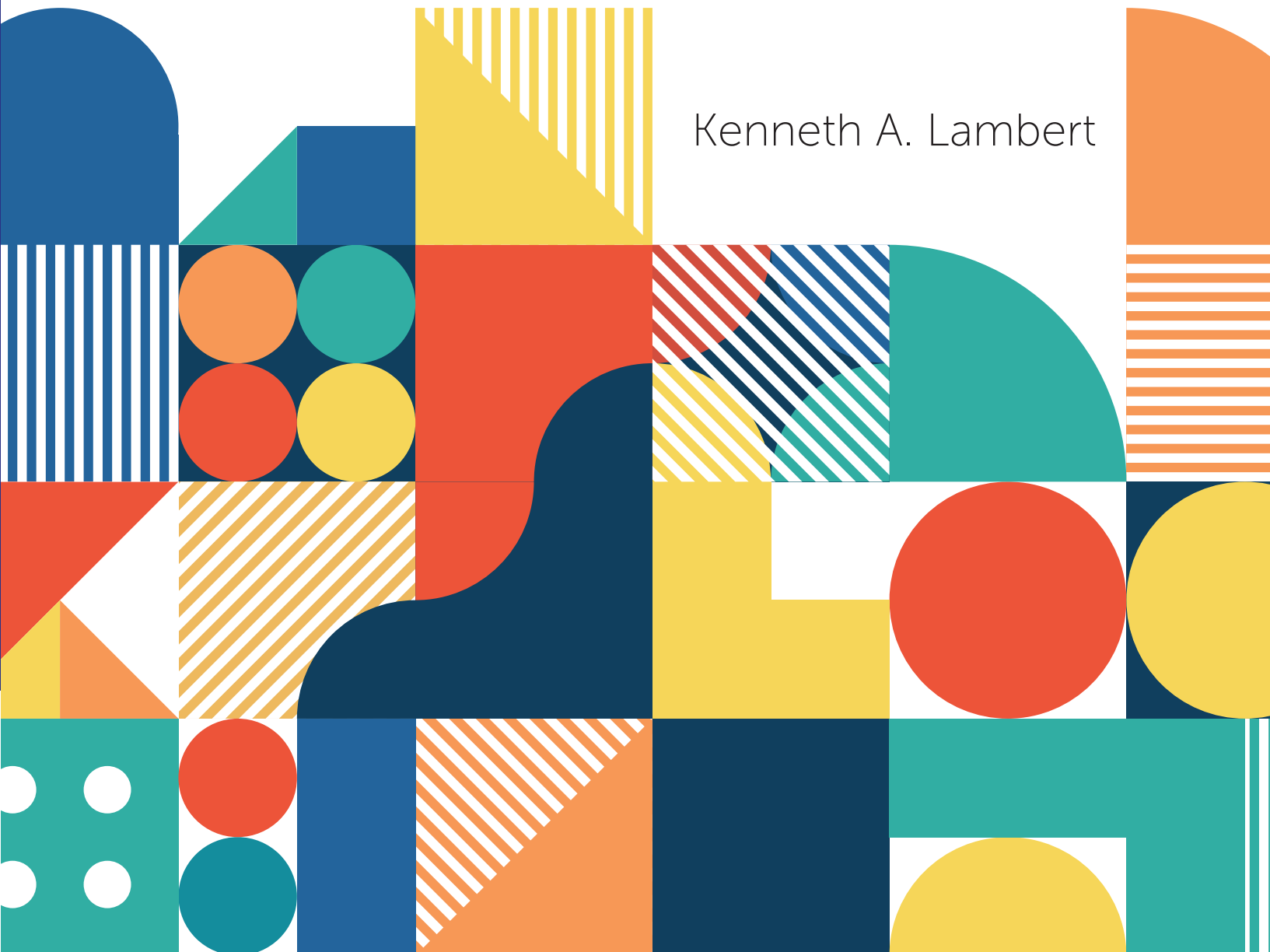
Cengage

Fundamentals of Python | First Programs

Third Edition



Kenneth A. Lambert

Fundamentals of Python

First Programs

Third Edition

Kenneth A. Lambert

Cengage

Australia • Brazil • Canada • Mexico • Singapore • United Kingdom • United States

Fundamentals of Python: First Programs,
Third Edition
Kenneth A. Lambert

SVP, Product: Cheryl Costantini

VP, Product: Thais Alencar

Portfolio Product Director: Rita Lombard

Portfolio Product Manager: Tran Pham

Product Assistant: Anh Nguyen

Learning Designer: Mary Convertino

Senior Content Manager: Michelle Ruelos Cannistraci

Digital Project Manager: John Smigelski

Technical Editor: Danielle Shaw

Developmental Editor: Spencer Peppet

VP, Product Marketing: Jason Sakos

Director, Product Marketing: Danae April

Product Marketing Manager: Mackenzie Paine

Content Acquisition Analyst: Ann Hoffman

Production Service: Straive

Designer: Erin Griffin

Cover Image Source: Armagadon/shutterstock.com

Library of Congress Control Number: 2023904384

ISBN: 978-0-357-88101-9

Cengage
200 Pier 4 Boulevard
Boston, MA 02210
USA

Printed in the United States of America
Print Number: 01 Print Year: 2023

Brief Contents

About the Author xi
Preface xiii

Chapter 1 Introduction 1

Chapter 2 Software Development, Data Types, and Expressions 27

Chapter 3 Loops and Selection Statements 53

Chapter 4 Strings and Text Files 87

Chapter 5 Lists and Dictionaries 115

Chapter 6 Design with Functions 145

Chapter 7 Design with Recursion 165

Chapter 8 Simple Graphics and Image Processing 189

Chapter 9 Graphical User Interfaces 223

Chapter 10 Design with Classes 265

Chapter 11 Data Analysis and Visualization 321

Chapter 12 Multithreading, Networks, and Client/Server Programming 357

Chapter 13 Searching, Sorting, and Complexity Analysis 393

Appendix A Python Resources 429

Appendix B Installing the `images` and `breezypythongui` Libraries 431

Appendix C The API for Image Processing 433

Appendix D Transition from Python to Java and C++ 435

Appendix E Suggestions for Further Reading 437

Glossary **439**
Index **453**

Contents

About the Author xi
Preface xiii

Chapter 1

Introduction 1

1.1 Two Fundamental Ideas of Computer Science: Algorithms and Information Processing 1
Algorithms 2
Information Processing 3

