starting out with >>>

PYTHON®

SIXTH EDITION

TONY G

starting out with >>>

PYTHON®

SIXTH EDITION

TONY GADDIS

Contents

5: Functions

6: Files and Exceptions

7: Lists and Tuples

starting out with >>> PYTHON®

SIXTH EDITION

TONY GADDIS

Dja65/123RF

STARTING OUT WITH PYTHON®

SIXTH EDITION

Tony Gaddis

Haywood Community College

Pearson

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

Director of Content Strategy: Dawn Murrin

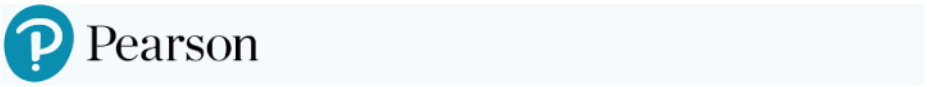
Product Development: Tracy Johnson

Content Strategy Management: Tracy Johnson

Content Production: Abhijeet Gope, Bob Engelhardt, Sandra Rodriguez, Scott Disanno, Pallavi Pandit, Adarsh Sushanth

Product Management: Holly Stark
Marketing Manager: Krista Clark
Rights and Permissions: Anjali Singh
Cover Image: dja65/123RF

Revel:
Revel for Starting Out with Python, Access Card 978-0-13-761913-9
Revel for Starting Out with Python, Instant Access 978-0-13-761915-3
Revel for Starting Out with Python, Combo Access Card 978-0-13-761914-6
Revel for Starting Out with Python, Inclusive Access 978-0-13-761918-4
Revel for Starting Out with Python, Print Offer [Loose-Leaf] 978-0-13-761910-8
Pearson+:
Pearson+ Starting Out with Python 978-0-13-787120-9
Starting Out with Python (Pearson eText Inclusive Access) 978-0-13-792251-2



Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity, depth, and breadth of all learners' lived experiences.

We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, sex, sexual orientation, socioeconomic status, ability, age, and religious or political beliefs.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational content accurately reflects the histories and lived experiences of the learners we serve.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content prompts deeper discussions with students and motivates them to expand their own learning (and worldview).

Accessibility

We are also committed to providing products that are fully accessible to all learners. As per Pearson's guidelines for accessible educational Web media, we test and retest the capabilities of our products against the highest standards for every release, following the WCAG guidelines in developing new products for copyright year 2022 and beyond.

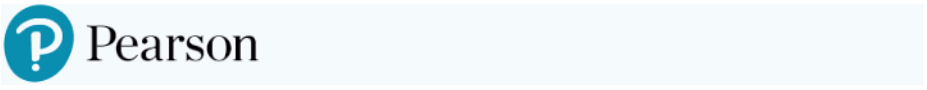
You can learn more about Pearson's commitment to accessibility at https://www.pearson.com/us/accessibility.html

Contact Us

While we work hard to present unbiased, fully accessible content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html

For accessibility-related issues, such as using assistive technology with Pearson products, alternative text requests, or accessibility documentation, email the Pearson Disability Support team at **disability.support@pearson.com**



Contents in a Glance

Note: For detailed Table of Contents please see navigation bar or at the beginning of your Print Offer.

Location of VideoNotes

VideoNotes are narrated step-by-step video tutorials that show how to solve problems completely, from design through coding.

VideoNote 4-2: The for Loop

VideoNote 4-3: Bug Collector Problem

Chapter 5: Functions

VideoNote 5-1: Defining and Calling a Function

VideoNote 5-2: Passing Arguments to a Function

VideoNote 5-3: Writing a Value-Returning Function

VideoNote 5-4: The Kilometer Converter Problem

VideoNote 5-5: The Feet to Inches Problem

Chapter 6: Files and Exceptions

VideoNote 6-1: Using Loops to Process Files

VideoNote 6-2: File Display

Chapter 7: Lists and Tuples

VideoNote 7-1: List Slicing

VideoNote 7-2: Lottery Number Generator Problem

Chapter 8: More About Strings

VideoNote 8-1: Vowels and Consonants Problem

Chapter 9: Dictionaries and Sets

VideoNote 9-1: Introduction to Dictionaries

VideoNote 9-2: Introduction to Sets

VideoNote 9-3: The Capital Quiz Problem

Chapter 10: Classes and Object-Oriented Programming

VideoNote 10-1: Classes and Objects

VideoNote 10-2: The Pet Class

Chapter 11: Inheritance

VideoNote 11-1: Person and Customer Classes

Chapter 12: Recursion

VideoNote 12-1: Recursive Multiplication

Chapter 13: GUI Programming

VideoNote 13-1: Creating a Simple GUI

VideoNote 13-2: Responding to Button Clicks

VideoNote 13-3: Name and Address Problem

Chapter 14: Database Programming

VideoNote 14-1: Opening and Closing a Database Connection

VideoNote 14-2: Creating a Table

VideoNote 14-3: Adding Data to a Table

VideoNote 14-4: The SELECT Statement

VideoNote 14-5: Updating Rows

VideoNote 14-6: Getting Started with the Population Database Problem

Appendix B: Introduction to IDLE

VideoNote B-1: Intro to IDLE

Preface

Welcome to *Starting Out with Python*, Sixth 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 Sixth 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:

