starting out with >>>

PYTHON®

FIFTH EDITION

TONY GADDIS

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
	Introduction to Turtle Graphics, p. 81
	The Sales Prediction Problem, p. 114
Chapter 3	The `if` Statement, p. 119
	The `if-else` Statement, p. 128
	The Areas of Rectangles Problem, p. 161
Chapter 4	The `while` Loop, p. 170
	The `for` Loop, p. 178
	The Bug Collector Problem, p. 213
Chapter 5	Defining and Calling a Function, p. 222
	Passing Arguments to a Function, p. 236
	Writing a Value-Returning Function, p. 261
	The Kilometer Converter Problem, p. 294
	The Feet to Inches Problem, p. 296
Chapter 6	Using Loops to Process Files, p. 321
	File Display, p. 358
Chapter 7	List Slicing, p. 370
	The Lottery Number Generator Problem, p. 426
Chapter 8	The Vowels and Consonants problem, p. 463
Chapter 9	Introduction to Dictionaries, p. 467
	Introduction to Sets, p. 493
	The Capital Quiz Problem, p. 518
Chapter 10	Classes and Objects, p. 525
	The `Pet` class, p. 578
Chapter 11	The `Person` and `Customer` Classes, p. 607
Chapter 12	The Recursive Multiplication Problem, p. 626
Chapter 13	Creating a Simple GUI application, p. 635
	Responding to Button Clicks, p. 647
	The Name and Address Problem, p. 713
Chapter 14	Opening and Closing a Database Connection, p. 723
	Creating a Table, p. 726
	Adding Data to a Table, p. 731
	The SELECT Statement, p. 739
	Updating Rows, p. 753
	Getting Started with the Population Database Problem, p. 796
Appendix B	Introduction to IDLE, p. 803

Preface

Welcome to *Starting Out with Python*, Fifth 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 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 Fifth Edition

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

- **Database Programming –** This edition adds a new chapter on database programming. Chapter 14 introduces the student to SQL and Python database programming with SQLite.
- **Comprehension Expressions –** This edition introduces and explains list comprehensions, dictionary comprehensions, and set comprehensions.

- **Updated String Topics –** Several new string topics have been added, including:
    - Throughout the text, this edition uses f-strings, which were introduced in Python 3.6, to display formatted output. F-strings use a concise and intuitive syntax and are easier to learn than the `format` function. The previous material on the `format` function has been moved to Appendix F.
    - A new discussion of string tokens has been added to Chapter 8.
    - A new example of reading and parsing CSV files has been added to Chapter 8.
    - The discussion of string concatenation in Chapter 2 has been expanded to include implicit concatenation of adjacent strings.
- **GUI Programming –** Several new GUI programming topics have been added to Chapter 13, including:
    - Adding borders to widgets
    - Internal and external padding
    - Listbox widgets and scrollbars
- **Turtle Graphics:** Two commands for reading user input with dialog boxes have been introduced:
    - `turtle.numinput`
    - `turtle.textinput`
- **Random List Element Selection –** The `random.choice()` function is introduced in Chapter 7 as a way to randomly select list elements.
- **New Function Topics –** Several new topics have been added to chapter 5, including:
    - The `pass` keyword is introduced
    - Expanded discussion of the value `None`, and why a function might return `None`.
    - This edition adopts the standard practice of conditionally executing the `main` function.

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 formatted screen output. Pseudocode and flowcharts are also introduced as tools for designing programs. The chapter also includes an optional introduction to the turtle graphics library.

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 discussed as well. The chapter also includes an optional turtle graphics section, with a discussion of how to use decision structures to test the state of the turtle.

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. The chapter also includes an optional section on using loops to draw designs with the turtle graphics library.

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. An optional section includes a discussion of modularizing turtle graphics code with 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 list comprehension expressions, slicing, and many of the list methods. One- and two-dimensional lists are covered. The chapter also includes a discussion of the `matplotlib` package, and how to use it to plot charts and graphs from lists.

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. This chapter also includes examples of string tokenizing and parsing CSV files.

