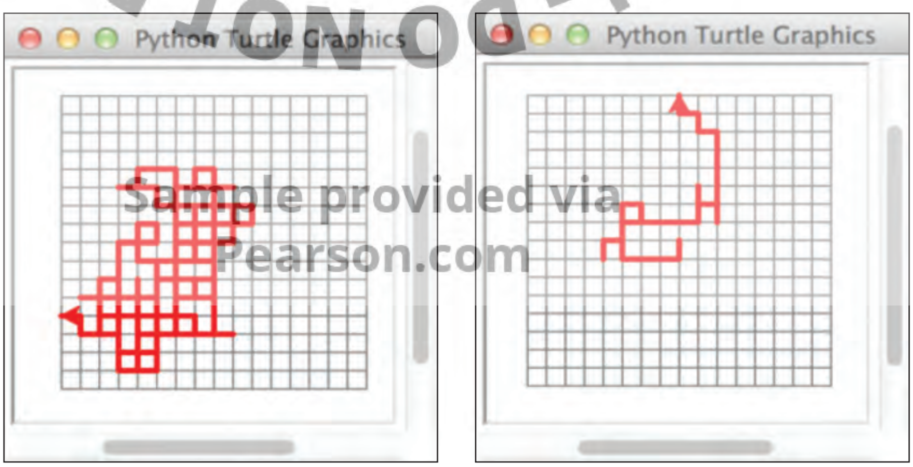
However, using the *break* statement makes the program simpler and easier to read in this case.

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.

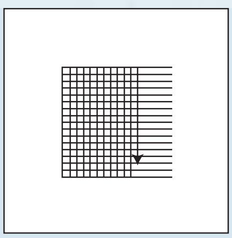


(a) (b)

Figure 5.2 The program simulates random walks in a lattice. (Screenshots courtesy of Apple.)

LISTING 5.15 RandomWalk.py

```
import turtle
from random import randint

turtle.speed(5) # Set turtle speed to medium

# Draw 16 by 16 lattices
turtle.color("gray") # Color for lattice
x = -80
for y in range(-80, 80 + 1, 10):
    turtle.penup()
    turtle.goto(x, y) # Draw a horizontal line
    turtle.pendown()
    turtle.forward(160)

y = 80
turtle.right(90)
for x in range(-80, 80 + 1, 10):
    turtle.penup()
    turtle.goto(x, y) # Draw a vertical line
    turtle.pendown()
    turtle.forward(160)

turtle.pensize(3)
turtle.color("red")

turtle.penup()
turtle.goto(0, 0) # Go to the center
turtle.pendown()

x = y = 0 # Current pen location at the center of lattice
while abs(x) < 80 and abs(y) < 80:
    r = randint(0, 3)
    if r == 0:
        x += 10  # Walk east
        turtle.setheading(0)
        turtle.forward(10)
    elif r == 1:
        y -= 10 # Walk south
        turtle.setheading(270)
        turtle.forward(10)
    elif r == 2:
        x -= 10 # Walk west
        turtle.setheading(180)
        turtle.forward(10)
    elif r == 3:
        y += 10 # Walk north
        turtle.setheading(90)
        turtle.forward(10)

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 (*Repeat additions*) Listing 5.4, SubtractionQuizLoop.py, generates five random subtraction questions. Revise the program to generate ten random addition questions for two integers between 1 and 15. Display the correct count and test time.

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                15.1
...
96              363.4
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 (*Find numbers divisible by 11 and 17*) Write a program that displays, five numbers per line, all the numbers from 1,000 to 5,000 that are divisible by 11 and 17. The numbers are separated by exactly one tab.

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`, `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** (*Find the factors of an integer*) Write a program that reads an integer and displays all its smallest factors, also known as *prime factors*. For example, if the input integer is 120, the output should be as follows:

2 2 2 3 5

```
Enter a positive integer: 120
The prime factors for 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
                      1   2   4   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
  1   2   4   8  16  32  64 128  64  32  16   8   4   2   1
```

***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.

```
Enter loan amount, for example 120000.95:  10000.65
Enter number of years as an integer, for example 5:  5
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:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{i!}$$

Write a program that displays the **e** value for `i = 10000`, `20000`, …, and `100000`. (Hint: Since $i! = i \times (i - 1) \times \ldots \times 2 \times 1$, 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   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
```

***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.

$$\frac{1}{1 + \sqrt{2}} + \frac{1}{\sqrt{2} + \sqrt{3}} + \frac{1}{\sqrt{3} + \sqrt{4}} + \ldots + \frac{1}{\sqrt{624} + \sqrt{625}}$$

***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 (*Simulation: heads or tails*) Write a program that simulates flipping a coin one million times and displays the number of heads and tails.

***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** (*Process string*) Write a program that prompts the user to enter a string and displays all the characters at positions 3, 6, 9 and so on.

```
Enter a string: 123456789abcdef
369cf
```

***5.43** (*Math: combinations*) Write a program that displays all possible combinations for picking two numbers from integers `1` to `7`. Also display the total number of combinations.

****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** (*Decimal to octal*) Write a program that prompts the user to enter a decimal integer and displays its corresponding octal value.

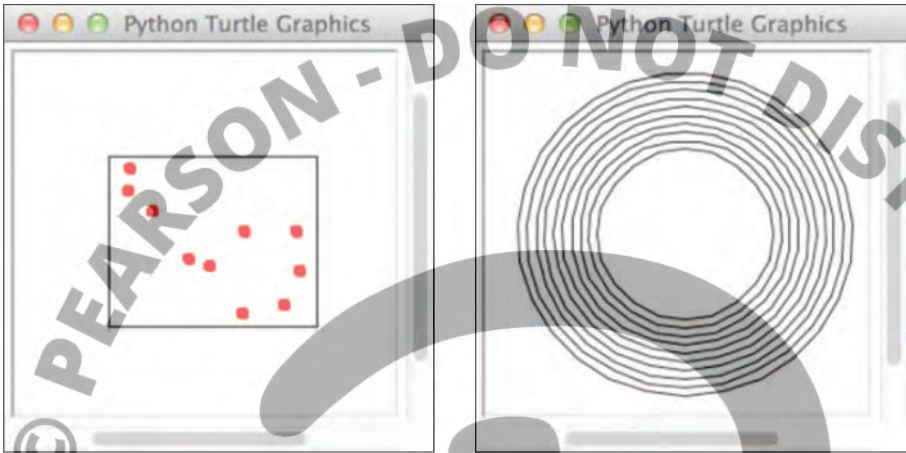
```
Enter an integer: 100
The octal representation is 144
```

****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:

$$mean = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \ldots + x_n}{n} \quad deviation = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}}$$