1.2 The Structure of a Modern Computer System 4
Computer Hardware 4
Computer Software 5

1.3 A Not-So-Brief History of Computing Systems 7
Before Electronic Digital Computers 8
Human Beings as Computers (1940–1945) 10
The First Electronic Digital Computers (1940–1950) 10
The First Programming Languages (1950–1965) 11
Integrated Circuits, Interaction, Time-Sharing, and Software Engineering (1965–1975) 12
Personal Computing and Networks (1975–1990) 12
Consultation, Communication, and E-Commerce (1990–2000) 14
Mobile Applications and Ubiquitous Computing (2000–present) 15

1.4 Getting Started with Python Programming 16
Running Code in the Interactive Shell 16
Input, Processing, and Output 18
Editing, Saving, and Running a Script 19
Behind the Scenes: How Python Works 20
Detecting and Correcting Syntax Errors 21
Summary 22
Key Terms 23
Review Questions 24
Programming Exercises 25
Debugging Exercise 25

Chapter 2

Software Development, Data Types, and Expressions 27

2.1 The Software Development Process 27

2.2 Strings, Assignment, and Comments 32
Data Types 33
String Literals 33
Escape Sequences 34
String Concatenation 34
Variables and the Assignment Statement 35
Program Comments and Docstrings 36

2.3 Numeric Data Types and Character Sets 37
Integers 37
Floating-Point Numbers 37
Character Sets 38

2.4 Expressions 39
Arithmetic Expressions 39
Mixed-Mode Arithmetic and Type Conversions 41

2.5 Using Functions and Modules 43
Calling Functions: Arguments and Return Values 43

The `math` Module 44
The Main Module 45
Program Format and Structure 46
Running a Script from a Terminal Command Prompt 46
Summary 48
Key Terms 49
Review Questions 49
Programming Exercises 50
Debugging Exercise 51

Chapter 3

Loops and Selection Statements 53

3.1 Definite Iteration: The `for` Loop 53
Executing a Statement a Given Number of Times 54
Count-Controlled Loops 55
Augmented Assignment 56
Loop Errors: Off-by-One Error 56
Traversing the Contents of a Data Sequence 57
Specifying the Steps in the Range 57
Loops That Count Down 58
3.2 Formatting Text for Output 58
3.3 Selection: `if` and `if-else` Statements 64
The Boolean Type, Comparisons, and Boolean Expressions 64
`if-else` Statements 65
One-Way Selection Statements 66
Multiway `if` Statements 67
Logical Operators and Compound Boolean Expressions 68
Short-Circuit Evaluation 70
Testing Selection Statements 70
3.4 Conditional Iteration: The `while` Loop 71
The Structure and Behavior of a `while` Loop 72
Count Control with a `while` Loop 73
The `while True` Loop and the `break` Statement 74
Random Numbers 75
Loop Logic, Errors, and Testing 76
Summary 81
Key Terms 82
Review Questions 82
Programming Exercises 84
Debugging Exercise 86

Chapter 4

Strings and Text Files 87

4.1 Accessing Characters and Substrings in Strings 87
The Structure of Strings 87
The Subscript Operator 88
Slicing for Substrings 89
Testing for a Substring with `in` Operator 90
4.2 Data Encryption 91
4.3 Strings and Number Systems 93
The Positional System for Representing Numbers 93
Converting Binary to Decimal 94
Converting Decimal to Binary 95
Conversion Shortcuts 95
Octal and Hexadecimal Numbers 96
4.4 String Methods 97
4.5 Text Files 100
Text Files and Their Format 100
Writing Text to a File 101
Writing Numbers to a File 101
Reading Text from a File 101

Reading Numbers from a File 102
Accessing and Manipulating Files and Directories on Disk 104
Summary 111
Key Terms 112
Review Questions 112
Programming Exercises 113
Debugging Exercise 114

Chapter 5

Lists and Dictionaries 115

5.1 Lists 115
List Literals and Basic Operators 116
Replacing an Element in a List 118
List Methods for Inserting and Removing Elements 119
Searching a List 120
Sorting a List 121
Mutator Methods and the Value `None` 121
Aliasing and Side Effects 122
Equality: Object Identity and Structural Equivalence 123
Example: Using a List to Find the Median of a Set of Numbers 123
Tuples 124

5.2 Defining Simple Functions 126
The Syntax of Simple Function Definitions 126
Parameters and Arguments 127
The `return` Statement 127
Boolean Functions 127
Defining a `main` Function 127

5.3 Dictionaries 132
Dictionary Literals 132
Adding Keys and Replacing Values 132
Accessing Values 133
Removing Keys 133
Traversing a Dictionary 133
Example: The Hexadecimal System Revisited 134
Example: Finding the Mode of a List of Values 135
Summary 141
Key Terms 142
Review Questions 142
Programming Exercises 143
Debugging Exercise 144