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, delete key-value pairs, and write dictionary comprehensions. The student learns to store values as unique elements in sets and perform common set operations such as union, intersection, difference, and symmetric difference. Set comprehensions are also introduced. 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, list boxes, 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. The Chapter includes a discussion of the `Canvas` widget, and how to use it to draw lines, rectangles, ovals, arcs, polygons, and text.

Chapter 14: Database Programming

This chapter introduces the student to database programming. The chapter first introduces the basic concepts of databases, such as tables, rows, and primary keys. Then the student learns to use SQLite to connect to a database in Python. SQL is introduced and the student learns to execute queries and statements that search for rows, add new rows, update existing rows, and delete rows. CRUD applications are demonstrated, and the chapter concludes with a discussion of relational data.

Appendix A: Installing Python

This appendix explains how to download and install the latest Python distribution.

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: Predefined Named Colors

This appendix lists the predefined color names that can be used with the turtle graphics library, `matplotlib` and `tkinter`.

Appendix E: More About the `import` Statement

This appendix discusses various ways to use the `import` statement. For example, you can use the `import` statement to import a module, a class, a function, or to assign an alias to a module.

Appendix F: Formatting Numeric Output with the `format()` Function

This appendix discusses the `format()` function and shows how to use its format specifiers to control the way that numeric values are displayed.

Appendix G: Installing Modules with the `pip` Utility

This appendix discusses how to use the `pip` utility to install third-party modules from the Python Package Index, or PyPI.