- **Updated for Python 3.9** – This edition uses new language features from versions of Python up through Python 3.9.
- **The `with` Statement -- Chapter 6** now introduces the `with` statement as a way to open files. Many examples of using the `with` statement with files have been added throughout the book.
- **Multiple Assignment --** This edition introduces multiple assignment in **Chapter 2**.
- **Single-Line `if` Statements --** A new section on single-line `if` statements has been added to **Chapter 3**.
- **Conditional Expressions -- Chapter 3** now introduces conditional expressions and the ternary operator.
- **Walrus Operator and Assignment Expressions --** A new section on the walrus operator and assignment expressions has been added to **Chapter 3**. **Chapter 3** shows an example using an assignment expression in an `if` statement, and **Chapter 4** shows an example of using an assignment expression in a `while` loop.

- **Using the `while` Loop as a Count-Controlled Loop -- Chapter 4** has a new section on counter variables and using the `while` statement to write count-controlled loops.
- **Single-Line `while` Loops --** A new section on single-line `while` Loops has been added to **Chapter 4**.
- **Using `break`, `continue`, and `else` With Loops -- Chapter 4** now has a section on using `break` and `continue` with loops, and Python's unique `else` clause with loops.
- **Keyword-Only Parameters -- Chapter 5** now discusses keyword-only parameters and how to implement them in a function.
- **Positional-Only Parameters -- Chapter 5** now discusses positional-only parameters and how to implement them in a function.
- **Default Arguments -- Chapter 5** now has a section on default function arguments.
- **Using `count` and `sum` with Lists --** A discussion of the `count` method and the `sum` function with lists has been added to **Chapter 7**.
- **Storing Mutable Objects in a Tuple --** A new section has been added to **Chapter 7** that discusses the immutability of tuples, and how mutable objects can be stored in tuples.
- **Dictionary Merge and Update Operators -- Chapter 9** has a new section that discusses the dictionary's merge and update operators.

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 highlevel 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 their 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.