Chapter 6

Design with Functions 145

6.1 A Quick Review of What Functions Are and How They Work 145
Functions as Abstraction Mechanisms 146
Functions Eliminate Redundancy 146
Functions Hide Complexity 147
Functions Support General Methods with Systematic Variations 147
Functions Support the Division of Labor 148

6.2 Problem Solving with Top-Down Design 148
The Design of the Text Analysis Program 148
The Design of the Sentence Generator Program 149
The Design of the Doctor Program 150

6.3 Managing a Program's Namespace 156
Module Variables, Parameters, and Temporary Variables 156
Scope 157
Lifetime 157
Using Keywords for Default and Optional Arguments 158
Summary 161
Key Terms 161
Review Questions 162
Programming Exercises 163
Debugging Exercise 164

Chapter 7

Design with Recursion 165

7.1 Design with Recursive Functions 165
Defining a Recursive Function 166
Recursive Algorithms 167
Tracing a Recursive Function 167
Using Recursive Definitions to Construct Recursive Functions 168
Recursion in Sentence Structure 168
Infinite Recursion 170
The Costs and Benefits of Recursion 170

7.2 Higher-Order Functions 179
Functions as First-Class Data Objects 179
Mapping 180
Filtering 181
Reducing 181
Using `lambda` to Create Anonymous Functions 182
Creating Jump Tables 182
Summary 185
Key Terms 185
Review Questions 186
Programming Exercises 187
Debugging Exercise 188

Chapter 8

Simple Graphics and Image Processing 189

8.1 Simple Graphics 189
Overview of Turtle Graphics 190
Turtle Operations 190
Setting Up a turtle.cfg File and Running IDLE 192
Object Instantiation and the `turtle` Module 192
Drawing Two-Dimensional Shapes 194
Examining an Object's Attributes 195
Manipulating a Turtle's Screen 196
Taking a Random Walk 196
Colors and the RGB System 197
Example: Filling Radial Patterns with Random Colors 198

8.2 Image Processing 203
Analog and Digital Information 204
Sampling and Digitizing Images 204
Image File Formats 204
Image-Manipulation Operations 205
The Properties of Images 205
The `images` Module 205
A Loop Pattern for Traversing a Grid 208
A Word on Tuples 209
Converting an Image to Black and White 209
Converting an Image to Grayscale 210
Copying an Image 211
Blurring an Image 212
Edge Detection 213
Reducing the Image Size 214
Summary 216
Key Terms 217
Review Questions 217
Programming Exercises 218
Debugging Exercise 221

Chapter 9

Graphical User Interfaces 223

9.1 The Behavior of Terminal-Based Programs and GUI-Based Programs 224
The Terminal-Based Version 224
The GUI-Based Version 225
Event-Driven Programming 226

9.2 Coding Simple GUI-Based Programs 226
A Simple "Hello World" Program 227
A Template for All GUI Programs 228
The Syntax of Class and Method Definitions 228
Subclassing and Inheritance as Abstraction Mechanisms 229

9.3 Windows and Window Components 229
Windows and Their Attributes 230
Window Layout 230
Types of Window Components and Their Attributes 232
Displaying Images 233
9.4 Command Buttons and Responding to Events 235
9.5 Input and Output with Entry Fields 237
Text Fields 237
Integer and Float Fields for Numeric Data 238
9.6 Defining and Using Instance Variables 240
9.7 Other Useful GUI Resources 246
Using Nested Frames to Organize Components 247
Multiline Text Areas 248
File Dialogs 250
Obtaining Input with Prompter Boxes 252
Check Buttons 253
Radio Buttons 254
Keyboard Events 256
Working with Colors 256
Using a Color Chooser 258
Summary 260
Key Terms 260
Review Questions 261
Programming Exercises 262
Debugging Exercise 263

Chapter 10

Design with Classes 265

10.1 Getting Inside Objects and Classes 266
A First Example: The `Student` Class 266
Docstrings 268
Method Definitions 269
The `__init__` Method and Instance Variables 269
The `__str__` Method 269
Accessors and Mutators 270
The Lifetime of Objects 270
Rules of Thumb for Defining a Simple Class 271
10.2 Data-Modeling Examples 279
Rational Numbers 279
Rational Number Arithmetic and Operator Overloading 281
Comparison Methods 282
Equality and the `__eq__` Method 282
The `__repr__` Method for Printing an Object in IDLE 283
Savings Accounts and Class Variables 284
Putting the Accounts into a Bank 286
Using `pickle` for Permanent Storage of Objects 288
Input of Objects and the `try-except` Statement 289
Playing Cards 290
10.3 Building a New Data Structure: The Two-Dimensional Grid 298
The Interface of the `Grid` Class 298
The Implementation of the `Grid` Class: Instance Variables for the Data 300
The Implementation of the `Grid` Class: Subscript and Search 301
10.4 Structuring Classes with Inheritance and Polymorphism 305
Inheritance Hierarchies and Modeling 305
Example 1: A Restricted Savings Account 306
Example 2: The Dealer and a Player in the Game of Blackjack 308
Polymorphic Methods 312
The Costs and Benefits of Object-Oriented Programming 312
Summary 315
Key Terms 316
Review Questions 317
Programming Exercises 318
Debugging Exercise 320

Chapter 11

Data Analysis and Visualization 321

11.1 Some Basic Functions for Analyzing a Data Set 322
Computing the Maximum, Minimum, and Mean 323
Computing the Median 323
Computing the Mode and Modes 324
Computing the Standard Deviation 325
Using the NumPy Library 326

11.2 Visualizing a Data Set 333
Pie Charts 335
Bar Charts 337
Scatter Plots 339
Line Plots 340
Histograms 342

11.3 Working with More Complex Data Sets 343
Creating a Data Set with `pandas` 344
Visualizing Data with `pandas` and `matplotlib.pyplot` 344
Accessing Columns and Rows in a Data Frame 345
Creating a Data Frame from a CSV File 346
Cleaning the Data in a Data Frame 347
Accessing Other Attributes of a Data Frame 348
Summary 354
Key Terms 354
Review Questions 354
Programming Exercises 355