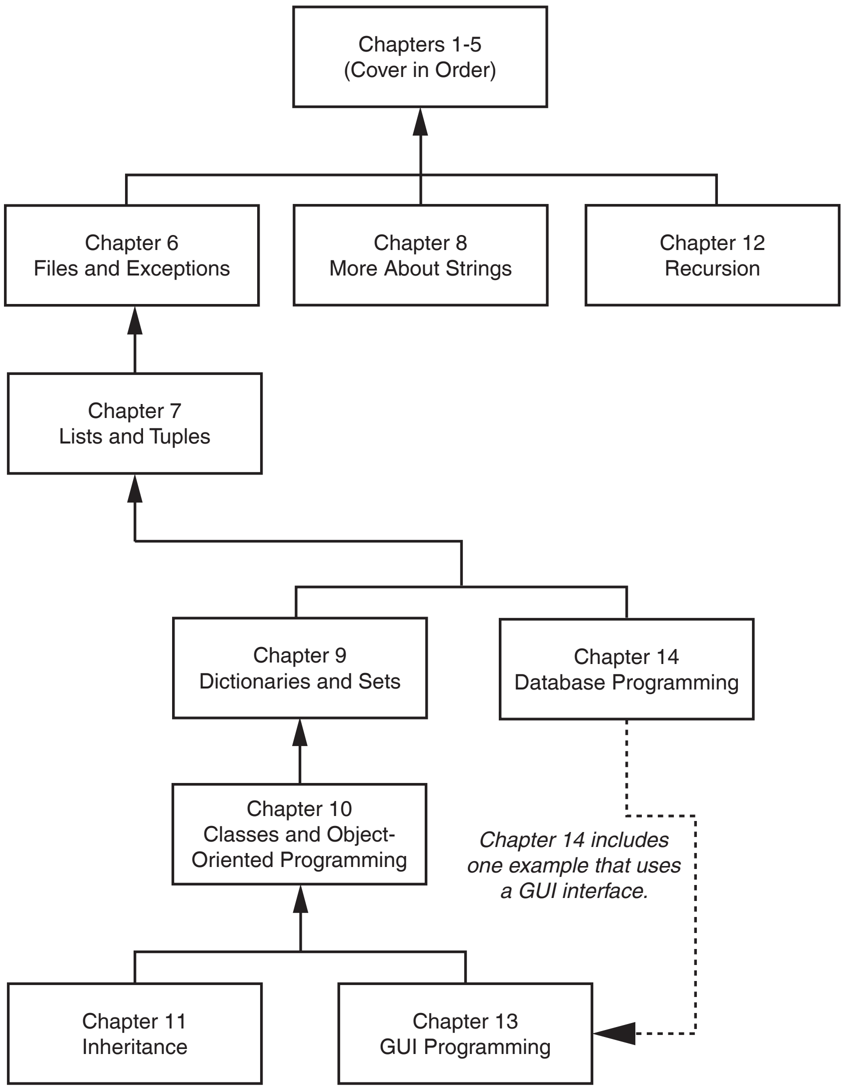
Appendix H: Answers to Checkpoints

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 Statements	Each major section of the text starts with a concept statement. 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/cs-resources. 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. Please note that the Checkpoint exercises in this textbook may differ from the Checkpoint questions embedded in the digital product Revel Starting Out with Python.
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. Please note that the end-of-chapter Programming Exercises in this textbook may differ from the end-of-chapter Programming Exercises embedded in the digital product Revel Starting Out with Python.

Supplements

Student Online Resources

Many student resources are available for this book from the publisher. The following items are available at www.pearsonhighered.com/cs-resources

- 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

Desmond K. H. Chun
Chabot Community College

Sonya Dennis
Morehouse College

Barbara Goldner
North Seattle Community College

Paul Gruhn
Manchester Community College

Bob Husson
Craven Community College

Diane Innes
Sandhills Community College

Daniel Jinguji
North Seattle Community College

John Kinuthia
Nazareth College of Rochester

Frank Liu
Sam Houston State University

Gary Marrer
Glendale Community College

Keith Mehl
Chabot College

Shyamal Mitra
University of Texas at Austin

Vince Offenback
North Seattle Community College

Smiljana Petrovic
Iona College

Raymond Pettit
Abilene Christian University

Janet Renwick
University of Arkansas–Fort Smith

Haris Ribic
SUNY at Binghamton

Ken Robol
Beaufort Community College

Eric Shaffer
University of Illinois at Urbana-Champaign

Tom Stokke
University of North Dakota

Anita Sutton
Germanna Community College

Ann Ford Tyson
Florida State University

Karen Ughetta
Virginia Western Community College

Christopher Urban
SUNY Institute of Technology

Nanette Veilleux
Simmons College

Brent Wilson
George Fox University

Linda F. Wilson
Texas Lutheran University

I would like to thank the faculty, staff, and administration at Haywood Community College for the opportunity to build a career teaching the subjects that I love. I would also like to thank my family and friends for their support in all of my projects.

It is a great honor to be published by Pearson, and I am extremely fortunate to have Tracy Johnson as my Editor and Content Manager. She and her colleagues Holly Stark, Erin Sullivan, Alicia Wilson, Scott Disanno, Carole Snyder, Bob Engelhardt, and Aishwarya Panday have worked tirelessly to produce and promote this book. 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 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

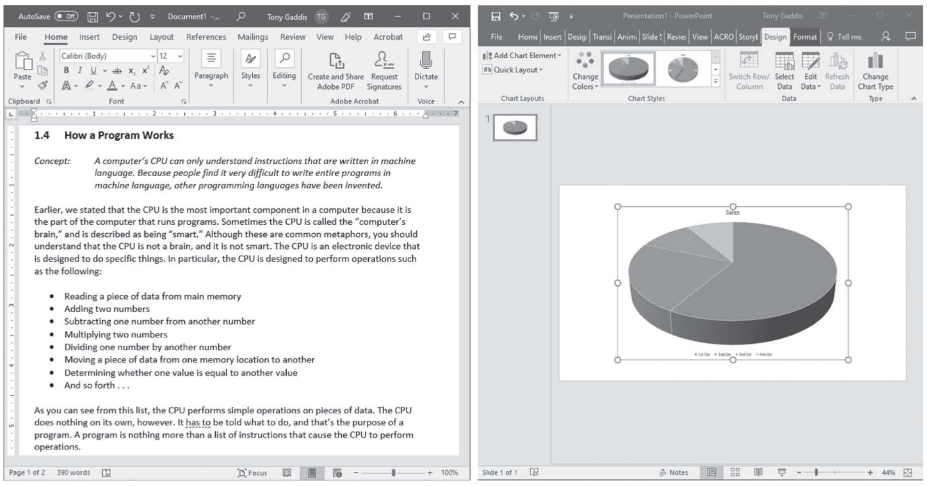
Think about some of the different ways that people use computers. In school, students use computers for tasks such as writing papers, searching for articles, sending email, and participating in online classes. At work, people use computers to analyze data, make presentations, conduct business transactions, 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 games. And don't forget that cell phones, tablets, 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 perform such a wide variety of tasks 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 a presentation program



