GLOBAL EDITION

Introduction to Python® Programming and Data Structures

THIRD EDITION

Y. Daniel Liang

Digital Resources for Students

Your new eBook provides 12-month access to digital resources that include VideoNotes, animations, case studies, supplements, and more on the Companion Website. Refer to the preface in the textbook for a detailed list of resources.

Follow these instructions to register for the Companion Website:

1. Go to www.pearsonglobaleditions.com.
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

ISSLIA-SHELL-SHONE-SNOBS-WERSH-UNRWA

Use the login name and password you created during registration to start using the online resources that accompany your textbook.

IMPORTANT:

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable.

For technical support, go to https://support.pearson.com/getsupport

INTRODUCTION TO PYTHON®

PROGRAMMING AND DATA STRUCTURES

Third Edition

Global Edition

Y. Daniel Liang

Georgia Southern University

Pearson

Product Management: Gargi Banerjee and K. K. Neelakantan
Content Strategy: Shabnam Dohutia, Aurko Mitra, and Anay Singh
Product Marketing: Wendy Gordon and Ashish Jain
Supplements: Bedasree Das
Production and Digital Studio: Vikram Medepalli, Naina Singh, and Niharika Thapa
Rights and Permissions: Anjali Singh
Cover image: voloshin311/Shutterstock

Please contact https://support.pearson.com/getsupport/s/ with any queries on this content.

Pearson Education Limited
KAO Two
KAO Park
Hockham Way
Harlow
CM17 9SR
United Kingdom

and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsonglobaleditions.com

Authorized adaptation from the United States edition, entitled Introduction to Python Programming and Data Structures, Revel, *1st edition*, *ISBN 978-0-13-518790-6 by Y. Daniel Liang published by Pearson Education © 2020.*

ISBN 10 (Print): 1-292-42412-5
ISBN 13 (Print): 978-1-292-42412-5
ISBN 13 (uPDF eBook): 978-1-292-42398-2

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

eBook formatted by B2R Technologies Pvt. Ltd.

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 programming 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 programming 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 effecting other contents of the book.

The book teaches problem solving in a problem-driven way that focuses on problem solving rather than syntax. We stimulate student interests 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 library are introduced in order 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. In order 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 are 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 programming is by doing. Basic concepts are explained by example and a large number of exercises with various levels of difficulty are provided for students to practice. 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.
- **Animation** simulates the execution of a program by stepping through the code. It helps students comprehend the code. More importantly, the visual illustration in Animation helps 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

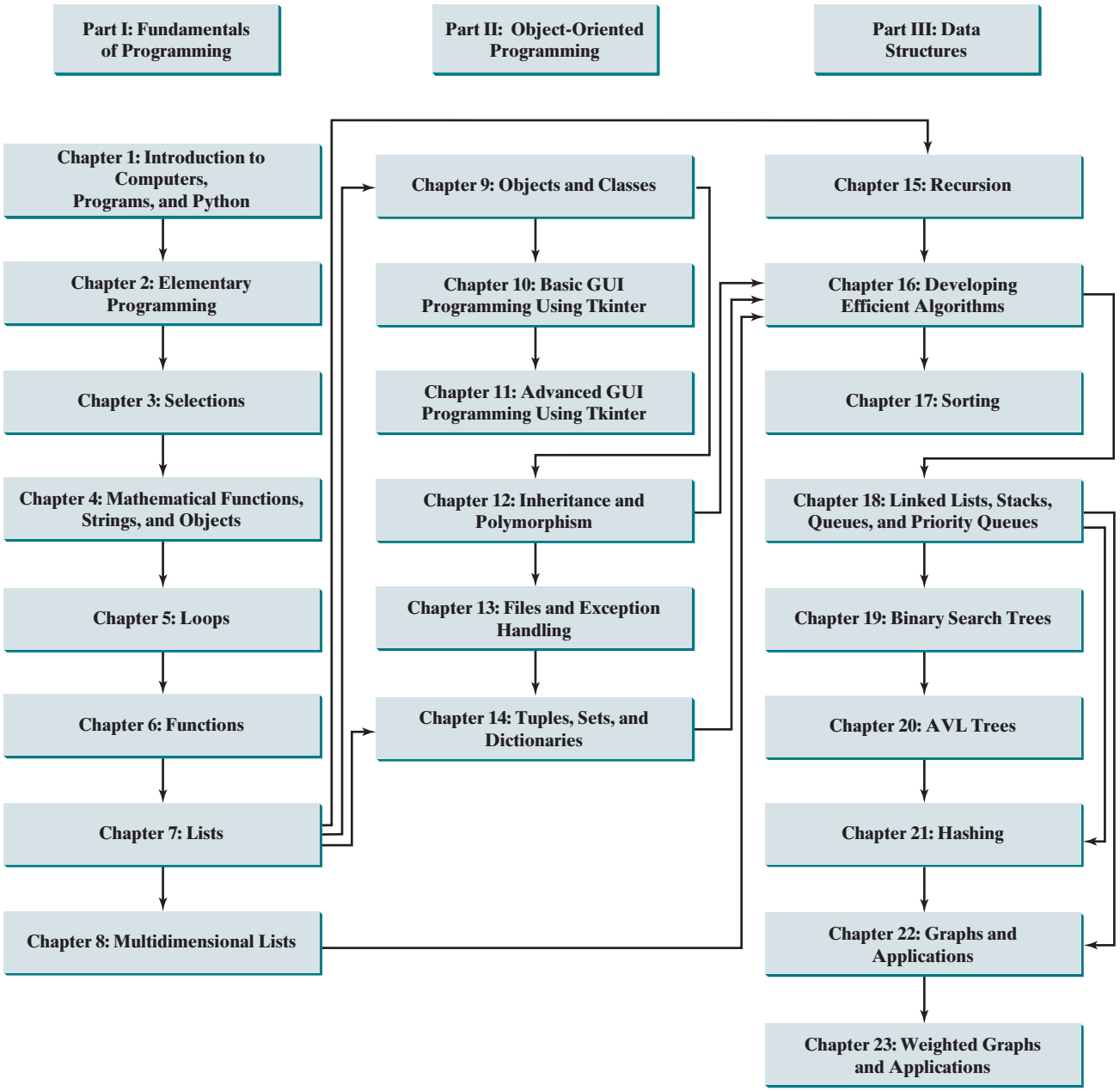
This new edition is completely revised in every detail to enhance clarity, presentation, content, examples, and exercises. The major improvements are as follows:

- **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 statistic functions are covered in **Chapter 7**, to enable students to write simple code for common statistic 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. (This is the hallmark for every new edition of the Liang book.)
- 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 the 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 understand that these are for instructors only. Such requests will not be answered.

Video Notes

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 www.pearsonglobaleditions.com.

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

This page is intentionally left blank

BRIEF CONTENTS

CONTENTS

VideoNotes

Locations of VideoNotes

www.pearsonglobaleditions.com

CHAPTER

1

INTRODUCTION TO COMPUTERS, PROGRAMS, AND PYTHON

Objectives

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

1.1 Introduction

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

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

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

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

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

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

1.2 What Is a Computer?

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

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

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

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

A computer's components are interconnected by a subsystem called a *bus*. You can think of a bus as a sort of system of roads running among the computer's components; data and power

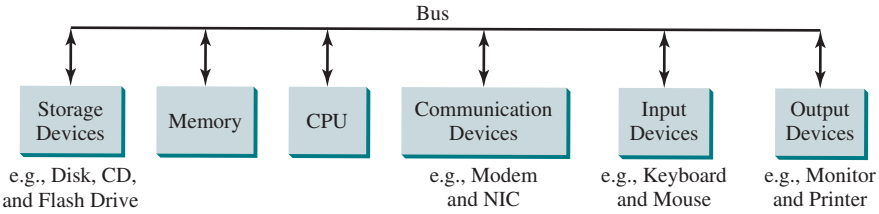


FIGURE 1.1 A computer consists of a CPU, memory, storage devices, input devices, output devices, and communication devices.

travel along the bus from one part of the computer to another. In personal computers, the bus is built into the computer's *motherboard*, which is a circuit case that connects all of the parts of a computer together.

