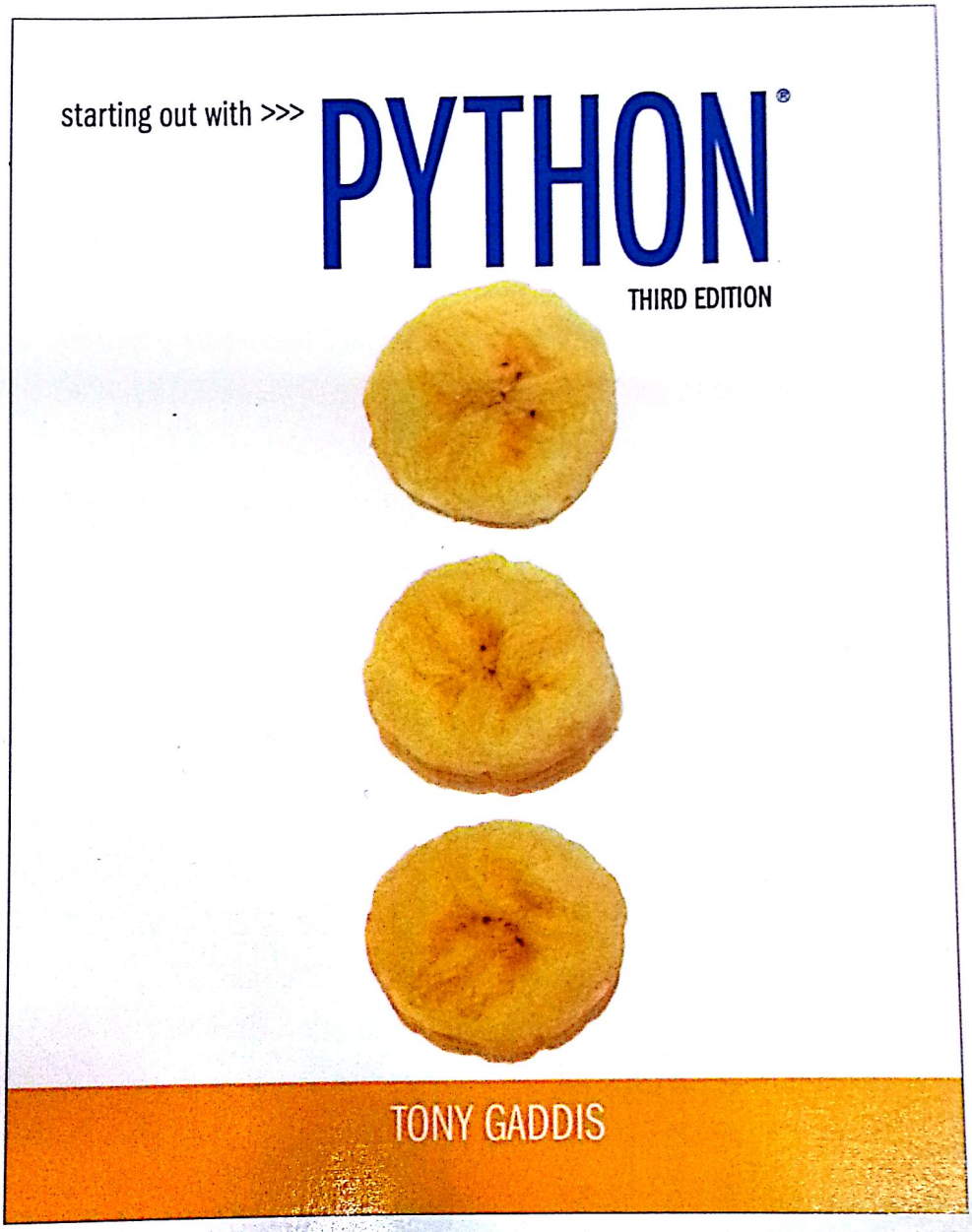
STUDENT VALUE EDITION



Before purchasing this text, please be sure this is the correct book for your course. Once this package has been opened, you may not be able to return it to your bookstore.

ISBN-13: 978-0-13-384849-6
ISBN-10: 0-13-384849-3

9 780133 848496 90000 >

PEARSON pearsonhighered.com

ONLINE ACCESS

Thank you for purchasing a new copy of ***Starting Out with Python,*** Third Edition. Your textbook includes one year of prepaid access to the book's Companion Website. This prepaid subscription provides you with full access to the following student support areas:

- VideoNotes
- Online Appendices
- Source Code

Note that this prepaid subscription does not include access to MyProgrammingLab, which is available at http://www.myprogramminglab.com for purchase.

Use a coin to scratch off the coating and reveal your student access code.
Do not use a knife or other sharp object as it may damage the code.

To access the ***Starting Out with Python,*** Third Edition, Companion Website for the first time, you will need to register online using a computer with an Internet connection and a web browser. The process takes just a couple of minutes and only needs to be completed once.

1. Go to **http://www.pearsonhighered.com/gaddis/**
2. Click on **Companion Website.**
3. Click on the **Register** button.
4. On the registration page, enter your student access code* found beneath the scratch-off panel. Do not type the dashes. You can use lower- or uppercase.
5. Follow the on-screen instructions. If you need help at any time during the online registration process, simply click the **Need Help?** icon.
6. Once your personal Login Name and Password are confirmed, you can begin using the *Starting* ***Out with Python*** Companion Website!

To log in after you have registered:

You only need to register for this Companion Website once. After that, you can log in any time at **http://www.pearsonhighered.com/gaddis/** by providing your Login Name and Password when prompted.

*Important: The access code can only be used once. This subscription is valid for one year upon activation and is not transferable. If this access code has already been revealed, it may no longer be valid. If this is the case, you can purchase a subscription by going to **http://www.pearsonhighered.com/gaddis/** and following the on-screen instructions.

STARTING OUT WITH

PYTHON®

THIRD EDITION

STARTING OUT WITH

PYTHON®

THIRD EDITION

Tony Gaddis
Haywood Community College

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director:	Marcia Horton
Acquisitions Editor:	Matt Goldstein
Program Manager:	Kayla Smith-Tarbox
Director of Marketing:	Christy Lesko
Marketing Manager:	Yezan Alayan
Marketing Assistant:	Jon Bryant
Director of Production:	Erin Gregg
Senior Managing Editor:	Scott Disanno
Senior Production Project Manager:	Marilyn Lloyd
Manufacturing Buyer:	Linda Sager
Art Director:	Jayne Conte
Cover Designer:	Bruce Kenselaar
Manager, Rights and Permissions:	Timothy Nicholls
Text Permissions:	Jenell Forschler
Cover Image:	© Aaron Amat/Fotolia
Media Project Manager:	Renata Butera
Full-Service Project Management:	Jogender Taneja/iEnergizer Aptara®, Inc.
Composition:	iEnergizer Aptara®, Inc.
Printer/Binder:	Courier Kendallville
Cover Printer:	Lehigh Phoenix Color
Text Font:	Sabon LT Std

Credits and acknowledgments borrowed from other sources and reproduced, with permission, appear on the Credits page in the endmatter of this textbook.

Library of Congress Cataloging-in-Publication Data

Gaddis, Tony.

Starting out with python / Tony Gaddis.—Third edition.

pages cm

Includes index.

ISBN-13: 978-0-13-358273-4

ISBN-10: 0-13-358273-6

1. Python (Computer program language) I. Title.

QA76.73.P98G34 2014

005.13'3—dc23

2013046440

V011

10 9 8 7 6 5 4 3 2

PEARSON

ISBN 10: 0-13-358273-6

ISBN 13: 978-0-13-358273-4

Contents in a Glance

Contents

LOCATION OF VIDEONOTES IN THE TEXT

Chapter 1	Using Interactive Mode in IDLE, p. 23 Performing Exercise 2, p. 28
Chapter 2	The `print` Function, p. 36 Reading Input from the Keyboard, p. 49 The Sales Prediction Problem, p. 77
Chapter 3	The `if` Statement, p. 81 The `if-else` Statement, p. 90 The Areas of Rectangles Problem, p. 115
Chapter 4	The `while` Loop, p. 122 The `for` Loop, p. 130 The Bug Collector Problem, p. 161
Chapter 5	Defining and Calling a function, p. 168 Passing Arguments to a Function, p. 181 Writing a Value-Returning Function, p. 206 The Kilometer Converter Problem, p. 229 The Feet to Inches Problem, p. 230
Chapter 6	Using Loops to Process Files, p. 252 File Display, p. 288
Chapter 7	List Slicing, p. 299 The Lottery Number Generator Problem, p. 334
Chapter 8	The Vowels and Consonants problem, p. 367
Chapter 9	Introduction to Dictionaries, p. 369 Introduction to Sets, p. 392 The CapitalQuiz Problem, p. 416
Chapter 10	Classes and Objects, p. 423 The Pet class, p. 476
Chapter 11	The `Person` and `Customer` Classes, p. 505
Chapter 12	The Recursive Multiplication Problem, p. 524
Chapter 13	Creating a Simple GUI application, p. 532 Responding to Button Clicks, p. 538 The Name and Address Problem, p. 562
Appendix B	Introduction to IDLE, p. 569

Preface

Welcome to *Starting Out with Python,* Third Edition. This book uses the Python language to teach programming concepts and problem-solving skills, without assuming any previous programming experience. With easy-to-understand examples, pseudocode, flowcharts, and other tools, the student learns how to design the logic of programs and then implement those programs using Python. This book is ideal for an introductory programming course or a programming logic and design course using Python as the language.

As with all the books in the *Starting Out With* series, the hallmark of this text is its clear, friendly, and easy-to-understand writing. In addition, it is rich in example programs that are concise and practical. The programs in this book include short examples that highlight specific programming topics, as well as more involved examples that focus on problem solving. Each chapter provides one or more case studies that provide step-by-step analysis of a specific problem and shows the student how to solve it.

Control Structures First, Then Classes

Python is a fully object-oriented programming language, but students do not have to understand object-oriented concepts to start programming in Python. This text first introduces the student to the fundamentals of data storage, input and output, control structures, functions, sequences and lists, file I/O, and objects that are created from standard library classes. Then the student learns to write classes, explores the topics of inheritance and polymorphism, and learns to write recursive functions. Finally, the student learns to develop simple event-driven GUI applications.

Changes in the Third Edition

This book's clear writing style remains the same as in the previous edition. However, many improvements have been made, which are summarized here:

- In the previous editions, Chapter 3 introduced simple, void functions, and then Chapter 6 covered value-returning functions. In this edition, the two chapters have been combined. Chapter 5: Functions covers simple void functions, value-returning functions, and modules.
- Several new programming problems have been added.

- Numerous examples of using the Python shell to test relational operators have been added to Chapter 3, Decision Structures.
- The book's programs have been tested with Python 3.3.2, the most recent version of Python at the time this edition was written.

Brief Overview of Each Chapter

Chapter 1: Introduction to Computers and Programming

This chapter begins by giving a very concrete and easy-to-understand explanation of how computers work, how data is stored and manipulated, and why we write programs in high-level languages. An introduction to Python, interactive mode, script mode, and the IDLE environment are also given.

Chapter 2: Input, Processing, and Output

This chapter introduces the program development cycle, variables, data types, and simple programs that are written as sequence structures. The student learns to write simple programs that read input from the keyboard, perform mathematical operations, and produce screen output. Pseudocode and flowcharts are also introduced as tools for designing programs.

Chapter 3: Decision Structures and Boolean Logic

In this chapter the student learns about relational operators and Boolean expressions and is shown how to control the flow of a program with decision structures. The `if`, `if-else`, and `if-elif-else` statements are covered. Nested decision structures and logical operators are also discussed.

Chapter 4: Repetition Structures

This chapter shows the student how to create repetition structures using the `while` loop and `for` loop. Counters, accumulators, running totals, and sentinels are discussed, as well as techniques for writing input validation loops.

Chapter 5: Functions

In this chapter the student first learns how to write and call void functions. The chapter shows the benefits of using functions to modularize programs and discusses the top-down design approach. Then, the student learns to pass arguments to functions. Common library functions, such as those for generating random numbers, are discussed. After learning how to call library functions and use their return value, the student learns to define and call his or her own functions. Then the student learns how to use modules to organize functions.

Chapter 6: Files and Exceptions

This chapter introduces sequential file input and output. The student learns to read and write large sets of data and store data as fields and records. The chapter concludes by discussing exceptions and shows the student how to write exception-handling code.

Chapter 7: Lists and Tuples

This chapter introduces the student to the concept of a sequence in Python and explores the use of two common Python sequences: lists and tuples. The student learns to use lists for arraylike operations, such as storing objects in a list, iterating over a list, searching for items in a list, and calculating the sum and average of items in a list. The chapter discusses slicing and many of the list methods. One- and two-dimensional lists are covered.

Chapter 8: More About Strings

In this chapter the student learns to process strings at a detailed level. String slicing and algorithms that step through the individual characters in a string are discussed, and several built-in functions and string methods for character and text processing are introduced.

Chapter 9: Dictionaries and Sets

This chapter introduces the dictionary and set data structures. The student learns to store data as key-value pairs in dictionaries, search for values, change existing values, add new key-value pairs, and delete key-value pairs. The student learns to store values as unique elements in sets and perform common set operations such as union, intersection, difference, and symmetric difference. The chapter concludes with a discussion of object serialization and introduces the student to the Python `pickle` module.

Chapter 10: Classes and Object-Oriented Programming

This chapter compares procedural and object-oriented programming practices. It covers the fundamental concepts of classes and objects. Attributes, methods, encapsulation and data hiding, `__init__` functions (which are similar to constructors), accessors, and mutators are discussed. The student learns how to model classes with UML and how to find the classes in a particular problem.

Chapter 11: Inheritance

The study of classes continues in this chapter with the subjects of inheritance and polymorphism. The topics covered include superclasses, subclasses, how `__init__` functions work in inheritance, method overriding, and polymorphism.

Chapter 12: Recursion

This chapter discusses recursion and its use in problem solving. A visual trace of recursive calls is provided, and recursive applications are discussed. Recursive algorithms for many tasks are presented, such as finding factorials, finding a greatest common denominator (GCD), and summing a range of values in a list, and the classic Towers of Hanoi example are presented.

Chapter 13: GUI Programming

This chapter discusses the basic aspects of designing a GUI application using the `tkinter` module in Python. Fundamental widgets, such as labels, buttons, entry fields, radio buttons, check buttons, and dialog boxes, are covered. The student also learns how events work in a GUI application and how to write callback functions to handle events.

Appendix A: Installing Python

This appendix explains how to download and install the Python 3 interpreter.

Appendix B: Introduction to IDLE

This appendix gives an overview of the IDLE integrated development environment that comes with Python.

Appendix C: The ASCII Character Set

As a reference, this appendix lists the ASCII character set.

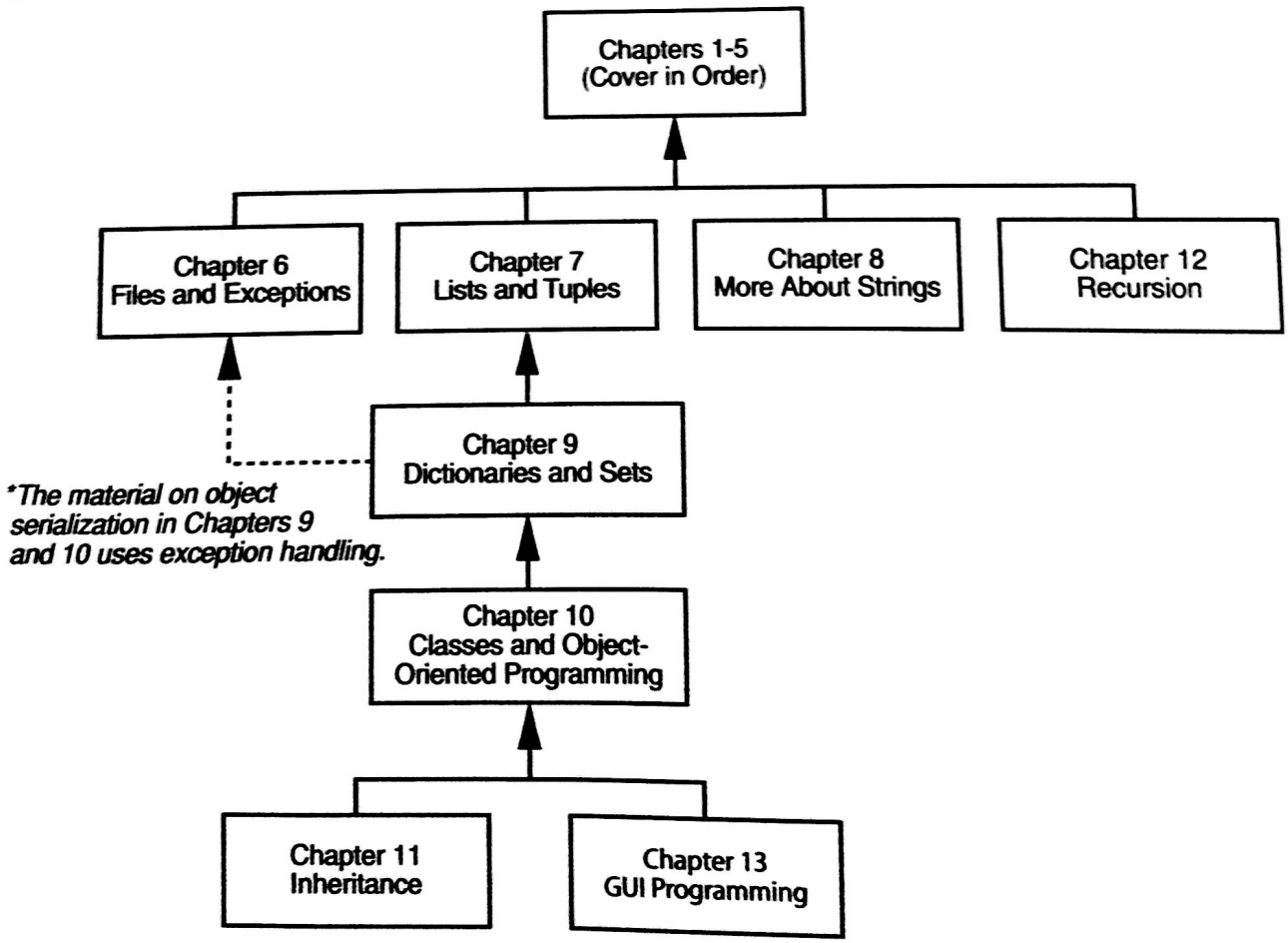
Appendix D: Answers to Checkpoints

This appendix gives the answers to the Checkpoint questions that appear throughout the text.

Organization of the Text

The text teaches programming in a step-by-step manner. Each chapter covers a major set of topics and builds knowledge as students progress through the book. Although the chapters can be easily taught in their existing sequence, you do have some flexibility in the order that you wish to cover them. Figure P-1 shows chapter dependencies. Each box represents a chapter or a group of chapters. An arrow points from a chapter to the chapter that must be covered before it.

Figure P-1 Chapter dependencies



Features of the Text

Concept	Each major section of the text starts with a concept statement.
Statements	This statement concisely summarizes the main point of the section.
Example Programs	Each chapter has an abundant number of complete and partial example programs, each designed to highlight the current topic.
 **In the Spotlight Case Studies**	Each chapter has one or more In the Spotlight case studies that provide detailed, step-by-step analysis of problems and show the student how to solve them.
 **VideoNotes**	Online videos developed specifically for this book are available for viewing at `www.pearsonhighered.com/gaddis/videonotes`. Icons appear throughout the text alerting the student to videos about specific topics.
 **Notes**	Notes appear at several places throughout the text. They are short explanations of interesting or often misunderstood points relevant to the topic at hand.
 **Tips**	Tips advise the student on the best techniques for approaching different programming problems.
 **Warnings**	Warnings caution students about programming techniques or practices that can lead to malfunctioning programs or lost data.
 **Checkpoints**	Checkpoints are questions placed at intervals throughout each chapter. They are designed to query the student's knowledge quickly after learning a new topic.
Review Questions	Each chapter presents a thorough and diverse set of review questions and exercises. They include Multiple Choice, True/False, Algorithm Workbench, and Short Answer.
Programming Exercises	Each chapter offers a pool of programming exercises designed to solidify the student's knowledge of the topics currently being studied.

Supplements

Student Online Resources

Many student resources are available for this book from the publisher. The following items are available on the Gaddis Series resource page at `www.pearsonhighered.com/gaddis`

- The source code for each example program in the book
- Access to the book's companion VideoNotes

Instructor Resources

The following supplements are available to qualified instructors only:

- Answers to all of the Review Questions
- Solutions for the exercises
- PowerPoint presentation slides for each chapter
- Test bank

Visit the Pearson Education Instructor Resource Center (`www.pearsonhighered.com/irc`) or contact your local Pearson Education campus representative for information on how to access them.