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
- And so forth . . .

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.

and programs can be written quickly using it. Python is also a powerful language, popular with professional software developers. In fact, it 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 of which computers are commonly made. Next, we will look at how computers store data and execute programs. Finally, you 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 of which a computer is made are referred to as the computer's hardware. The programs that run on a computer are referred to as software.

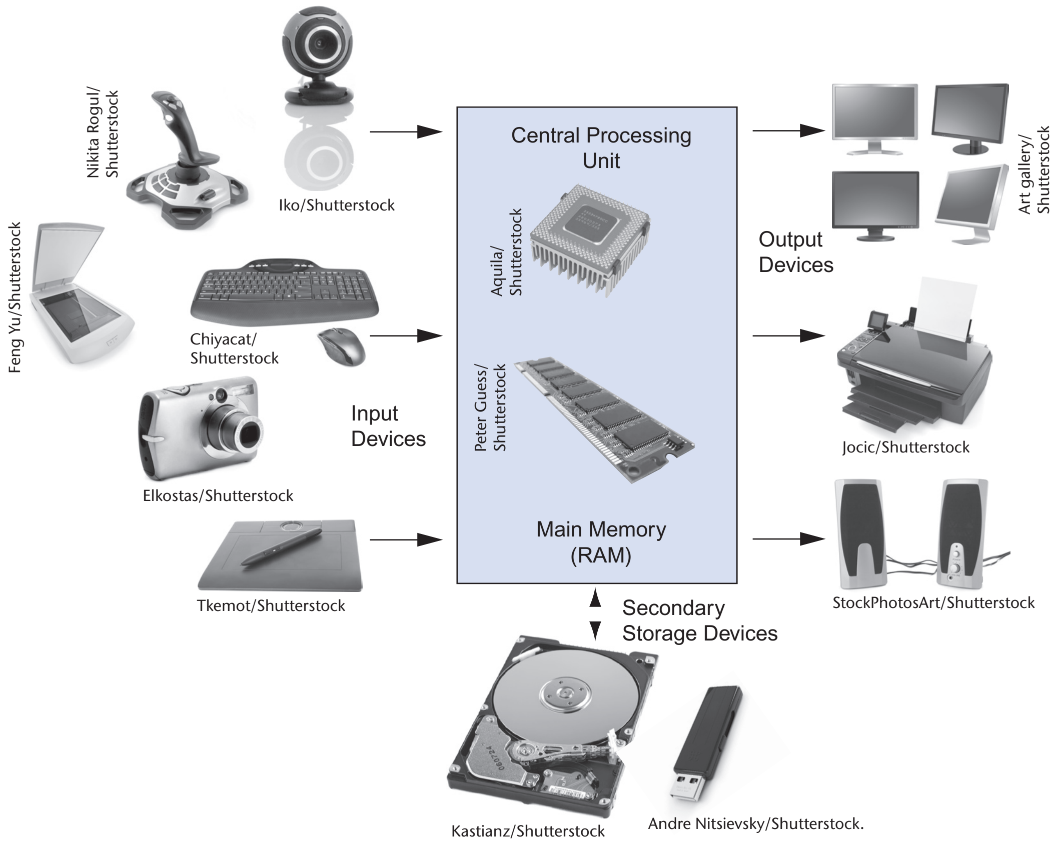
Hardware

The term *hardware* refers to all of the physical devices, or *components*, of which a computer is made. 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