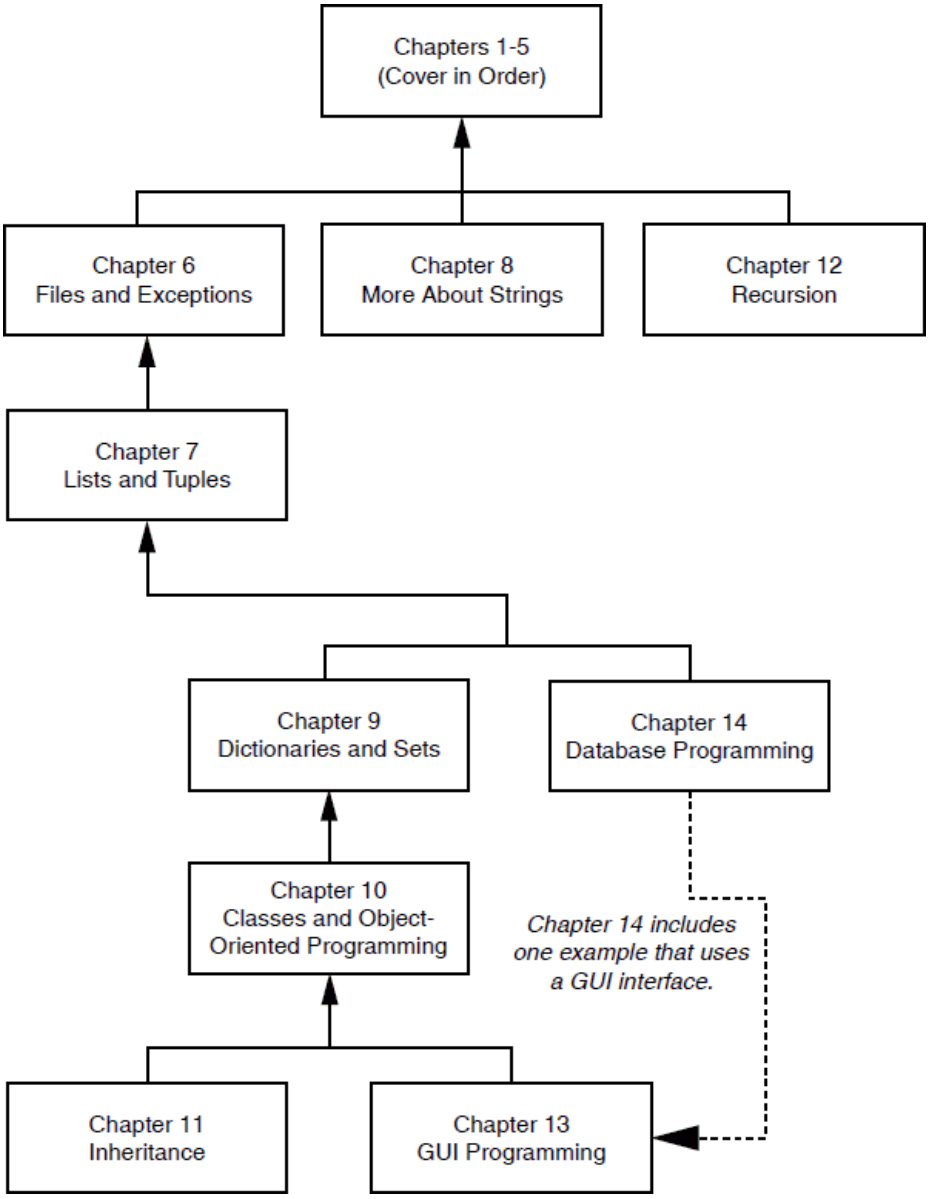
Appendix H: Answers to Checkpoints

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

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

Each chapter has one or more *In the Spotlight* case studies that Case Studies provide detailed, step-by-step analysis of problems and show the student how to solve them.

VideoNotes

VideoNotes are narrated step-by-step video tutorials that show how to solve problems completely, from design through coding.

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

(Noninteractive) Checkpoint Questions from the Book 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 https://media.pearsoncmg.com/ph/esm/ecs_gaddis_sowpython_6e/cw/

Instructor Resources

The following supplements are available to qualified instructors only:

- Solutions for the Section Quizzes, Chapter Quizzes, Programming Projects, Checkpoint self-review items, and Live Code Example practice activities are contained in the Revel Solutions Manual
- Solutions for the review questions and exercises

- PowerPoint presentation slides for each chapter
- Test bank
- TestGen
- Errata

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:

Desmond K. H. Chun	Raymond Pettit
Chabot Community College	*Abilene Christian University*
Sonya Dennis	Janet Renwick
Morehouse College	*University of Arkansas–Fort Smith*
Barbara Goldner	Haris Ribic
North Seattle Community College	*SUNY at Binghamton*
Paul Gruhn	Ken Robol
Manchester Community College	*Beaufort Community College*
Bob Husson	Eric Shaffer
Craven Community College	*University of Illinois at Urbana-Champaign*
Diane Innes	Tom Stokke
Sandhills Community College	*University of North Dakota*
Daniel Jinguji	Anita Sutton
North Seattle Community College	*Germanna Community College*
John Kinuthia	Ann Ford Tyson
Nazareth College of Rochester	*Florida State University*

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

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, Krista Clark, Scott Disanno, Sandra Rodriguez, Bob Engelhardt, Abhijeet Gope, Adarsh Sushanth, Pallavi Pandit, and Anu Sivakolundu have worked tirelessly to produce and promote this book. Thanks to you all!

About the Author

Tony Gaddis

Tony Gaddis's career includes twenty years as a faculty member at Haywood Community College. He has taught computer science courses, both in-person and online, to a wide variety of students. He particularly enjoys the challenges of teaching non-majors and others who initially struggle with the concepts of programming. He is an award-winning instructor who was previously selected as the NC Community College Teacher of the Year.

Tony is also a prolific author. His Starting Out With… series of textbooks, published by Pearson, covers a wide range of programming languages and pedagogical approaches for the CS1 and CS2 classrooms. Each of Tony's books encapsulates his passion for teaching and his experience explaining difficult concepts to beginning students.

Ordering Options

Revel

For more information about the Revel for this title, please visit: https://www.pearsonhighered.com/revel/educators/features/

Enhanced Revel Features: https://www.pearson.com/au/revel/educators/enhanced-features/index.html

Revel Instructor Help: https://help.pearsoncmg.com/glp/en-us/pearson/instructor/Content/index.htm

Pearson+

For more information about Pearson+ for this title, please visit https://www.pearson.com/en-us/pearsonplus.html and https://plc.pearson.com/en-US/our-products-and-services/pearson-plus

Pearson+ Help Center: https://www.pearson.com/en-us/pearsonplus/support.html

Chapter 1

Introduction to Computers and Programming

Topics

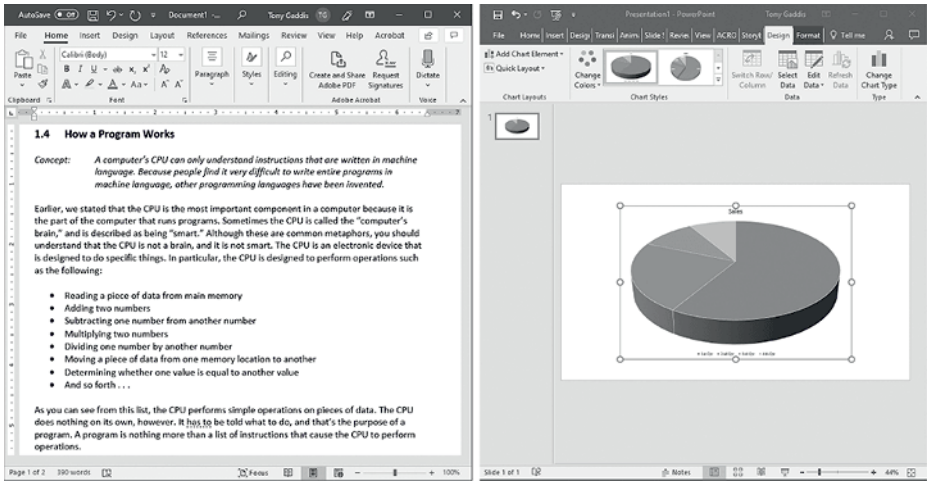
1.1: Introduction

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.