Acknowledgments

I would like to thank the following faculty reviewers for their insight, expertise, and thoughtful recommendations:

Paul Amer
University of Delaware

James Atlas
University of Delaware

James Carrier
Guilford Technical Community College

John Cavazos
University of Delaware

Barbara Goldner
North Seattle Community College

Paul Gruhn
Manchester Community College

Diane Innes
Sandhills Community College

Daniel Jinguji
North Seattle Community College

Gary Marrer
Glendale Community College

Keith Mehl
Chabot College

Vince Offenback
North Seattle Community College

Smiljana Petrovic
Iona College

Raymond Pettit
Abilene Christian University

Janet Renwick
University of Arkansas–Fort Smith

Tom Stokke
University of North Dakota

Karen Ughetta
Virginia Western Community College

Reviewers of Previous Editions

Desmond K. H. Chun
Chabot Community College

Bob Husson
Craven Community College

Shyamal Mitra
University of Texas at Austin

Ken Robol
Beaufort Community College

Eric Shaffer
University of Illinois at Urbana-Champaign

Ann Ford Tyson
Florida State University

Linda F. Wilson
Texas Lutheran University

I would like to thank my family for their love and support in all my many projects. I am extremely fortunate to have Matt Goldstein as my editor. I am also fortunate to have Yez Alayan as marketing manager and Kathryn Ferranti as marketing coordinator. They do a great job getting my books out to the academic community. I work with a great production team led by Marilyn Lloyd and Kayla Smith-Tarbox. Thanks to you all!

About the Author

Tony Gaddis is the principal author of the *Starting Out With* series of textbooks. Tony has nearly two decades of experience teaching computer science courses, primarily at Haywood Community College. He is a highly acclaimed instructor who was previously selected as the North Carolina Community College "Teacher of the Year" and has received the Teaching Excellence award from the National Institute for Staff and Organizational Development. The *Starting Out With* series includes introductory books covering C++, Java™, Microsoft® Visual Basic®, Microsoft® C#®, Python®, Programming Logic and Design, Alice, and App Inventor, all published by Pearson. More information about all these books can be found at `www.pearsonhighered.com/gaddisbooks`.

CHAPTER 1

Introduction to Computers and Programming

TOPICS

1.1 Introduction

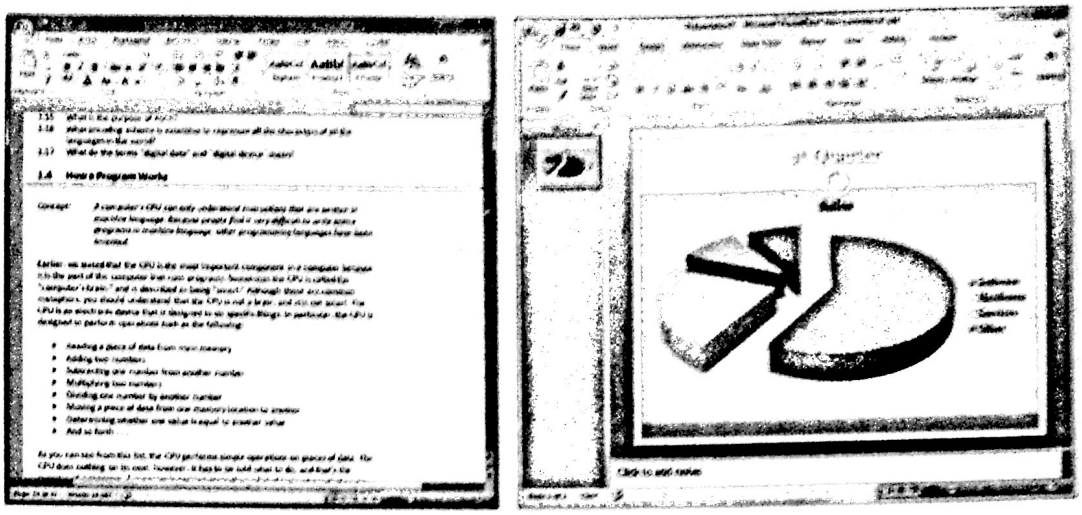
Think about some of the different ways that people use computers. In school, students use computers for tasks such as writing papers, searching for articles, sending email, and participating in online classes. At work, people use computers to analyze data, make presentations, conduct business transactions, communicate with customers and coworkers, control machines in manufacturing facilities, and do many other things. At home, people use computers for tasks such as paying bills, shopping online, communicating with friends and family, and playing computer games. And don't forget that cell phones, iPods®, smart phones, car navigation systems, and many other devices are computers too. The uses of computers are almost limitless in our everyday lives.

Computers can do such a wide variety of things because they can be programmed. This means that computers are not designed to do just one job, but to do any job that their programs tell them to do. A *program* is a set of instructions that a computer follows to perform a task. For example, Figure 1-1 shows screens using Microsoft Word and PowerPoint, two commonly used programs.

Programs are commonly referred to as *software*. Software is essential to a computer because it controls everything the computer does. All of the software that we use to make our computers useful is created by individuals working as programmers or software developers. A *programmer,* or *software developer,* is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, you will find programmers' work used in business, medicine, government, law enforcement, agriculture, academics, entertainment, and many other fields.

This book introduces you to the fundamental concepts of computer programming using the Python language. The Python language is a good choice for beginners because it is easy to learn

Figure 1-1 A word processing program and an image editing program



and programs can be written quickly using it. Python is also a powerful language, popular with professional software developers. In fact, it is has been reported that Python is used by Google, NASA, YouTube, various game companies, the New York Stock Exchange, and many others.

Before we begin exploring the concepts of programming, you need to understand a few basic things about computers and how they work. This chapter will build a solid foundation of knowledge that you will continually rely on as you study computer science. First, we will discuss the physical components that computers are commonly made of. Next, we will look at how computers store data and execute programs. Finally, we will get a quick introduction to the software that you will use to write Python programs.

1.2 Hardware and Software

CONCEPT: **The physical devices that a computer is made of are referred to as the computer's hardware. The programs that run on a computer are referred to as software.**

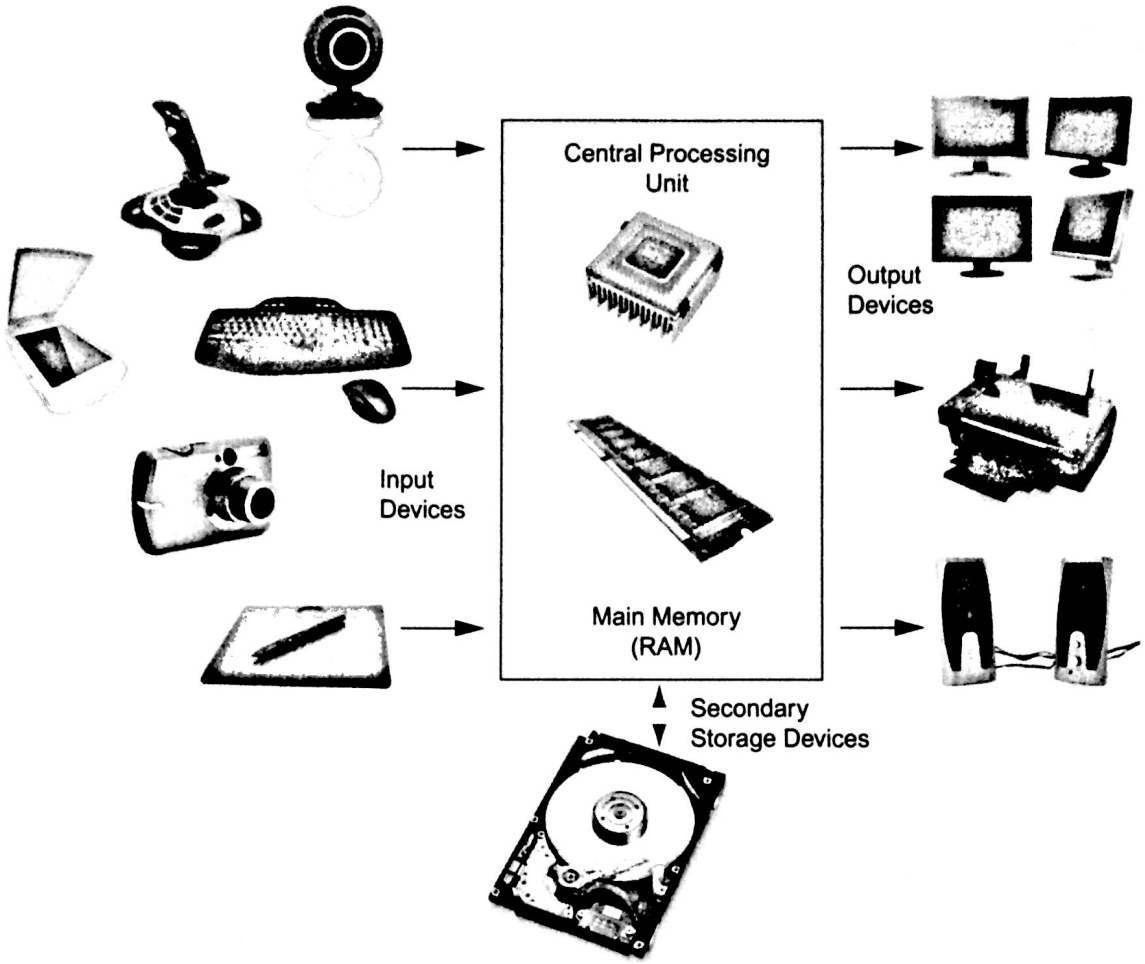
Hardware

The term *hardware* refers to all of the physical devices, or *components*, that a computer is made of. A computer is not one single device, but a system of devices that all work together. Like the different instruments in a symphony orchestra, each device in a computer plays its own part.

If you have ever shopped for a computer, you've probably seen sales literature listing components such as microprocessors, memory, disk drives, video displays, graphics cards, and so on. Unless you already know a lot about computers, or at least have a friend that does, understanding what these different components do might be challenging. As shown in Figure 1-2, a typical computer system consists of the following major components:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices

Figure 1-2 Typical components of a computer system



- Input devices
- Output devices

Let's take a closer look at each of these components.

The CPU

When a computer is performing the tasks that a program tells it to do, we say that the computer is *running* or *executing* the program. The *central processing unit*, or *CPU*, is the part of a computer that actually runs programs. The CPU is the most important component in a computer because without it, the computer could not run software.

In the earliest computers, CPUs were huge devices made of electrical and mechanical components such as vacuum tubes and switches. Figure 1-3 shows such a device. The two women in the photo are working with the historic ENIAC computer. The *ENIAC*, which is considered by many to be the world's first programmable electronic computer, was built in 1945 to calculate artillery ballistic tables for the U.S. Army. This machine, which was primarily one big CPU, was 8 feet tall, 100 feet long, and weighed 30 tons.

Today, CPUs are small chips known as *microprocessors*. Figure 1-4 shows a photo of a lab technician holding a modern microprocessor. In addition to being much smaller than the old electromechanical CPUs in early computers, microprocessors are also much more powerful.

Figure 1-3 The ENIAC computer (courtesy of U.S. Army Historic Computer Images)

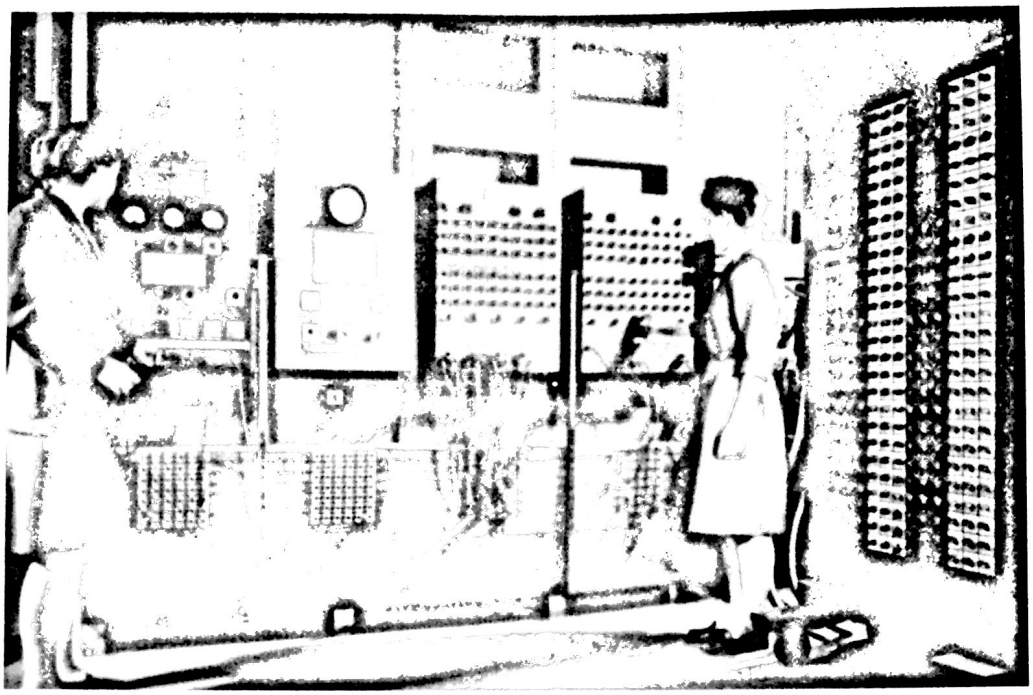
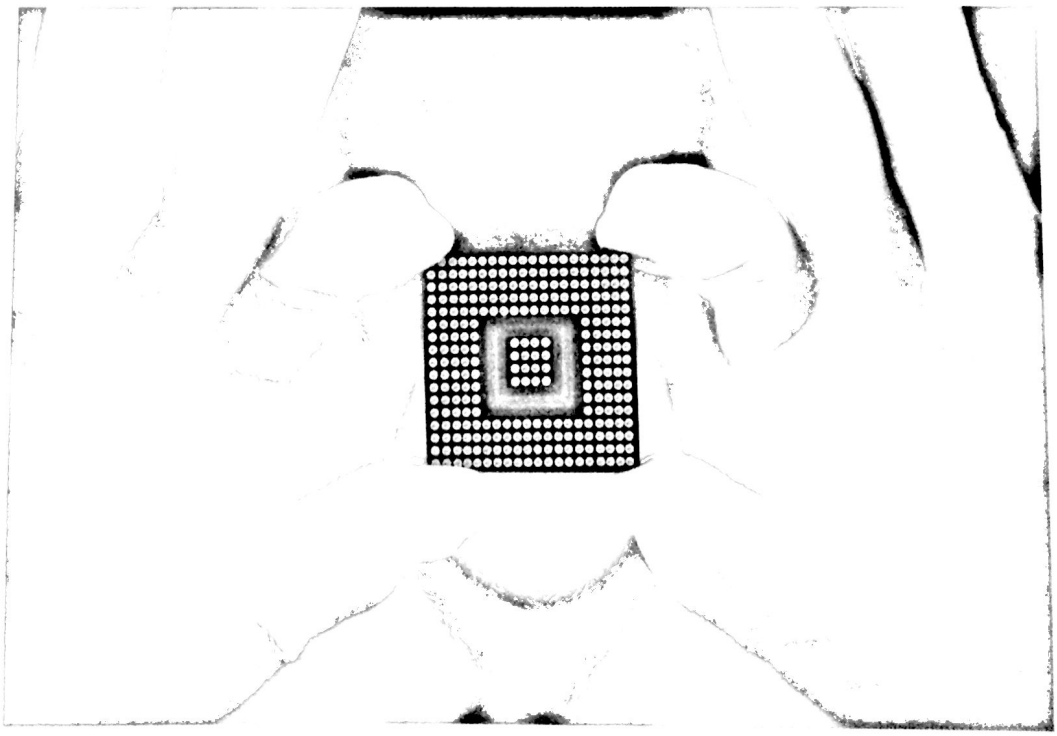


Figure 1-4 A lab technician holds a modern microprocessor (Creativa/Shutterstock)

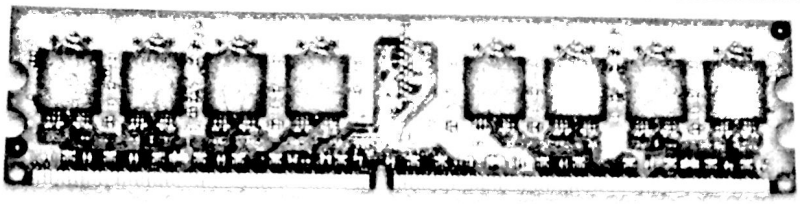


Main Memory

You can think of *main memory* as the computer's work area. This is where the computer stores a program while the program is running, as well as the data that the program is working with. For example, suppose you are using a word processing program to write an essay for one of your classes. While you do this, both the word processing program and the essay are stored in main memory.

Main memory is commonly known as *random-access memory*, or *RAM*. It is called this because the CPU is able to quickly access data stored at any random location in RAM. RAM is usually a *volatile* type of memory that is used only for temporary storage while a program is running. When the computer is turned off, the contents of RAM are erased. Inside your computer, RAM is stored in chips, similar to the ones shown in Figure 1-5.

Figure 1-5 Memory chips (Garsya/Shutterstock)



Secondary Storage Devices

Secondary storage is a type of memory that can hold data for long periods of time, even when there is no power to the computer. Programs are normally stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory records, is saved to secondary storage as well.

The most common type of secondary storage device is the *disk drive*. A traditional disk drive stores data by magnetically encoding it onto a spinning circular disk. *Solid-state drives*, which store data in solid-state memory, are increasingly becoming popular. A solid-state drive has no moving parts and operates faster than a traditional disk drive. Most computers have some sort of secondary storage device, either a traditional disk drive or a solid-state drive, mounted inside their case. External storage devices, which connect to one of the computer's communication ports, are also available. External storage devices can be used to create backup copies of important data or to move data to another computer.

In addition to external storage devices, many types of devices have been created for copying data and for moving it to other computers. For many years floppy disk drives were popular. A *floppy disk drive* records data onto a small floppy disk, which can be removed from the drive. Floppy disks have many disadvantages, however. They hold only a small amount of data, are slow to access data, and can be unreliable. Floppy disk drives are rarely used today, in favor of superior devices such as USB drives. *USB drives* are small devices that plug into the computer's USB (universal serial bus) port and

appear to the system as a disk drive. These drives do not actually contain a disk, however. They store data in a special type of memory known as *flash memory*. USB drives, which are also known as *memory sticks* and *flash drives*, are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the *CD* (compact disc) and the *DVD* (digital versatile disc) are also popular for data storage. Data is not recorded magnetically on an optical disc, but is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they are good mediums for creating backup copies of data.

Input Devices

Input is any data the computer collects from people and from other devices. The component that collects the data and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, scanner, microphone, and digital camera. Disk drives and optical drives can also be considered input devices because programs and data are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any data the computer produces for people or for other devices. It might be a sales report, a list of names, or a graphic image. The data is sent to an *output device*, which formats and presents it. Common output devices are video displays and printers. Disk drives and CD recorders can also be considered output devices because the system sends data to them in order to be saved.

Software

If a computer is to function, software is not optional. Everything that a computer does, from the time you turn the power switch on until you shut the system down, is under the control of software. There are two general categories of software: system software and application software. Most computer programs clearly fit into one of these two categories. Let's take a closer look at each.

System Software

The programs that control and manage the basic operations of a computer are generally referred to as *system software*. System software typically includes the following types of programs:

Operating Systems An *operating system* is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all of the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer. Popular operating systems for laptop and desktop computers include Windows, Mac OS, and Linux. Popular operating systems for mobile devices include Android and iOS.

Utility Programs A *utility program* performs a specialized task that enhances the computer's operation or safeguards data. Examples of utility programs are virus scanners, file compression programs, and data backup programs.

Software Development Tools *Software development tools* are the programs that programmers use to create, modify, and test software. Assemblers, compilers, and interpreters are examples of programs that fall into this category.

Application Software

Programs that make a computer useful for everyday tasks are known as *application software*. These are the programs that people normally spend most of their time running on their computers. Figure 1-1, at the beginning of this chapter, shows screens from two commonly used applications: Microsoft Word, a word processing program, and PowerPoint, a presentation program. Some other examples of application software are spreadsheet programs, email programs, web browsers, and game programs.

Checkpoint

1.1 What is a program?

1.2 What is hardware?

1.3 List the five major components of a computer system.

1.4 What part of the computer actually runs programs?

1.5 What part of the computer serves as a work area to store a program and its data while the program is running?

1.6 What part of the computer holds data for long periods of time, even when there is no power to the computer?

1.7 What part of the computer collects data from people and from other devices?

1.8 What part of the computer formats and presents data for people or other devices?

1.9 What fundamental set of programs control the internal operations of the computer's hardware?

1.10 What do you call a program that performs a specialized task, such as a virus scanner, a file compression program, or a data backup program?

1.11 Word processing programs, spreadsheet programs, email programs, web browsers, and game programs belong to what category of software?