Figure 1-2 Typical components of a computer system



- Secondary storage devices
- Input devices
- Output devices

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

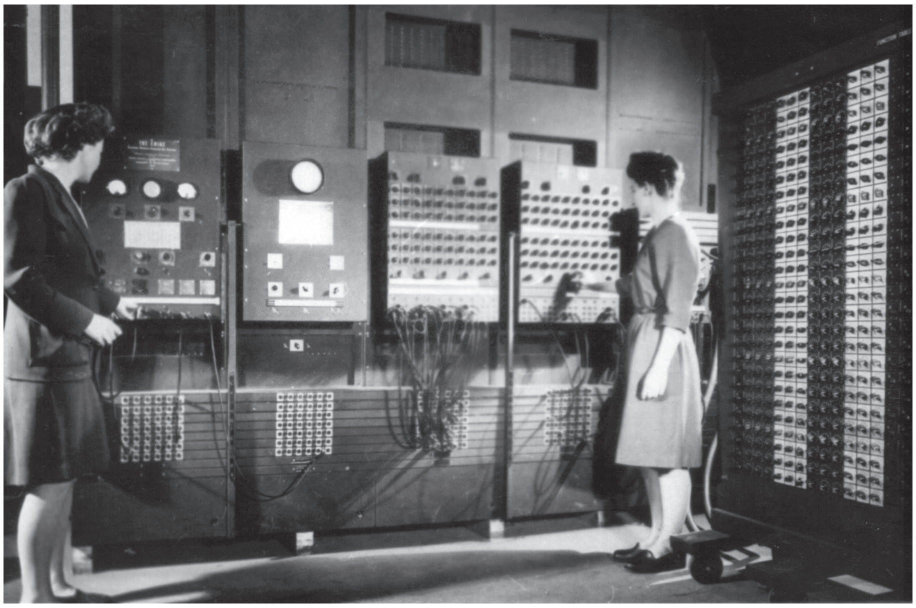
The CPU

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.

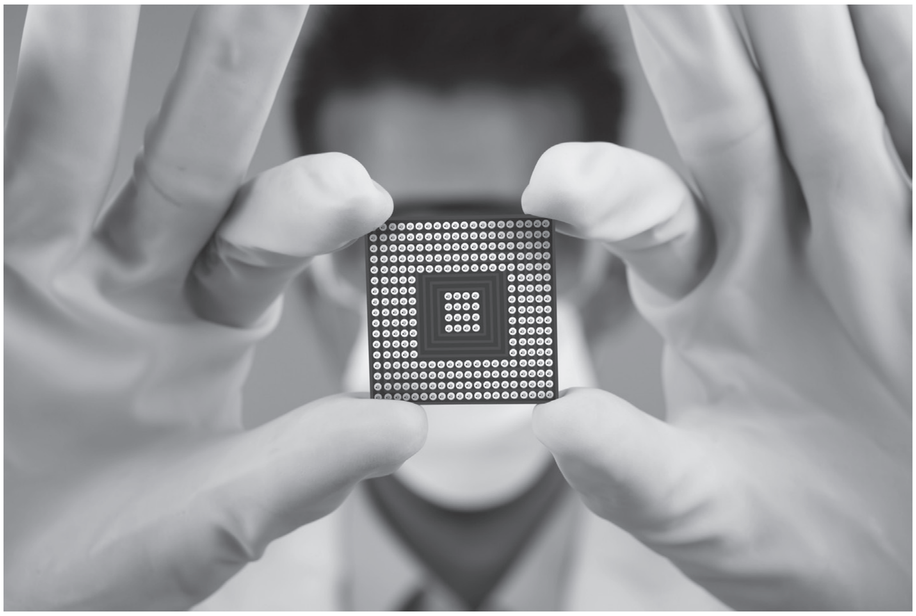
Today, CPUs are small chips known as *microprocessors*. Figure 1-4 shows a photo of a lab technician holding a modern microprocessor. In addition to being much smaller than the old electromechanical CPUs in early computers, microprocessors are also much more powerful.