1.2.1 Central Processing Unit

The *central processing unit (CPU)* is the computer's brain. It retrieves instructions from memory and executes them. The CPU usually has two components: a *control unit* and an *arithmetic/logic unit*. The control unit controls and coordinates the actions of the other components. The arithmetic/logic unit performs numeric operations (addition, subtraction, multiplication, and division) and logical operations (comparisons).

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

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

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

1.2.2 Bits and Bytes

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

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

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

Data of various kinds, such as numbers and characters, are encoded as a series of bytes. As a programmer, you don't need to worry about the encoding and decoding of data, which the computer system performs automatically, based on the encoding scheme. An *encoding scheme*

is a set of rules that govern how a computer translates characters and numbers into data the computer can actually work with. Most schemes translate each character into a predetermined string of bits. In the popular ASCII encoding scheme, for example, the character C is represented as **01000011** in one byte.

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

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

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

1.2.3 Memory

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

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

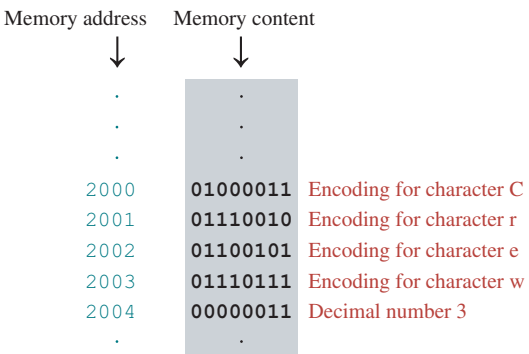


FIGURE 1.2 Memory stores data and program instructions in uniquely addressed memory locations.

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

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

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

1.2.4 Storage Devices

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

There are three main types of storage devices:

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

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

Hard Disk Drives and Solid-State Drives

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

CDs and DVDs

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

DVD stands for digital versatile disc or digital video disc. DVDs and CDs look alike, and you can use either to store data. A DVD can hold more information than a CD; a standard DVD's storage capacity is 4.7 GB. Like CDs, there are two types of DVDs: DVD-R (Recordable) and DVD-RW (ReWritable).

USB Flash Drives

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

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

Cloud Storage

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

1.2.5 Input and Output Devices

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

The Keyboard

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

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

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

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

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

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

The Mouse

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

The Monitor

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

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

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

Touchscreens

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

1.2.6 Communication Devices

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

- A *dial-up modem* uses a phone line to dial a phone number to connect to the Internet and can transfer data at a speed up to 56,000 bps (bits per second).
- A *digital subscriber line (DSL)* connection also uses a standard phone line, but it can transfer data 20 times faster than a standard dial-up modem.
- A *cable modem* uses the cable TV line maintained by the cable company and is generally faster than DSL.

- A *network interface card (NIC)* is a device that connects a computer to a *local area network (LAN)*. LANs are commonly used to connect computers within a limited area such as a school, a home, and an office. A high-speed NIC called *1000BaseT* can transfer data at 1,000 million bits per second (mbps).
- Wireless networking is now extremely popular in homes, businesses, and schools. Every laptop computer sold today is equipped with a wireless adapter that enables the computer to connect to a local area network and the Internet.

1.3 Programming Languages

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

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

1.3.1 Machine Language

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

```
1101101010011010
```

1.3.2 Assembly Language

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

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

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

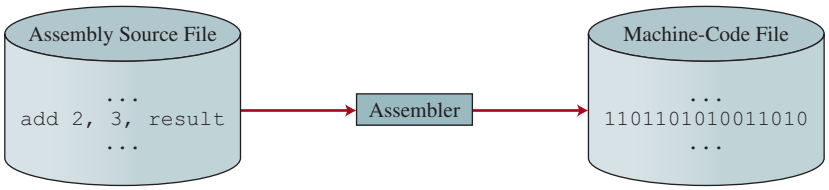


FIGURE 1.3 An assembler translates assembly-language instructions into machine code.

Writing code in assembly language is easier than in machine language. However, it is still tedious to write code in assembly language. An instruction in assembly language essentially corresponds to an instruction in machine code. Writing in assembly requires

that you know how the CPU works. Assembly language is referred to as a *low-level language* because assembly language is close in nature to machine language and is machine dependent.

1.3.3 High-Level Language

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

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

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

TABLE 1.1 Popular High-Level Programming Languages

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

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

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

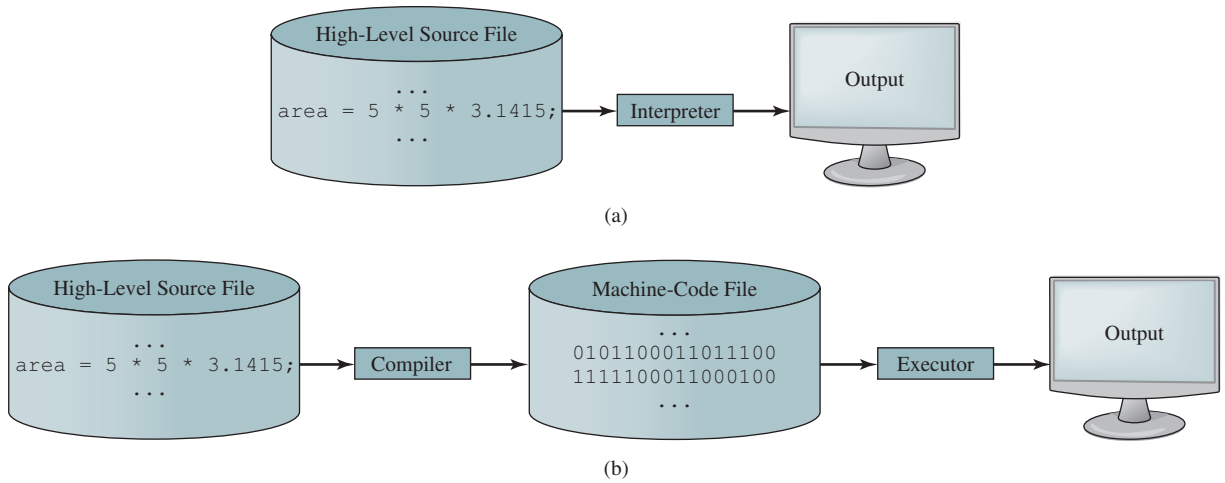


FIGURE 1.4 (a) An interpreter translates and executes a program one statement at a time. (b) A compiler translates the entire source program into a machine-language file for execution.

1.4 Operating Systems

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

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

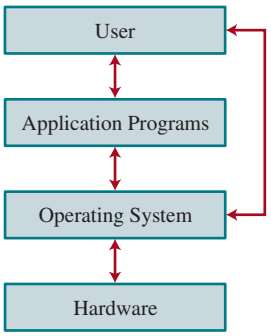


FIGURE 1.5 Users and applications access the computer's hardware via the operating system.

The major tasks of an operating system are as follows:

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

1.4.1 Controlling and Monitoring System Activities

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the monitor, keeping track of files and folders on storage devices, and controlling peripheral devices, such as disk drives and printers. An operating system must also ensure

that different programs and users working at the same time do not interfere with each other. In addition, the OS is responsible for security, ensuring that unauthorized users and programs are not allowed to access the system.

1.4.2 Allocating and Assigning System Resources

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

1.4.3 Scheduling Operations

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

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

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

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

1.5 The History of Python

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

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

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

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

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

Python is now being developed and maintained by a large team of volunteers and is available for free from the Python Software Foundation. Two versions of Python are currently coexistent: Python 2 and Python 3. Python 3 is a newer version, but it is not backward-compatible with Python 2. This means that if you write a program using the Python 2 syntax, it may not work in Python 3.