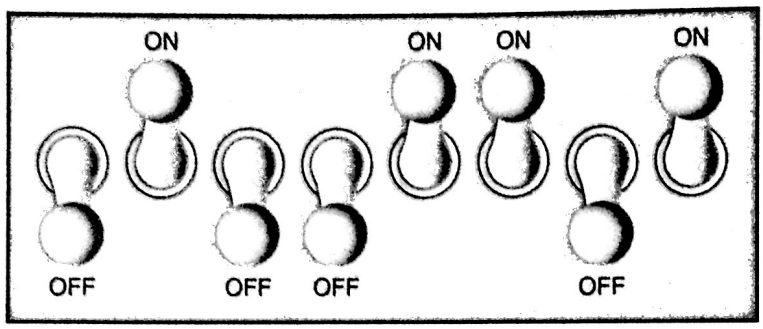
1.3 How Computers Store Data

CONCEPT: All data that is stored in a computer is converted to sequences of 0s and 1s.

A computer's memory is divided into tiny storage locations known as *bytes*. One byte is only enough memory to store a letter of the alphabet or a small number. In order to do anything meaningful, a computer has to have lots of bytes. Most computers today have millions, or even billions, of bytes of memory.

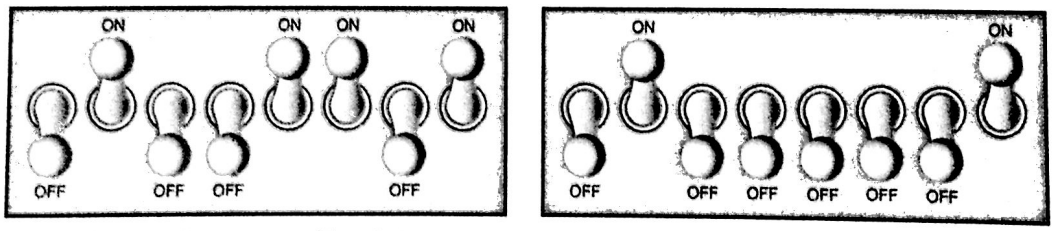
Each byte is divided into eight smaller storage locations known as bits. The term *bit* stands for *binary digit*. Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer systems, bits are tiny electrical components that can hold either a positive or a negative charge. Computer scientists think of a positive charge as a switch in the *on* position, and a negative charge as a switch in the *off* position. Figure 1-6 shows the way that a computer scientist might think of a byte of memory: as a collection of switches that are each flipped to either the on or off position.

Figure 1-6 Think of a byte as eight switches



When a piece of data is stored in a byte, the computer sets the eight bits to an on/off pattern that represents the data. For example, the pattern on the left in Figure 1-7 shows how the number 77 would be stored in a byte, and the pattern on the right shows how the letter A would be stored in a byte. We explain below how these patterns are determined.

Figure 1-7 Bit patterns for the number 77 and the letter A



The number 77 stored in a byte.

The letter A stored in a byte.

Storing Numbers

A bit can be used in a very limited way to represent numbers. Depending on whether the bit is turned on or off, it can represent one of two different values. In computer systems, a bit that is turned off represents the number 0 and a bit that is turned on represents the number 1. This corresponds perfectly to the *binary numbering system*. In the binary numbering system (or *binary*, as it is usually called) all numeric values are written as sequences of 0s and 1s. Here is an example of a number that is written in binary:

```
10011101
```

The position of each digit in a binary number has a value assigned to it. Starting with the rightmost digit and moving left, the position values are 2^0, 2^1, 2^2, 2^3, and so forth, as shown in Figure 1-8. Figure 1-9 shows the same diagram with the position values calculated. Starting with the rightmost digit and moving left, the position values are 1, 2, 4, 8, and so forth.

Figure 1-8 The values of binary digits as powers of 2

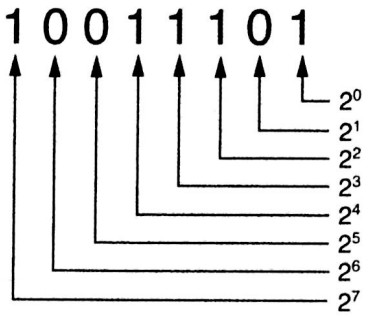
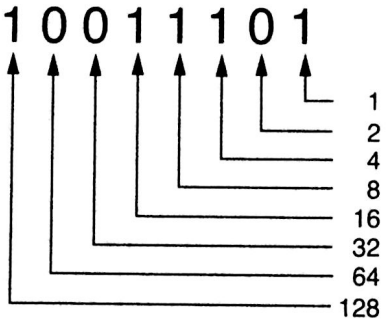


Figure 1-9 The values of binary digits



To determine the value of a binary number you simply add up the position values of all the 1s. For example, in the binary number 10011101, the position values of the 1s are 1, 4, 8, 16, and 128. This is shown in Figure 1-10. The sum of all of these position values is 157. So, the value of the binary number 10011101 is 157.

Figure 1-10 Determining the value of 10011101

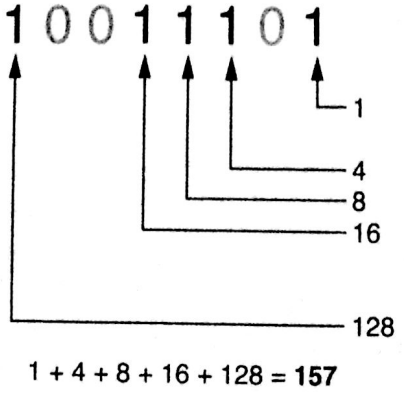
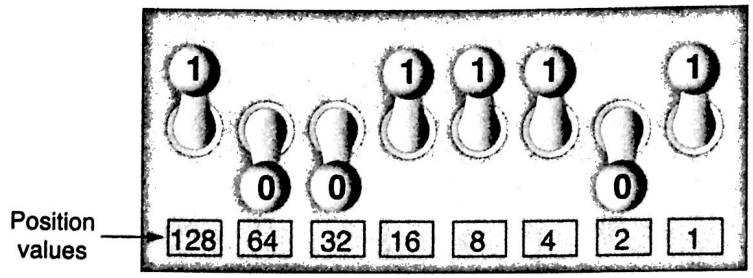


Figure 1-11 shows how you can picture the number 157 stored in a byte of memory. Each 1 is represented by a bit in the on position, and each 0 is represented by a bit in the off position.

Figure 1-11 The bit pattern for 157

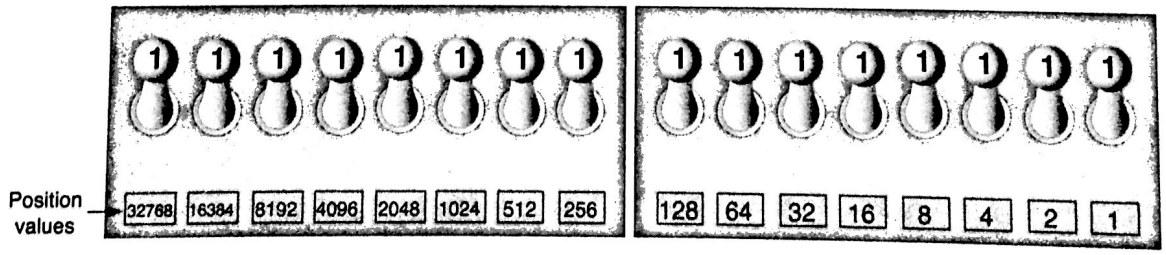


128 + 16 + 8 + 4 + 1 = **157**

When all of the bits in a byte are set to 0 (turned off), then the value of the byte is 0. When all of the bits in a byte are set to 1 (turned on), then the byte holds the largest value that can be stored in it. The largest value that can be stored in a byte is 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255. This limit exists because there are only eight bits in a byte.

What if you need to store a number larger than 255? The answer is simple: use more than one byte. For example, suppose we put two bytes together. That gives us 16 bits. The position values of those 16 bits would be 2^0, 2^1, 2^2, 2^3, and so forth, up through 21^5. As shown in Figure 1-12, the maximum value that can be stored in two bytes is 65,535. If you need to store a number larger than this, then more bytes are necessary.

Figure 1-12 Two bytes used for a large number



32768 + 16384 + 8192 + 4096 + 2048 + 1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **65535**

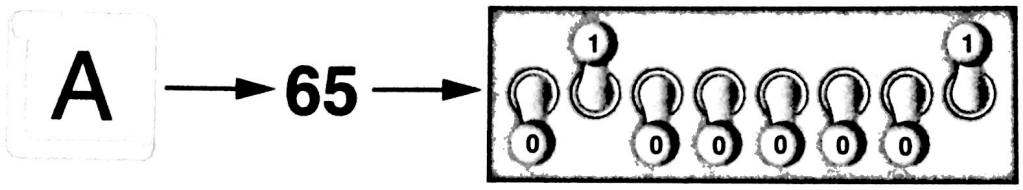
TIP: In case you're feeling overwhelmed by all this, relax! You will not have to actually convert numbers to binary while programming. Knowing that this process is taking place inside the computer will help you as you learn, and in the long term this knowledge will make you a better programmer.

Storing Characters

Any piece of data that is stored in a computer's memory must be stored as a binary number. That includes characters, such as letters and punctuation marks. When a character is stored in memory, it is first converted to a numeric code. The numeric code is then stored in memory as a binary number.

Over the years, different coding schemes have been developed to represent characters in computer memory. Historically, the most important of these coding schemes is *ASCII*, which stands for the *American Standard Code for Information Interchange*. ASCII is a set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters. For example, the ASCII code for the uppercase letter A is 65. When you type an uppercase A on your computer keyboard, the number 65 is stored in memory (as a binary number, of course). This is shown in Figure 1-13.

Figure 1-13 The letter A is stored in memory as the number 65



TIP: The acronym ASCII is pronounced "askee."

In case you are curious, the ASCII code for uppercase B is 66, for uppercase C is 67, and so forth. Appendix C shows all of the ASCII codes and the characters they represent.

The ASCII character set was developed in the early 1960s and was eventually adopted by most all computer manufacturers. ASCII is limited, however, because it defines codes for only 128 characters. To remedy this, the Unicode character set was developed in the early 1990s. *Unicode* is an extensive encoding scheme that is compatible with ASCII, but can also represent characters for many of the languages in the world. Today, Unicode is quickly becoming the standard character set used in the computer industry.

Advanced Number Storage

Earlier you read about numbers and how they are stored in memory. While reading that section, perhaps it occurred to you that the binary numbering system can be used to represent only integer numbers, beginning with 0. Negative numbers and real numbers (such as 3.14159) cannot be represented using the simple binary numbering technique we discussed.

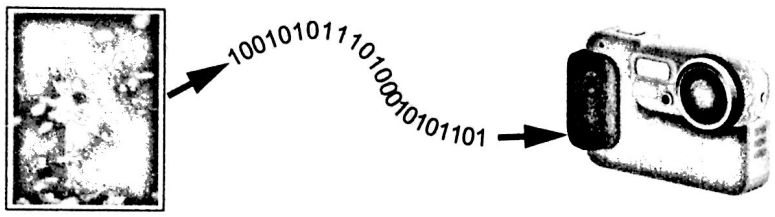
Computers are able to store negative numbers and real numbers in memory, but to do so they use encoding schemes along with the binary numbering system. Negative numbers are encoded using a technique known as *two's complement*, and real numbers are encoded in *floating-point notation*. You don't need to know how these encoding schemes work, only that they are used to convert negative numbers and real numbers to binary format.

Other Types of Data

Computers are often referred to as digital devices. The term *digital* can be used to describe anything that uses binary numbers. *Digital data* is data that is stored in binary, and a *digital device* is any device that works with binary data. In this section we have discussed how numbers and characters are stored in binary, but computers also work with many other types of digital data.

For example, consider the pictures that you take with your digital camera. These images are composed of tiny dots of color known as *pixels*. (The term pixel stands for *picture element*.) As shown in Figure 1-14, each pixel in an image is converted to a numeric code that represents the pixel's color. The numeric code is stored in memory as a binary number.

Figure 1-14 A digital image is stored in binary format



The music that you play on your CD player, iPod, or MP3 player is also digital. A digital song is broken into small pieces known as *samples*. Each sample is converted to a binary number, which can be stored in memory. The more samples that a song is divided into, the more it sounds like the original music when it is played back. A CD quality song is divided into more than 44,000 samples per second!

Checkpoint

1.12 What amount of memory is enough to store a letter of the alphabet or a small number?

1.13 What do you call a tiny "switch" that can be set to either on or off?

1.14 In what numbering system are all numeric values written as sequences of 0s and 1s?

1.15 What is the purpose of ASCII?

1.16 What encoding scheme is extensive enough to represent the characters of many of the languages in the world?

1.17 What do the terms "digital data" and "digital device" mean?

1.4 How a Program Works

CONCEPT: **A computer's CPU can only understand instructions that are written in machine language. Because people find it very difficult to write entire programs in machine language, other programming languages have been invented.**

Earlier, we stated that the CPU is the most important component in a computer because it is the part of the computer that runs programs. Sometimes the CPU is called the "computer's brain" and is described as being "smart." Although these are common metaphors, you should understand that the CPU is not a brain, and it is not smart. The CPU is an electronic device that is designed to do specific things. In particular, the CPU is designed to perform operations such as the following:

- Reading a piece of data from main memory
- Adding two numbers
- Subtracting one number from another number
- Multiplying two numbers
- Dividing one number by another number
- Moving a piece of data from one memory location to another
- Determining whether one value is equal to another value

As you can see from this list, the CPU performs simple operations on pieces of data. The CPU does nothing on its own, however. It has to be told what to do, and that's the purpose of a program. A program is nothing more than a list of instructions that cause the CPU to perform operations.

Each instruction in a program is a command that tells the CPU to perform a specific operation. Here's an example of an instruction that might appear in a program:

```
10110000
```

To you and me, this is only a series of 0s and 1s. To a CPU, however, this is an instruction to perform an operation.[1] It is written in 0s and 1s because CPUs only understand instructions that are written in *machine language*, and machine language instructions always have an underlying binary structure.

A machine language instruction exists for each operation that a CPU is capable of performing. For example, there is an instruction for adding numbers, there is an instruction for subtracting one number from another, and so forth. The entire set of instructions that a CPU can execute is known as the CPU's *instruction set*.

NOTE: There are several microprocessor companies today that manufacture CPUs. Some of the more well-known microprocessor companies are Intel, AMD, and Motorola. If you look carefully at your computer, you might find a tag showing a logo for its microprocessor.

Each brand of microprocessor has its own unique instruction set, which is typically understood only by microprocessors of the same brand. For example, Intel microprocessors understand the same instructions, but they do not understand instructions for Motorola microprocessors.

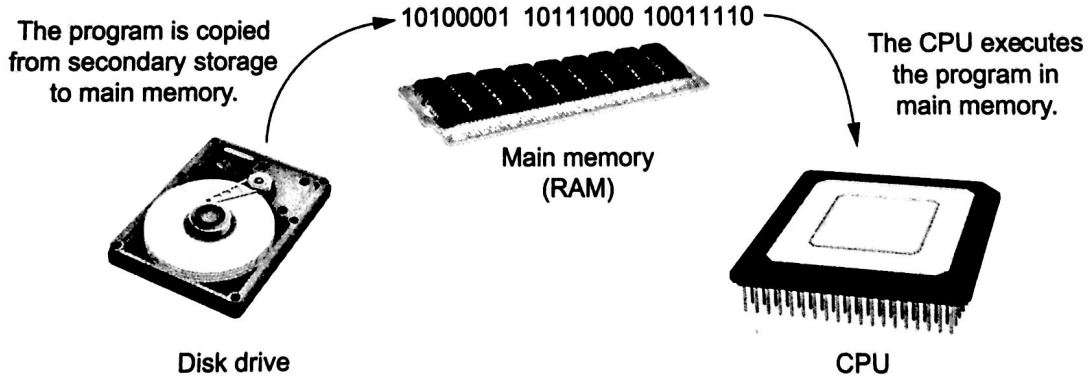
[1] The example shown is an actual instruction for an Intel microprocessor. It tells the microprocessor to move a value into the CPU.

The machine language instruction that was previously shown is an example of only one instruction. It takes a lot more than one instruction, however, for the computer to do anything meaningful. Because the operations that a CPU knows how to perform are so basic in nature, a meaningful task can be accomplished only if the CPU performs many operations. For example, if you want your computer to calculate the amount of interest that you will earn from your savings account this year, the CPU will have to perform a large number of instructions, carried out in the proper sequence. It is not unusual for a program to contain thousands or even millions of machine language instructions.

Programs are usually stored on a secondary storage device such as a disk drive. When you install a program on your computer, the program is typically copied to your computer's disk drive from a CD-ROM, or perhaps downloaded from a website.

Although a program can be stored on a secondary storage device such as a disk drive, it has to be copied into main memory, or RAM, each time the CPU executes it. For example, suppose you have a word processing program on your computer's disk. To execute the program you use the mouse to double-click the program's icon. This causes the program to be copied from the disk into main memory. Then, the computer's CPU executes the copy of the program that is in main memory. This process is illustrated in Figure 1-15.

Figure 1-15 A program is copied into main memory and then executed

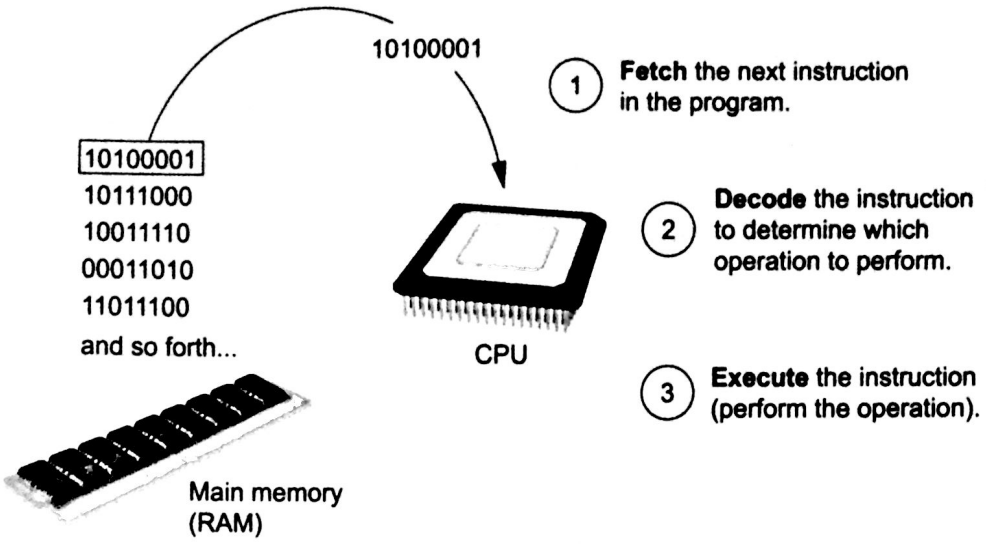


When a CPU executes the instructions in a program, it is engaged in a process that is known as the *fetch-decode-execute cycle*. This cycle, which consists of three steps, is repeated for each instruction in the program. The steps are

1. **Fetch** A program is a long sequence of machine language instructions. The first step of the cycle is to fetch, or read, the next instruction from memory into the CPU.
2. **Decode** A machine language instruction is a binary number that represents a command that tells the CPU to perform an operation. In this step the CPU decodes the instruction that was just fetched from memory, to determine which operation it should perform.
3. **Execute** The last step in the cycle is to execute, or perform, the operation.

Figure 1-16 illustrates these steps.

Figure 1-16 The fetch-decode-execute cycle



From Machine Language to Assembly Language

Computers can only execute programs that are written in machine language. As previously mentioned, a program can have thousands or even millions of binary instructions, and writing such a program would be very tedious and time consuming. Programming in machine language would also be very difficult because putting a 0 or a 1 in the wrong place will cause an error.

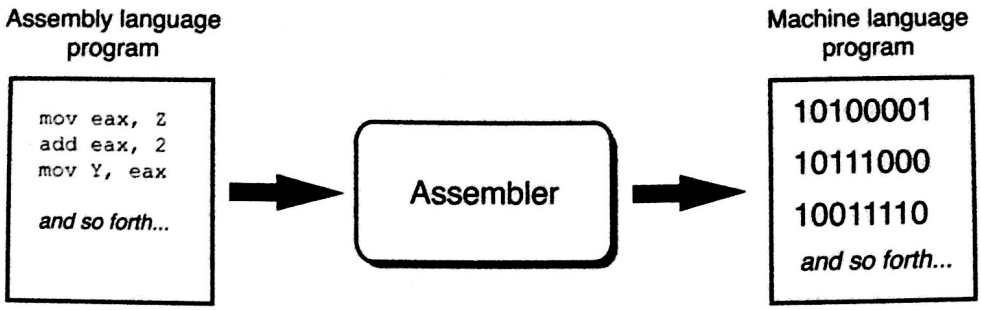
Although a computer's CPU only understands machine language, it is impractical for people to write programs in machine language. For this reason, *assembly language* was created in the early days of computing[2] as an alternative to machine language. Instead of using binary numbers for instructions, assembly language uses short words that are known as *mnemonics*. For example, in assembly language, the mnemonic `add` typically means to add numbers, `mul` typically means to multiply numbers, and `mov` typically means to move a value to a location in memory. When a programmer uses assembly language to write a program, he or she can write short mnemonics instead of binary numbers.