Figure 1-1

A word processing program and a presentation program



Used with permission from Microsoft.

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 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.

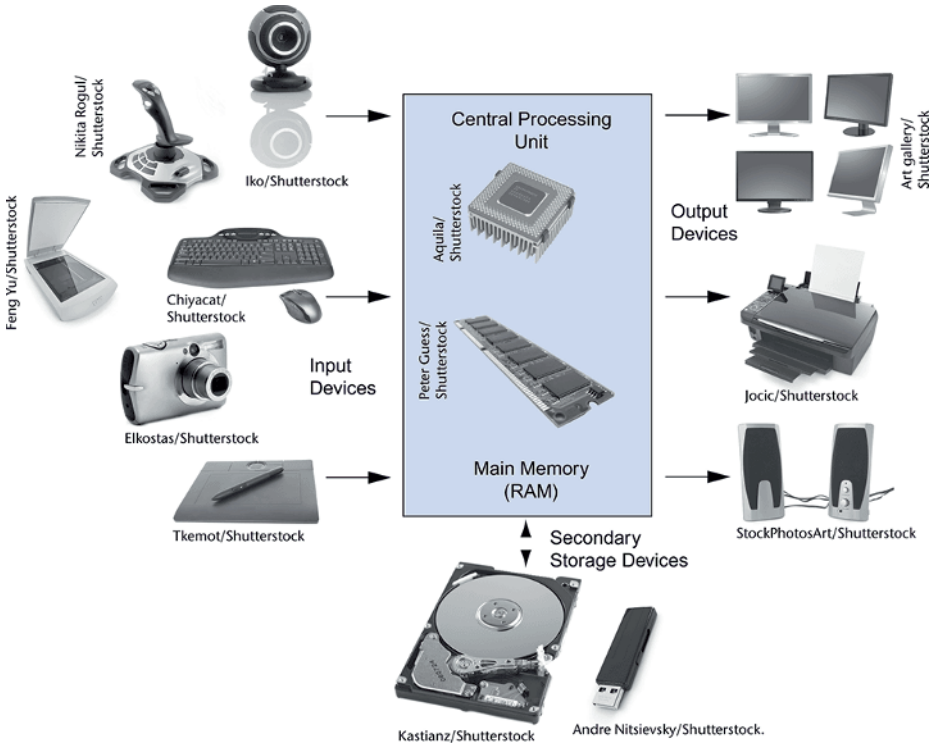
Hardware

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
- Secondary storage devices
- Input devices
- Output devices

Figure 1-2
Typical components of a computer system



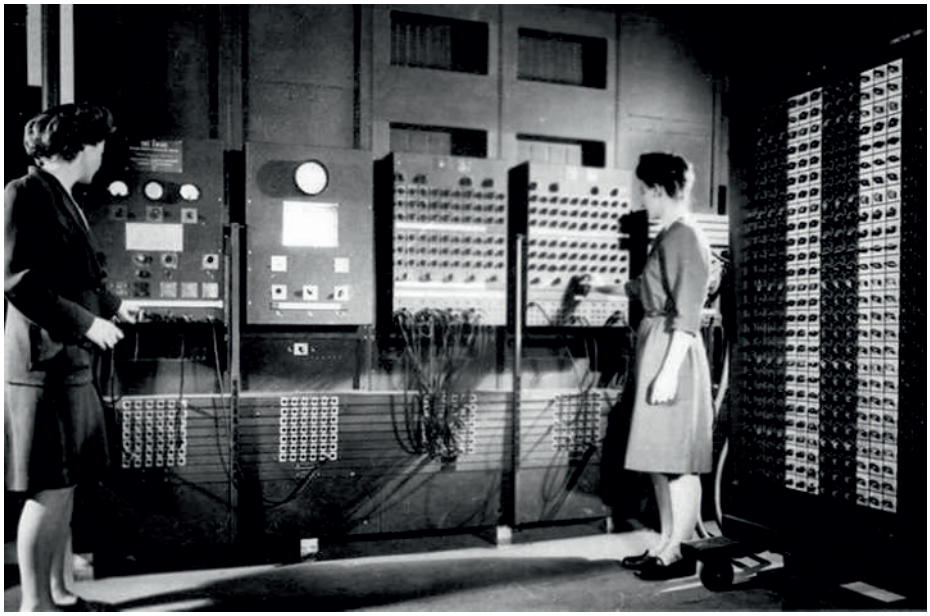
Let's take a closer look at each of these components.

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.