Python provides a tool that automatically converts code written in Python 2 into Python 3. Python 2 will eventually be replaced by Python 3. This book teaches programming using Python 3.

1.6 Getting Started with Python

A Python program is executed from the Python interpreter.

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

VideoNote
Start with Python

Note

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

1.6.1 Launching Python

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

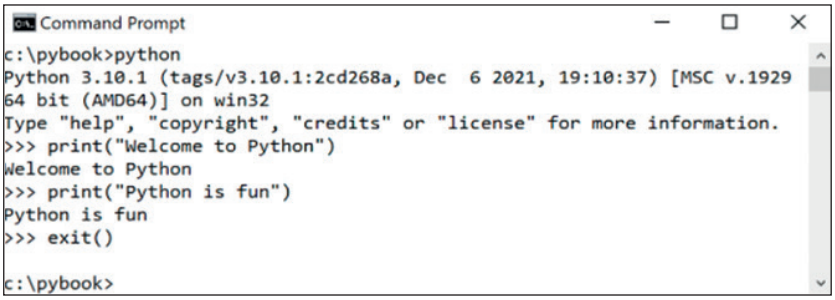


FIGURE 1.6 You can launch Python from Windows command prompt. (Courtesy of Microsoft Corporation.)

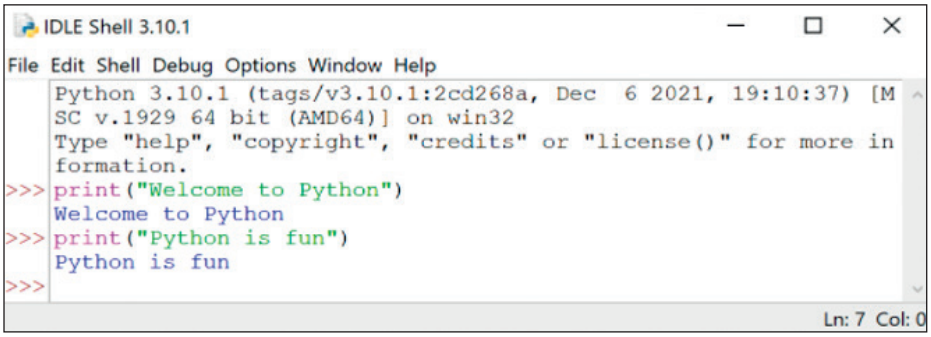


FIGURE 1.7 You can use Python from IDLE. (Courtesy of Microsoft Corporation.)

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

Note

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

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

Note

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

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

Note

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

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

Note

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

1.6.2 Creating Python Source Code Files

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

```
python filename.py
```

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

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

Note

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

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

LISTING 1.1 Welcome.py

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

```
Welcome to Python
Python is fun
```

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

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

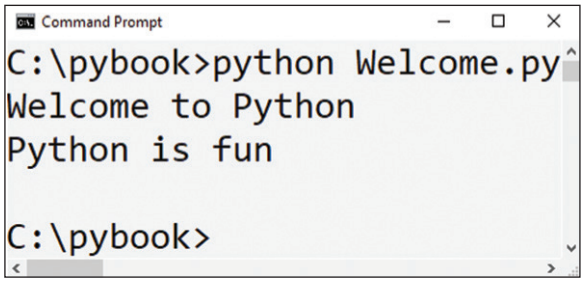


FIGURE 1.8 You can run a Python script file from a command window. (Courtesy of Microsoft Corporation.)

In Listing 1.1, line 1 is a *comment* that documents what the program is and how it is constructed. Comments help programmers communicate and understand a program. They are not programming statements and thus are ignored by the interpreter. In Python, comments start with a pound sign (#) and extend to the end of the physical line.

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

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

Indentation Error

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

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

Punctuation Error

(b) No punctuations at the end of a statement

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

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

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

TABLE 1.2 Special Characters

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

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

LISTING 1.2 WelcomeWithThreeMessages.py

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

```
Welcome to Python
Python is fun
Problem Driven
```

1.6.3 Using Python to Perform Mathematical Computations

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

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

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

LISTING 1.3 ComputeExpression.py

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

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

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

1.7 Programming Style and Documentation

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

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

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

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

1.7.1 Appropriate Comments and Comment Styles

Include a summary comment at the beginning of the program to explain what the program does, its key features, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is difficult to read. It is important to make comments concise so that they do not crowd the program or make it difficult to read.

1.7.2 Proper Spacing

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

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

```
print(3 + 4 * 4)
```

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

```
print(3+4*4)
```

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

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

1.8 Programming Errors

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

1.8.1 Syntax Errors

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

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

```
Command Prompt - python
Python 3.9.11 (tags/v3.9.11:2de452f, Mar 16 20
Type "help", "copyright", "credits" or "licens
>>> print("Programming is fun)
  File "<stdin>", line 1
    print("Programming is fun)
                              ^
SyntaxError: EOL while scanning string literal
>>>
```

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

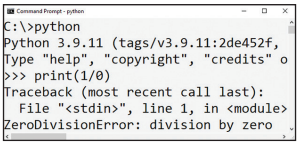
Tip

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

1.8.2 Runtime Errors

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

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



1.8.3 Logic Errors

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

LISTING 1.4 ShowLogicErrors.py

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

```
Fahrenheit 35 is Celsius degree
–12.555555555555554
```

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

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

1.9 Getting Started with Graphics Programming

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

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

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

1.9.1 Drawing and Adding Color to a Figure


VideoNote
Start Turtle Graphics

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

1. Launch Python from IDLE or from the command window by typing `python` at the command prompt.

2. At the Python statement prompt `>>>`, type the following statement to import the `turtle` module. This statement imports all functions defined in the `turtle` module and make them available for you to use.

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

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

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

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

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

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

   Your window should look like the one shown in Figure 1.9b.

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

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

   Your window should now look like the one shown in Figure 1.9c.

   To draw the rest of Figure 1.9, continue with these steps.

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

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

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

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

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

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

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

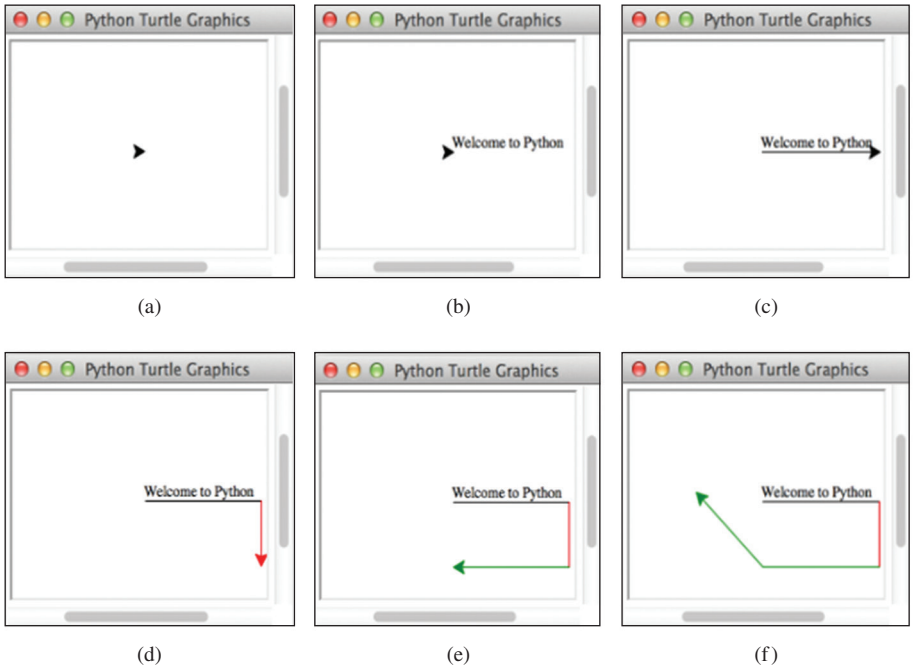


FIGURE 1.9 (a) Turtle location is displayed. (b) A string is displayed. (c) Turtle is moved forward 100 pixels. (d) Turtle is turned right. (e) Turtle is turned left. (f) Turtle is turned right 45 degrees. (Screenshots courtesy of Apple.)