NOTE: There are many different versions of assembly language. It was mentioned earlier that each brand of CPU has its own machine language instruction set. Each brand of CPU typically has its own assembly language as well.

Assembly language programs cannot be executed by the CPU, however. The CPU only understands machine language, so a special program known as an *assembler* is used to translate an assembly language program to a machine language program. This process is shown in Figure 1-17. The machine language program that is created by the assembler can then be executed by the CPU.

[2] The first assembly language was most likely that developed in the 1940s at Cambridge University for use with a historic computer known as the EDSAC.

Figure 1-17 An assembler translates an assembly language program to a machine language program



High-Level Languages

Although assembly language makes it unnecessary to write binary machine language instructions, it is not without difficulties. Assembly language is primarily a direct substitute for machine language, and like machine language, it requires that you know a lot about the CPU. Assembly language also requires that you write a large number of instructions for even the simplest program. Because assembly language is so close in nature to machine language, it is referred to as a *low-level language*.

In the 1950s, a new generation of programming languages known as *high-level languages* began to appear. A high-level language allows you to create powerful and complex programs without knowing how the CPU works and without writing large numbers of low-level instructions. In addition, most high-level languages use words that are easy to understand. For example, if a programmer were using COBOL (which was one of the early high-level languages created in the 1950s), he or she would write the following instruction to display the message *Hello world* on the computer screen:

```
DISPLAY "Hello world"
```

Python is a modern, high-level programming language that we will use in this book. In Python you would display the message *Hello world* with the following instruction:

```
print('Hello world')
```

Doing the same thing in assembly language would require several instructions and an intimate knowledge of how the CPU interacts with the computer's output device. As you can see from this example, high-level languages allow programmers to concentrate on the tasks they want to perform with their programs rather than the details of how the CPU will execute those programs.

Since the 1950s, thousands of high-level languages have been created. Table 1-1 lists several of the more well-known languages.

Key Words, Operators, and Syntax: An Overview

Each high-level language has its own set of predefined words that the programmer must use to write a program. The words that make up a high-level programming language are known as *key words* or *reserved words*. Each key word has a specific meaning, and cannot be used for any other purpose. Table 1-2 shows all of the Python key words.

Table 1-1 Programming languages

Language	Description
Ada	Ada was created in the 1970s, primarily for applications used by the U.S. Department of Defense. The language is named in honor of Countess Ada Lovelace, an influential and historic figure in the field of computing.
BASIC	Beginners All-purpose Symbolic Instruction Code is a general-purpose language that was originally designed in the early 1960s to be simple enough for beginners to learn. Today, there are many different versions of BASIC.
FORTRAN	FORmula TRANslator was the first high-level programming language. It was designed in the 1950s for performing complex mathematical calculations.
COBOL	Common Business-Oriented Language was created in the 1950s and was designed for business applications.
Pascal	Pascal was created in 1970 and was originally designed for teaching programming. The language was named in honor of the mathematician, physicist, and philosopher Blaise Pascal.
C and C++	C and C++ (pronounced "c plus plus") are powerful, general-purpose languages developed at Bell Laboratories. The C language was created in 1972, and the C++ language was created in 1983.
C#	Pronounced "c sharp." This language was created by Microsoft around the year 2000 for developing applications based on the Microsoft .NET platform.
Java	Java was created by Sun Microsystems in the early 1990s. It can be used to develop programs that run on a single computer or over the Internet from a web server.
JavaScript	JavaScript, created in the 1990s, can be used in web pages. Despite its name, JavaScript is not related to Java.
Python	Python, the language we use in this book, is a general-purpose language created in the early 1990s. It has become popular in business and academic applications.
Ruby	Ruby is a general-purpose language that was created in the 1990s. It is increasingly becoming a popular language for programs that run on web servers.
Visual Basic	Visual Basic (commonly known as VB) is a Microsoft programming language and software development environment that allows programmers to create Windows-based applications quickly. VB was originally created in the early 1990s.

Table 1-2 The Python key words

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

In addition to key words, programming languages have *operators* that perform various operations on data. For example, all programming languages have math operators that perform arithmetic. In Python, as well as most other languages, the + sign is an operator that adds two numbers. The following adds 12 and 75:

```
12 + 75
```

There are numerous other operators in the Python language, many of which you will learn about as you progress through this text.

In addition to key words and operators, each language also has its own *syntax,* which is a set of rules that must be strictly followed when writing a program. The syntax rules dictate how key words, operators, and various punctuation characters must be used in a program. When you are learning a programming language, you must learn the syntax rules for that particular language.

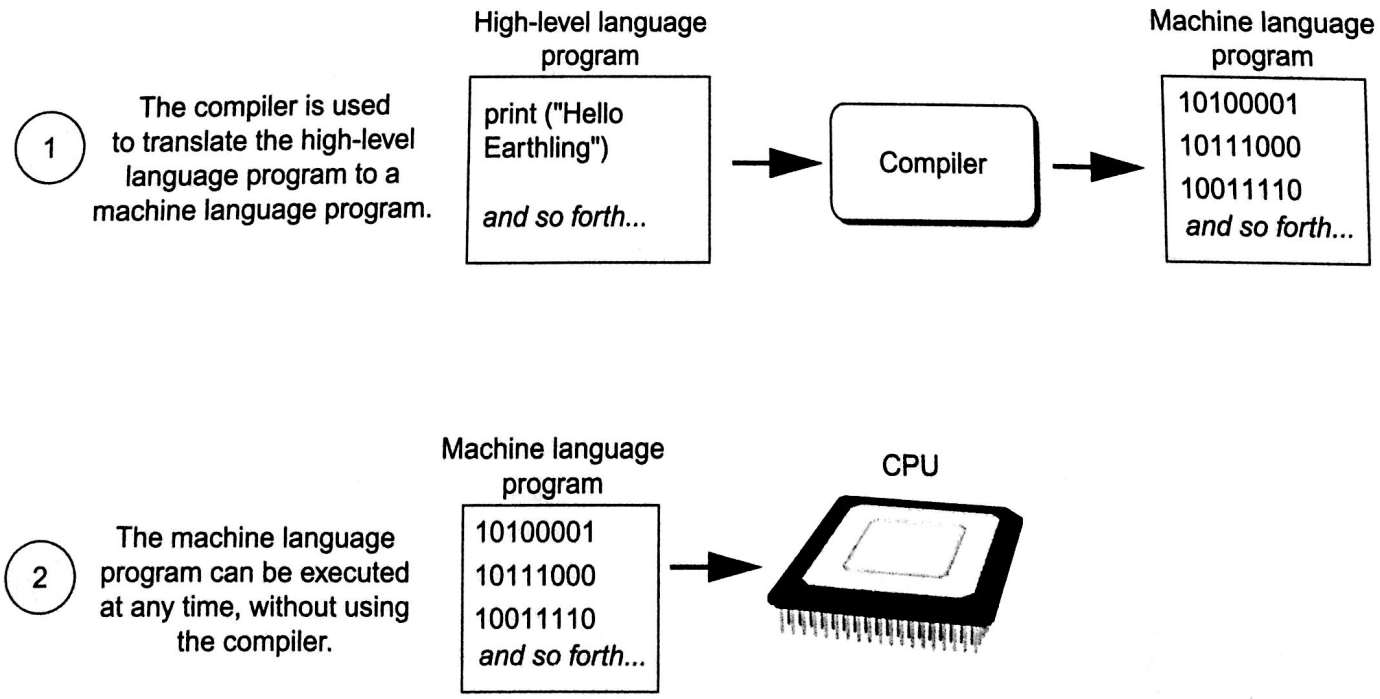
The individual instructions that you use to write a program in a high-level programming language are called *statements*. A programming statement can consist of key words, operators, punctuation, and other allowable programming elements, arranged in the proper sequence to perform an operation.

Compilers and Interpreters

Because the CPU understands only machine language instructions, programs that are written in a high-level language must be translated into machine language. Depending on the language that a program has been written in, the programmer will use either a compiler or an interpreter to make the translation.

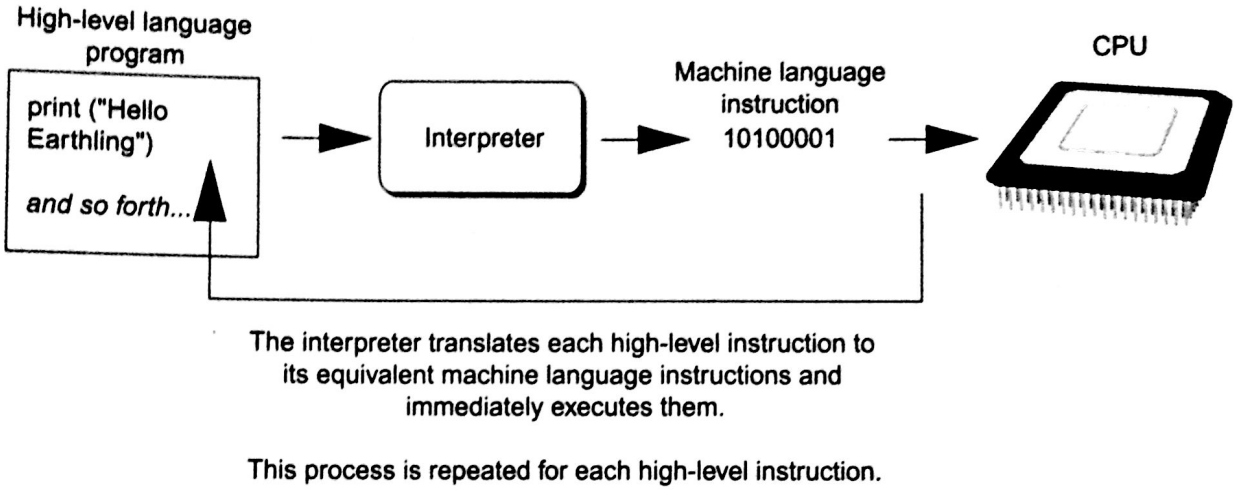
A *compiler* is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed. This is shown in Figure 1-18. As shown in the figure, compiling and executing are two different processes.

Figure 1-18 Compiling a high-level program and executing it



The Python language uses an *interpreter,* which is a program that both translates and executes the instructions in a high-level language program. As the interpreter reads each individual instruction in the program, it converts it to machine language instructions and then immediately executes them. This process repeats for every instruction in the program. This process is illustrated in Figure 1-19. Because interpreters combine translation and execution, they typically do not create separate machine language programs.

Figure 1-19 Executing a high-level program with an interpreter



The statements that a programmer writes in a high-level language are called *source code,* or simply *code.* Typically, the programmer types a program's code into a text editor and then saves the code in a file on the computer's disk. Next, the programmer uses a compiler to translate the code into a machine language program, or an interpreter to translate and execute the code. If the code contains a syntax error, however, it cannot be translated. A *syntax error* is a mistake such as a misspelled key word, a missing punctuation character, or the incorrect use of an operator. When this happens the compiler or interpreter displays an error message indicating that the program contains a syntax error. The programmer corrects the error and then attempts once again to translate the program.

NOTE: Human languages also have syntax rules. Do you remember when you took your first English class, and you learned all those rules about commas, apostrophes, capitalization, and so forth? You were learning the syntax of the English language.

Although people commonly violate the syntax rules of their native language when speaking and writing, other people usually understand what they mean. Unfortunately, compilers and interpreters do not have this ability. If even a single syntax error appears in a program, the program cannot be compiled or executed. When an interpreter encounters a syntax error, it stops executing the program.

Checkpoint

1.18 A CPU understands instructions that are written only in what language?

1.19 A program has to be copied into what type of memory each time the CPU executes it?

1.20 When a CPU executes the instructions in a program, it is engaged in what process?

1.21 What is assembly language?

1.22 What type of programming language allows you to create powerful and complex programs without knowing how the CPU works?

1.23 Each language has a set of rules that must be strictly followed when writing a program. What is this set of rules called?

1.24 What do you call a program that translates a high-level language program into a separate machine language program?

1.25 What do you call a program that both translates and executes the instructions in a high-level language program?

1.26 What type of mistake is usually caused by a misspelled key word, a missing punctuation character, or the incorrect use of an operator?

1.5 Using Python

CONCEPT: The Python interpreter can run Python programs that are saved in files or interactively execute Python statements that are typed at the keyboard. Python comes with a program named IDLE that simplifies the process of writing, executing, and testing programs.

Installing Python

Before you can try any of the programs shown in this book, or write any programs of your own, you need to make sure that Python is installed on your computer and properly configured. If you are working in a computer lab, this has probably been done already. If you are using your own computer, you can follow the instructions in Appendix A to download and install Python.

The Python Interpreter

You learned earlier that Python is an interpreted language. When you install the Python language on your computer, one of the items that is installed is the Python interpreter. The *Python interpreter* is a program that can read Python programming statements and execute them. (Sometimes we will refer to the Python interpreter simply as the interpreter.)

You can use the interpreter in two modes: interactive mode and script mode. In *interactive mode,* the interpreter waits for you to type Python statements on the keyboard. Once you type a statement, the interpreter executes it and then waits for you to type another statement. In *script mode,* the interpreter reads the contents of a file that contains Python statements. Such a file is known as a *Python program* or a *Python script*. The interpreter executes each statement in the Python program as it reads it.

Interactive Mode

Once Python has been installed and set up on your system, you start the interpreter in interactive mode by going to the operating system's command line and typing the following command:

```
python
```

If you are using Windows, you can alternatively click the *Start* button, then *All Programs*. You should see a program group named something like *Python 3.3*. (The "3.3" is the version of Python that is installed. At the time this is being written, Python 3.3 is the latest version.) Inside this program group you should see an item named *Python (command line)*. Clicking this menu item will start the Python interpreter in interactive mode.

NOTE: When the Python interpreter is running in interactive mode, it is commonly called the *Python shell*.

When the Python interpreter starts in interactive mode, you will see something like the following displayed in a console window:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53)
[MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

The >>> that you see is a prompt that indicates the interpreter is waiting for you to type a Python statement. Let's try it out. One of the simplest things that you can do in Python is print a message on the screen. For example, the following statement prints the message *Python programming is fun!* on the screen:

```
print('Python programming is fun!')
```

You can think of this as a command that you are sending to the Python interpreter. If you type the statement exactly as it is shown, the message *Python programming is fun!* is printed on the screen. Here is an example of how you type this statement at the interpreter's prompt:

```
>>> print('Python programming is fun!') Enter
```

After typing the statement, you press the Enter key, and the Python interpreter executes the statement, as shown here:

```
>>> print('Python programming is fun!') Enter
Python programming is fun!
>>>
```

After the message is displayed, the >>> prompt appears again, indicating that the interpreter is waiting for you to enter another statement. Let's look at another example. In the following sample session, we have entered two statements:

```
>>> print('To be or not to be') Enter
To be or not to be
>>> print('That is the question.') Enter
That is the question.
>>>
```

If you incorrectly type a statement in interactive mode, the interpreter will display an error message. This will make interactive mode useful to you while you learn Python. As you learn new parts of the Python language, you can try them out in interactive mode and get immediate feedback from the interpreter.

To quit the Python interpreter in interactive mode on a Windows computer, press Ctrl-Z (pressing both keys together) followed by Enter. On a Mac, Linux, or UNIX computer, press Ctrl-D.

NOTE: In Chapter 2 we discuss the details of statements like the ones previously shown. If you want to try them now in interactive mode, make sure you type them exactly as shown.

Writing Python Programs and Running Them in Script Mode

Although interactive mode is useful for testing code, the statements that you enter in interactive mode are not saved as a program. They are simply executed and their results displayed on the screen. If you want to save a set of Python statements as a program, you save those statements in a file. Then, to execute the program, you use the Python interpreter in script mode.

For example, suppose you want to write a Python program that displays the following three lines of text:

```
Nudge nudge
Wink wink
Know what I mean?
```

To write the program you would use a simple text editor like Notepad (which is installed on all Windows computers) to create a file containing the following statements:

```
print('Nudge nudge')
print('Wink wink')
print('Know what I mean?')
```

NOTE: It is possible to use a word processor to create a Python program, but you must be sure to save the program as a plain text file. Otherwise the Python interpreter will not be able to read its contents.

When you save a Python program, you give it a name that ends with the `.py` extension, which identifies it as a Python program. For example, you might save the program previously shown with the name `test.py`. To run the program you would go to the directory in which the file is saved and type the following command at the operating system command line:

```
python test.py
```

This starts the Python interpreter in script mode and causes it to execute the statements in the file `test.py`. When the program finishes executing, the Python interpreter exits.

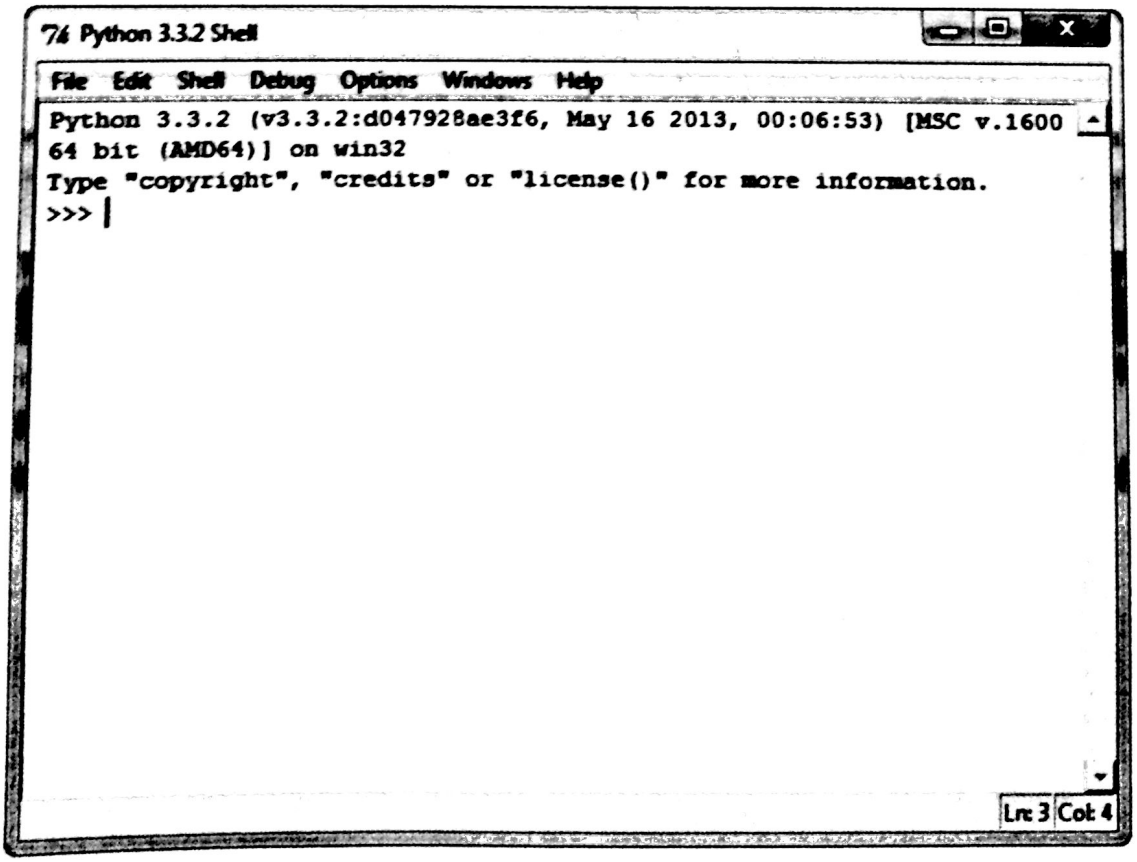
The IDLE Programming Environment

The previous sections described how the Python interpreter can be started in interactive mode or script mode at the operating system command line. As an alternative, you can use an *integrated development environment*, which is a single program that gives you all of the tools you need to write, execute, and test a program.

Recent versions of Python include a program named *IDLE*, which is automatically installed when the Python language is installed. (IDLE stands for Integrated DeveLopment Environment.) When you run IDLE, the window shown in Figure 1-20 appears. Notice that the >>> prompt appears in the IDLE window, indicating that the interpreter is running in interactive mode. You can type Python statements at this prompt and see them executed in the IDLE window.

IDLE also has a built-in text editor with features specifically designed to help you write Python programs. For example, the IDLE editor "colorizes" code so that key words and

Figure 1-20 IDLE



other parts of a program are displayed in their own distinct colors. This helps make programs easier to read. In IDLE you can write programs, save them to disk, and execute them. Appendix B provides a quick introduction to IDLE and leads you through the process of creating, saving, and executing a Python program.

NOTE: Although IDLE is installed with Python, there are several other Python IDEs available. Your instructor might prefer that you use a specific one in class.