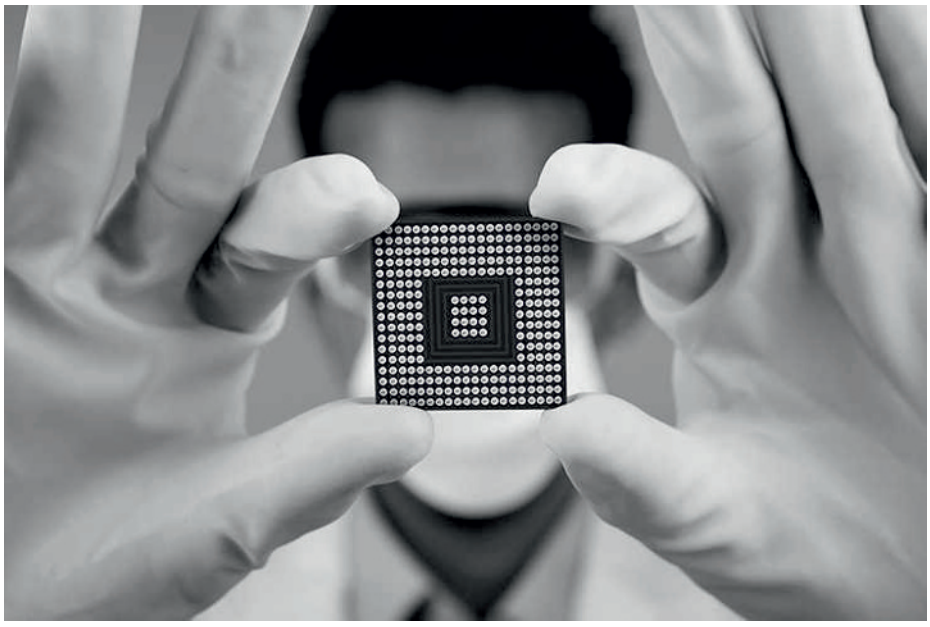
Figure 1-3
The ENIAC computer



Courtesy of U.S. Army Historic Computer Images

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 1-4
A lab technician holds a modern microprocessor



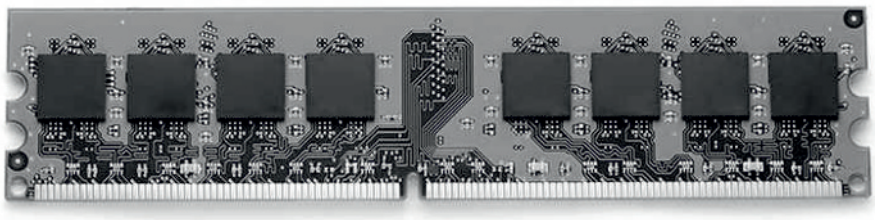
Creativa Images/Shutterstock

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.

1.2: (Noninteractive) Checkpoint Questions from the Book

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?

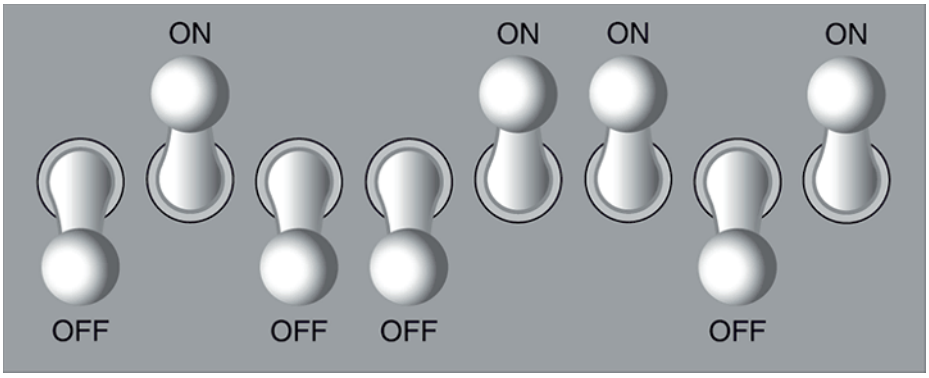
1.3: How Computers Store Data

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.

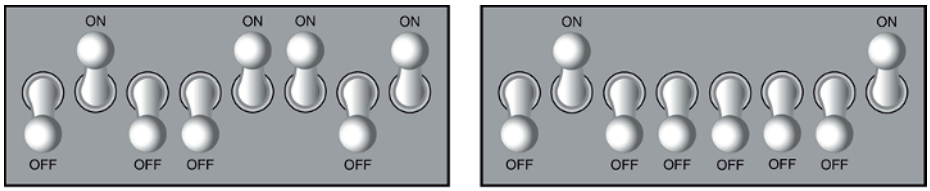
Each byte is divided into eight smaller storage locations known as bits. The term ***bit*** stands for ***binary digit***. Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer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