1.9.2 Moving the Pen to Any Location

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

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

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

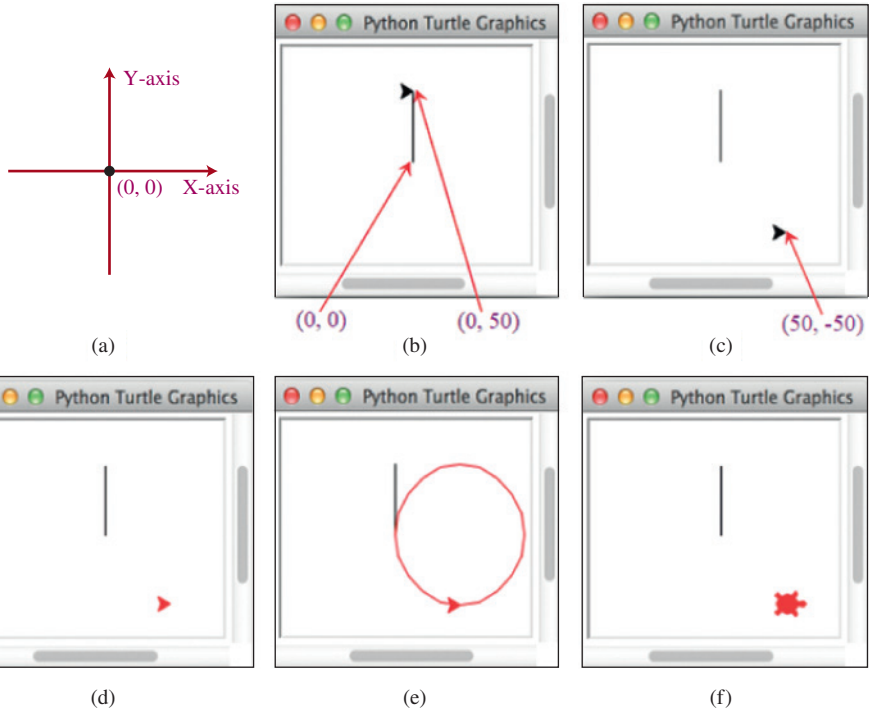


FIGURE 1.10 (a) The center of the Turtle Graphics window is at the coordinates (0, 0). (b) Move to (0, 50). (c) Move the pen to (50, −50). (d) Set color to red. (e) Draw a circle using the circle command. (f) Cursor is changed. (Screenshots courtesy of Apple.)

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

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

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

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

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

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

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

1.9.3 Drawing the Olympic Rings Logo

Listing 1.5 shows a program for drawing the Olympics rings logo, as shown in Figure 1.11.

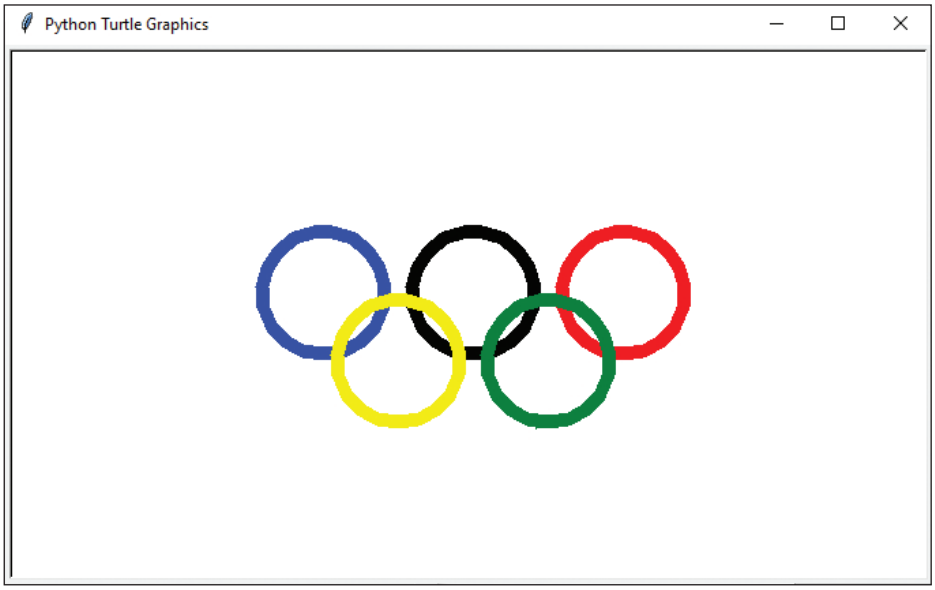
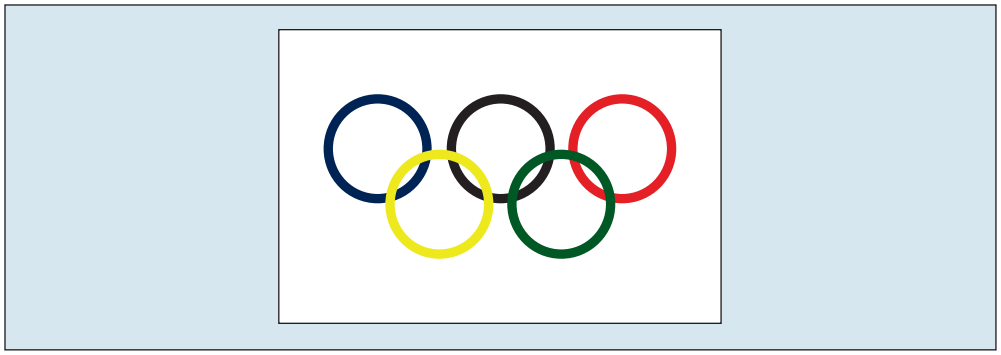


FIGURE 1.11 The program draws the Olympics rings logo. (Screenshots courtesy of Apple.)

LISTING 1.5 OlympicSymbol.py

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



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

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

Key Terms

`.py file`
assembler
assembly language
bit
bus
byte
cable modem
calling a function
central processing unit (CPU)
comment
compiler
console
dot pitch
DSL (digital subscriber line)
encoding scheme
function
hardware
high-level language
indentation
interactive mode
interpreter
invoking a function
IDLE (Interactive DeveLopment Environment)
logic error
low-level language
machine language
memory
module
motherboard
network interface card (NIC)
operating system (OS)
pixel
program
programming
programming language
runtime error
screen resolution
script file
script mode
software
source code
source file
source program
storage devices
syntax error
syntax rules

Chapter Summary

1. A computer is an electronic device that stores and processes data.
2. A computer includes both *hardware* and *software*.
3. Hardware is the physical aspect of the computer that can be touched.
4. Computer *programs*, known as *software*, are the invisible instructions that control the hardware and make it perform tasks.
5. *Computer programming* is the writing of instructions (i.e., code) for computers to perform.
6. The *central processing unit (CPU)* is a computer's brain. It retrieves instructions from memory and executes them.
7. Computers use zeros and ones because digital devices have two stable electrical states, off and on, referred to by convention as zero and one.
8. A *bit* is a binary digit 0 or 1.
9. A *byte* is a sequence of 8 bits.
10. A kilobyte is about 1,000 bytes, a megabyte about 1 million bytes, a gigabyte about 1 billion bytes, and a terabyte about 1,000 gigabytes.
11. *Memory* stores data and program instructions for the CPU to execute.
12. A *memory unit* is an ordered sequence of bytes.
13. Memory is volatile because information that hasn't been saved is lost when the power is turned off.
14. Programs and data are permanently stored on *storage devices* and are moved to memory when the computer actually uses them.
15. The *machine language* is a set of primitive instructions built into every computer.
16. *Assembly language* is a low-level programming language in which a mnemonic is used to represent each machine-language instruction.
17. *High-level languages* are English-like and easy to learn and program.
18. A program written in a high-level language is called *source code.*
19. A *compiler* is a software program that translates the *source program* into a *machine-language* program.
20. The *operating system (OS)* is a program that manages and controls a computer's activities.
21. You can run Python on Windows, UNIX, and Mac.
22. Python is *interpreted*, meaning that Python translates each statement and processes it one at a time.