Review Questions

Multiple Choice

1. A(n) _____ is a set of instructions that a computer follows to perform a task.
   a. compiler
   b. program
   c. interpreter
   d. programming language
2. The physical devices that a computer is made of are referred to as _____.
   a. hardware
   b. software
   c. the operating system
   d. tools
3. The part of a computer that runs programs is called _____.
   a. RAM
   b. secondary storage
   c. main memory
   d. the CPU
4. Today, CPUs are small chips known as _____.
   a. ENIACs
   b. microprocessors
   c. memory chips
   d. operating systems
5. The computer stores a program while the program is running, as well as the data that the program is working with, in _____.
   a. secondary storage
   b. the CPU
   c. main memory
   d. the microprocessor
6. This is a volatile type of memory that is used only for temporary storage while a program is running.
   a. RAM
   b. secondary storage
   c. the disk drive
   d. the USB drive

7. A type of memory that can hold data for long periods of time, even when there is no power to the computer, is called _____.
   a. RAM
   b. main memory
   c. secondary storage
   d. CPU storage
8. A component that collects data from people or other devices and sends it to the computer is called _____.
   a. an output device
   b. an input device
   c. a secondary storage device
   d. main memory
9. A video display is a(n) _____ device.
   a. output
   b. input
   c. secondary storage
   d. main memory
10. A _____ is enough memory to store a letter of the alphabet or a small number.
    a. byte
    b. bit
    c. switch
    d. transistor
11. A byte is made up of eight _____.
    a. CPUs
    b. instructions
    c. variables
    d. bits
12. In the _____ numbering system, all numeric values are written as sequences of 0s and 1s.
    a. hexadecimal
    b. binary
    c. octal
    d. decimal
13. A bit that is turned off represents the following value: _____.
    a. 1
    b. –1
    c. 0
    d. "no"
14. A set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters is _____.
    a. binary numbering
    b. ASCII
    c. Unicode
    d. ENIAC

15. An extensive encoding scheme that can represent characters for many languages in the world is _____.
    a. binary numbering
    b. ASCII
    c. Unicode
    d. ENIAC
16. Negative numbers are encoded using the _____ technique.
    a. two's complement
    b. floating point
    c. ASCII
    d. Unicode
17. Real numbers are encoded using the _____ technique.
    a. two's complement
    b. floating point
    c. ASCII
    d. Unicode
18. The tiny dots of color that digital images are composed of are called _____.
    a. bits
    b. bytes
    c. color packets
    d. pixels
19. If you were to look at a machine language program, you would see _____.
    a. Python code
    b. a stream of binary numbers
    c. English words
    d. circuits
20. In the _____ part of the fetch-decode-execute cycle, the CPU determines which operation it should perform.
    a. fetch
    b. decode
    c. execute
    d. immediately after the instruction is executed
21. Computers can only execute programs that are written in _____.
    a. Java
    b. assembly language
    c. machine language
    d. Python
22. The _____ translates an assembly language program to a machine language program.
    a. assembler
    b. compiler
    c. translator
    d. interpreter

23. The words that make up a high-level programming language are called _____.
    a. binary instructions
    b. mnemonics
    c. commands
    d. key words
24. The rules that must be followed when writing a program are called _____.
    a. syntax
    b. punctuation
    c. key words
    d. operators
25. A(n) _____ program translates a high-level language program into a separate machine language program.
    a. assembler
    b. compiler
    c. translator
    d. utility

True or False

1. Today, CPUs are huge devices made of electrical and mechanical components such as vacuum tubes and switches.
2. Main memory is also known as RAM.
3. Any piece of data that is stored in a computer's memory must be stored as a binary number.
4. Images, like the ones you make with your digital camera, cannot be stored as binary numbers.
5. Machine language is the only language that a CPU understands.
6. Assembly language is considered a high-level language.
7. An interpreter is a program that both translates and executes the instructions in a high-level language program.
8. A syntax error does not prevent a program from being compiled and executed.
9. Windows, Linux, Android, iOS, and Mac OSX are all examples of application software.
10. Word processing programs, spreadsheet programs, email programs, web browsers, and games are all examples of utility programs.

Short Answer

1. Why is the CPU the most important component in a computer?
2. What number does a bit that is turned on represent? What number does a bit that is turned off represent?
3. What would you call a device that works with binary data?
4. What are the words that make up a high-level programming language called?
5. What are the short words that are used in assembly language called?
6. What is the difference between a compiler and an interpreter?
7. What type of software controls the internal operations of the computer's hardware?

Exercises

1. To make sure that you can interact with the Python interpreter, try the following steps on your computer:
   - Start the Python interpreter in interactive mode.
   - At the >>> prompt type the following statement and then press Enter:

     ```
     print('This is a test of the Python interpreter.') Enter
     ```

   - After pressing the Enter key the interpreter will execute the statement. If you typed everything correctly, your session should look like this:

     ```
     >>> print('This is a test of the Python interpreter.') Enter
     This is a test of the Python interpreter.
     >>>
     ```

   - If you see an error message, enter the statement again, and make sure you type it exactly as shown.
   - Exit the Python interpreter. (In Windows, press Ctrl-Z followed by Enter. On other systems press Ctrl-D.)


VideoNote
Performing
Exercise 2

2. To make sure that you can interact with IDLE, try the following steps on your computer:
   - Start IDLE. To do this in Windows, click the *Start* button, then *All Programs*. In the Python program group click *IDLE (Python GUI)*.
   - When IDLE starts, it should appear similar to the window previously shown in Figure 1-20. At the >>> prompt type the following statement and then press Enter:

     ```
     print('This is a test of IDLE.') Enter
     ```

   - After pressing the Enter key the Python interpreter will execute the statement. If you typed everything correctly, your session should look like this:

     ```
     >>> print('This is a test of IDLE.') Enter
     This is a test of IDLE.
     >>>
     ```

   - If you see an error message, enter the statement again and make sure you type it exactly as shown.
   - Exit IDLE by clicking File, then Exit (or pressing Ctrl-Q on the keyboard).
3. Use what you've learned about the binary numbering system in this chapter to convert the following decimal numbers to binary:

   11

   65

   100

   255
4. Use what you've learned about the binary numbering system in this chapter to convert the following binary numbers to decimal:

   1101

   1000

   101011

5. Look at the ASCII chart in Appendix C and determine the codes for each letter of your first name.
6. Use the Internet to research the history of the Python programming language, and answer the following questions:
   - Who was the creator of Python?
   - When was Python created?
   - In the Python programming community, the person who created Python is commonly referred to as the "BDFL." What does this mean?

5. Look at the ASCII chart in Appendix C and determine the codes for each letter of your first name.
6. Use the Internet to research the history of the Python programming language, and answer the following questions:
   - Who was the creator of Python?
   - When was Python created?
   - In the Python programming community, the person who created Python is commonly referred to as the "BDFL." What does this mean?

CHAPTER 2

Input, Processing, and Output

TOPICS

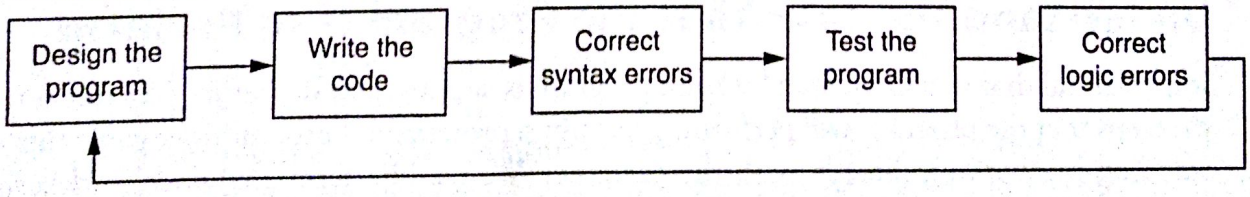
2.1 Designing a Program

CONCEPT: Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudocode and flowcharts to create models of programs.

The Program Development Cycle

In Chapter 1 you learned that programmers typically use high-level languages such as Python to create programs. There is much more to creating a program than writing code, however. The process of creating a program that works correctly typically requires the five phases shown in Figure 2-1. The entire process is known as the *program development cycle*.

Figure 2-1 The program development cycle



Let's take a closer look at each stage in the cycle.

1. **Design the Program** All professional programmers will tell you that a program should be carefully designed before the code is actually written. When programmers begin

a new project, they never jump right in and start writing code as the first step. They start by creating a design of the program. There are several ways to design a program, and later in this section we will discuss some techniques that you can use to design your Python programs.

2. **Write the Code** After designing the program, the programmer begins writing code in a high-level language such as Python. Recall from Chapter 1 that each language has its own rules, known as syntax, that must be followed when writing a program. A language's syntax rules dictate things such as how key words, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.
3. **Correct Syntax Errors** If the program contains a syntax error, or even a simple mistake such as a misspelled key word, the compiler or interpreter will display an error message indicating what the error is. Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all of the syntax errors and simple typing mistakes have been corrected, the program can be compiled and translated into a machine language program (or executed by an interpreter, depending on the language being used).
4. **Test the Program** Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A *logic error* is a mistake that does not prevent the program from running, but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.)
5. **Correct Logic Errors** If the program produces incorrect results, the programmer *debugs* the code. This means that the programmer finds and corrects logic errors in the program. Sometimes during this process, the programmer discovers that the program's original design must be changed. In this event, the program development cycle starts over and continues until no errors can be found.

More About the Design Process

The process of designing a program is arguably the most important part of the cycle. You can think of a program's design as its foundation. If you build a house on a poorly constructed foundation, eventually you will find yourself doing a lot of work to fix the house! A program's design should be viewed no differently. If your program is designed poorly, eventually you will find yourself doing a lot of work to fix the program.

The process of designing a program can be summarized in the following two steps:

1. Understand the task that the program is to perform.
2. Determine the steps that must be taken to perform the task.

Let's take a closer look at each of these steps.

Understand the Task That the Program Is to Perform

It is essential that you understand what a program is supposed to do before you can determine the steps that the program will perform. Typically, a professional programmer gains this understanding by working directly with the customer. We use the term *customer* to describe the person, group, or organization that is asking you to write a program. This could be a customer in the traditional sense of the word, meaning someone who is paying you to write a program. It could also be your boss, or the manager of a department within your company. Regardless of whom it is, the customer will be relying on your program to perform an important task.

To get a sense of what a program is supposed to do, the programmer usually interviews the customer. During the interview, the customer will describe the task that the program should perform, and the programmer will ask questions to uncover as many details as possible about the task. A follow-up interview is usually needed because customers rarely mention everything they want during the initial meeting, and programmers often think of additional questions.

The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements. A *software requirement* is simply a single task that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.

TIP: If you choose to become a professional software developer, your customer will be anyone who asks you to write programs as part of your job. As long as you are a student, however, your customer is your instructor! In every programming class that you will take, it's practically guaranteed that your instructor will assign programming problems for you to complete. For your academic success, make sure that you understand your instructor's requirements for those assignments and write your programs accordingly.

Determine the Steps That Must Be Taken to Perform the Task

Once you understand the task that the program will perform, you begin by breaking down the task into a series of steps. This is similar to the way you would break down a task into a series of steps that another person can follow. For example, suppose someone asks you how to boil water. You might break down that task into a series of steps as follows:

1. Pour the desired amount of water into a pot.
2. Put the pot on a stove burner.
3. Turn the burner to high.
4. Watch the water until you see large bubbles rapidly rising. When this happens, the water is boiling.

This is an example of an *algorithm*, which is a set of well-defined logical steps that must be taken to perform a task. Notice that the steps in this algorithm are sequentially ordered. Step 1 should be performed before step 2, and so on. If a person follows these steps exactly as they appear, and in the correct order, he or she should be able to boil water successfully.

A programmer breaks down the task that a program must perform in a similar way. An algorithm is created, which lists all of the logical steps that must be taken. For example, suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee. Here are the steps that you would take:

1. Get the number of hours worked.
2. Get the hourly pay rate.
3. Multiply the number of hours worked by the hourly pay rate.
4. Display the result of the calculation that was performed in step 3.

Of course, this algorithm isn't ready to be executed on the computer. The steps in this list have to be translated into code. Programmers commonly use two tools to help them accomplish this: pseudocode and flowcharts. Let's look at each of these in more detail.

Pseudocode

Because small mistakes like misspelled words and forgotten punctuation characters can cause syntax errors, programmers have to be mindful of such small details when writing code. For this reason, programmers find it helpful to write a program in pseudocode (pronounced "sue doe code") before they write it in the actual code of a programming language such as Python.

The word "pseudo" means fake, so *pseudocode* is fake code. It is an informal language that has no syntax rules and is not meant to be compiled or executed. Instead, programmers use pseudocode to create models, or "mock-ups," of programs. Because programmers don't have to worry about syntax errors while writing pseudocode, they can focus all of their attention on the program's design. Once a satisfactory design has been created with pseudocode, the pseudocode can be translated directly to actual code. Here is an example of how you might write pseudocode for the pay calculating program that we discussed earlier:

Input the hours worked
Input the hourly pay rate
Calculate gross pay as hours worked multiplied by pay rate
Display the gross pay

Each statement in the pseudocode represents an operation that can be performed in Python. For example, Python can read input that is typed on the keyboard, perform mathematical calculations, and display messages on the screen.

Flowcharts

Flowcharting is another tool that programmers use to design programs. A *flowchart* is a diagram that graphically depicts the steps that take place in a program. Figure 2-2 shows how you might create a flowchart for the pay calculating program.

Notice that there are three types of symbols in the flowchart: ovals, parallelograms, and a rectangle. Each of these symbols represents a step in the program, as described here:

- The ovals, which appear at the top and bottom of the flowchart, are called *terminal symbols*. The *Start* terminal symbol marks the program's starting point and the *End* terminal symbol marks the program's ending point.
- Parallelograms are used as *input symbols* and *output symbols*. They represent steps in which the program reads input or displays output.
- Rectangles are used as *processing symbols*. They represent steps in which the program performs some process on data, such as a mathematical calculation.

The symbols are connected by arrows that represent the "flow" of the program. To step through the symbols in the proper order, you begin at the *Start* terminal and follow the arrows until you reach the *End* terminal.

Checkpoint

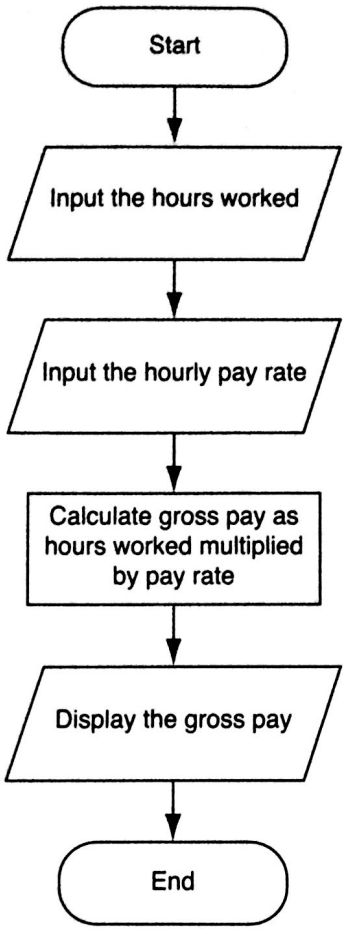
2.1 Who is a programmer's customer?

2.2 What is a software requirement?

2.3 What is an algorithm?

2.4 What is pseudocode?

Figure 2-2 Flowchart for the pay calculating program



2.5 What is a flowchart?

2.6 What do each of the following symbols mean in a flowchart?
- Oval
- Parallelogram
- Rectangle

2.2 Input, Processing, and Output

CONCEPT: **Input is data that the program receives. When a program receives data, it usually processes it by performing some operation with it. The result of the operation is sent out of the program as output.**

Computer programs typically perform the following three-step process:

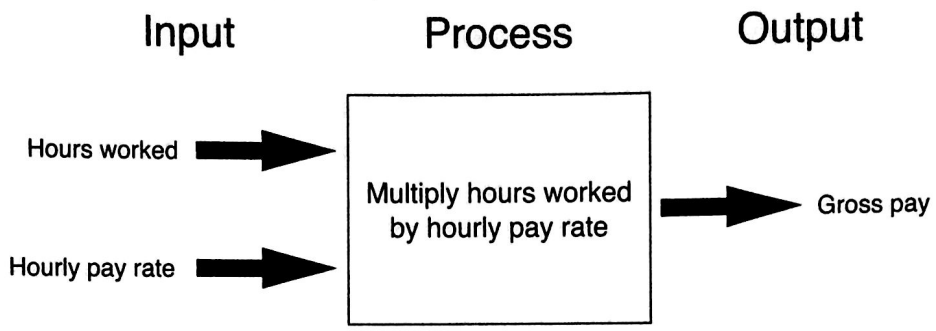
1. Input is received.
2. Some process is performed on the input.
3. Output is produced.

Input is any data that the program receives while it is running. One common form of input is data that is typed on the keyboard. Once input is received, some process, such as

a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output.

Figure 2-3 illustrates these three steps in the pay calculating program that we discussed earlier. The number of hours worked and the hourly pay rate are provided as input. The program processes this data by multiplying the hours worked by the hourly pay rate. The results of the calculation are then displayed on the screen as output.

Figure 2-3 The input, processing, and output of the pay calculating program



In this chapter we will discuss basic ways that you can perform input, processing, and output using Python.

2.3 Displaying Output with the `print` Function

CONCEPT: You use the `print` function to display output in a Python program.


VideoNote
The `print` Function

A *function* is a piece of prewritten code that performs an operation. Python has numerous built-in functions that perform various operations. Perhaps the most fundamental built-in function is the `print` function, which displays output on the screen. Here is an example of a statement that executes the `print` function:

```
print('Hello world')
```

In interactive mode, if you type this statement and press the Enter key, the message *Hello world* is displayed. Here is an example:

```
>>> print('Hello world')
Hello world
>>>
```

When programmers execute a function, they say that they are *calling* the function. When you call the `print` function, you type the word `print`, followed by a set of parentheses. Inside the parentheses, you type an *argument*, which is the data that you want displayed on the screen. In the previous example, the argument is `'Hello world'`. Notice that the quote marks are not displayed when the statement executes. The quote marks simply specify the beginning and the end of the text that you wish to display.

Suppose your instructor tells you to write a program that displays your name and address on the computer screen. Program 2-1 shows an example of such a program, with the output that it will produce when it runs. (The line numbers that appear in a program listing in

this book are *not* part of the program. We use the line numbers in our discussion to refer to parts of the program.)

Program 2-1 (output.py)

```
1 print('Kate Austen')
2 print('123 Full Circle Drive')
3 print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

It is important to understand that the statements in this program execute in the order that they appear, from the top of the program to the bottom. When you run this program, the first statement will execute, followed by the second statement, and followed by the third statement.

Strings and String Literals

Programs almost always work with data of some type. For example, Program 2-1 uses the following three pieces of data:

```
'Kate Austen'
'123 Full Circle Drive
'Asheville, NC 28899'
```

These pieces of data are sequences of characters. In programming terms, a sequence of characters that is used as data is called a *string*. When a string appears in the actual code of a program it is called a *string literal*. In Python code, string literals must be enclosed in quote marks. As mentioned earlier, the quote marks simply mark where the string data begins and ends.

In Python you can enclose string literals in a set of single-quote marks (') or a set of double-quote marks ("). The string literals in Program 2-1 are enclosed in single-quote marks, but the program could also be written as shown in Program 2-2.

Program 2-2 (double_quotes.py)

```
1 print("Kate Austen")
2 print("123 Full Circle Drive")
3 print("Asheville, NC 28899")
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

If you want a string literal to contain either a single-quote or an apostrophe as part of the string, you can enclose the string literal in double-quote marks. For example, Program 2-3 prints two strings that contain apostrophes.

Program 2-3 (apostrophe.py)

```
1 print("Don't fear!")
2 print("I'm here!")
```

Program Output

```
Don't fear!
I'm here!
```

Likewise, you can use single-quote marks to enclose a string literal that contains double-quotes as part of the string. Program 2-4 shows an example.

Program 2-4 (display_quote.py)

```
1 print('Your assignment is to read "Hamlet" by tomorrow.')
```

Program Output

```
Your assignment is to read "Hamlet" by tomorrow.
```

Python also allows you to enclose string literals in triple quotes (either """ or '''). Triple-quoted strings can contain both single quotes and double quotes as part of the string. The following statement shows an example:

```
print("""I'm reading "Hamlet" tonight.""")
```

This statement will print

```
I'm reading "Hamlet" tonight.
```

Triple quotes can also be used to surround multiline strings, something for which single and double quotes cannot be used. Here is an example:

```
print("""One
Two
Three""")
```

This statement will print

```
One
Two
Three
```

Checkpoint

2.7 Write a statement that displays your name.

2.8 Write a statement that displays the following text:

```
Python's the best!
```

2.9 Write a statement that displays the following text:

```
The cat said "meow."
```

2.4 Comments

CONCEPT: **Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the Python interpreter ignores them. They are intended for people who may be reading the source code.**

Comments are short notes placed in different parts of a program, explaining how those parts of the program work. Although comments are a critical part of a program, they are ignored by the Python interpreter. Comments are intended for any person reading a program's code, not the computer.