Figure 1-3 The ENIAC computer



courtesy of U.S. Army Historic Computer Images

Figure 1-4 A lab technician holds a modern microprocessor



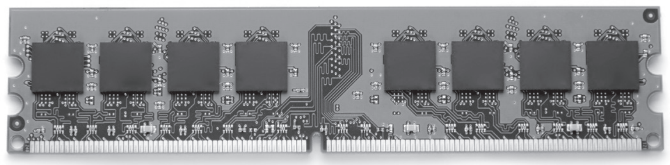
Creativa Images/Shutterstock

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 example, *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.

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, touchscreen, 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 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 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, macOS, 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?

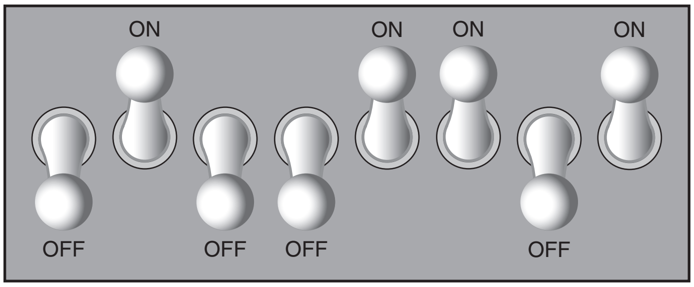
1.3 How Computers Store Data

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.

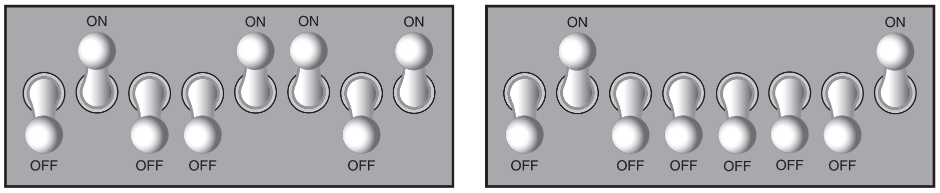
Each byte is divided into eight smaller storage locations known as bits. The term *bit* stands for *binary digit*. Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer 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

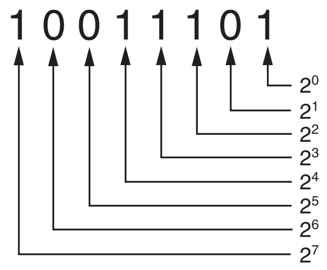
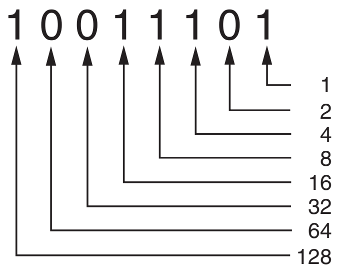


Figure 1-9 The values of binary digits



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

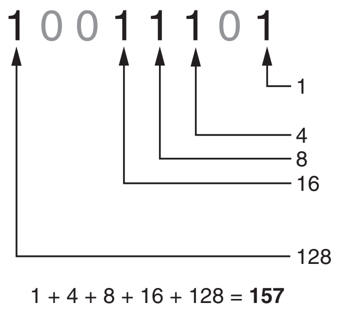
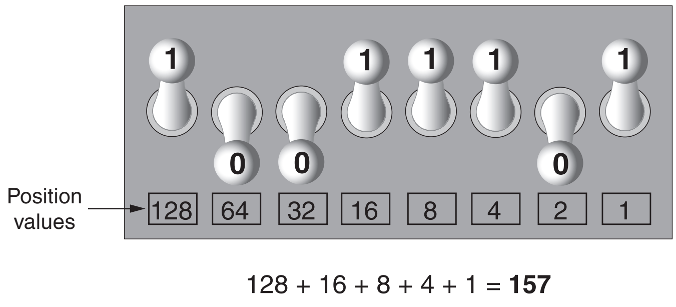


Figure 1-11 shows how you can picture the number 157 stored in a byte of memory. Each 1 is represented by a bit in the on position, and each 0 is represented by a bit in the off position.