23. You can enter Python statements interactively from the Python statement prompt >>> or store all your code in one file and execute it together.

24. To run a Python source file from the Windows command prompt, use the **python *filename*.py** command.

25. In Python, *comments* are preceded by a pound sign (#) on a line and extends to the end of the line.

26. Python source programs are case sensitive.

27. Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors. *Syntax* and *runtime errors* cause a program to terminate abnormally. *Logic errors* occur when a program does not perform the way it was intended to.

Programming Exercises

Note

Solutions to even-numbered exercises in this book are on the Companion Website. Solutions to all exercises are on the Instructor Resource Website. The level of difficulty is rated easy (no star), moderate (*), hard (**), or challenging (***).

Section 1.6

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

```
Welcome to Python
Welcome to Computer Science
Programming is fun
```

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

```
Welcome to Python
Welcome to Python
Welcome to Python
Welcome to Python
Welcome to Python
```

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

```
####  #   # ##### #   #  ###  #   #
#   #  # #    #   #   # #   # ##  #
####    #     #   ##### #   # # # #
#       #     #   #   # #   # #  ##
#       #     #   #   #  ###  #   #
```

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

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

1.5 (*Compute expressions*) Write a program that displays the result of $\frac{14.8 - 2.4 \times 3 + 7.6}{5.2 + 3.3}$

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

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

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

Write a program that displays the result of $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11}\right)$ and $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15}\right)$

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

$$area = length\ of\ a\ side \times height$$
$$perimeter = 4 \times length\ of\ a\ side$$

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

$$area = width \times height$$

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

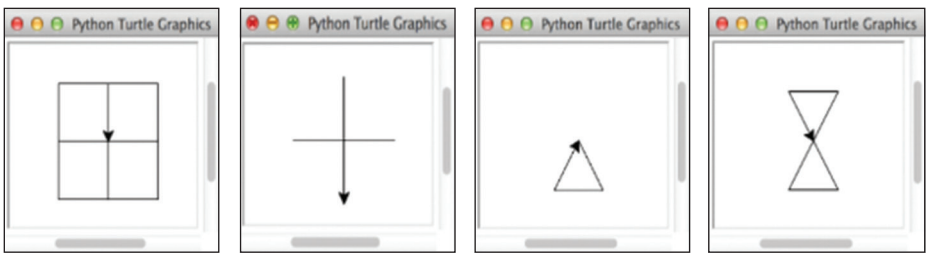
***1.11** (*Population projection*) The US Census Bureau projects population based on the following assumptions:

- One birth every 7 seconds
- One death every 13 seconds
- One new immigrant every 45 seconds

Write a program to display the population for each of the next five years. Assume that the current population is 312032486 and one year has 365 days.

Section 1.9

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



(a) (b) (c) (d)

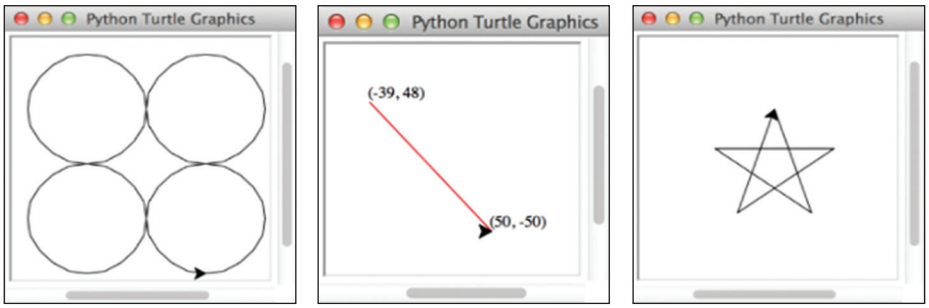
FIGURE 1.12 Four squares are drawn in (a), a cross is drawn in (b), a triangle is drawn in (c), and two triangles are drawn in (d). (Screenshots courtesy of Apple.)

1.13 (*Turtle: draw a cross*) Write a program that draws a cross as shown in Figure 1.12b.

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

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

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



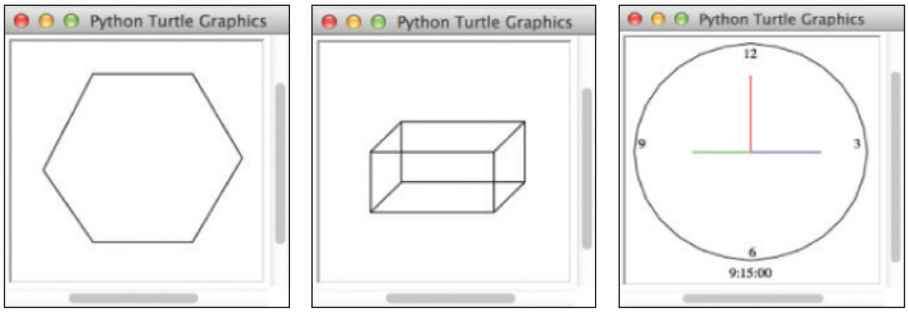
(a) (b) (c)

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

1.17 (*Turtle: draw a line*) Write a program that draws a red line connecting two points (`-39, 48`) and (`50, -50`) and displays the coordinates of the two points, as shown in Figure 1.13b.

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

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



(a) (b) (c)

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

1.20 (*Turtle: display a rectanguloid*) Write a program that displays a rectanguloid, as shown in Figure 1.14b.

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

Note

Additional programming exercises with solutions are provided to instructors in the Instructor Resources.

CHAPTER

2

ELEMENTARY PROGRAMMING

Objectives

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

2.1 Introduction

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

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

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

2.2 Writing a Simple Program

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

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

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

1. Get the circle's radius from the user.
2. Compute the area applying the following formula:

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

3. Display the result.

Tip

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

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

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

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

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

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

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

The complete program is shown in Listing 2.1.

LISTING 2.1 ComputeArea.py

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

```
The area for the circle of radius 20 is 1256.636
```

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

```
radius = 20
```

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

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

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

line#	radius	area
2	20	
5		1256.636

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

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

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

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

2.3 Reading Input from the Console

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

In Listing 2.1, a radius is set in the source code. To use a different radius, you have to modify the source code. You can use the **input** function to ask the user to input a value for the radius.

The following statement prompts the user to enter a value, and then it assigns the value to the variable:

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

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

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

LISTING 2.2 ComputeAreaWithConsoleInput.py

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

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

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

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

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

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

LISTING 2.3 ComputeAverage.py

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

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

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

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

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

Note

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

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

is equivalent to

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

Note

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

2.4 Identifiers

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

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

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

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

Note

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

Tip

Descriptive identifiers make programs easy to read. Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, `numberOfStudents` is better than `numStuds`, `numOfStuds`, or `numOfStudents`. We use descriptive names for complete programs in the text. However, we will occasionally use variables

names such as `i`, `j`, `k`, `x`, and `y` in the code snippets for brevity. These names also provide a generic tone to the code snippets.

Tip

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

2.5 Variables, Assignment Statements, and Expressions

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

VideoNote

Assignment Statement

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

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

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

```
variable = expression
```

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

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

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

```
x = x + 1
```

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

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

```
1 = x # Wrong
```

Note

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

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

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

which is equivalent to

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

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

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

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

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

Caution

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

```
interestRate = 0.05
interest = interestrate * 45
```

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

2.6 Simultaneous Assignments

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

Python supports *simultaneous assignment* in syntax like this:

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

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

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

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

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

2.7 Named Constants

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

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

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

There are three benefits of using constants:

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

2.8 Numeric Data Types and Operators

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

Perform Computation

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

Note

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

```
print(0B1111) # Displays 15
print(0O7777) # Displays 4095
print(0XFFFF) # Displays 65535
```

Hexadecimal numbers, binary numbers, and octal numbers are introduced in Appendix C.