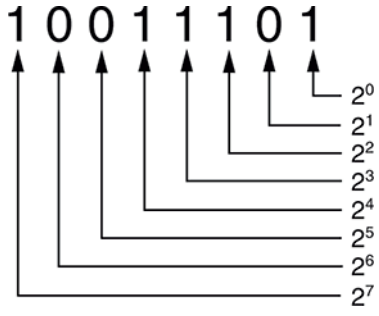
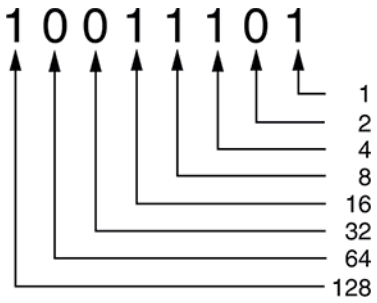


Figure 1-9

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

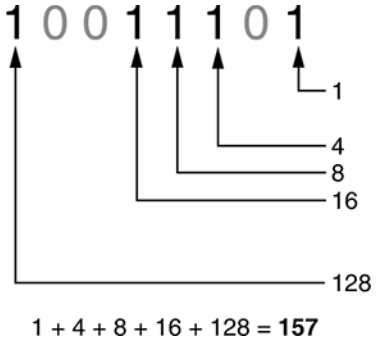
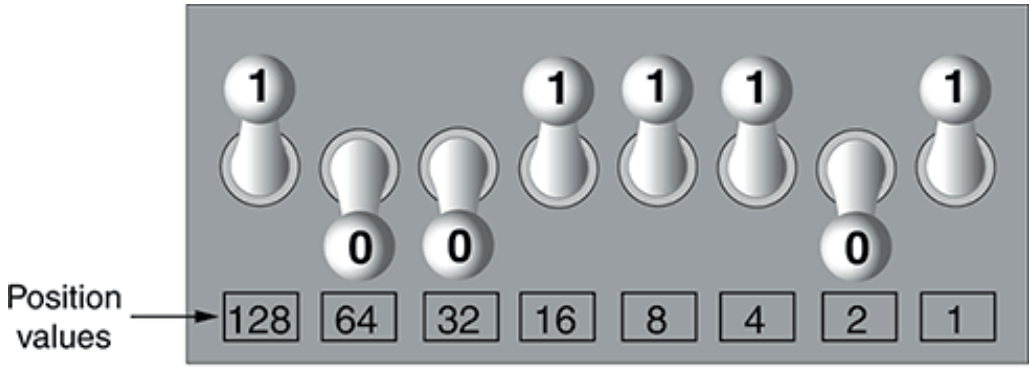


Figure 1-11 shows how you can picture the number 157 stored in a byte of memory. Each 1 is represented by a bit in the on position, and each 0 is represented by a bit in the off position.

Figure 1-11

The bit pattern for 157

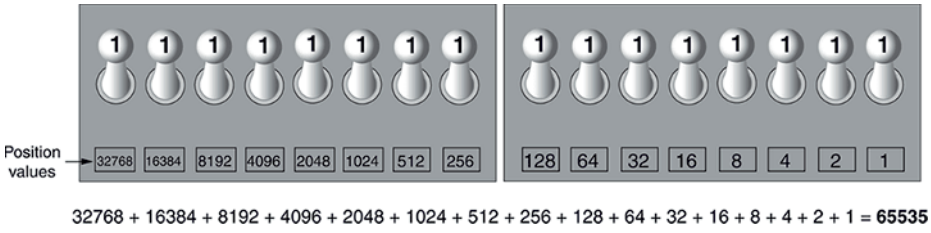


128 + 16 + 8 + 4 + 1 = **157**

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 2^{15}. 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



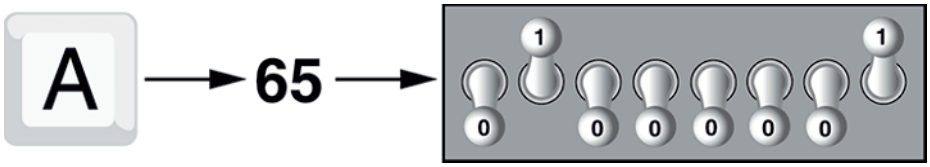
TIP: In case you're feeling overwhelmed by all this, relax! You will not have to actually convert numbers to binary while programming. Knowing that this process is taking place inside the computer will help you as you learn, and in the long term this knowledge will make you a better programmer.

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.

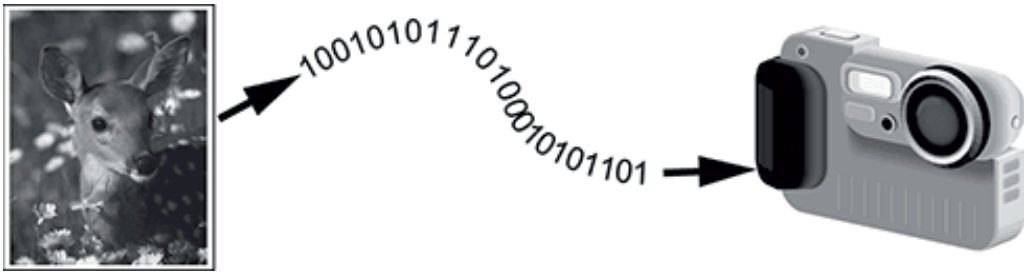
Other Types of Data

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



Jupiterimages/PHOTOS.com/Getty Images

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!