Chapter 12

Multithreading, Networks, and Client/Server Programming 357

12.1 Threads and Processes 357
Threads 358
Sleeping Threads 360
Producer, Consumer, and Synchronization 362

12.2 The Readers and Writers Problem 368
Using the `SharedCell` Class 369
Implementing the Interface of the `SharedCell` Class 370
Implementing the Helper Methods of the `SharedCell` Class 371
Testing the `SharedCell` Class with a `Counter` Object 372
Defining a Thread-Safe Class 373

12.3 Networks, Clients, and Servers 374
IP Addresses 374
Ports, Servers, and Clients 375
Sockets and a Day/Time Client Script 375
A Day/Time Server Script 377
A Two-Way Chat Script 379
Handling Multiple Clients Concurrently 380
Summary 388
Key Terms 389
Review Questions 389
Programming Exercises 390
Debugging Exercise 392

Chapter 13

Searching, Sorting, and Complexity Analysis 393

13.1 Measuring the Efficiency of Algorithms 394
Measuring the Run Time of an Algorithm 394
Counting Instructions 396

13.2 Complexity Analysis 398
Orders of Complexity 398
Big-O Notation 400

The Role of the Constant of Proportionality 400
Measuring the Memory Used by an Algorithm 400
13.3 Search Algorithms 401
Search for a Minimum 401
Sequential Search of a List 402
Best-Case, Worst-Case, and Average-Case Performance 403
Binary Search of a List 403
13.4 Basic Sort Algorithms 405
Selection Sort 405
Bubble Sort 406
Insertion Sort 408
Best-Case, Worst-Case, and Average-Case Performance Revisited 409
13.5 Faster Sorting 409
Quicksort 410
Partitioning 410
Complexity Analysis of Quicksort 410
Implementation of Quicksort 411
Merge Sort 413
Implementing the Merging Process 413
Complexity Analysis of Merge Sort 416
13.6 An Exponential Algorithm: Recursive Fibonacci 416
Converting Fibonacci to a Linear Algorithm 417
Summary 423
Key Terms 424
Review Questions 424
Programming Exercises 425

Appendix A

Python Resources 429

Appendix B

Installing the `images` and `breezypythongui` Libraries 431

Appendix C

The API for Image Processing 433

Appendix D

Transition from Python to Java and C++ 435

Appendix E

Suggestions for Further Reading 437

Glossary 439
Index 453

About the Author

Kenneth A. Lambert is Professor of Computer Science Emeritus at Washington and Lee University. He taught introductory programming courses for 37 years and has been an active researcher in computer science education. Lambert has co-authored a series of introductory C++ textbooks with Douglas Nance and Thomas Naps and a series of introductory Java textbooks with Martin Osborne. He is the author of Python textbooks for CS1 and CS2 college-level courses and a Python textbook for teens. He is also the co-creator of the BreezySwing framework and is the creator of the breezypythongui framework.

Dedication

To my grandchildren—Lucy, Wyatt, Cuba, and Van

Kenneth A. Lambert

Lexington, VA

Preface

"Everyone should learn how to code." That's my favorite quote from Suzanne Keen, formerly the Thomas Broadus Professor of English and Dean of the College at Washington and Lee University, where I have taught computer science for more than 30 years. The quote also states the reason why I wrote the first and second editions of *Fundamentals of Python: First Programs*, and why I now offer you this third edition. The book is intended for an introductory course in programming and problem solving. It covers the material taught in a typical Computer Science 1 (CS1) course at the undergraduate or high school level.

This book covers five major aspects of computing:

1. **Programming basics**—Data types, control structures, algorithm development, and program design with functions are basic ideas that you need to master in order to solve problems with computers. This book examines these core topics in detail and gives you practice employing your understanding of them to solve a wide range of problems.
2. **Object-oriented programming (OOP)**—Object-oriented programming is the dominant programming paradigm used to develop large software systems. This book introduces you to the fundamental principles of OOP and enables you to apply them successfully.
3. **Data and information processing**—Most useful programs rely on data structures to solve problems. These data structures include strings, arrays, files, lists, and dictionaries. This book introduces you to these commonly used data structures and includes examples that illustrate criteria for selecting the appropriate data structures for given problems.
4. **Software development life cycle**—Rather than isolate software development techniques in one or two chapters, this book deals with them throughout in the context of numerous case studies. Among other things, you'll learn that coding a program is often not the most difficult or challenging aspect of problem solving and software development.
5. **Contemporary applications of computing**—The best way to learn about programming and problem solving is to create interesting programs with real-world applications. In this book, you'll begin by creating applications that involve numerical problems and text processing. For example, you'll learn the basics of encryption techniques, such as those that are used to make your credit card number and other information secure on the Internet. But unlike many other introductory texts, this one does not restrict itself to problems involving numbers and text. Most contemporary applications involve graphical user interfaces, event-driven programming, graphics, image manipulation, network communications, and data analysis. These topics are not consigned to the margins but are presented in depth after you have mastered the basics of programming.

Why Python?

Computer technology and applications have become increasingly more sophisticated over the past three decades, and so has the computer science curriculum, especially at the introductory level. Today's students learn a bit of programming and problem solving and are then expected to move quickly into topics like software development, complexity analysis, and data structures that 35 years ago were relegated to advanced courses. In addition, the ascent of object-oriented programming as the dominant paradigm of problem solving has led instructors and textbook authors to implant powerful, industrial-strength programming languages such as

C++ and Java in the introductory curriculum. As a result, instead of experiencing the rewards and excitement of solving problems with computers, beginning computer science students often become overwhelmed by the combined tasks of mastering advanced concepts as well as the syntax of a programming language.