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

TABLE 2.1 Numeric Operators

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

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

Note

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

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

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

2.8.1 The /, //, and ** Operators

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

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

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

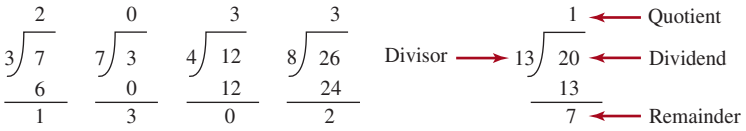
```
>>> 5 // 2
2
>>> 2 // 4
0
>>>
```

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

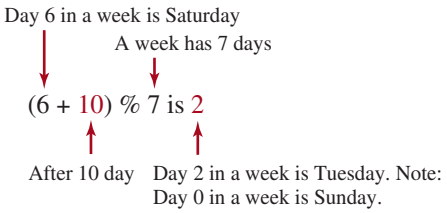
```
>>> 2.3 ** 3.5
18.45216910555504
>>> (-2.5) ** 2
6.25
>>>
```

2.8.2 The % Operator

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



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



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

LISTING 2.4 DisplayTime.py

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

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

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

2.8.3 Scientific Notation

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

Note

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

Caution

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

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

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

2.9 Case Study: Minimum Number of Changes

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

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

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

Here are the steps in developing the program:

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

The complete program is shown in Listing 2.5.

LISTING 2.5 ComputeChange.py

```
1  # Receive the amount
2  amount = float(input("Enter an amount, e.g., 11.56: "))
3
4  # Convert the amount to cents
5  remainingAmount = int(amount * 100)
6
```

```
 7  # Find the number of one dollars
 8  numberOfOneDollars = remainingAmount // 100
 9  remainingAmount = remainingAmount % 100
10
11  # Find the number of quarters in the remaining amount
12  numberOfQuarters = remainingAmount // 25
13  remainingAmount = remainingAmount % 25
14
15  # Find the number of dimes in the remaining amount
16  numberOfDimes = remainingAmount // 10
17  remainingAmount = remainingAmount % 10
18
19  # Find the number of nickels in the remaining amount
20  numberOfNickels = remainingAmount // 5
21  remainingAmount = remainingAmount % 5
22
23  # Find the number of pennies in the remaining amount
24  numberOfPennies = remainingAmount
25
26  # Display results
27  print("Your amount", amount, "consists of"),
28  print("  ", numberOfOneDollars, "dollars"),
29  print("  ", numberOfQuarters, "quarters"),
30  print("  ", numberOfDimes,  "dimes"),
31  print("  ", numberOfNickels, "nickels"),
32  print("  ", numberOfPennies, "pennies")
```

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

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

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

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

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

Caution

One serious problem with this example is the possible loss of precision when converting a float amount to the integer `remainingAmount`. This could lead to an inaccurate result. If you try to enter the amount `10.03`, `10.03 * 100` might be `1002.9999999999999`. You will find that the program displays `10` dollars and `2` pennies. To fix the problem, enter the amount as an integer value representing cents (see Programming Exercise 3.7).

2.10 Evaluating Expressions and Operator Precedence

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

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

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

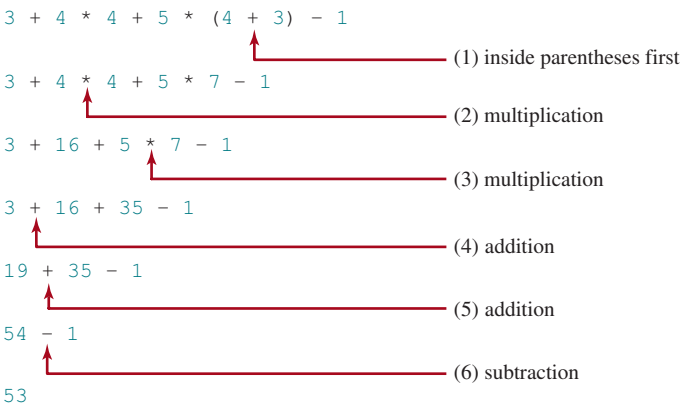
can be translated into a Python expression as:

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

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

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

Here is an example of how an expression is evaluated:



2.11 Augmented Assignment Operators

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

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

```
count = count + 1
```

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

```
count += 1
```

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

TABLE 2.2 Augmented Assignment Operators

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

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

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

is same as

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

Caution

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

2.12 Type Conversions and Rounding

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

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

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

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

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

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

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

We will discuss the **round** function more in Chapter 4.

Note

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

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

Listing 2.6 shows a program that displays the sales tax with two digits after the decimal point.

LISTING 2.6 SalesTax.py

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

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

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

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

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

2.13 Case Study: Displaying the Current Time

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

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

The **time()** function in the **time** module returns the current time in seconds with millisecond precision elapsed since the time **00:00:00** on January 1, 1970 GMT, as shown in Figure 2.1. This time is known as the *UNIX epoch*. The epoch is the point when time starts. **1970** was the

year when the UNIX operating system was formally introduced. For example, `time.time()` returns `1285543663.205`, which means `1285543663` seconds and `205` milliseconds.

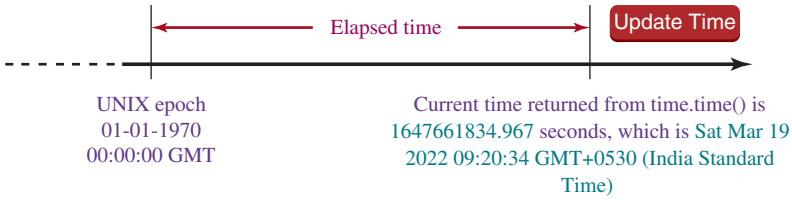


FIGURE 2.1 The `time.time()` function returns the seconds with millisecond precision since the UNIX epoch.

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

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

Listing 2.7 gives the complete program.

LISTING 2.7 ShowCurrentTime.py

```
1  import time
2
3  currentTime = time.time() # Get current time
4
5  # Obtain the total seconds since midnight, Jan 1, 1970
6  totalSeconds = int(currentTime)
7
8  # Get the current second
9  currentSecond = totalSeconds % 60
10
11 # Obtain the total minutes
12 totalMinutes = totalSeconds // 60
13
14 # Compute the current minute in the hour
15 currentMinute = totalMinutes % 60
16
17 # Obtain the total hours
18 totalHours = totalMinutes // 60
19
```

```
20  # Compute the current hour
21  currentHour = totalHours % 24
22
23  # Display results
24  print("Current time is", currentHour, ":",
25      currentMinute, ":", currentSecond, "GMT")
```

```
Current time is 20 : 44 : 49 GMT
```

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

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

2.14 Software Development Process

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

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

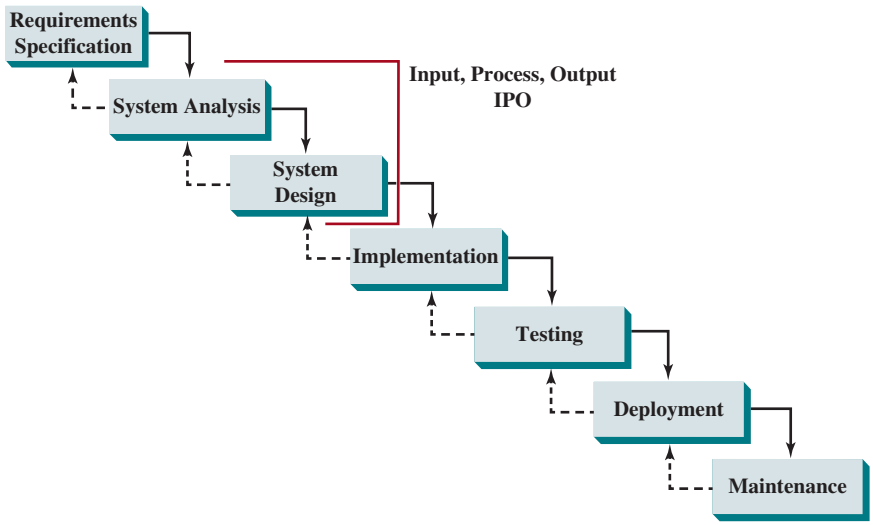


FIGURE 2.2 At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

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

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

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

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

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

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

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

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

Stage 1: Requirements Specification

The program must satisfy the following requirements:

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

Stage 2: System Analysis

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

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

$$totalPayment = totalPayment \times numberOfYears \times 12$$

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

Note

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

Note

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

Stage 3: System Design

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

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

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

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

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

Step 3.5. Display the monthly payment and total payment.