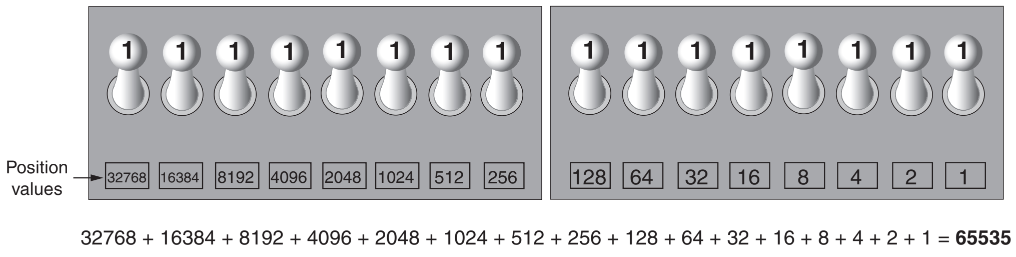
Figure 1-11 The bit pattern for 157



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



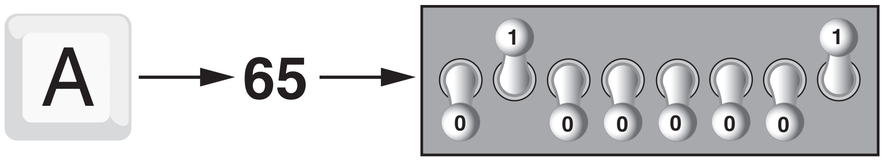
TIP: In case you're feeling overwhelmed by all this, relax! You will not have to actually convert numbers to binary while programming. Knowing that this process is taking place inside the computer will help you as you learn, and in the long term this knowledge will make you a better programmer.

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 almost 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.

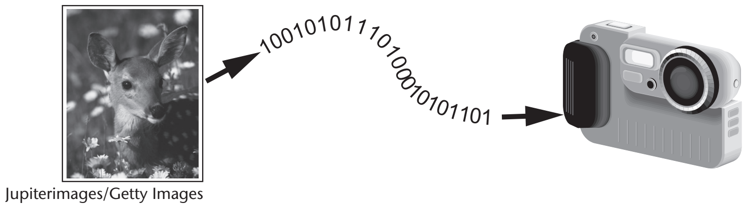
Computers are able to store negative numbers and real numbers in memory, but to do so they use encoding schemes along with the binary numbering system. Negative numbers are encoded using a technique known as *two's complement*, and real numbers are encoded in *floating-point notation*. You don't need 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 format, 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 stream from an online source, or play on an 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. For example, 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.

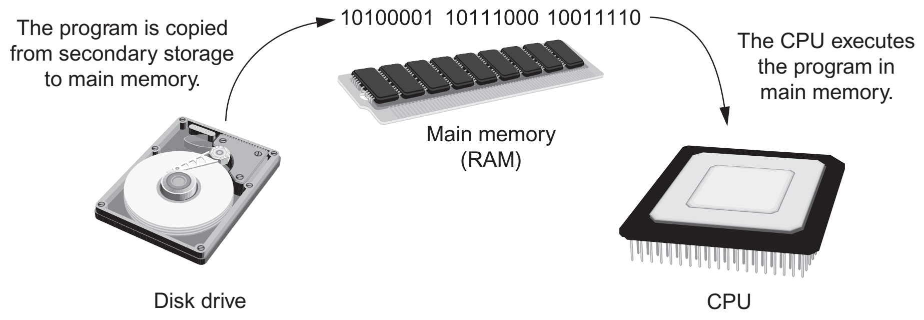
[1] The example shown is an actual instruction for an Intel microprocessor. It tells the microprocessor to move a value into the CPU.

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 downloaded from a website, or installed from an online app store.

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


The program is copied from secondary storage to main memory.
10100001 10111000 10011110
The CPU executes the program in main memory.
Main memory (RAM)
Disk drive
CPU

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.