In Python you begin a comment with the # character. When the Python interpreter sees a # character, it ignores everything from that character to the end of the line. For example, look at Program 2-5. Lines 1 and 2 are comments that briefly explain the program's purpose.

Program 2-5 (comment1.py)

```
1 # This program displays a person's
2 # name and address.
3 print('Kate Austen')
4 print('123 Full Circle Drive')
5 print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Programmers commonly write end-line comments in their code. An *end-line comment* is a comment that appears at the end of a line of code. It usually explains the statement that appears in that line. Program 2-6 shows an example. Each line ends with a comment that briefly explains what the line does.

Program 2-6 (comment2.py)

```
1 print('Kate Austen')              # Display the name.
2 print('123 Full Circle Drive')    # Display the address.
3 print('Asheville, NC 28899')      # Display the city, state, and ZIP.
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

As a beginning programmer, you might be resistant to the idea of liberally writing comments in your programs. After all, it can seem more productive to write code that actually does something! It is crucial that you take the extra time to write comments, however. They will almost certainly save you and others time in the future when you have to modify or debug the program. Large and complex programs can be almost impossible to read and understand if they are not properly commented.

2.5 Variables

CONCEPT: **A variable is a name that represents a value stored in the computer's memory.**

Programs usually store data in the computer's memory and perform operations on that data. For example, consider the typical online shopping experience: you browse a website and add the items that you want to purchase to the shopping cart. As you add items to the shopping cart, data about those items is stored in memory. Then, when you click the checkout button, a program running on the website's computer calculates the cost of all the items you have in your shopping cart, applicable sales taxes, shipping costs, and the total of all these charges. When the program performs these calculations, it stores the results in the computer's memory.

Programs use variables to access and manipulate data that is stored in memory. A *variable* is a name that represents a value in the computer's memory. For example, a program that calculates the sales tax on a purchase might use the variable name `tax` to represent that value in memory. And a program that calculates the distance between two cities might use the variable name `distance` to represent that value in memory. When a variable represents a value in the computer's memory, we say that the variable *references* the value.

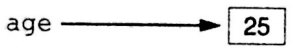
Creating Variables with Assignment Statements

You use an *assignment statement* to create a variable and make it reference a piece of data. Here is an example of an assignment statement:

```
age = 25
```

After this statement executes, a variable named age will be created, and it will reference the value 25. This concept is shown in Figure 2-4. In the figure, think of the value 25 as being stored somewhere in the computer's memory. The arrow that points from age to the value 25 indicates that the name age references the value.

Figure 2-4 The age variable references the value 25



An assignment statement is written in the following general format:

```
variable = expression
```

The equal sign (=) is known as the *assignment operator*. In the general format, `variable` is the name of a variable and `expression` is a value, or any piece of code that results in a value. After an assignment statement executes, the variable listed on the left side of the = operator will reference the value given on the right side of the = operator.

To experiment with variables, you can type assignment statements in interactive mode, as shown here:

```
>>> width = 10 Enter
>>> length = 5 Enter
>>>
```

The first statement creates a variable named `width` and assigns it the value 10. The second statement creates a variable named `length` and assigns it the value 5. Next, you can use the `print` function to display the values referenced by these variables, as shown here:

```
>>> print(width) Enter
10
>>> print(length) Enter
5
>>>
```

When you pass a variable as an argument to the `print` function, you do not enclose the variable name in quote marks. To demonstrate why, look at the following interactive session:

```
>>> print('width') Enter
width
>>> print(width) Enter
10
>>>
```

In the first statement, you passed `'width'` as an argument to the `print` function, and the function printed the string *width*. In the second statement, you passed `width` (with no quote marks) as an argument to the `print` function, and the function displayed the value referenced by the `width` variable.

In an assignment statement, the variable that is receiving the assignment must appear on the left side of the = operator. As shown in the following interactive session, an error occurs if the item on the left side of the = operator is not a variable:

```
>>> 25 = age Enter
SyntaxError: can't assign to literal
>>>
```

The code in Program 2-7 demonstrates a variable. Line 2 creates a variable named room and assigns it the value 503. The statements in lines 3 and 4 display a message. Notice that line 4 displays the value that is referenced by the room variable.

Program 2-7 (variable_demo.py)

```
1 # This program demonstrates a variable.
2 room = 503
3 print('I am staying in room number')
4 print(room)
```

Program Output

```
I am staying in room number
503
```

Program 2-8 shows a sample program that uses two variables. Line 2 creates a variable named top_speed, assigning it the value 160. Line 3 creates a variable named distance, assigning it the value 300. This is illustrated in Figure 2-5.

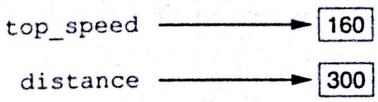
Program 2-8 (variable_demo2.py)

```
1 # Create two variables: top_speed and distance.
2 top_speed = 160
3 distance = 300
4
5 # Display the values referenced by the variables.
6 print('The top speed is')
7 print(top_speed)
8 print('The distance traveled is')
9 print(distance)
```

Program Output

```
The top speed is
160
The distance traveled is
300
```

Figure 2-5 Two variables



WARNING! You cannot use a variable until you have assigned a value to it. An error will occur if you try to perform an operation on a variable, such as printing it, before it has been assigned a value.

Sometimes a simple typing mistake will cause this error. One example is a misspelled variable name, as shown here:

```
temperature = 74.5 # Create a variable
print(tempereture) # Error! Misspelled variable name
```

In this code, the variable `temperature` is created by the assignment statement. The variable name is spelled differently in the `print` statement, however, which will cause an error. Another example is the inconsistent use of uppercase and lowercase letters in a variable name. Here is an example:

```
temperature = 74.5 # Create a variable
print(Temperature) # Error! Inconsistent use of case
```

In this code the variable `temperature` (in all lowercase letters) is created by the assignment statement. In the `print` statement, the name `Temperature` is spelled with an uppercase T. This will cause an error because variable names are case sensitive in Python.

Variable Naming Rules

Although you are allowed to make up your own names for variables, you must follow these rules:

- You cannot use one of Python's key words as a variable name. (See Table 1-2 for a list of the key words.)
- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means the variable name `ItemsOrdered` is not the same as `itemsordered`.

In addition to following these rules, you should always choose names for your variables that give an indication of what they are used for. For example, a variable that holds the temperature might be named `temperature`, and a variable that holds a car's speed might be named `speed`. You may be tempted to give variables names like `x` and `b2`, but names like these give no clue as to what the variable's purpose is.

Because a variable's name should reflect the variable's purpose, programmers often find themselves creating names that are made of multiple words. For example, consider the following variable names:

```
grosspay
payrate
hotdogssoldtoday
```

Unfortunately, these names are not easily read by the human eye because the words aren't separated. Because we can't have spaces in variable names, we need to find another way to separate the words in a multiword variable name and make it more readable to the human eye.

One way to do this is to use the underscore character to represent a space. For example, the following variable names are easier to read than those previously shown:

```
gross_pay
pay_rate
hot_dogs_sold_today
```

This style of naming variables is popular among Python programmers and is the style we will use in this book. There are other popular styles, however, such as the *camelCase* naming convention. camelCase names are written in the following manner:

- The variable name begins with lowercase letters.
- The first character of the second and subsequent words is written in uppercase.

For example, the following variable names are written in camelCase:

```
grossPay
payRate
hotDogsSoldToday
```

NOTE: This style of naming is called camelCase because the uppercase characters that appear in a name may suggest a camel's humps.

Table 2-1 lists several sample variable names and indicates whether each is legal or illegal in Python.

Table 2-1 Sample variable names

Variable Name	Legal or Illegal?
`units_per_day`	Legal
`dayOfWeek`	Legal
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`June1997`	Legal
`Mixture#3`	Illegal. Variable names may only use letters, digits, or underscores.

Displaying Multiple Items with the `print` Function

If you refer to Program 2-7 you will see that we used the following two statements in lines 3 and 4:

```
print('I am staying in room number')
print(room)
```

We called the `print` function twice because we needed to display two pieces of data. Line 3 displays the string literal `'I am staying in room number'`, and line 4 displays the value referenced by the `room` variable.

This program can be simplified, however, because Python allows us to display multiple items with one call to the `print` function. We simply have to separate the items with commas as shown in Program 2-9.

Program 2-9 (variable_demo3.py)

```
1 # This program demonstrates a variable.
2 room = 503
3 print('I am staying in room number', room)
```

Program Output

```
I am staying in room number 503
```

In line 3 we passed two arguments to the `print` function. The first argument is the string literal `'I am staying in room number'`, and the second argument is the `room` variable. When the `print` function executed, it displayed the values of the two arguments in the order that we passed them to the function. Notice that the `print` function automatically printed a space separating the two items. When multiple arguments are passed to the `print` function, they are automatically separated by a space when they are displayed on the screen.

Variable Reassignment

Variables are called "variable" because they can reference different values while a program is running. When you assign a value to a variable, the variable will reference that value until you assign it a different value. For example, look at Program 2-10. The statement in line 3 creates a variable named `dollars` and assigns it the value 2.75. This is shown in the top part of Figure 2-6. Then, the statement in line 8 assigns a different value, 99.95, to the `dollars` variable. The bottom part of Figure 2-6 shows how this changes the `dollars` variable. The old value, 2.75, is still in the computer's memory, but it can no longer be used because it isn't referenced by a variable. When a value in memory is no longer referenced by a variable, the Python interpreter automatically removes it from memory through a process known as *garbage collection.*

Program 2-10 (variable_demo4.py)

```
1 # This program demonstrates variable reassignment.
2 # Assign a value to the dollars variable.
3 dollars = 2.75
```

(program continues)

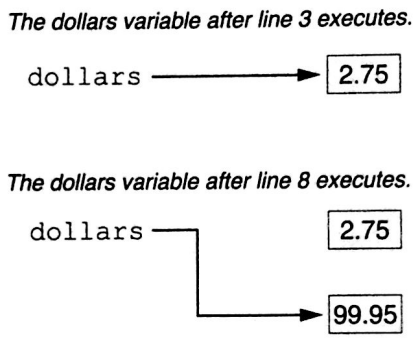
Program 2-10 *(continued)*

```
4    print('I have', dollars, 'in my account.')
5
6    # Reassign dollars so it references
7    # a different value.
8    dollars = 99.95
9    print('But now I have', dollars, 'in my account!')
```

Program Output

```
I have 2.75 in my account.
But now I have 99.95 in my account!
```

Figure 2-6 Variable reassignment in Program 2-10



Numeric Data Types and Literals

In Chapter 1 we discussed the way that computers store data in memory. (See section 1.3) You might recall from that discussion that computers use a different technique for storing real numbers (numbers with a fractional part) than for storing integers. Not only are these types of numbers stored differently in memory, but similar operations on them are carried out in different ways.

Because different types of numbers are stored and manipulated in different ways, Python uses *data types* to categorize values in memory. When an integer is stored in memory, it is classified as an `int`, and when a real number is stored in memory, it is classified as a `float`.

Let's look at how Python determines the data type of a number. Several of the programs that you have seen so far have numeric data written into their code. For example, the following statement, which appears in Program 2-9, has the number 503 written into it.

```
room = 503
```

This statement causes the value 503 to be stored in memory, and it makes the `room` variable reference it. The following statement, which appears in Program 2-10, has the number 2.75 written into it.

```
dollars = 2.75
```

This statement causes the value 2.75 to be stored in memory, and it makes the `dollars` variable reference it. A number that is written into a program's code is called a *numeric*

literal. When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:

- A numeric literal that is written as a whole number with no decimal point is considered an `int`. Examples are `7`, `124`, and `-9`.
- A numeric literal that is written with a decimal point is considered a `float`. Examples are `1.5`, `3.14159`, and `5.0`.

So, the following statement causes the number 503 to be stored in memory as an `int`:

```
room = 503
```

And the following statement causes the number 2.75 to be stored in memory as a `float`:

```
dollars = 2.75
```

When you store an item in memory, it is important for you to be aware of the item's data type. As you will see, some operations behave differently depending on the type of data involved, and some operations can only be performed on values of a specific data type.

As an experiment, you can use the built-in `type` function in interactive mode to determine the data type of a value. For example, look at the following session:

```
>>> type(1) Enter
<class 'int'>
>>>
```

In this example, the value 1 is passed as an argument to the `type` function. The message that is displayed on the next line, `<class 'int'>`, indicates that the value is an `int`. Here is another example:

```
>>> type(1.0) Enter
<class 'float'>
>>>
```

In this example, the value 1.0 is passed as an argument to the `type` function. The message that is displayed on the next line, `<class 'float'>`, indicates that the value is a `float`.

WARNING! You cannot write currency symbols, spaces, or commas in numeric literals. For example, the following statement will cause an error:

```
value = $4,567.99 # Error!
```

This statement must be written as:

```
value = 4567.99 # Correct
```

Storing Strings with the `str` Data Type

In addition to the `int` and `float` data types, Python also has a data type named `str`, which is used for storing strings in memory. The code in Program 2-11 shows how strings can be assigned to variables.

Program 2-11 (string_variable.py)

```
1 # Create variables to reference two strings.
2 first_name = 'Kathryn'
3 last_name = 'Marino'
4
5 # Display the values referenced by the variables.
6 print(first_name, last_name)
```

Program Output

```
Kathryn Marino
```

Reassigning a Variable to a Different Type

Keep in mind that in Python, a variable is a just a name that refers to a piece of data in memory. It is a mechanism that makes it easy for you, the programmer, to store and retrieve data. Internally, the Python interpreter keeps track of the variable names that you create and the pieces of data to which those variable names refer. Any time you need to retrieve one of those pieces of data, you simply use the variable name that refers to it.

A variable in Python can refer to items of any type. After a variable has been assigned an item of one type, it can be reassigned an item of a different type. To demonstrate, look at the following interactive session. (We have added line numbers for easier reference.)

```
1 >>> x = 99 Enter
2 >>> print(x) Enter
3 99
4 >>> x = 'Take me to your leader' Enter
5 >>> print(x) Enter
6 Take me to your leader.
7 >>>
```

The statement in line 1 creates a variable named x and assigns it the `int` value 99. Figure 2-7 shows how the variable x references the value 99 in memory. The statement in line 2 calls the `print` function, passing x as an argument. The output of the `print` function is shown in line 3. Then, the statement in line 4 assigns a string to the x variable. After this statement executes, the x variable no longer refers to an `int`, but to the string `'Take me to your leader'`. This is shown in Figure 2-8. Line 5 calls the `print` function again, passing x as an argument. Line 6 shows the `print` function's output.

Figure 2-7 The variable x references an integer

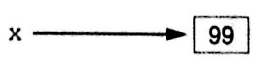
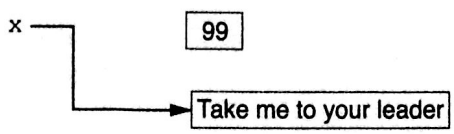


Figure 2-8 The variable x references a string



Checkpoint

2.10 What is a variable?

2.11 Which of the following are illegal variable names in Python, and why?

```
x
99bottles
july2009
theSalesFigureForFiscalYear
r&d
grade_report
```

2.12 Is the variable name `Sales` the same as `sales`? Why or why not?

2.13 Is the following assignment statement valid or invalid? If it is invalid, why?

```
72 = amount
```

2.14 What will the following code display?

```
val = 99
print('The value is', 'val')
```

2.15 Look at the following assignment statements:

```
value1 = 99
value2 = 45.9
value3 = 7.0
value4 = 7
value5 = 'abc'
```

After these statements execute, what is the Python data type of the values referenced by each variable?

2.16 What will be displayed by the following program?

```
my_value = 99
my_value = 0
print(my_value)
```

2.6 Reading Input from the Keyboard

CONCEPT: Programs commonly need to read input typed by the user on the keyboard. We will use the Python functions to do this.

Most of the programs that you will write will need to read input and then perform an operation on that input. In this section, we will discuss a basic input operation: reading data that has been typed on the keyboard. When a program reads data from the keyboard, usually it stores that data in a variable so it can be used later by the program.

In this book we use Python's built-in `input` function to read input from the keyboard. The `input` function reads a piece of data that has been entered at the keyboard and returns that piece of data, as a string, back to the program. You normally use the `input` function in an assignment statement that follows this general format:

```
variable = input(prompt)
```

In the general format, *prompt* is a string that is displayed on the screen. The string's purpose is to instruct the user to enter a value; *variable* is the name of a *variable* that references the data that was entered on the keyboard. Here is an example of a statement that uses the `input` function to read data from the keyboard:

```
name = input('What is your name? ')
```

When this statement executes, the following things happen:

- The string `'What is your name? '` is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard and then to press the Enter key.
- When the Enter key is pressed, the data that was typed is returned as a string and assigned to the `name` variable.

To demonstrate, look at the following interactive session:

```
>>> name = input('What is your name? ') Enter
What is your name? Holly Enter
>>> print(name) Enter
Holly
>>>
```

When the first statement was entered, the interpreter displayed the prompt `'What is your name? '` and waited for the user to enter some data. The user entered *Holly* and pressed the Enter key. As a result, the string `'Holly'` was assigned to the `name` variable. When the second statement was entered, the interpreter displayed the value referenced by the `name` variable.

Program 2-12 shows a complete program that uses the `input` function to read two strings as input from the keyboard.

Program 2-12 (string_input.py)

```
1 # Get the user's first name.
2 first_name = input('Enter your first name: ')
3
4 # Get the user's last name.
5 last_name = input('Enter your last name: ')
6
7 # Print a greeting to the user.
8 print('Hello', first_name, last_name)
```

Program Output (with input shown in bold)

```
Enter your first name: Vinny Enter
Enter your last name: Brown Enter
Hello Vinny Brown
```

Take a closer look in line 2 at the string we used as a prompt:

```
'Enter your first name: '
```

Notice that the last character in the string, inside the quote marks, is a space. The same is true for the following string, used as prompt in line 5:

```
'Enter your last name: '
```

We put a space character at the end of each string because the `input` function does not automatically display a space after the prompt. When the user begins typing characters, they appear on the screen immediately after the prompt. Making the last character in the prompt a space visually separates the prompt from the user's input on the screen.

Reading Numbers with the `input` Function

The `input` function always returns the user's input as a string, even if the user enters numeric data. For example, suppose you call the `input` function, type the number 72, and press the Enter key. The value that is returned from the `input` function is the string `'72'`. This can be a problem if you want to use the value in a math operation. Math operations can be performed only on numeric values, not strings.

Fortunately, Python has built-in functions that you can use to convert a string to a numeric type. Table 2-2 summarizes two of these functions.

Table 2-2 Data Conversion Functions

Function	Description
`int(item)`	You pass an argument to the `int()` function and it returns the argument's value converted to an `int`.
`float(item)`	You pass an argument to the `float()` function and it returns the argument's value converted to a `float`.

For example, suppose you are writing a payroll program and you want to get the number of hours that the user has worked. Look at the following code:

```
string_value = input('How many hours did you work? ')
hours = int(string_value)
```

The first statement gets the number of hours from the user and assigns that value as a string to the `string_value` variable. The second statement calls the `int()` function, passing `string_value` as an argument. The value referenced by `string_value` is converted to an `int` and assigned to the `hours` variable.

This example illustrates how the `int()` function works, but it is inefficient because it creates two variables: one to hold the string that is returned from the `input` function and another to hold the integer that is returned from the `int()` function. The following code shows a better approach. This one statement does all the work that the previously shown two statements do, and it creates only one variable:

```
hours = int(input('How many hours did you work? '))
```

This one statement uses *nested function* calls. The value that is returned from the `input` function is passed as an argument to the `int()` function. This is how it works:

- It calls the `input` function to get a value entered at the keyboard.
- The value that is returned from the `input` function (a string) is passed as an argument to the `int()` function.
- The `int` value that is returned from the `int()` function is assigned to the `hours` variable.

After this statement executes, the `hours` variable is assigned the value entered at the keyboard, converted to an `int`.

Let's look at another example. Suppose you want to get the user's hourly pay rate. The following statement prompts the user to enter that value at the keyboard, converts the value to a `float`, and assigns it to the `pay_rate` variable:

```
pay_rate = float(input('What is your hourly pay rate? '))
```

This is how it works:

- It calls the `input` function to get a value entered at the keyboard.
- The value that is returned from the `input` function (a string) is passed as an argument to the `float()` function.
- The `float` value that is returned from the `float()` function is assigned to the `pay_rate` variable.

After this statement executes, the `pay_rate` variable is assigned the value entered at the keyboard, converted to a `float`.

Program 2-13 shows a complete program that uses the `input` function to read a string, an `int`, and a `float`, as input from the keyboard.