Stage 4: Implementation

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

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

Listing 2.8 gives the complete program.

LISTING 2.8 ComputeLoan.py

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

```
18  # Display results
19  print("The monthly payment is", int(monthlyPayment * 100) / 100)
20  print("The total payment is", int(totalPayment * 100) /100)
```

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

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

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

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

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

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

Stage 5: Testing

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

Tip

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

2.15 Case Study: Computing Distances

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

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

LISTING 2.9 ComputeDistance.py

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

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

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

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

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

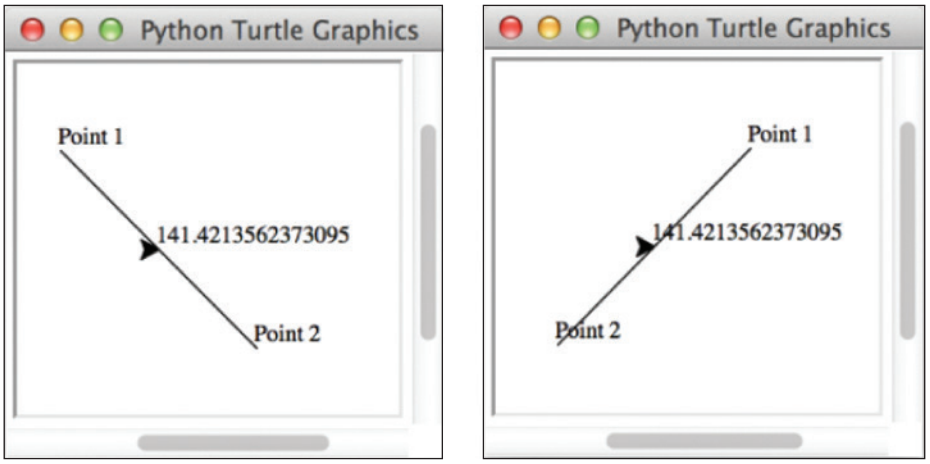
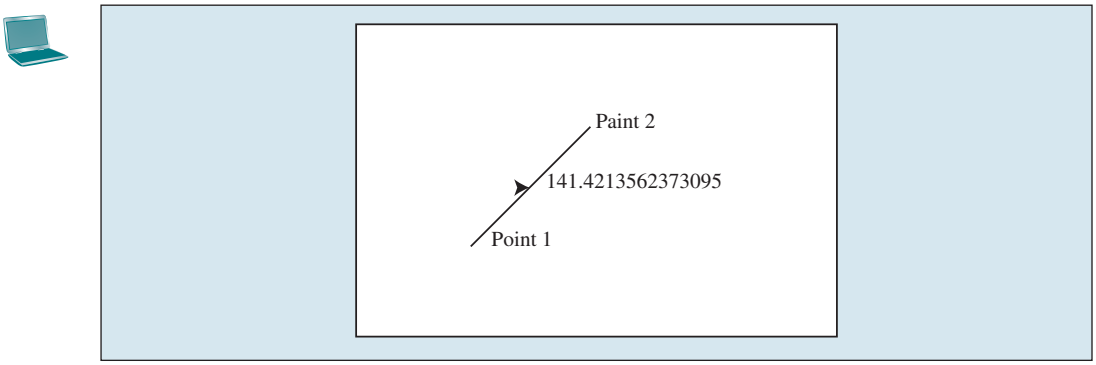


FIGURE 2.3 The program displays a line and its length. (Screenshots courtesy of Apple.)

LISTING 2.10 ComputeDistanceGraphics.py

```
1  import turtle
2
3  # Prompt the user for inputing two points
4  x1 = float(input("Enter x-coordinate for Point 1: "))
5  y1 = float(input("Enter y-coordinate for Point 1: "))
6  x2 = float(input("Enter x-coordinate for Point 2: "))
7  y2 = float(input("Enter y-coordinate for Point 2: "))
8
9  # Compute the distance
10 distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
11
12 # Display two points and the connecting line
13 turtle.penup()
14 turtle.goto(x1, y1) # Move to (x1, y1)
15 turtle.pendown()
16 turtle.write("Point 1", font=("Times", 12))
```

```
17  turtle.goto(x2, y2) # draw a line to (x2, y2)
18  turtle.write("Point 2", font=("Times", 12))
19
20  # Move to the center point of the line
21  turtle.penup()
22  turtle.goto((x1 + x2) / 2, (y1 + y2) / 2)
23  turtle.write(distance, font=("Times", 12))
24
25  turtle.done()
```



The program prompts the user to enter the values for two points, (`x1, y1`) and (`x2, y2`), and computes their distance (lines 4–10). It then moves to (`x1, y1`) (line 14), displays the text Point 1 (line 16), draws a line from (`x1, y1`) to (`x2, y2`) (line 17), and displays the text Point 2 (line 18). Finally, it moves to the center of the line (line 22) and displays the distance (line 23).

Key Terms

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

Chapter Summary

1. You can get input using the `input` function and convert a string into a numerical value using the `int` function or the `float` function.

2. *Identifiers* are the names used for elements in a program.

3. An identifier is a sequence of characters of any length that consists of letters, digits, underscores (_), and asterisk signs (*). An identifier must start with a letter or an underscore; it cannot start with a digit. An identifier cannot be a keyword.

4. *Variables* are used to store data in a program.

5. The equal sign (=) is used as the *assignment operator*.

6. A variable must be assigned a value before it can be used.

7. There are two types of numeric data in Python: integers and real numbers. Integer types (*int* for short) are for whole numbers, and real types (also called *float*) are for numbers with a decimal point.

8. Python provides *assignment operators* that perform numeric operations: + (addition), – (subtraction), * (multiplication), / (division), // (integer division), % (remainder), and ** (exponent).

9. The numeric operators in a Python expression are applied the same way as in an arithmetic expression.

10. Python provides augmented assignment operators: += (addition assignment), –= (subtraction assignment), *= (multiplication assignment), /= (float division assignment), //= (integer division assignment), and %= (remainder assignment). These operators combine the +, –, *, /, //, and % operators and the assignment operator into augmented operators.

11. When evaluating an expression with values of an int type and a float type, Python automatically converts the int value to a float type value.

12. You can convert a float to an int using the `int(value)` function.

13. *System analysis* seeks to analyze the data flow and to identify the system's input and output.

14. *System design* is the stage when programmers develop a process for obtaining the output from the input.