```
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.



(a) (b)

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.

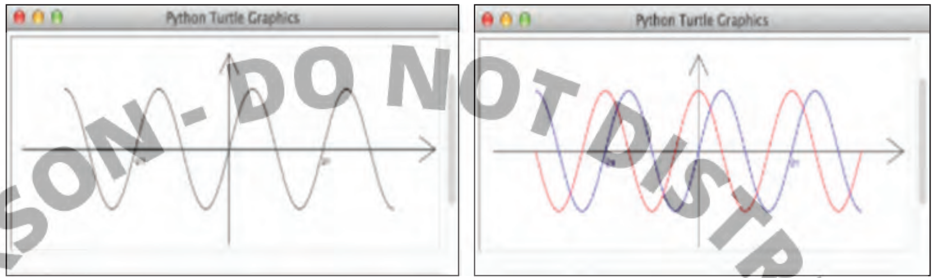
(a) (b) (c)

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.



(a) (b)

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.)

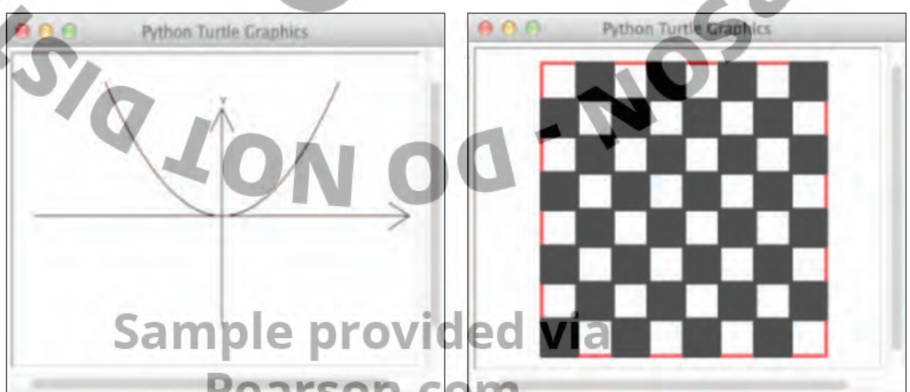
Hint: The Unicode for π is `\u03c0`. To display -2π use `turtle.write("-2\u03c0")`. For a trigonometric function like `sin(x)`, `x` is in radians. Use the following loop to plot the sine function:

```
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.

****5.54** (*Turtle: plot the square function*) Write a program that draws a diagram for the function $f(x) = x^2$ (see Figure 5.6a).



(a) (b)

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** (*Count uppercase letters*) Write a program that prompts the user to enter a string and displays the number of the uppercase letters in the string.

```
Enter a string: Programming is fun
The number of uppercase letter in Programming is fun is 1
```

***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** (*Reverse a string*) Write a program that prompts the user to enter a string and displays the string in reverse order.

```
Enter a string: Welcome
The reversed string is emocleW
```

***5.59** (*Count vowels and consonants*) Assume letters **A**, **E**, **I**, **O**, and **U** as the vowels. Write a program that prompts the user to enter a string and displays the number of vowels and consonants in the string.

```
Enter a String: Programming is fun
The number of Vowels is 5
The number of consonants is 11
```

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.

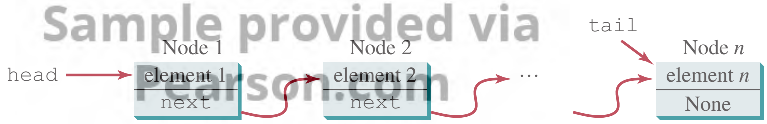


FIGURE 18.1 A linked list consists of any number of nodes chained together.

Animation: Linked List

Pedagogical Note

For an interactive demo on how linked lists work, see http://liveexample.pearsoncmg.com/liang/animation/web/LinkedList.html.

A node can be defined as a class, as follows:

```
class Node:
    def __init__(self, e):
        self.elmenet = 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.

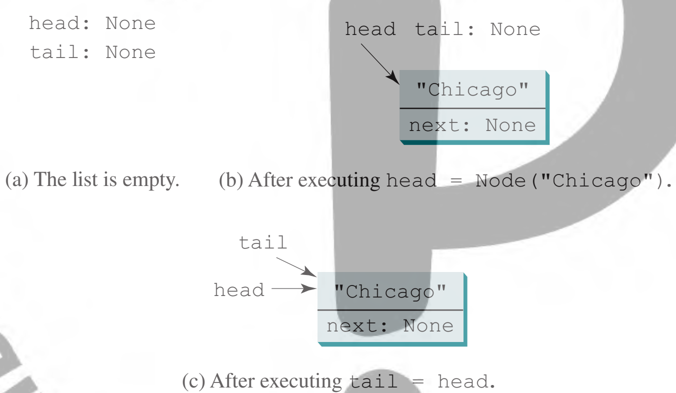


FIGURE 18.2 Append the first node to the list.