Program 2-13 (input.py)

```
1  # Get the user's name, age, and income.
2  name = input('What is your name? ')
3  age = int(input('What is your age? '))
4  income = float(input('What is your income? '))
5
6  # Display the data.
7  print('Here is the data you entered:')
8  print('Name:', name)
9  print('Age:', age)
10 print('Income:', income)
```

Program Output (with input shown in bold)

```
What is your name? Chris [Enter]
What is your age? 25 [Enter]
What is your income? 75000.0
Here is the data you entered:
Name: Chris
Age: 25
Income: 75000.0
```

Let's take a closer look at the code:

- Line 2 prompts the user to enter his or her name. The value that is entered is assigned, as a string, to the `name` variable.
- Line 3 prompts the user to enter his or her age. The value that is entered is converted to an `int` and assigned to the `age` variable.
- Line 4 prompts the user to enter his or her income. The value that is entered is converted to a `float` and assigned to the `income` variable.
- Lines 7 through 10 display the values that the user entered.

The `int()` and `float()` functions work only if the item that is being converted contains a valid numeric value. If the argument cannot be converted to the specified data type, an error known as an exception occurs. An *exception* is an unexpected error that occurs while a program is running, causing the program to halt if the error is not properly dealt with. For example, look at the following interactive mode session:

```
>>> age = int(input('What is your age? ')) Enter
What is your age? xyz Enter
Traceback (most recent call last):
    File "<pyshell#81>", line 1, in <module>
        age = int(input('What is your age? '))
ValueError: invalid literal for int() with base 10: 'xyz'
>>>
```

NOTE: In this section, we mentioned the user. The *user* is simply any hypothetical person that is using a program and providing input for it. The user is sometimes called the *end user*.

Checkpoint

2.17 You need the user of a program to enter a customer's last name. Write a statement that prompts the user to enter this data and assigns the input to a variable.

2.18 You need the user of a program to enter the amount of sales for the week. Write a statement that prompts the user to enter this data and assigns the input to a variable.

2.7 Performing Calculations

CONCEPT: Python has numerous operators that can be used to perform mathematical calculations.

Most real-world algorithms require calculations to be performed. A programmer's tools for performing calculations are *math operators*. Table 2-3 lists the math operators that are provided by the Python language.

Programmers use the operators shown in Table 2-3 to create math expressions. A *math expression* performs a calculation and gives a value. The following is an example of a simple math expression:

```
12 + 2
```

Table 2-3 Python math operators

Symbol	Operation	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the result as a floating-point number
//	Integer division	Divides one number by another and gives the result as an integer
%	Remainder	Divides one number by another and gives the remainder
**	Exponent	Raises a number to a power

The values on the right and left of the + operator are called *operands*. These are values that the + operator adds together. If you type this expression in interactive mode, you will see that it gives the value 14:

```
>>> 12 + 2 Enter
14
>>>
```

Variables may also be used in a math expression. For example, suppose we have two variables named `hours` and `pay_rate`. The following math expression uses the `*` operator to multiply the value referenced by the `hours` variable by the value referenced by the `pay_rate` variable:

```
hours * pay_rate
```

When we use a math expression to calculate a value, normally we want to save that value in memory so we can use it again in the program. We do this with an assignment statement. Program 2-14 shows an example.

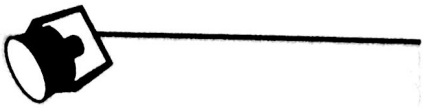
Program 2-14 (simple_math.py)

```
1  # Assign a value to the salary variable.
2  salary = 2500.0
3
4  # Assign a value to the bonus variable.
5  bonus = 1200.0
6
7  # Calculate the total pay by adding salary
8  # and bonus. Assign the result to pay.
9  pay = salary + bonus
10
11 # Display the pay.
12 print('Your pay is', pay)
```

Program Output

```
Your pay is 3700.0
```

Line 2 assigns 2500.0 to the `salary` variable, and line 5 assigns 1200.0 to the `bonus` variable. Line 9 assigns the result of the expression `salary + bonus` to the `pay` variable. As you can see from the program output, the `pay` variable holds the value 3700.0.



In the Spotlight:

Calculating a Percentage

If you are writing a program that works with a percentage, you have to make sure that the percentage's decimal point is in the correct location before doing any math with the percentage. This is especially true when the user enters a percentage as input. Most users enter the number 50 to mean 50 percent, 20 to mean 20 percent, and so forth. Before you perform any calculations with such a percentage, you have to divide it by 100 to move its decimal point two places to the left.

Let's step through the process of writing a program that calculates a percentage. Suppose a retail business is planning to have a storewide sale where the prices of all items will be 20 percent off. We have been asked to write a program to calculate the sale price of an item after the discount is subtracted. Here is the algorithm:

1. *Get the original price of the item.*
2. *Calculate 20 percent of the original price. This is the amount of the discount.*
3. *Subtract the discount from the original price. This is the sale price.*
4. *Display the sale price.*

In step 1 we get the original price of the item. We will prompt the user to enter this data on the keyboard. In our program we will use the following statement to do this. Notice that the value entered by the user will be stored in a variable named `original_price`.

```
original_price = float(input("Enter the item's original price: "))
```

In step 2, we calculate the amount of the discount. To do this we multiply the original price by 20 percent. The following statement performs this calculation and assigns the result to the `discount` variable.

```
discount = original_price * 0.2
```

In step 3, we subtract the discount from the original price. The following statement does this calculation and stores the result in the `sale_price` variable.

```
sale_price = original_price - discount
```

Last, in step 4, we will use the following statement to display the sale price:

```
print('The sale price is', sale_price)
```

Program 2-15 shows the entire program, with example output.

Program 2-15 (sale_price.py)

```
1 # This program gets an item's original price and
2 # calculates its sale price, with a 20% discount.
3
```

(program continues)

Program 2-15 *(continued)*

```
4  # Get the item's original price.
5  original_price = float(input("Enter the item's original price: "))
6
7  # Calculate the amount of the discount.
8  discount = original_price * 0.2
9
10 # Calculate the sale price.
11 sale_price = original_price - discount
12
13 # Display the sale price.
14 print('The sale price is', sale_price)
```

Program Output (with input shown in bold)

```
Enter the item's original price: 100.00 Enter
The sale price is 80.0
```

Floating-Point and Integer Division

Notice in Table 2-3 that Python has two different division operators. The / operator performs floating-point division, and the // operator performs integer division. Both operators divide one number by another. The difference between them is that the / operator gives the result as a floating-point value, and the // operator gives the result as an integer. Let's use the interactive mode interpreter to demonstrate:

```
>>> 5 / 2 Enter
2.5
>>>
```

In this session we used the / operator to divide 5 by 2. As expected, the result is 2.5. Now let's use the // operator to perform integer division:

```
>>> 5 // 2 Enter
2
>>>
```

As you can see, the result is 2. The // operator works like this:

- When the result is positive, it is *truncated*, which means that its fractional part is thrown away.
- When the result is negative, it is rounded *away from zero* to the nearest integer.

The following interactive session demonstrates how the // operator works when the result is negative:

```
>>> -5 // 2 Enter
-3
>>>
```

Operator Precedence

You can write statements that use complex mathematical expressions involving several operators. The following statement assigns the sum of 17, the variable x, 21, and the variable y to the variable answer.

```
answer = 17 + x + 21 + y
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12.0 + 6.0 / 3.0
```

What value will be assigned to outcome? The number 6.0 might be used as an operand for either the addition or division operator. The outcome variable could be assigned either 6.0 or 14.0, depending on when the division takes place. Fortunately, the answer can be predicted because Python follows the same order of operations that you learned in math class.

First, operations that are enclosed in parentheses are performed first. Then, when two operators share an operand, the operator with the higher *precedence* is applied first. The precedence of the math operators, from highest to lowest, are:

1. Exponentiation: **
2. Multiplication, division, and remainder: * / // %
3. Addition and subtraction: + -

Notice that the multiplication (*), floating-point division (/), integer division (//), and remainder (%) operators have the same precedence. The addition (+) and subtraction (-) operators also have the same precedence. When two operators with the same precedence share an operand, the operators execute from left to right.

Now, let's go back to the previous math expression:

```
outcome = 12.0 + 6.0 / 3.0
```

The value that will be assigned to outcome is 14.0 because the division operator has a higher *precedence* than the addition operator. As a result, the division takes place before the addition. The expression can be diagrammed as shown in Figure 2-9.

Figure 2-9 Operator precedence

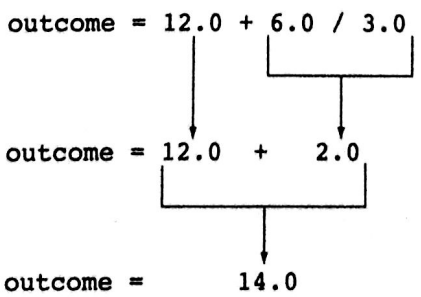


Table 2-4 shows some other sample expressions with their values.

Table 2-4 Some expressions

Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2
`8 + 12 * 2 - 4`	28
`6 - 3 * 2 + 7 - 1`	6

NOTE: There is an exception to the left-to-right rule. When two ** operators share an operand, the operators execute right-to-left. For example, the expression 2**3**4 is evaluated as 2**(3**4).

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the variables a and b are added together, and their sum is divided by 4:

```
result = (a + b) / 4
```

Without the parentheses, however, b would be divided by 4 and the result added to a. Table 2-5 shows more expressions and their values.

Table 2-5 More expressions and their values

Expression	Value
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5
`8 + 12 * (6 - 2)`	56
`(6 - 3) * (2 + 7) / 3`	9

In the Spotlight:

Calculating an Average

Determining the average of a group of values is a simple calculation: add all of the values and then divide the sum by the number of values. Although this is a straightforward calculation, it is easy to make a mistake when writing a program that calculates an average. For example, let's assume that the variables a, b, and c each hold a value and we want to calculate the average of those values. If we are careless, we might write a statement such as the following to perform the calculation:

```
average = a + b + c / 3.0
```

Can you see the error in this statement? When it executes, the division will take place first. The value in c will be divided by 3, and then the result will be added to a + b. That is not the correct way to calculate an average. To correct this error we need to put parentheses around a + b + c, as shown here:

```
average = (a + b + c) / 3.0
```

Let's step through the process of writing a program that calculates an average. Suppose you have taken three tests in your computer science class, and you want to write a program that will display the average of the test scores. Here is the algorithm:

1. *Get the first test score.*
2. *Get the second test score.*
3. *Get the third test score.*
4. *Calculate the average by adding the three test scores and dividing the sum by 3.*
5. *Display the average.*

In steps 1, 2, and 3 we will prompt the user to enter the three test scores. We will store those test scores in the variables `test1`, `test2`, and `test3`. In step 4 we will calculate the average of the three test scores. We will use the following statement to perform the calculation and store the result in the `average` variable:

```
average = (test1 + test2 + test3) / 3.0
```

Last, in step 5, we display the average. Program 2-16 shows the program.

Program 2-16 (test_score_average.py)

```
 1  # Get three test scores and assign them to the
 2  # test1, test2, and test3 variables.
 3  test1 = float(input('Enter the first test score: '))
 4  test2 = float(input('Enter the second test score: '))
 5  test3 = float(input('Enter the third test score: '))
 6
 7  # Calculate the average of the three scores
 8  # and assign the result to the average variable.
 9  average = (test1 + test2 + test3) / 3.0
10
11  # Display the average.
12  print('The average score is', average)
```

Program Output (with input shown in bold)

```
Enter the first test score: 90 Enter
Enter the second test score: 80 Enter
Enter the third test score: 100 Enter
The average score is 90.0
```

The Exponent Operator

In addition to the basic math operators for addition, subtraction, multiplication, and division, Python also provides an exponent operator. Two asterisks written together (`**`) is the exponent operator, and its purpose it to raise a number to a power. For example, the following statement raises the `length` variable to the power of 2 and assigns the result to the `area` variable:

```
area = length**2
```

The following session with the interactive interpreter shows the values of the expressions `4**2`, `5**3`, and `2**10`:

```
>>> 4**2 Enter
16
>>> 5**3 Enter
125
>>> 2**10 Enter
1024
>>>
```

The Remainder Operator

In Python, the % symbol is the remainder operator. (This is also known as the *modulus operator.*) The remainder operator performs division, but instead of returning the quotient, it returns the remainder. The following statement assigns 2 to `leftover`:

```
leftover = 17 % 3
```

This statement assigns 2 to leftover because 17 divided by 3 is 5 with a remainder of 2. The remainder operator is useful in certain situations. It is commonly used in calculations that convert times or distances, detect odd or even numbers, and perform other specialized operations. For example, Program 2-17 gets a number of seconds from the user, and it converts that number of seconds to hours, minutes, and seconds. For example, it would convert 11,730 seconds to 3 hours, 15 minutes, and 30 seconds.

Program 2-17 (time_converter.py)

```
1  # Get a number of seconds from the user.
2  total_seconds = float(input('Enter a number of seconds: '))
3
4  # Get the number of hours.
5  hours = total_seconds // 3600
6
7  # Get the number of remaining minutes.
8  minutes = (total_seconds // 60) % 60
9
10 # Get the number of remaining seconds.
11 seconds = total_seconds % 60
12
13 # Display the results.
14 print('Here is the time in hours, minutes, and seconds:')
15 print('Hours:', hours)
16 print('Minutes:', minutes)
17 print('Seconds:', seconds)
```

Program Output (with input shown in bold)

```
Enter a number of seconds: 11730 Enter
Here is the time in hours, minutes, and seconds:
```

```
Hours: 3.0
Minutes: 15.0
Seconds: 30.0
```

Let's take a closer look at the code:

- Line 2 gets a number of seconds from the user, converts the value to a `float`, and assigns it to the `total_seconds` variable.
- Line 5 calculates the number of hours in the specified number of seconds. There are 3600 seconds in an hour, so this statement divides `total_seconds` by 3600. Notice that we used the integer division operator (`//`) operator. This is because we want the number of hours with no fractional part.
- Line 8 calculates the number of remaining minutes. This statement first uses the `//` operator to divide `total_seconds` by 60. This gives us the total number of minutes. Then, it uses the `%` operator to divide the total number of minutes by 60 and get the remainder of the division. The result is the number of remaining minutes.
- Line 11 calculates the number of remaining seconds. There are 60 seconds in a minute, so this statement uses the `%` operator to divide the `total_seconds` by 60 and get the remainder of the division. The result is the number of remaining seconds.
- Lines 14 through 17 display the number of hours, minutes, and seconds.

Converting Math Formulas to Programming Statements

You probably remember from algebra class that the expression $2xy$ is understood to mean 2 times x times y. In math, you do not always use an operator for multiplication. Python, as well as other programming languages, requires an operator for any mathematical operation. Table 2-6 shows some algebraic expressions that perform multiplication and the equivalent programming expressions.

Table 2-6 Algebraic expressions

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	`6 * B`
(3)(12)	3 times 12	`3 * 12`
$4xy$	4 times x times y	`4 * x * y`

When converting some algebraic expressions to programming expressions, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following formula:

$$x = \frac{a + b}{c}$$

To convert this to a programming statement, $a + b$ will have to be enclosed in parentheses:

```
x = (a + b)/c
```

Table 2-7 shows additional algebraic expressions and their Python equivalents.

Table 2-7 Algebraic and programming expressions

Algebraic Expression	Python Statement
$y = 3\frac{x}{2}$	`y = 3 * x / 2`
$z = 3bc + 4$	`z = 3 * b * c + 4`
$a = \frac{x + 2}{b - 1}$	`a = (x + 2) / (b - 1)`

In the Spotlight:

Converting a Math Formula to a Programming Statement

Suppose you want to deposit a certain amount of money into a savings account and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have $10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the present value, or the amount that you need to deposit today.
- F is the future value that you want in the account. (In this case, F is $10,000.)
- r is the annual interest rate.
- n is the number of years that you plan to let the money sit in the account.

It would be convenient to write a computer program to perform the calculation because then we can experiment with different values for the variables. Here is an algorithm that we can use:

1. *Get the desired future value.*
2. *Get the annual interest rate.*
3. *Get the number of years that the money will sit in the account.*
4. *Calculate the amount that will have to be deposited.*
5. *Display the result of the calculation in step 4.*

In steps 1 through 3, we will prompt the user to enter the specified values. We will assign the desired future value to a variable named `future_value`, the annual interest rate to a variable named `rate`, and the number of years to a variable named `years`.

In step 4, we calculate the present value, which is the amount of money that we will have to deposit. We will convert the formula previously shown to the following statement. The statement stores the result of the calculation in the `present_value` variable.

```
present_value = future_value / (1.0 + rate)**years
```

In step 5, we display the value in the `present_value` variable. Program 2-18 shows the program.

Program 2-18 (future_value.py)

```
1  # Get the desired future value.
2  future_value = float(input('Enter the desired future value: '))
3
4  # Get the annual interest rate.
5  rate = float(input('Enter the annual interest rate: '))
6
7  # Get the number of years that the money will appreciate.
8  years = int(input('Enter the number of years the money will grow: '))
9
10 # Calculate the amount needed to deposit.
11 present_value = future_value / (1.0 + rate)**years
12
13 # Display the amount needed to deposit.
14 print('You will need to deposit this amount:', present_value)
```

Program Output

```
Enter the desired future value: 10000.0 Enter
Enter the annual interest rate: 0.05 Enter
Enter the number of years the money will grow: 10 Enter
You will need to deposit this amount: 6139.13253541
```

NOTE: Unlike the output shown for this program, dollar amounts are usually rounded to two decimal places. Later in this chapter you will learn how to format numbers so they are rounded to a specified number of decimal places.

Mixed-Type Expressions and Data Type Conversion

When you perform a math operation on two operands, the data type of the result will depend on the data type of the operands. Python follows these rules when evaluating mathematical expressions:

- When an operation is performed on two `int` values, the result will be an `int`.
- When an operation is performed on two `float` values, the result will be a `float`.
- When an operation is performed on an `int` and a `float`, the `int` value will be temporarily converted to a `float` and the result of the operation will be a `float`. (An expression that uses operands of different data types is called a *mixed-type expression*.)

The first two situations are straightforward: operations on `int`s produce `int`s, and operations on `float`s produce `float`s. Let's look at an example of the third situation, which involves mixed-type expressions:

```
my_number = 5 * 2.0
```

When this statement executes, the value 5 will be converted to a `float` (5.0) and then multiplied by 2.0. The result, 10.0, will be assigned to `my_number`.

The `int` to `float` conversion that takes place in the previous statement happens implicitly. If you need to explicitly perform a conversion, you can use either the `int()` or `float()` functions. For example, you can use the `int()` function to convert a floating-point value to an integer, as shown in the following code:

```
fvalue = 2.6
ivalue = int(fvalue)
```

The first statement assigns the value 2.6 to the `fvalue` variable. The second statement passes `fvalue` as an argument to the `int()` function. The `int()` function returns the value 2, which is assigned to the `ivalue` variable. After this code executes, the `fvalue` variable is still assigned the value 2.6, but the `ivalue` variable is assigned the value 2.

As demonstrated in the previous example, the `int()` function converts a floating-point argument to an integer by truncating it. As previously mentioned, that means it throws away the number's fractional part. Here is an example that uses a negative number:

```
fvalue = -2.9
ivalue = int(fvalue)
```

In the second statement, the value −2 is returned from the `int()` function. After this code executes, the `fvalue` variable references the value −2.9, and the `ivalue` variable references the value −2.

You can use the `float()` function to explicitly convert an `int` to a `float`, as shown in the following code:

```
ivalue = 2
fvalue = float(ivalue)
```

After this code executes, the `ivalue` variable references the integer value 2, and the `fvalue` variable references the floating-point value 2.0.

Breaking Long Statements into Multiple Lines

Most programming statements are written on one line. If a programming statement is too long, however, you will not be able to view all of it in your editor window without scrolling horizontally. In addition, if you print your program code on paper and one of the statements is too long to fit on one line, it will wrap around to the next line and make the code difficult to read.

Python allows you to break a statement into multiple lines by using the *line continuation character,* which is a backslash (\). You simply type the backslash character at the point you want to break the statement, and then press the Enter key. Here is a `print` function call that is broken into two lines with the line continuation character:

```
print('We sold', units_sold, \
      'for a total of', sales_amount)
```

The line continuation character that appears at the end of the first line tells the interpreter that the statement is continued on the next line. Here is a statement that performs a mathematical calculation and has been broken up to fit on two lines:

```
result = var1 * 2 + var2 * 3 + \
         var3 * 4 + var4 * 5
```

Here is one last example:

```
print("Monday's sales are", monday, \
      "and Tuesday's sales are", tuesday, \
      "and Wednesday's sales are", wednesday)
```

This long statement is broken into three lines. Notice that the first two lines end with a backslash.

Checkpoint

2.19 Complete the following table by writing the value of each expression in the Value column.

Expression	Value
`6 + 3 * 5`	_____
`12 / 2 - 4`	_____
`9 + 14 * 2 - 6`	_____
`(6 + 2) * 3`	_____
`14 / (11 - 4)`	_____
`9 + 12 * (8 - 3)`	_____

2.20 What value will be assigned to `result` after the following statement executes?

```
result = 9 // 2
```

2.21 What value will be assigned to `result` after the following statement executes?

```
result = 9 % 2
```

2.8 More About Data Output

So far we have discussed only basic ways to display data. Eventually, you will want to exercise more control over the way data appear on the screen. In this section, you will learn more details about the Python `print` function, and you'll see techniques for formatting output in specific ways.