This book uses the Python programming language as a way of making the first year of studying computer science more manageable and attractive for students and instructors alike. Python has the following pedagogical benefits:

- Python has simple, conventional syntax. Python statements are very close to those of pseudocode algorithms, and Python expressions use the conventional notation found in algebra. Thus, students can spend less time learning the syntax of a programming language and more time learning to solve interesting problems.
- Python has safe semantics. Any expression or statement whose meaning violates the definition of the language produces an error message.
- Python scales well. It is very easy for beginners to write simple programs in Python. Python also includes all of the advanced features of a modern programming language, such as support for data structures and object-oriented software development, for use when they become necessary.
- Python is highly interactive. Expressions and statements can be entered at an interpreter's prompts to allow the programmer to try out experimental code and receive immediate feedback. Longer code segments can then be composed and saved in script files to be loaded and run as modules or standalone applications.
- Python is general purpose. In today's context, this means that the language includes resources for contemporary applications, including media computing and networks.
- Python is free and is in widespread use in industry. Students can download Python to run on a variety of devices. There is a large Python user community, and expertise in Python programming has great résumé value.

To summarize these benefits, Python is a comfortable and flexible vehicle for expressing ideas about computation, both for beginners and for experts. If students learn these ideas well in the first course, they should have no problems making a quick transition to other languages needed for courses later in the curriculum. Most importantly, beginning students will spend less time staring at a computer screen and more time thinking about interesting problems to solve.

Organization of the Text

The approach of this text is easygoing, with each new concept introduced only when it is needed.

Chapter 1 introduces computer science by focusing on two fundamental ideas, algorithms and information processing. A brief overview of computer hardware and software, followed by an extended discussion of the history of computing, sets the context for computational problem solving.

Chapters 2 and 3 cover the basics of problem solving and algorithm development using the standard control structures of expression evaluation, sequencing, Boolean logic, selection, and iteration with the basic numeric data types. Emphasis in these chapters is on problem solving that is both systematic and experimental, involving algorithm design, testing, and documentation.

Chapters 4 and 5 introduce the use of the strings, text files, lists, and dictionaries. These data structures are both remarkably easy to manipulate in Python and support some interesting applications. Chapter 5 also introduces simple function definitions as a way of organizing algorithmic code.

Chapter 6 explores the technique and benefits of procedural abstraction with function definitions. Top-down design and stepwise refinement with functions are examined as means of structuring code to solve complex problems. Details of namespace organization (parameters, temporary variables, and module variables) and communication among software components are discussed.

Chapter 7 examines recursive design with functions. A section on functional programming with higher-order functions shows how to exploit functional design patterns to simplify solutions.

Chapter 8 focuses on the use of existing objects and classes to compose programs. Special attention is paid to the application programming interface (API), or set of methods, of a class of objects and the manner in which objects cooperate to solve problems. This chapter also introduces two contemporary applications of computing: graphics and image processing. These are areas in which object-based programming is particularly useful.

Chapter 9 introduces the definition of new classes to construct graphical user interfaces (GUIs). The chapter contrasts the event-driven model of GUI programs with the process-driven model of terminal-based programs. The chapter explores the creation and layout of GUI components, as well as the design of GUI-based applications using the model/view pattern. The initial approach to defining new classes in this chapter is unusual for an introductory textbook: students learn that the easiest way to define a new class is to customize an existing class using subclassing and inheritance.

Chapter 10 continues the exploration of object-oriented design with the definition of entirely new classes. Several examples of simple class definitions from different application domains are presented. Some of these are then integrated into more realistic applications to show how object-oriented software components can be used to build complex systems. Emphasis is on designing appropriate interfaces for classes that exploit polymorphism.

Chapter 11 introduces tools and techniques for performing data analysis, a fast-growing application area of computer science. Topics include the acquisition and cleaning of data sets, applying functions to determine relationships among data, and deploying graphs, plots, and charts to visualize these relationships.

Chapter 12 covers advanced material related to several important areas of computing: concurrent programming, networks, and client/server applications. This chapter thus gives students challenging experiences near the end of the first course. This chapter introduces multithreaded programs and the construction of simple network-based client/server applications.

Chapter 13 covers some topics addressed at the beginning of a traditional CS2 course. This chapter introduces complexity analysis with big-O notation. Enough material is presented to enable you to perform simple analyses of the running time and memory usage of algorithms and data structures, using search and sort algorithms as examples.

New to This Edition

The third edition includes the following new or updated content and features:

- A new chapter (Chapter 7) on design with recursion. This chapter incorporates and expands on material on recursive functions and higher-order functions from Chapter 6 of the second edition.
- A new chapter (Chapter 11) on data analysis and visualization. This chapter introduces tools and techniques for acquiring data sets, cleaning them, and applying functions to them to determine relationships which can be visualized in plots, charts, and graphs.
- Updated coverage of the history of computing in Chapter 1.
- New fail-safe programming sections added to most chapters to demonstrate best practices for programming securely.
- New list of key terms in each chapter.
- Updated end-of-chapter review questions and programming exercises.
- End-of-chapter programming exercises mapped to the learning objectives for each chapter.
- New debugging exercises in each chapter provide examples of challenging programming errors and give you experience in diagnosing and correcting them.
- Several new case studies as well as new or updated programming exercises.
- Text revisions throughout with a focus on readability.

Features of the Text

This book explains and develops concepts carefully, using frequent examples and diagrams. New concepts are then applied in complete programs to show how they aid in solving problems. The chapters place an early and consistent emphasis on good writing habits and neat, readable documentation.