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.

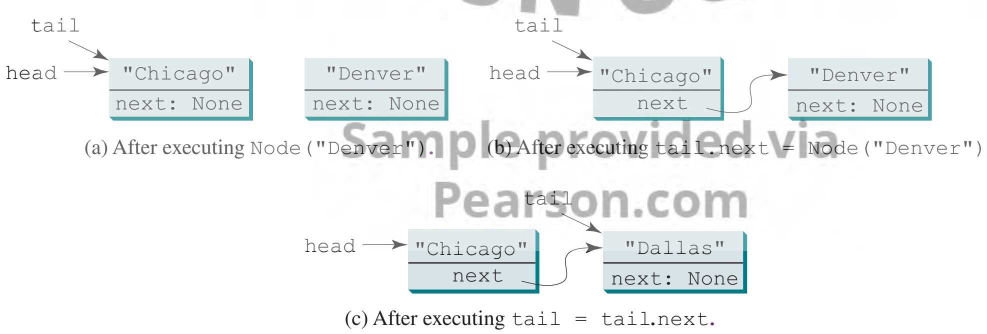


FIGURE 18.3 Append the second node to the list.

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.

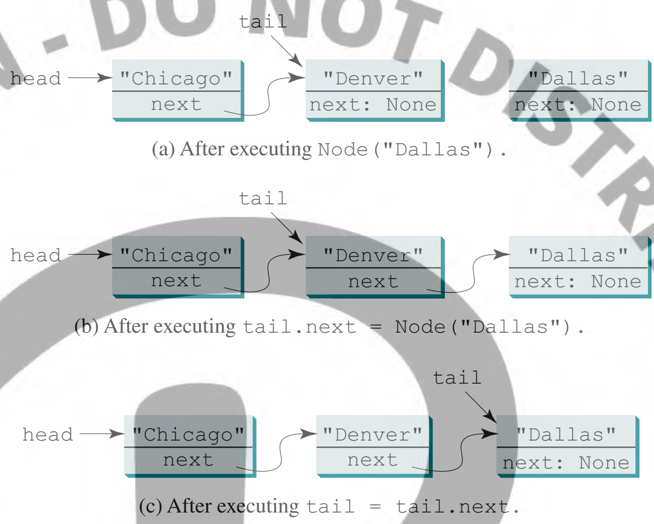


FIGURE 18.4 Append the third node to the list.

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`

```
from LinkedList import LinkedList

lst = LinkedList() # Create a linked list

```

```
# Add elements to the list
lst.add("America") # Add America to the list
print("(1)", lst)

lst.insert(0, "Canada") # Add Canada to the beginning of the list
print("(2)", lst)

lst.add("Russia") # Add Russia to the end of the list
print("(3)", lst)

lst.addLast("France") # Add France to the end of the list
print("(4)", lst)

lst.insert(2, "Germany") # Add Germany to the list at index 2
print("(5)", lst)

lst.insert(5, "Norway") # Add Norway to the list at index 5
print("(6)", lst)

lst.insert(0, "Poland") # Same as list.addFirst("Poland")
print("(7)", lst)

# Remove elements from the list
lst.removeAt(0) # Remove the element at index 0
print("(8)", lst)

lst.removeAt(2) # Remove the element at index 2
print("(9)", lst)

lst.removeAt(lst.getSize() - 1) # Remove the last element
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.

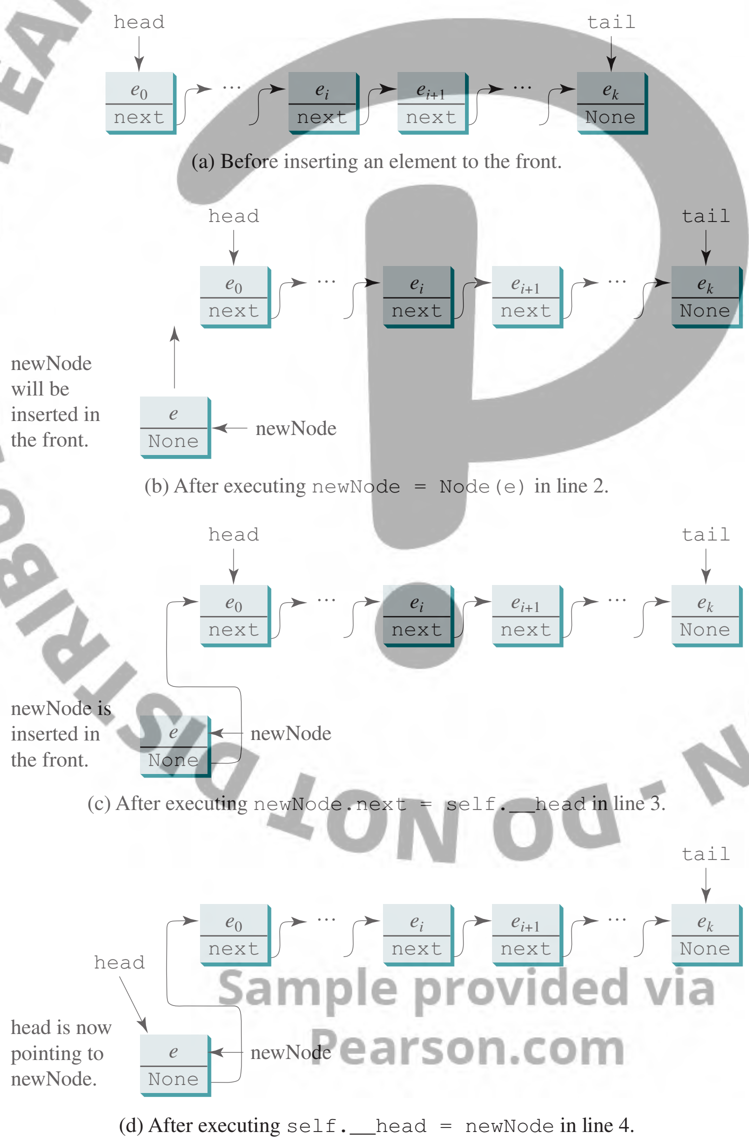


FIGURE 18.6 A new element is added to the beginning of the list.

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)`

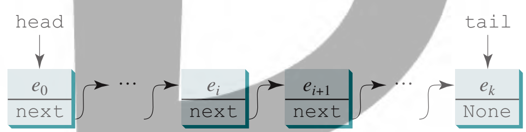
The `addLast(e)` method creates a node to hold the element and appends the node at the end of the list. It can be implemented as follows:

```
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
```

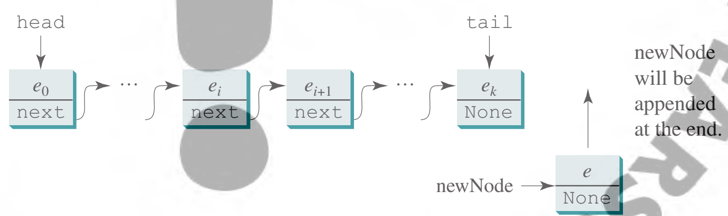
The `addLast(e)` method creates a new node to store the element (line 2) and appends it to the end of the list, as shown in Figure 18.7b. Consider two cases:

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.

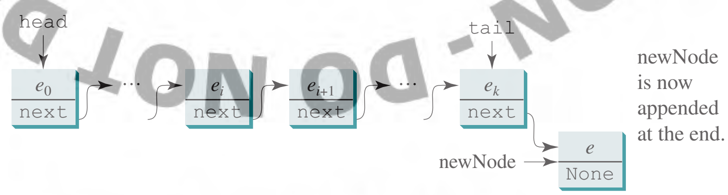
In any case, after the node is created, the size should be increased by `1` (line 10).



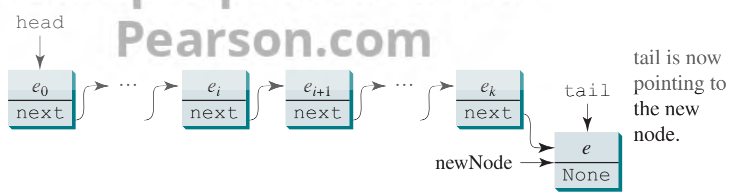
(a) Before appending an element to the end.



(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).

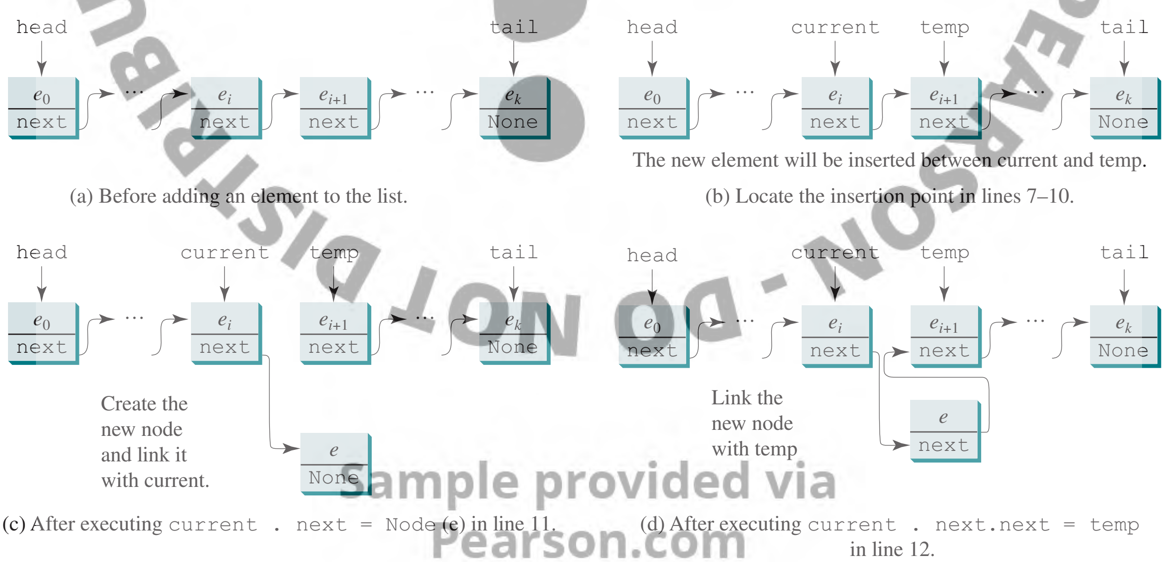


Figure 18.8 A new element is inserted in the middle of the list.

Clearly, the `insert(index, e)` method takes O(n) time.

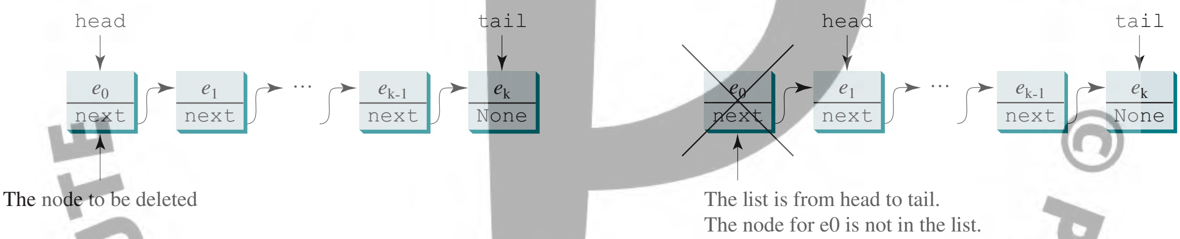
18.4.4 Implementing removeFirst()

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).



(a) Before deleting the first element. (b) After executing `self.__head = self.__head . next` in line 6.

Figure 18.9 The first node is deleted from the list.

Clearly, the `removeFirst()` method takes O(1) time.

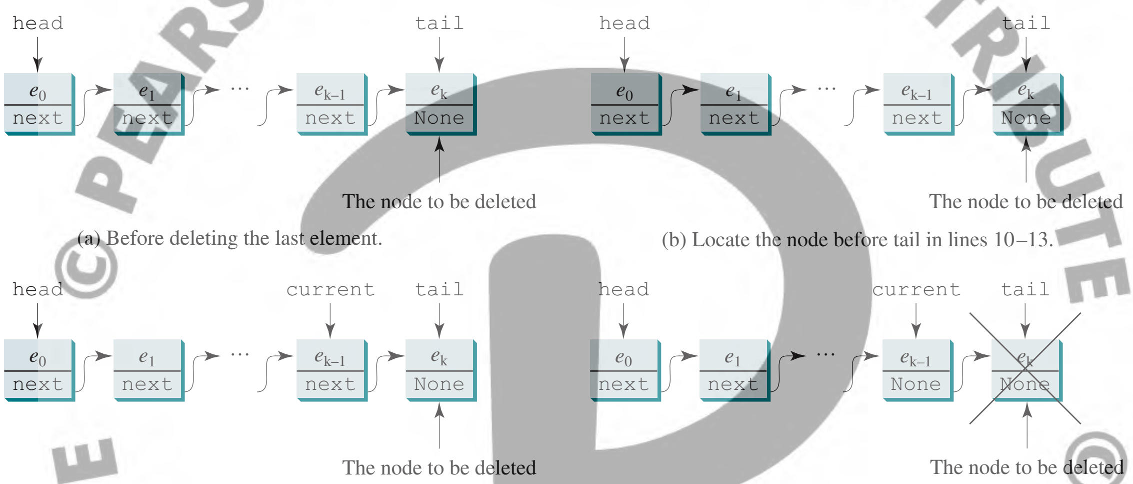
18.4.5 Implementing `removeLast()`

The `removeLast()` method removes the last element from the list. It can be implemented as follows:

```
def removeLast(self):
    if self.__size == 0:
        return None # Nothing to remove
    elif self.__size == 1: # Only one element in the list
        temp = self.__head
        self.__head = self.__tail = None # list becomes empty
        self.__size = 0
        return temp.element
    else:
        current = self.__head
        for i in range(self.__size - 2):
            current = current.next

        temp = self.__tail
        self.__tail = current
        self.__tail.next = None
        self.__size -= 1
        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).



(a) Before deleting the last element.

(b) Locate the node before tail in lines 10–13.

(c) After executing `self.__tail = current` in the 16.

(d) After executing `current . next = None` in the 17.

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:

```
def removeAt(self, index):
    if index < 0 or index >= self.__size:
        return None # Out of range
    elif index == 0:
        return self.removeFirst() # Remove first
    elif index == self.__size - 1:
        return self.removeLast() # Remove last
    else:
        previous = self.__head
        for i in range(1, index):
            previous = previous.next

        current = previous.next
        previous.next = current.next
        self.__size -= 1
        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.

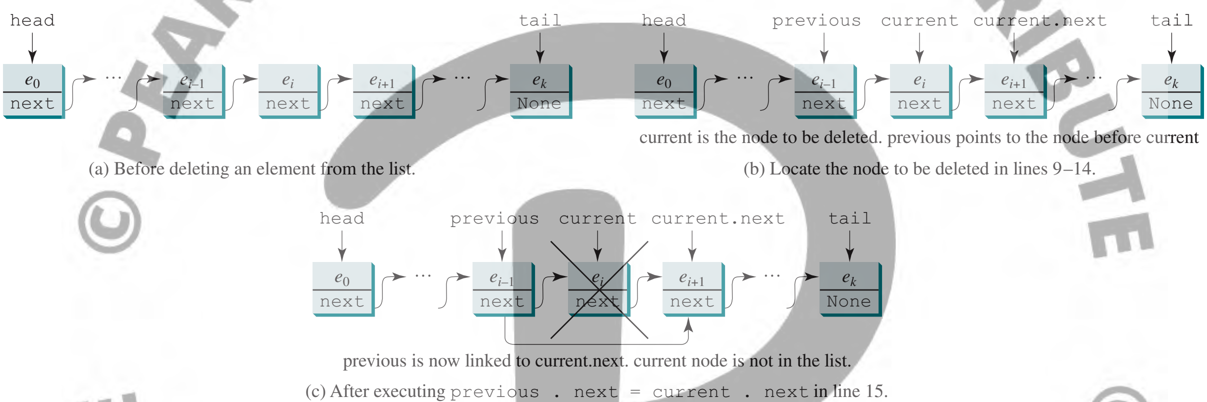


FIGURE 18.11 A node is deleted from the list.

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

```python
class LinkedList:
    def __init__(self):
        self.__head = None
        self.__tail = None
        self.__size = 0

    # Return the head element in the list
    def getFirst(self):
        if self.__size == 0:
            return None
        else:
            return self.__head.element

    # Return the last element in the list
    def getLast(self):
        if self.__size == 0:
            return None
        else:
            return self.__tail.element

```

```
    # Add an element to the beginning of the list
    def addFirst(self, e):
        newNode = Node(e) # Create a new node
        newNode.next = self.__head # link the new node with the head
        self.__head = newNode # head points to the new node
        self.__size += 1 # Increase list size

        if self.__tail == None: # the new node is the only node in list
            self.__tail = self.__head