1.3: (Noninteractive) Checkpoint Questions from the Book

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 ***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.

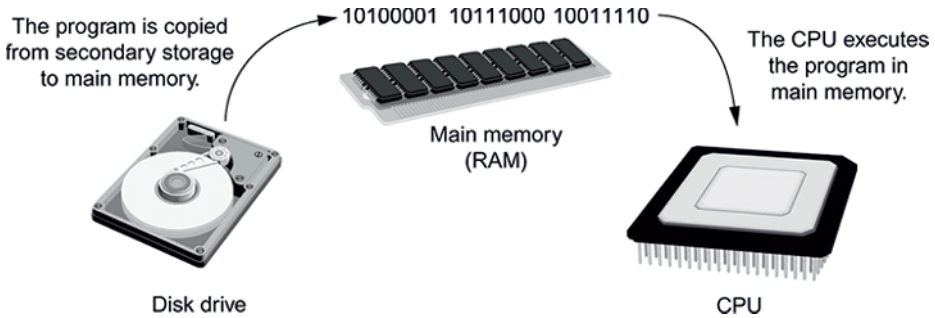
Each brand of microprocessor has its own unique instruction set, which is typically understood only by microprocessors of the same brand. For example, Intel microprocessors understand the same instructions, but they do not understand instructions for Motorola microprocessors.

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



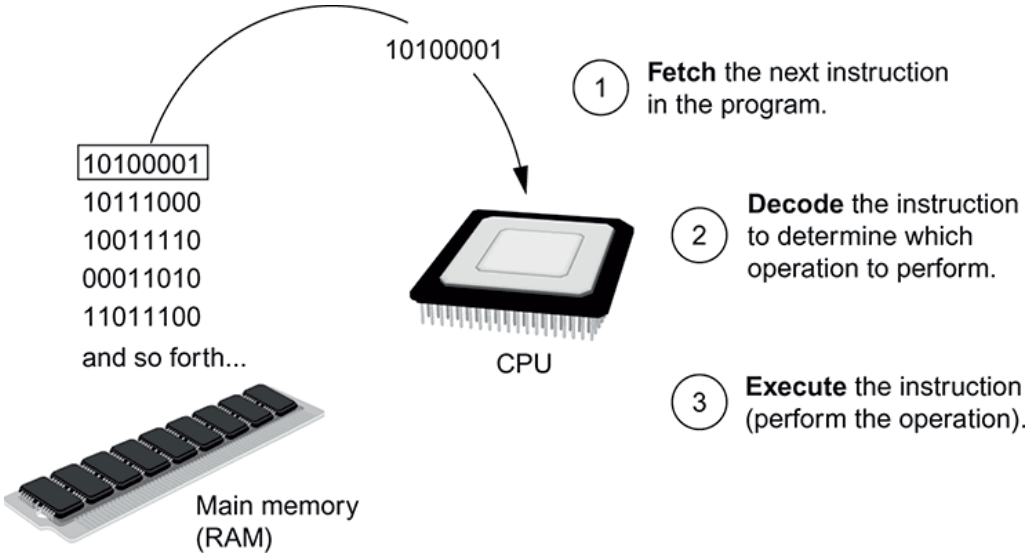
When a CPU executes the instructions in a program, it is engaged in a process that is known as the ***fetch-decode-execute cycle***. This cycle, which consists of three steps, is repeated for each instruction in the program. The steps are:

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

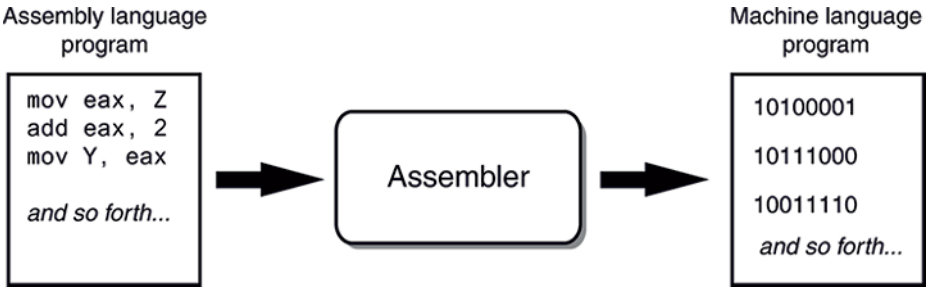
Computers can only execute programs that are written in machine language. As previously mentioned, a program can have thousands or even millions of binary instructions, and writing such a program would be very tedious and time consuming. Programming in machine language would also be very difficult, because 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.

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.

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 Ada Lovelace, a 19th century mathematician who published an algorithm that is considered by many to be the first computer program.
BASIC	**B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode 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	**FOR**mula **TRAN**slator was the first high-level programming language. It was designed in the 1950s for performing complex mathematical calculations.
COBOL	**C**ommon **B**usiness-**O**riented **L**anguage 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.
Rust	The Rust programming language is designed for high performance, memory safety, and concurrent execution. It was announced in 2010 by Mozilla Research.
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.