15. The essence of system analysis and design is input, process, and output. This is called *IPO*.

Programming Exercises

Pedagogical Note

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

Debugging Tip

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

Sections 2.2– 2.10

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

```
1 km / 1 h = 1000 m / 3600 s
```

```
Enter speed in km/h: 81
81.0 km/h is equal to 22.5 m/s
```

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

```
area = 6 * side * side
volume = side * side * side
```

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

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

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

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

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

***2.5** (*Financial application: calculate tips*) Write a program that reads the subtotal and the gratuity rate and computes the gratuity and total. For example, if the user enters `10` for the subtotal and `15%` for the gratuity rate, the program displays `1.5` as the gratuity and `11.5` as the total.

```
Enter the subtotal: 100.57
Enter the gratuity rate: 15
The gratuity is 15.08 and total is 115.65
```

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

```
Enter an amount as integer, e.g., 1156 for 11 dollars 56
cents: 435
Your amount 435 consists of
    4      dollars
    1      quarters
    1      dimes
    0      nickels
    0      pennies
```

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

```
Enter the number of minutes: 1000000000
1000000000 minutes is approximately 1902 years and 214 days
```

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

Potential Energy = mass * gravity * height

Take gravity = 9.8 m/s^2

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

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

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

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

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

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

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

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

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

```
Enter the initial speed of car in m/s: 0
Enter the final speed of car in m/s: 32
Enter the distance covered by car in meters: 112
The acceleration is 4.571428571428571 m/s^2
```

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

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

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

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

```
a   b      pow(a, b)
1   2      1
2   3      8
3   4      81
4   5      1024
5   6      15625
```

***2.13** (*Find the greatest digit*) Write a program that prompts the user to enter a four-digit integer and displays the greatest digit among the four digits.

```
Enter a four-digit integer: 4573
The largest digit is 7
```

***2.14** (*Geometry: area of a triangle*) Write a program that prompts the user to enter the three points (`x1, y1`), (`x2, y2`), and (`x3, y3`) of a triangle and displays its area. The formula for computing the area of a triangle is

$$s = (side1 + side2 + side3) / 2$$

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

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

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

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

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

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

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

```
Enter v0: 5.5
Enter v1: 50.9
Enter t: 4.5
The average acceleration is 10.08888888888889
```

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

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

Sections 2.11–2.12

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

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

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

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

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

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

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

```
interest = balance × (annualInterestRate / 1200)
```

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

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

**2.21 (*Financial application: compound value*) Suppose you save $100 each month into a savings account with an annual interest rate of 5%. Therefore, 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 a monthly saving amount and displays the account value after the sixth month. Keep two digits after the decimal point.

```
Enter monthly saving amount: 100
After the sixth month, the account value is 608.81
```

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

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

*2.23 (*Slope of a line*) Write a program that prompts the user to enter the coordinates of two points (x1, y1) and (x2, y2) and displays the slope of the line connects the two points. The formula of the slope is $(y_2 - y_1) / (x_2 - x_1)$

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

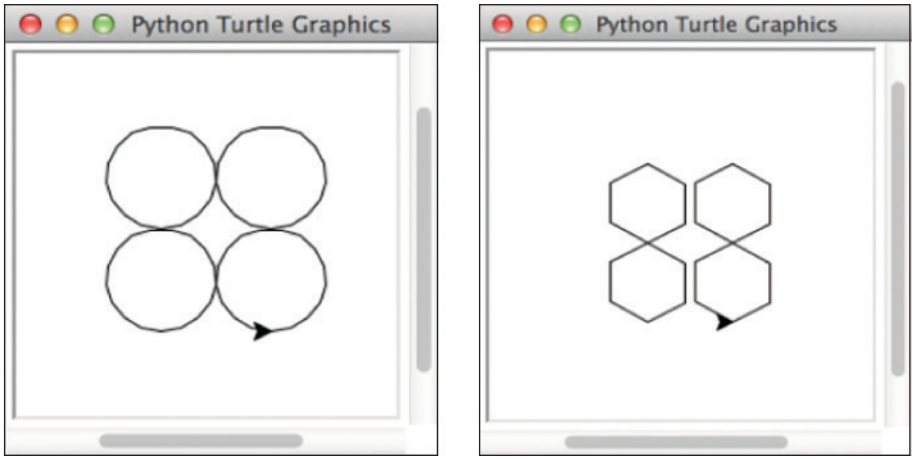
*2.24 (*Cost of driving*) Write a program that prompts the user to enter the distance to drive, the fuel efficiency of the car in miles per gallon, and the price per gallon, and displays the cost of the trip.

```
Enter the driving distance: 900.5
Enter miles per gallon: 25.5
Enter prices per gallon: 3.55
The cost of driving is $125.36372549019607
```

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

```
Enter an integra between 0 and 1000: 999
The sum of all digits in 999 is 27
```

Section 2.13

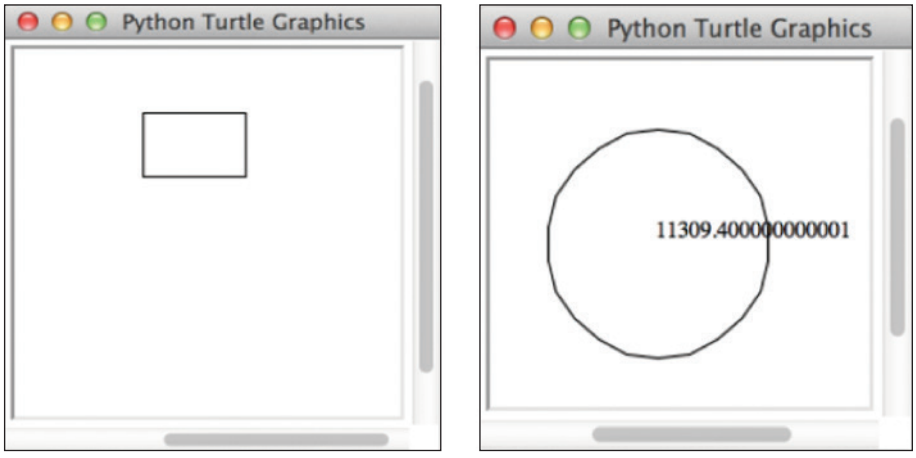


(a) (b)

FIGURE 2.4 Four circles are drawn in (a) and four hexagons are drawn in (b). (Screenshots courtesy of Apple.)

2.26 (*Turtle: draw four circles*) Write a program that prompts the user to enter the radius and draws four circles in the center of the screen, as shown in Figure 2.4a.

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



(a) (b)

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

**2.28 (*Turtle: draw a rectangle*) Write a program that prompts the user to enter the center of a rectangle, its width and height, and then displays the rectangle, as shown in Figure 2.5a.

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

CHAPTER 3

SELECTIONS

Objectives

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

3.1 Introduction

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

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

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

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

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

3.2 Boolean Types, Values, and Expressions

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

Boolean Expressions

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

TABLE 3.1 Relational Operators

Python Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	less than	`radius < 0`	`False`
`<=`	≤	less than or equal to	`radius <= 0`	`False`
`>`	>	greater than	`radius > 0`	`True`
`>=`	≥	greater than or equal to	`radius >= 0`	`True`
`==`	=	equal to	`radius == 0`	`False`
`!=`	≠	not equal to	`radius != 0`	`True`

Caution

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

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

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

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

```
lightsOn = True
```

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

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

For example,

```
print(int(True))
```

displays `1` and

```
print(int(False))
```

displays `0`.

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

For example,

```
print(bool(0))
```

displays `False` and

```
print(bool(4))
```

displays `True`.

3.3 Generating Random Numbers

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

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

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

The program may be set up to work as follows:

Step 1: Generate two single-digit integers for `number1` (e.g., `4`) and `number2` (e.g., `5`),
Step 2: Prompt the student to answer, `"What is 4 + 5?"`
Step 3: Check whether the student's answer is correct.

LISTING 3.1 AdditionQuiz.py

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

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

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

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

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

Python also provides another function `randrange(a, b)` for generating a random integer between `a` and `b - 1`. The argument `a` may be omitted if it is `0`. So, `randrange(b)` is the same as `randrange(0, b)`. `randrange(a, b)` is also equivalent to `randint(a, b - 1)`. For example, `randrange(0, 10)` and `randint(0, 9)` are the same. Since `randint` is more intuitive, the book generally uses `randint` in the examples.

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

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

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

3.4 `if` Statements

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

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

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

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

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

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

The flowchart in Figure 3.1a illustrates how Python executes the syntax of an `if` statement. A *flowchart* is a diagram that describes an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Process operations