    # Add an element to the end of the list
    def addLast(self, e):
        newNode = Node(e) # Create a new node for e

        if self.__tail == None:
            self.__head = self.__tail = newNode # The only node in list
        else:
            self.__tail.next = newNode # Link the new with the last node
            self.__tail = self.__tail.next # tail now points to the last node

        self.__size += 1 # Increase size

    # Same as addLast
    def add(self, e):
        self.addLast(e)

    # Insert a new element at the specified index in this list
    # The index of the head element is 0
    def insert(self, index, e):
        if index == 0:
            self.addFirst(e) # Insert first
        elif index >= self.__size:
            self.addLast(e) # Insert last
        else: # Insert in the middle
            current = self.__head
            for i in range(1, index):
                current = current.next
            temp = current.next
            current.next = Node(e)
            (current.next).next = temp
            self.__size += 1

    # Remove the head node and
    # return the object that is contained in the removed node.
    def removeFirst(self):
        if self.__size == 0:
            return None # Nothing to delete
        else:
            temp = self.__head # Keep the first node temporarily
            self.__head = self.__head.next # Move head to point the next node
            self.__size -= 1 # Reduce size by 1
            if self.__head == None:
                self.__tail = None # List becomes empty
            return temp.element # Return the deleted element

    # Remove the last node and
    # return the object that is contained in the removed node
    def removeLast(self):
        if self.__size == 0:
            return None # Nothing to remove
```

```
        elif self.__size == 1: # Only one element in the list
            temp = self.__head
            self.__head = self.__tail = None # list becomes empty
            self.__size = 0
            return temp.element
        else:
            current = self.__head

            for i in range(self.__size - 2):
                current = current.next

            temp = self.__tail
            self.__tail = current
            self.__tail.next = None
            self.__size -= 1
            return temp.element

    # Remove the element at the specified position in this list.
    # Return the element that was removed from the list.
    def removeAt(self, index):
        if index < 0 or index >= self.__size:
            return None # Out of range
        elif index == 0:
            return self.removeFirst() # Remove first
        elif index == self.__size - 1:
            return self.removeLast() # Remove last
        else:
            previous = self.__head

            for i in range(1, index):
                previous = previous.next

            current = previous.next
            previous.next = current.next
            self.__size -= 1
            return current.element

    # Return true if the list is empty
    def isEmpty(self):
        return self.__size == 0

    # Return the size of the list
    def getSize(self):
        return self.__size

    def __str__(self):
        result = "["

        current = self.__head
        for i in range(self.__size):
            result += str(current.element)
            current = current.next
            if current != None:
                result += ", " # Separate two elements with a comma
            else:
                result += "]" # Insert the closing ] in the string

        return result

    # Clear the list */
    def clear(self):
        self.__head = self.__tail = None
```

```python

    # Return true if this list contains the element o
    def contains(self, e):
        print("Implementation left as an exercise")
        return True

    # Remove the element and return true if the element is in the list
    def remove(self, e):
        print("Implementation left as an exercise")
        return True

    # Return the element from this list at the specified index
    def get(self, index):
        print("Implementation left as an exercise")
        return None

    # Return the index of the head matching element in this list.
    # Return -1 if no match.
    def indexOf(self, e):
        print("Implementation left as an exercise")
        return 0

    # Return the index of the last matching element in this list
    # Return -1 if no match.
    def lastIndexOf(self, e):
        print("Implementation left as an exercise")
        return 0

    # Replace the element at the specified position in this list
    # with the specified element. */
    def set(self, index, e):
        print("Implementation left as an exercise")
        return None

    # Return elements via indexer
    def __getitem__(self, index):
        return self.get(index)

    # Return an iterator for a linked list
    def __iter__(self):
        return LinkedListIterator(self.__head)

# The Node class
class Node:
    def __init__(self, e):
        self.element = e
        self.next = None

class LinkedListIterator:
    def __init__(self, head):
        self.current = head

    def __next__(self):
        if self.current == None:
            raise StopIteration
        else:
            element = self.current.element
            self.current = self.current.next
            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)`, `constains(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

Methods for list/Complexity		*Methods for LinkedList/Complexity*	
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

```
from LinkedList import LinkedList
import time

startTime = time.time()
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.

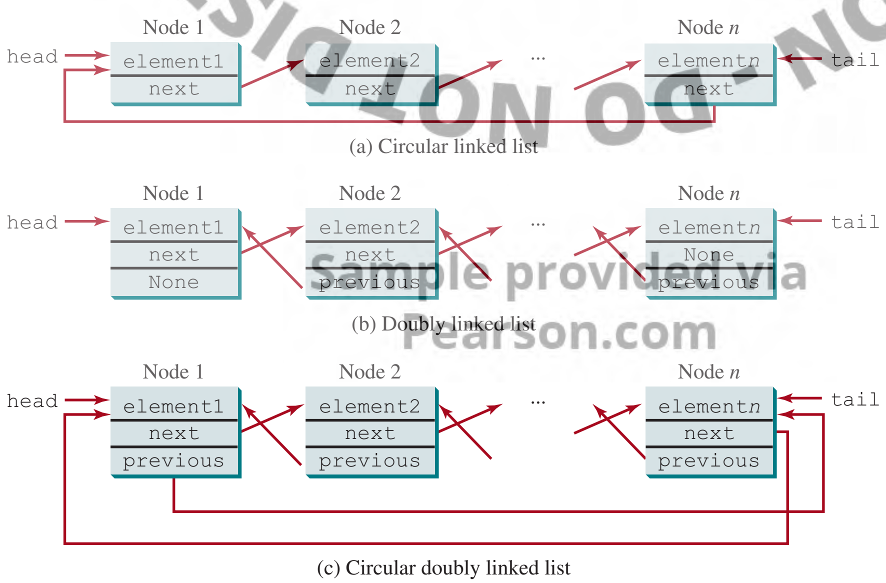


Figure 18.12 Linked lists may appear in various forms.

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 pointer 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