Keywords, 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 ***keywords*** or ***reserved words***. Each keyword has a specific meaning, and cannot be used for any other purpose. **Table 1-2** shows all of the Python keywords.

Table 1-2

The Python keywords

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

In addition to keywords, 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 keywords 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 keywords, 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.

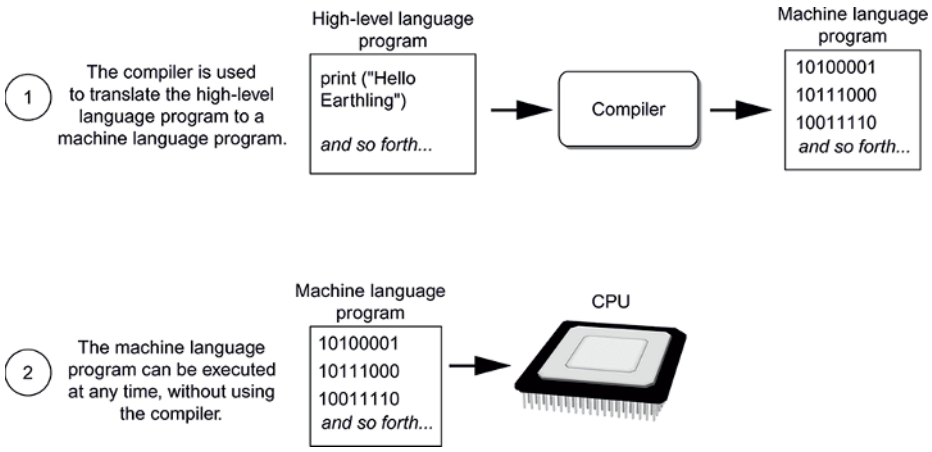
The individual instructions that you use to write a program in a high-level programming language are called ***statements***. A programming statement can consist of keywords, operators, punctuation, and other allowable programming elements, arranged in the proper sequence to perform an operation.

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 in which a program has been written, the programmer will use either a compiler or an interpreter to make the translation.

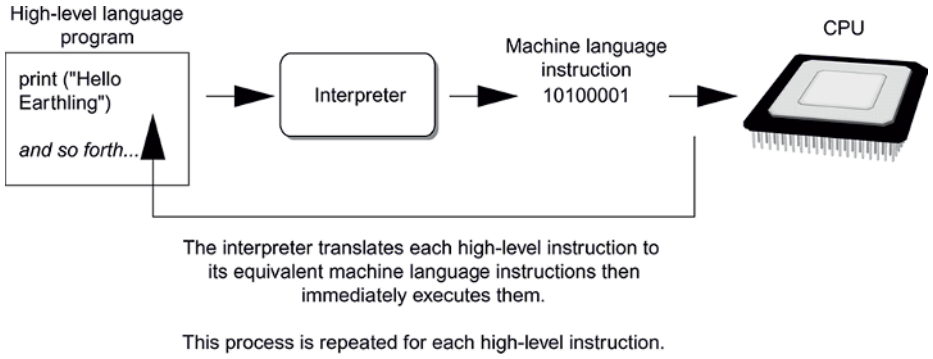
A ***compiler*** is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed. This is shown in **Figure 1-18**. As shown in the figure, compiling and executing are two different processes.

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 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 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 keyword, 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 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.

1.4: (Noninteractive) Checkpoint Questions from the Book

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 keyword, 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 type *python* in the Windows search box. In the search results, you will see a program named something like *Python 3.5*. (The "3.5" is the version of Python that is installed. At the time this is being written, Python 3.5 is the latest version.) Clicking this 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 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52)
[MSC v.1928 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 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, or Linux computer, press Ctrl-D.

NOTE: In **Chapter 2**, we will 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.

---

a. `RowNumber`

b. `RowID`

c. `ID`

d. `RowIndex`

27. A _____ key is two or more columns that are combined to create a unique value.

a. combined

b. compound

c. composite

d. joined

28. The `sqlite3` module defines an exception named _____that is thrown any time a database error occurs.

a. `DBException`

b. `DataError`

c. `SQLException`

d. `Error`

29. This is a column in one table that references a primary key in another table.

a. secondary key

b. fake key

c. foreign key

d. duplicate key

True or False

1. For storing large amounts of data, a DBMS is preferable to a traditional file.

2. All identity columns in a table must contain the same value.

3. A table can have more than one primary key.

4. When you connect to a database in SQLite, if the database does not exist an exception will be thrown.

5. When making changes to an SQLite database, those changes are not saved to the database until the `Connection` object's `commit` method is called.

6. With SQLite, you use the `Cursor` object's `execute` method to pass an SQL statement to the DBMS.

7. A database can have only one table.

8. It is possible to update more than one row with the SQL `UPDATE` statement.

9. When you create a table in SQLite, the `RowID` column is not automatically created.

10. In SQLite, when you designate a column as `INTEGER PRIMARY KEY`, that column becomes an alias for the `RowID` column.

Short Answer

1. If you are writing an application to store the customer and inventory records for a large business, why would you not want to use traditional text or binary files?

Index