Suppressing the `print` Function's Ending Newline

The `print` function normally displays a line of output. For example, the following three statements will produce three lines of output:

```
print('One')
print('Two')
print('Three')
```

Each of the statements shown here displays a string and then prints a *newline character*. You do not see the newline character, but when it is displayed, it causes the output to

advance to the next line. (You can think of the newline character as a special command that causes the computer to start a new line of output.)

If you do not want the `print` function to start a new line of output when it finishes displaying its output, you can pass the special argument `end=' '` to the function, as shown in the following code:

```
print('One', end=' ')
print('Two', end=' ')
print('Three')
```

Notice that in the first two statements, the argument `end=' '` is passed to the `print` function. This specifies that the `print` function should print a space instead of a newline character at the end of its output. Here is the output of these statements:

```
One Two Three
```

Sometimes you might not want the `print` function to print anything at the end of its output, not even a space. If that is the case, you can pass the argument `end=''` to the `print` function, as shown in the following code:

```
print('One', end='')
print('Two', end='')
print('Three')
```

Notice that in the argument `end=''` there is no space between the quote marks. This specifies that the `print` function should print nothing at the end of its output. Here is the output of these statements:

```
OneTwoThree
```

Specifying an Item Separator

When multiple arguments are passed to the `print` function, they are automatically separated by a space when they are displayed on the screen. Here is an example, demonstrated in interactive mode:

```
>>> print('One', 'Two', 'Three') Enter
One Two Three
>>>
```

if you do not want a space printed between the items, you can pass the argument `sep=''` to the `print` function, as shown here:

```
>>> print('One', 'Two', 'Three', sep='') Enter
OneTwoThree
>>>
```

You can also use this special argument to specify a character other than the space to separate multiple items. Here is an example:

```
>>> print('One', 'Two', 'Three', sep='*') Enter
One*Two*Three
>>>
```

Notice that in this example, we passed the argument `sep='*'` to the `print` function. This specifies that the printed items should be separated with the * character. Here is another example:

```
>>> print('One', 'Two', 'Three', sep='---') Enter
One---Two---Three
>>>
```

Escape Characters

An *escape character* is a special character that is preceded with a backslash (\), appearing inside a string literal. When a string literal that contains escape characters is printed, the escape characters are treated as special commands that are embedded in the string.

For example, `\n` is the newline escape character. When the `\n` escape character is printed, it isn't displayed on the screen. Instead, it causes output to advance to the next line. For example, look at the following statement:

```
print('One\nTwo\nThree')
```

When this statement executes, it displays

```
One
Two
Three
```

Python recognizes several escape characters, some of which are listed in Table 2-8.

Table 2-8 Some of Python's escape characters

Escape Character	Effect
`\n`	Causes output to be advanced to the next line.
`\t`	Causes output to skip over to the next horizontal tab position.
`\'`	Causes a single quote mark to be printed.
`\"`	Causes a double quote mark to be printed.
`\\`	Causes a backslash character to be printed.

The `\t` escape character advances the output to the next horizontal tab position. (A tab position normally appears after every eighth character.) The following statements are illustrative:

```
print('Mon\tTues\tWed')
print('Thur\tFri\tSat')
```

This statement prints `Monday`, then advances the output to the next tab position, then prints `Tuesday`, then advances the output to the next tab position, then prints `Wednesday`. The output will look like this:

```
Mon     Tues    Wed
Thur    Fri     Sat
```

You can use the `\'` and `\"` escape characters to display quotation marks. The following statements are illustrative:

```
print("Your assignment is to read \"Hamlet\" by tomorrow.")
print('I\'m ready to begin.')
```

These statements display the following:

```
Your assignment is to read "Hamlet" by tomorrow.
I'm ready to begin.
```

You can use the \\ escape character to display a backslash, as shown in the following:

```
print('The path is C:\\temp\\data.')
```

This statement will display

```
The path is C:\temp\data.
```

Displaying Multiple Items with the + Operator

Earlier in this chapter, you saw that the + operator is used to add two numbers. When the + operator is used with two strings, however, it performs *string concatenation*. This means that it appends one string to another. For example, look at the following statement:

```
print('This is ' + 'one string.')
```

This statement will print

```
This is one string.
```

String concatenation can be useful for breaking up a string literal so a lengthy call to the `print` function can span multiple lines. Here is an example:

```
print('Enter the amount of ' + \
      'sales for each day and ' + \
      'press Enter.')
```

This statement will display the following:

```
Enter the amount of sales for each day and press Enter.
```

Formatting Numbers

You might not always be happy with the way that numbers, especially floating-point numbers, are displayed on the screen. When a floating-point number is displayed by the `print` function, it can appear with up to 12 significant digits. This is shown in the output of Program 2-19.

Program 2-19 (no_formatting.py)

```
1 # This program demonstrates how a floating-point
2 # number is displayed with no formatting.
3 amount_due = 5000.0
4 monthly_payment = amount_due / 12.0
5 print('The monthly payment is', monthly_payment)
```

Program Output

```
The monthly payment is 416.666666667
```

Because this program displays a dollar amount, it would be nice to see that amount rounded to two decimal places. Fortunately, Python gives us a way to do just that, and more, with the built-in `format` function.

When you call the built-in `format` function, you pass two arguments to the function: a numeric value and a format specifier. The *format specifier* is a string that contains special characters specifying how the numeric value should be formatted. Let's look at an example:

```
format(12345.6789, '.2f')
```

The first argument, which is the floating-point number 12345.6789, is the number that we want to format. The second argument, which is the string `'.2f'`, is the format specifier. Here is the meaning of its contents:

- The `.2` specifies the precision. It indicates that we want to round the number to two decimal places.
- The `f` specifies that the data type of the number we are formatting is a floating-point number. (If you are formatting an integer, you cannot use `f` for the type. We discuss integer formatting momentarily.)

The `format` function returns a string containing the formatted number. The following interactive mode session demonstrates how you use the `format` function along with the `print` function to display a formatted number:

```
>>> print(format(12345.6789, '.2f')) Enter
12345.68
>>>
```

Notice that the number is rounded to two decimal places. The following example shows the same number, rounded to one decimal place:

```
>>> print(format(12345.6789, '.1f')) Enter
12345.7
>>>
```

Here is another example:

```
>>> print('The number is', format(1.234567, '.2f')) Enter
The number is 1.23
>>>
```

Program 2-20 shows how we can modify Program 2-19 so that it formats its output using this technique.

Program 2-20 (formatting.py)

```
1 # This program demonstrates how a floating-point
2 # number can be formatted.
3 amount_due = 5000.0
4 monthly_payment = amount_due / 12
5 print('The monthly payment is', \
6       format(monthly_payment, '.2f'))
```

Program Output

```
The monthly payment is 416.67
```

Formatting in Scientific Notation

If you prefer to display floating-point numbers in scientific notation, you can use the letter `e` or the letter `E` instead of `f`. Here are some examples:

```
>>> print(format(12345.6789, 'e')) Enter
1.234568e+04
>>> print(format(12345.6789, '.2e')) Enter
1.23e+04
>>>
```

The first statement simply formats the number in scientific notation. The number is displayed with the letter `e` indicating the exponent. (If you use uppercase `E` in the format specifier, the result will contain an uppercase `E` indicating the exponent.) The second statement additionally specifies a precision of two decimal places.

Inserting Comma Separators

If you want the number to be formatted with comma separators, you can insert a comma into the format specifier, as shown here:

```
>>> print(format(12345.6789, ',.2f')) Enter
12,345.68
>>>
```

Here is an example that formats an even larger number:

```
>>> print(format(123456789.456, ',.2f')) Enter
123,456,789.46
>>>
```

Notice that in the format specifier the comma is written before (to the left of) the precision designator. Here is an example that specifies the comma separator but does not specify precision:

```
>>> print(format(12345.6789, ',f')) Enter
12,345.678900
>>>
```

Program 2-21 demonstrates how the comma separator and a precision of two decimal places can be used to format larger numbers as currency amounts.

Program 2-21 (dollar_display.py)

```
1 # This program demonstrates how a floating-point
2 # number can be displayed as currency.
3 monthly_pay = 5000.0
4 annual_pay = monthly_pay * 12
5 print('Your annual pay is $', \
6       format(annual_pay, ',.2f'), \
7       sep='')
```

Program Output

```
Your annual pay is $60,000.00
```

Notice that in line 7 we passed the argument `sep=''` to the `print` function. As we mentioned earlier, this specifies that no space should be printed between the items that are being displayed. If we did not pass this argument, a space would be printed after the $ sign.

Specifying a Minimum Field Width

The format specifier can also include a minimum field width, which is the minimum number of spaces that should be used to display the value. The following example prints a number in a field that is 12 spaces wide:

```
>>> print('The number is', format(12345.6789, '12,.2f')) Enter
The number is 12,345.68
>>>
```

In this example, the 12 that appears in the format specifier indicates that the number should be displayed in a field that is a minimum of 12 spaces wide. In this case, the number that is displayed is shorter than the field that it is displayed in. The number `12,345.68` uses only 9 spaces on the screen, but it is displayed in a field that is 12 spaces wide. When this is the case, the number is right justified in the field. If a value is too large to fit in the specified field width, the field is automatically enlarged to accommodate it.

Note that in the previous example, the field width designator is written before (to the left of) the comma separator. Here is an example that specifies field width and precision, but does not use comma separators:

```
>>> print('The number is', format(12345.6789, '12.2f')) Enter
The number is      12345.68
>>>
```

Field widths can help when you need to print numbers aligned in columns. For example, look at Program 2-22. Each of the variables is displayed in a field that is seven spaces wide.

Program 2-22 (columns.py)

```
1  # This program displays the following
2  # floating-point numbers in a column
3  # with their decimal points aligned.
4  num1 = 127.899
5  num2 = 3465.148
6  num3 = 3.776
7  num4 = 264.821
8  num5 = 88.081
9  num6 = 799.999
10
11 # Display each number in a field of 7 spaces
12 # with 2 decimal places.
13 print(format(num1, '7.2f'))
14 print(format(num2, '7.2f'))
```

(program continues)

Program 2-22 *(continued)*

```
15 print(format(num3, '7.2f'))
16 print(format(num4, '7.2f'))
17 print(format(num5, '7.2f'))
18 print(format(num6, '7.2f'))
```

Program Output

```
 127.90
3465.15
   3.78
 264.82
  88.08
 800.00
```

Formatting a Floating-Point Number as a Percentage

Instead of using `f` as the type designator, you can use the `%` symbol to format a floating-point number as a percentage. The `%` symbol causes the number to be multiplied by 100 and displayed with a `%` sign following it. Here is an example:

```
>>> print(format(0.5, '%')) Enter
50.000000%
>>>
```

Here is an example that specifies 0 as the precision:

```
>>> print(format(0.5, '.0%')) Enter
50%
>>>
```

Formatting Integers

All the previous examples demonstrated how to format floating-point numbers. You can also use the `format` function to format integers. There are two differences to keep in mind when writing a format specifier that will be used to format an integer:

- You use `d` as the type designator.
- You cannot specify precision.

Let's look at some examples in the interactive interpreter. In the following session, the number 123456 is printed with no special formatting:

```
>>> print(format(123456, 'd')) Enter
123456
>>>
```

In the following session, the number 123456 is printed with a comma separator:

```
>>> print(format(123456, ',d')) Enter
123,456
>>>
```

In the following session, the number 123456 is printed in a field that is 10 spaces wide:

```
>>> print(format(123456, '10d')) Enter
    123456
>>>
```

In the following session, the number 123456 is printed with a comma separator in a field that is 10 spaces wide:

```
>>> print(format(123456, '10,d')) Enter
    123,456
>>>
```

Review Questions

Multiple Choice

1. A _____ error does not prevent the program from running, but causes it to produce incorrect results.
   a. syntax
   b. hardware
   c. logic
   d. fatal
2. A _____ is a single function that the program must perform in order to satisfy the customer.
   a. task
   b. software requirement
   c. prerequisite
   d. predicate
3. A(n) _____ is a set of well-defined logical steps that must be taken to perform a task.
   a. logarithm
   b. plan of action
   c. logic schedule
   d. algorithm
4. An informal language that has no syntax rules and is not meant to be compiled or executed is called _____.
   a. faux code
   b. pseudocode
   c. Python
   d. a flowchart
5. A _____ is a diagram that graphically depicts the steps that take place in a program.
   a. flowchart
   b. step chart
   c. code graph
   d. program graph

6. A _____ is a sequence of characters.
   a. char sequence
   b. character collection
   c. string
   d. text block
7. A _____ is a name that references a value in the computer's memory.
   a. variable
   b. register
   c. RAM slot
   d. byte
8. A _____ is any hypothetical person using a program and providing input for it.
   a. designer
   b. user
   c. guinea pig
   d. test subject
9. A string literal in Python must be enclosed in
   a. parentheses.
   b. single-quotes.
   c. double-quotes.
   d. either single-quotes or double-quotes.
10. Short notes placed in different parts of a program explaining how those parts of the program work are called _____.
    a. comments
    b. reference manuals
    c. tutorials
    d. external documentation
11. A(n) _____ makes a variable reference a value in the computer's memory.
    a. variable declaration
    b. assignment statement
    c. math expression
    d. string literal
12. This symbol marks the beginning of a comment in Python.
    a. `&`
    b. `*`
    c. `**`
    d. `#`
13. Which of the following statements will cause an error?
    a. `x = 17`
    b. `17 = x`
    c. `x = 99999`
    d. `x = '17'`

14. In the expression 12 + 7, the values on the right and left of the + symbol are called _____.
    a. operands
    b. operators
    c. arguments
    d. math expressions
15. This operator performs integer division.
    a. `//`
    b. `%`
    c. `**`
    d. `/`
16. This is an operator that raises a number to a power.
    a. `%`
    b. `*`
    c. `**`
    d. `/`
17. This operator performs division, but instead of returning the quotient it returns the remainder.
    a. `%`
    b. `*`
    c. `**`
    d. `/`
18. Suppose the following statement is in a program: `price = 99.0`. After this statement executes, the price variable will reference a value of this data type.
    a. `int`
    b. `float`
    c. `currency`
    d. `str`
19. This built-in function can be used to read input that has been typed on the keyboard.
    a. `input()`
    b. `get_input()`
    c. `read_input()`
    d. `keyboard()`
20. This built-in function can be used to convert an `int` value to a `float`.
    a. `int_to_float()`
    b. `float()`
    c. `convert()`
    d. `int()`

True or False

1. Programmers must be careful not to make syntax errors when writing pseudocode programs.
2. In a math expression, multiplication and division takes place before addition and subtraction.

3. Variable names can have spaces in them.
4. In Python the first character of a variable name cannot be a number.
5. If you print a variable that has not been assigned a value, the number 0 will be displayed.

Short Answer

1. What does a professional programmer usually do first to gain an understanding of a problem?
2. What is pseudocode?
3. Computer programs typically perform what three steps?
4. If a math expression adds a `float` to an `int`, what will the data type of the result be?
5. What is the difference between floating-point division and integer division?

Algorithm Workbench

1. Write Python code that prompts the user to enter his or her height and assigns the user's input to a variable named `height`.
2. Write Python code that prompts the user to enter his or her favorite color and assigns the user's input to a variable named `color`.
3. Write assignment statements that perform the following operations with the variables `a`, `b`, and `c`.
   a. Adds 2 to `a` and assigns the result to `b`
   b. Multiplies `b` times 4 and assigns the result to `a`
   c. Divides `a` by 3.14 and assigns the result to `b`
   d. Subtracts 8 from `b` and assigns the result to `a`
4. Assume the variables `result`, `w`, `x`, `y`, and `z` are all integers, and that `w = 5, x = 4, y = 8, and z = 2`. What value will be stored in `result` after each of the following statements execute?

   a. `result = x + y`
   b. `result = z * 2`
   c. `result = y / x`
   d. `result = y - z`
   e. `result = w // z`
5. Write a Python statement that assigns the sum of 10 and 14 to the variable `total`.
6. Write a Python statement that subtracts the variable `down_payment` from the variable `total` and assigns the result to the variable `due`.
7. Write a Python statement that multiplies the variable `subtotal` by 0.15 and assigns the result to the variable `total`.
8. What would the following display?

```
a = 5
b = 2
c = 3
result = a + b * c
print(result)
```

9. What would the following display?

```
num = 99
num = 5
print(num)
```

10. Assume the variable `sales` references a `float` value. Write a statement that displays the value rounded to two decimal points.

11. Assume the following statement has been executed:

```
number = 1234567.456
```

Write a Python statement that displays the value referenced by the `number` variable formatted as

```
1,234,567.5
```

12. What will the following statement display?

```
print('George', 'John', 'Paul', 'Ringo', sep='@')
```

Programming Exercises

1. Personal Information

Write a program that displays the following information:

- Your name
- Your address, with city, state, and ZIP
- Your telephone number
- Your college major

2. Sales Prediction

A company has determined that its annual profit is typically 23 percent of total sales. Write a program that asks the user to enter the projected amount of total sales, and then displays the profit that will be made from that amount.

Hint: Use the value 0.23 to represent 23 percent.

3. Land Calculation

One acre of land is equivalent to 43,560 square feet. Write a program that asks the user to enter the total square feet in a tract of land and calculates the number of acres in the tract.

Hint: Divide the amount entered by 43,560 to get the number of acres.

4. Total Purchase

A customer in a store is purchasing five items. Write a program that asks for the price of each item, and then displays the subtotal of the sale, the amount of sales tax, and the total. Assume the sales tax is 7 percent.

5. Distance Traveled

Assuming there are no accidents or delays, the distance that a car travels down the interstate can be calculated with the following formula:

$$Distance = Speed \times Time$$

A car is traveling at 70 miles per hour. Write a program that displays the following:

- The distance the car will travel in 6 hours
- The distance the car will travel in 10 hours
- The distance the car will travel in 15 hours

6. Sales Tax

Write a program that will ask the user to enter the amount of a purchase. The program should then compute the state and county sales tax. Assume the state sales tax is 5 percent and the county sales tax is 2.5 percent. The program should display the amount of the purchase, the state sales tax, the county sales tax, the total sales tax, and the total of the sale (which is the sum of the amount of purchase plus the total sales tax).

Hint: Use the value 0.025 to represent 2.5 percent, and 0.05 to represent 5 percent.

7. Miles-per-Gallon

A car's miles-per-gallon (MPG) can be calculated with the following formula:

$$MPG = \frac{Miles\ driven}{Gallons\ of\ gas\ used}$$

Write a program that asks the user for the number of miles driven and the gallons of gas used. It should calculate the car's MPG and display the result.

8. Tip, Tax, and Total

Write a program that calculates the total amount of a meal purchased at a restaurant. The program should ask the user to enter the charge for the food, and then calculate the amount of a 18 percent tip and 7 percent sales tax. Display each of these amounts and the total.

9. Celsius to Fahrenheit Temperature Converter

Write a program that converts Celsius temperatures to Fahrenheit temperatures. The formula is as follows:

$$F = \frac{9}{5}C + 32$$

The program should ask the user to enter a temperature in Celsius, and then display the temperature converted to Fahrenheit.

10. Ingredient Adjuster

A cookie recipe calls for the following ingredients:

- 1.5 cups of sugar
- 1 cup of butter
- 2.75 cups of flour

The recipe produces 48 cookies with this amount of the ingredients. Write a program that asks the user how many cookies he or she wants to make, and then displays the number of cups of each ingredient needed for the specified number of cookies.

11. Male and Female Percentages

Write a program that asks the user for the number of males and the number of females registered in a class. The program should display the percentage of males and females in the class.

Hint: Suppose there are 8 males and 12 females in a class. There are 20 students in the class. The percentage of males can be calculated as 8 ÷ 20 = 0.4, or 40%. The percentage of females can be calculated as 12 ÷ 20 = 0.6, or 60%.

12. Stock Transaction Program

Last month Joe purchased some stock in Acme Software, Inc. Here are the details of the purchase:

- The number of shares that Joe purchased was 2,000.
- When Joe purchased the stock, he paid $40.00 per share.
- Joe paid his stockbroker a commission that amounted to 3 percent of the amount he paid for the stock.

Two weeks later Joe sold the stock. Here are the details of the sale:

- The number of shares that Joe sold was 2,000.
- He sold the stock for $42.75 per share.
- He paid his stockbroker another commission that amounted to 3 percent of the amount he received for the stock.

Write a program that displays the following information:

- The amount of money Joe paid for the stock.
- The amount of commission Joe paid his broker when he bought the stock.
- The amount that Joe sold the stock for.
- The amount of commission Joe paid his broker when he sold the stock.
- Display the amount of money that Joe had left when he sold the stock and paid his broker (both times). If this amount is positive, then Joe made a profit. If the amount is negative, then Joe lost money.