```
from LinkedList import LinkedList

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 `FibonacciIterator` 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

```
def fib():
    fn1 = 0 # Current two consecutive fibonacci numbers
    fn2 = 1
    while True:
        current = fn1
        fn1, fn2 = fn2, fn1 + fn2
        yield current # Yield a Fibonacci number

def main():
    iterator = fib()
    # Display all Fibonacci numbers <= 10000
    for i in iterator:
        if i <= 10000:
            print(i, end = ' ')
        else: break

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.

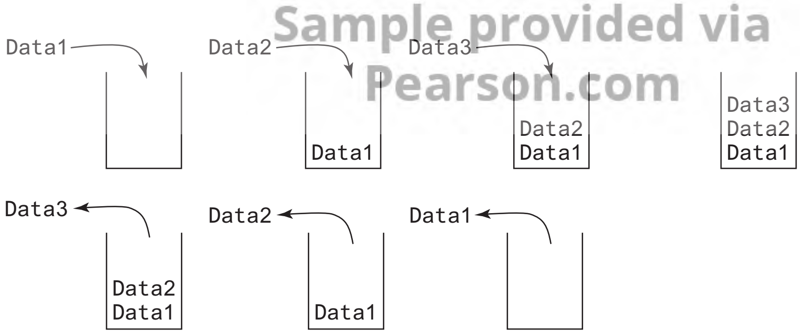


FIGURE 18.13 A `stack` holds data in a last-in, first-out fashion.

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.

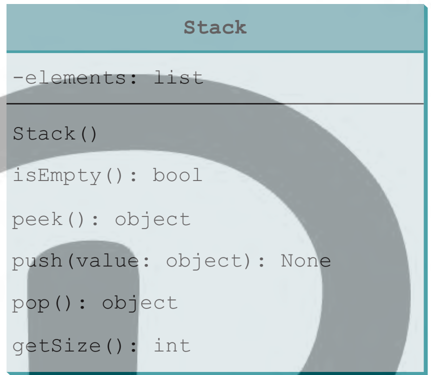


FIGURE 18.14 The **Stack** class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 18.7 Stack.py

```
class Stack:
    def __init__(self):
        self.__elements = []

    # Return true if the stack is empty
    def isEmpty(self):
        return len(self.__elements) == 0

    # Returns the element at the top of the stack
    # without removing it from the stack.
    def peek(self):
        if self.isEmpty():
            return None
        else:
            return self.__elements[len(self.__elements) - 1]

    # Stores an element into the top of the stack
    def push(self, value):
        self.__elements.append(value)

    # Removes the element at the top of the stack and returns it
    def pop(self):
        if self.isEmpty():
            return None
    else:
        return self.__elements.pop()

```

```
    # Return the size of the stack
    def getSize(self):
        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

Queues can be implemented using linked lists.

Key Point

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.

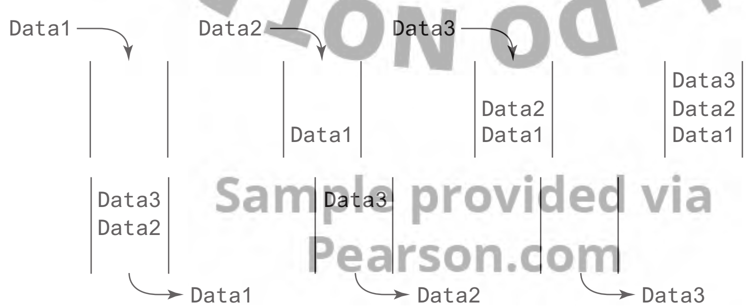


FIGURE 18.15 A queue holds objects in a first-in, first-out fashion.

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.

Queue
-elements: LinkedList
Queue() enqueue(e: object): None dequeue(): object getSize(): int isEmpty(): bool __str__(): str

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 **getSize**() 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
```

```

# Add elements to the queue
queue.enqueue("Nylah") # Add Nylah to the queue
print("(1)", queue)

queue.enqueue("Ashley") # Add Ashley to the queue
print("(2)", queue)

queue.enqueue("Curtis") # Add Curtis to the queue
queue.enqueue("Marisa") # Add Marisa to the queue
print("(3)", queue)

# Remove elements from the queue
print("(4)", queue.dequeue())
print("(5)", queue.dequeue())
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.

PriorityQueue
-heap: Heap
enqueue(element: object): None
dequeue(): object
getSize(): int

Figure 18.17 PriorityQueue uses a heap to provide a largest-in, first-out data structure.

Listing 18.11 PriorityQueue.py

```
from Heap import Heap

class PriorityQueue:
    def __init__(self):
        self.__heap = Heap()

    # Adds an element to this queue
    def enqueue(self, e):
        self.__heap.add(e)

    # Removes an element from this queue
    def dequeue(self):
        if self.getSize() == 0:
            return None
        else:
            return self.__heap.remove()

    # Return the size of the queue
    def getSize(self):
        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 18.12 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']
```


Key Point

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.

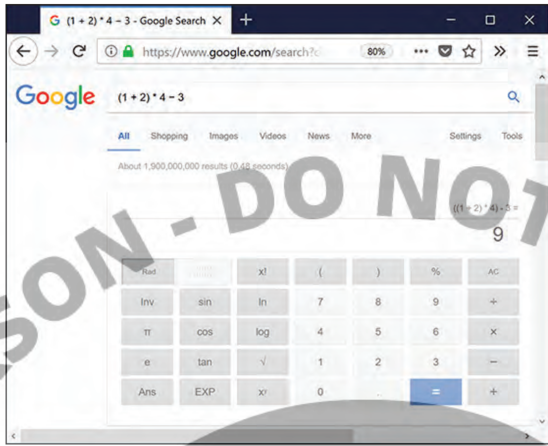


FIGURE 18.18 You can evaluate an arithmetic expression from Google.
(Screenshot of Google.)