The book includes several other important features:

- **Chapter Objectives:** Each chapter begins with a set of learning objectives which describe the skills and concepts you will acquire from a careful reading of the chapter.
- **Chapter Summary:** Each chapter ends with a summary of the major concepts covered in the chapter.
- **Key Terms:** When a technical term is introduced in the text, it appears in boldface. The list of terms appears after the chapter summary. Definitions of the key terms are provided in the glossary.

Exercise

Exercises: Most major sections of each chapter end with exercise questions that reinforce the reading by asking basic questions about the material in the section.

Case Study

Case Studies: The Case Studies present complete Python programs ranging from the simple to the substantial. To emphasize the importance and usefulness of the software development life cycle, case studies are discussed in the framework of a user request, followed by analysis, design, implementation, and suggestions for testing, with well-defined tasks performed at each stage. Some case studies are extended in end-of-chapter programming exercises.

Fail-Safe Programming

Fail-Safe Programming: Fail-Safe Programming sections include a discussion of ways to make a program detect and respond gracefully to disturbances in its runtime environment.

Review Questions

Review Questions: Multiple-choice review questions allow you to revisit the concepts presented in each chapter.

Programming Exercises

Programming Exercises: Each chapter ends with a set of programming projects of varying difficulty. Each programming exercise is mapped to one or more relevant chapter learning objectives and gives you the opportunity to design and implement a complete program that utilizes major concepts presented in that chapter.

Debugging Exercises

Debugging Exercises: Debugging exercises illustrate a typical program error with suggestions for repairing it.

- **A software toolkit for image processing:** This book comes with an open-source Python toolkit for the easy image processing discussed in Chapter 8. The toolkit can be obtained with the ancillaries at **www.cengage.com** or at **https://kennethalambert.com/python/**
- **A software toolkit for GUI programming:** This book comes with an open-source Python toolkit for the easy GUI programming introduced in Chapter 9. The toolkit can be obtained with the ancillaries at **www.cengage.com** or at **https://kennethalambert.com/breezypythongui/**
- **Appendices:** Five appendices include information on obtaining Python resources, installing the toolkits, using the toolkits' interfaces, and suggestions for further reading.
- **Glossary:** Definitions of key terms are collected in a glossary.

Inclusivity and Diversity

Cengage is committed to providing educational content that is inclusive and welcoming to *all* learners. Research demonstrates that students who experience a sense of belonging in class more successfully make meaning out of, and find relevance in, what they encounter in learning content. To improve both the learning process and outcomes, our

materials seek to affirm the fullness of human diversity with respect to ability, language, culture, gender, age, socio-economics, and other forms of human difference that students may bring to the classroom.

Across the computing industry, standard coding language, such as "Master" and "Slave" is being retired in favor of language that is more inclusive, such as "Supervisor/Worker," "Primary/Replica," or "Leader/Follower." At this time, different software development and social media companies are adopting their own replacement language and currently there is no shared standard. In addition, the terms "Master" and "Slave" remain deeply embedded in legacy code and understanding this terminology remains necessary for new programmers. When required for understanding, Cengage will introduce the non-inclusive term in the first instance but will then provide an appropriate replacement terminology for the remainder of the discussion or example. We appreciate your feedback as we work to make our products more inclusive for all.

For more information about Cengage's commitment to inclusivity and diversity, please visit **https://www.cengage.com/inclusion-diversity/**

Course Solutions

Online Learning Platform: MindTap

Today's leading online learning platform, MindTap for *Fundamentals of Python*, *Third Edition* provides complete control to craft a personalized, engaging learning experience that challenges students, builds confidence, and elevates performance.

MindTap introduces students to core concepts from the beginning of the course, using a simplified learning path that progresses from understanding to application and delivers access to eTextbooks, study tools, interactive media, auto-graded assessments, and performance analytics.

MindTap activities for *Fundamentals of Python: First Programs* are designed to help students build the skills needed in today's workforce. Research shows employers seek critical thinkers, troubleshooters, and creative problem-solvers to stay relevant in our fast-paced, technology-driven world. MindTap achieves this with assignments and activities that provide hands-on practice and real-life relevance. Students are guided through assignments that reinforce basic knowledge and understanding before moving on to more challenging problems.

All MindTap activities and assignments are tied to defined chapter learning objectives. Hands-on coding labs provide real-life application and practice. Readings and dynamic visualizations support the lecture, while a post-course assessment measures exactly how much a student has learned. MindTap provides the analytics and reporting to easily see where the class stands in terms of progress, engagement, and completion rates. The content and learning path can be used as provided, customized directly in the MindTap platform, or integrated into the Learning Management System (LMS) to meet the needs of a particular course . Instructors can control what students see and when they see it. Learn more at **https://www.cengage.com/mindtap**.

In addition to the readings, the MindTap for *Fundamentals of Python: First Programs, Third Edition* includes the following:

- **Coding labs.** These supplemental assignments provide real-world application and encourage students to practice new programming concepts in a complete online IDE. New and improved Guided Feedback provides personalized and immediate feedback to students as they proceed through their coding assignments so that they can understand and correct errors in their code.
- **Gradeable assessments and activities.** All assessments and activities from the readings are available as gradeable assignments within MindTap, including Exercises and Review Questions.
- **Video quizzes.** These graded assessments provide a visual explanation of foundational programming concepts that can be applied across multiple languages. Questions accompany each video to confirm understanding of new material.

- **Interactive activities.** These embedded interactive flowcharts, tabbed explorations, and click-to-reveal experiences are designed to engage students and help them assess their understanding of introductory computer science concepts as they progress through their chapter readings.
- **Interactive study aids.** Flashcards and PowerPoint lectures help users review main concepts from the units.