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 `operand`Stack.

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 `operator`Stack.

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 `operator` Stack.

1.4 If the extracted item is a `(` symbol, push it to `operator`Stack.

1.5 If the extracted item is a `)` symbol, repeatedly process the operators from the top of `operator`Stack 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`

```
import Stack

def main():
```

```
    expression = input("Enter an expression: ").strip()
    try:
        print(expression, "=", evaluateExpression(expression))
    except:
        print("Wrong expression: ", expression)

# Evaluate an expression
def evaluateExpression(expression):
    # Create operandStack to store operands
    operandStack = Stack.Stack()

    # Create operatorStack to store operators
    operatorStack = Stack.Stack()

    # Insert blanks around (, ), +, -, /, and *
    expression = insertBlanks(expression)

    # Extract operands and operators
    tokens = expression.split()

    # Phase 1: Scan tokens
    for token in tokens:
        if len(token) == 0: # Blank space
            continue # Back to the while loop to extract the next token
        elif token[0] == '+' or token[0] == '-':
            # Process all +, -, *, / in the top of the operator stack
            while not operatorStack.isEmpty() and \
                (operatorStack.peek() == '+' or
                 operatorStack.peek() == '-' or
                 operatorStack.peek() == '*' or
                 operatorStack.peek() == '/'):
                 processAnOperator(operandStack, operatorStack)

            # Push the + or - operator into the operator stack
            operatorStack.push(token[0])
        elif token[0] == '*' or token[0] == '/':
            # Process all *, / in the top of the operator stack
            while not operatorStack.isEmpty() and \
                (operatorStack.peek() == '*' or
                 operatorStack.peek() == '/'):
                processAnOperator(operandStack, operatorStack)

            # Push the * or / operator into the operator stack
            operatorStack.push(token[0])
        elif token.strip()[0] == '(':
            operatorStack.push('(') # Push '(' to stack
        elif token.strip()[0] == ')':
            # Process all the operators in the stack until seeing '('
            while operatorStack.peek() != '(':
                processAnOperator(operandStack, operatorStack)

            operatorStack.pop() # Pop the '(' symbol from the stack
        else: # An operand scanned
            # Push an operand to the stack
            operandStack.push(float(token))

    # Phase 2: process all the remaining operators in the stack
    while not operatorStack.isEmpty():
        processAnOperator(operandStack, operatorStack)

```

```
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()
```

```
Enter an expression: (13 + 2) * 4 - 3
(13 + 2) * 4 - 3 = 57.0
```

The program reads an expression as a string (line 4) and invokes the `evaluateExpressionfunction` (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 (*Implement set operations in* `LinkedList`) Define a new class named `MyLinkedList` that extends `LinkedList` with the following set methods:

```
# Add the elements in otherList to this list.
# Return true if this list changed as a result of the call
def addAll(self, otherList):
# Remove all the elements in otherList from this list
# Return true if this list changed as a result of the call
def removeAll(self, otherList):

# Retain the elements in this list that are also in otherList
# Return true if this list changed as a result of the call
def retainAll(self, otherList):
```

Use https://liangpy.pearsoncmg.com/test/Exercise18_01py3e.txt to test your code.

```
Enter list1: red green red black
Enter list2: red black yellow yellow
After list1.addAll(list2), list1 is [red, green, red, black,
red, black, yellow, yellow]
After list1.removeAll(list2), list1 is [green]
After list1.retainAll(list2), list1 is [red, red, black]
```

***18.2** (*Implement* `LinkedList`) The implementations of methods `clear()`, `contains(e)`, `get(index)`, and `lastIndexOf(e)` are omitted in the text. Implement these methods.

***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

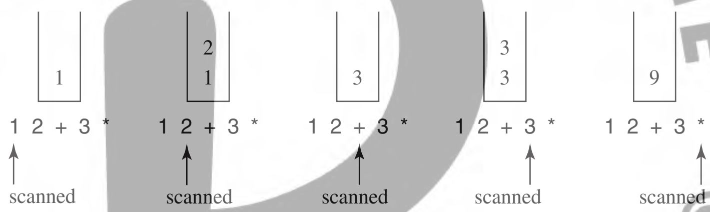
18.5 (*Implement a Queue*) The following code listing contains skeleton code for a multithreaded program, where the main thread generates `50` messages and places them in a Queue and another thread takes them from the Queue and prints them.

```
import threading
import queue

def main():
    threading.Thread(target=dequeuer, daemon=True).start()

    for i in range(50):
        # Add code to create a message,
        # add the message to the queue
        # and print the message
    print('All messages queued\n', end='')

    q.join()
    print('All work completed')

def dequeuer():
    while True:
        # Add code to read a message from the queue,
        # print the message
        # and call task_done() to inform the queue

q = queue.Queue()
main()
```

Complete the implementation in the for loop in `main()` and the while loop in `dequeuer()`.

***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** (*Postfix notation*) Postfix notation is a way of writing expressions without using parentheses. For example, the expression `(1 + 2) * 3` would be written as `1 2 + 3 *`. A postfix expression is evaluated using a stack. Scan a postfix expression from left to right. A variable or constant is pushed to the stack. When an operator is encountered, apply the operator with the top two operands in the stack and replace the two operands with the result. The following diagram shows how to evaluate `1 2 + 3 *`.



Write a program that prompts the user to enter a postfix expression and evaluate it.

****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.

(a) (b)

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, ..., 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
```