Supplemental Package

Instructor and Student Resources

Additional instructor and student resources for this product are available online.

Instructor assets include an Instructor's Manual, Educator's Guide, PowerPoint® slides, and a test bank powered by Cognero®. Student assets include data sets. Sign up or sign in at **www.cengage.com** to search for and access this product and its online resources.

- **Instructor Manual.** The Instructor Manual that accompanies this textbook includes additional instructional material to assist in class preparation, including items such as Overviews, Chapter Objectives, Teaching Tips, Quick Quizzes, Class Discussion Topics, Additional Projects, Additional Resources, and Key Terms.
- **Test Bank.** Cengage Testing Powered by Cognero is a flexible, online system that allows you to:
  - Author, edit, and manage test bank content from multiple Cengage solutions.
  - Create multiple test versions in an instant.
  - Deliver tests from your LMS, your classroom, or wherever you want.
- **PowerPoint Presentations.** This text provides PowerPoint slides to accompany each chapter. Slides may be used to guide classroom presentations, to make available to students for chapter review, or to print as classroom handouts. Files are provided for every figure in the text. Instructors may use the files to customize PowerPoint slides, illustrate quizzes, or create handouts.
- **Solution and Answer Guide.** Solutions and rationales to review questions and exercises are provided to assist with grading and student understanding.
- **Solutions.** Solutions to all programming exercises and case studies are available. If an input file is needed to run a programming exercise, it is included with the solution file.
- **Data Files.** Data files necessary to complete some of the steps in the programming exercises are available. If an input file is needed to run a program, it is included with the source code.
- **Educator's Guide.** The Educator's Guide contains a detailed outline of the corresponding MindTap course.
- **Transition Guide.** The Transition Guide outlines information on what has changed from the Second Edition.

Supplements can be found at **https://faculty.cengage.com/**. Sign In or create an account, then search for this title. You can save the title for easy access and then download the resources that you need.

Acknowledgments

I would like to thank my good friend, Martin Osborne, for many years of advice, friendly criticism, and encouragement on several of my book projects. I am also grateful to the many students and faculty colleagues at Washington and Lee University who have used earlier editions of this book and given helpful feedback on it over the life of those editions.

In addition, I would like to thank the following reviewers for the time and effort they contributed to *Fundamentals of Python*: Eric Williamson, Liberty University and Jason Carman, Horry, Georgetown Technical College.

Thank you also to Danielle Shaw, who helped to assure that the content of all data and solution files used for this text were correct and accurate.

Finally, thanks to the individuals at Cengage who made this book possible: Tran Pham, Product Manager; Mary Convertino, Learning Designer; Michelle Ruelos Cannistraci, Senior Content Manager; Troy Dundas, Technical Content Developer; Spencer Peppet, Developmental Editor; Ann Shaffer, Developmental Editor; and Ethan Wheel, Product Assistant.

Chapter 1

Introduction

Learning Objectives

When you complete this chapter, you will be able to:

- **1.1** Describe the basic features of an algorithm
- **1.2** Explain how hardware and software collaborate in a computer's architecture
- **1.3** Summarize a brief history of computing
- **1.4** Compose and run a simple Python program

As a reader of this book, you almost certainly have played a video game and listened to digital music. It's likely that you have watched a movie on Netflix after preparing a snack in a microwave. Chances are that today you will make a phone call, send or receive a text message, take a photo, or consult your favorite social network on a smartphone, which is a small computer. You and your friends have most likely used a desktop or laptop computer to do significant coursework in high school or college.

Computer technology is almost everywhere: in our homes, schools, and in the places where we work and play. Computer technology is essential to modern entertainment, education, medicine, manufacturing, communications, government, and commerce. We have digital lifestyles in an information-based economy. Some people even claim that nature itself performs computations on information structures present in DNA and in the relationships among subatomic particles.

In the following chapters you will learn about computer science, which is the study of computation that has made this new technology and this new world possible. You will also learn how to use computers effectively and appropriately to enhance your own life and the lives of others.

1.1 Two Fundamental Ideas of Computer Science: Algorithms and Information Processing

Like most areas of study, computer science focuses on a broad set of interrelated ideas. Two of the most basic ones are algorithms and information processing. In this section, these ideas are introduced in an informal way. You will examine them in more detail in later chapters.

Algorithms

People computed long before the invention of modern computing devices, and many continue to use devices that we might consider primitive. For example, consider how merchants made change for customers in marketplaces before the existence of credit cards, pocket calculators, or cash registers. Making change can be a complex activity. It takes some mental effort to get it right every time. Let's consider what's involved in this process.

According to one method, the first step is to compute the difference between the purchase price and the amount of money that the customer gives the merchant. The result of this calculation is the total amount that the merchant must return to the purchaser. For example, if you buy a dozen eggs at the farmers' market for $2.39 and you give the farmer a $10 bill, she should return $7.61 to you. To produce this amount, the merchant selects the appropriate coins and bills that add up to $7.61.

According to another method, the merchant starts with the purchase price and goes toward the amount given. First, coins are selected to bring the price to the next dollar amount (in this case, $0.61 = 2 quarters, 1 dime, and 1 penny), then dollars are selected to bring the price to the next five-dollar amount (in this case, $2), and then, in this case, a $5 bill completes the transaction. As you will see in this book, there can be many possible methods or algorithms that solve the same problem, and the choice of the best one is a skill you will acquire with practice.

Few people can subtract three-digit numbers without resorting to some manual aids, such as pencil and paper. As you learned in grade school, you can carry out subtraction with pencil and paper by following a sequence of well-defined steps. You have probably done this many times but never made a list of the specific steps involved. Making such lists to solve problems is something computer scientists do all the time. For example, the following list of steps describes the process of subtracting two numbers using a pencil and paper:

Step 1 Write down the two numbers, with the larger number above the smaller number and their digits aligned in columns from the right.

Step 2 Assume that you will start with the rightmost column of digits and work your way left through the various columns.

Step 3 Write down the difference between the two digits in the current column of digits, borrowing a 1 from the top number's next column to the left if necessary.

Step 4 If there is no next column to the left, stop. Otherwise, move to the next column to the left, and go back to Step 3.

If the **computing agent** (in this case a human being) follows each of these simple steps correctly, the entire process results in a correct solution to the given problem. We assume in Step 3 that the agent already knows how to compute the difference between the two digits in any given column, borrowing if necessary.

To make change, most people can select the combination of coins and bills that represent the correct change amount without any manual aids, other than the coins and bills. But the mental calculations involved can still be described in a manner similar to the preceding steps, and we can resort to writing them down on paper if there is a dispute about the correctness of the change.

The sequence of steps that describes each of these computational processes is called an **algorithm**. Informally, an algorithm is like a recipe. It provides a set of instructions that tells us how to do something, such as make change, bake bread, or put together a piece of furniture. More precisely, an algorithm describes a process that ends with a solution to a problem. The algorithm is also one of the fundamental ideas of computer science. An algorithm has the following features:

1. An algorithm consists of a finite number of instructions.
2. Each individual instruction in an algorithm is well defined. This means that the action described by the instruction can be performed effectively or be **executed** by a computing agent. For example,

any computing agent capable of arithmetic can compute the difference between two digits. So, an algorithmic step that says "compute the difference between two digits" would be well defined. On the other hand, a step that says "divide a number by 0" is not well defined, because no computing agent could carry it out.

3. An algorithm describes a process that eventually halts after arriving at a solution to a problem. For example, the process of subtraction halts after the computing agent writes down the difference between the two digits in the leftmost column of digits.
4. An algorithm solves a general class of problems. For example, an algorithm that describes how to make change should work for any two amounts of money whose difference is greater than or equal to $0.00.

Creating a list of steps that describe how to make change might not seem like a major accomplishment to you. But the ability to break a task down into its component parts is one of the main jobs of a computer programmer. Once you write an algorithm to describe a particular type of computation, you can build a machine to do the computing. Put another way, if you can develop an algorithm to solve a problem, you can automate the task of solving the problem. You might not feel compelled to write a computer program to automate the task of making change, because you can probably already make change yourself fairly easily. But suppose you needed to do a more complicated task—such as sorting a list of 100 names. In that case, a computer program would be very handy.

Computers can be designed to run a small set of algorithms for performing specialized tasks, such as operating a microwave. But we can also build computers, like the one on your desktop, that are capable of performing a task described by any algorithm. These computers are truly general-purpose problem-solving machines. They are unlike any machines that were built before, and they have formed the basis of the completely new world in which we live.

Later in this book, we introduce a notation for expressing algorithms and some suggestions for designing algorithms. You will see that algorithms and algorithmic thinking are critical underpinnings of any computer system.

Information Processing

Since people first learned to write several thousand years ago, they have processed information. Information itself has taken many forms in its history, from the marks impressed on clay tablets in ancient Mesopotamia; to the first written texts in ancient Greece; to the printed words in the books, newspapers, and magazines mass-produced since the European Renaissance; to the abstract symbols of modern mathematics and science used during the past 350 years. Only recently, however, have human beings developed the capacity to automate the processing of information by building computers. In the modern world of computers, information is also commonly referred to as **data**. But what is information?

Like mathematical calculations, **information processing** can be described with algorithms. In our earlier example of making change, the subtraction steps involved manipulating symbols used to represent numbers and money. In carrying out the instructions of any algorithm, a computing agent manipulates information. The computing agent starts with some given information (known as **input**), transforms this information according to well-defined rules, and produces new information, known as **output**.

It is important to recognize that the algorithms that describe information processing can also be represented as information. Computer scientists have been able to represent algorithms in a form that can be executed effectively and efficiently by machines. They have also designed real machines, called electronic digital computers, which are capable of executing algorithms.

Computer scientists more recently discovered how to represent many other things, such as images, music, human speech, and video, as information. Many of the media and communication devices that we now take for granted would be impossible without this new kind of information processing. We examine many of these achievements in more detail in later chapters.

Exercise 1-1

These short end-of-section exercises are intended to stimulate your thinking about computing.

1. List three common types of computing agents.
2. Write an algorithm that describes the second part of the process of making change (counting out the coins and bills).
3. Write an algorithm that describes a common task, such as baking a cake.
4. Describe an instruction that is not well defined and thus could not be included as a step in an algorithm. Give an example of such an instruction.
5. In what sense is a laptop computer a general-purpose problem-solving machine?
6. List four devices that use computers and describe the information that they process. (*Hint:* Think of the inputs and outputs of the devices.)

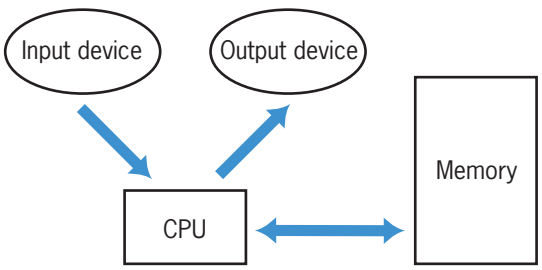
1.2 The Structure of a Modern Computer System

We now give a brief overview of the structure of modern computer systems. A modern computer system consists of **hardware** and **software**. Hardware consists of the physical devices required to execute algorithms. Software is the set of these algorithms, represented as **programs**, in particular **programming languages**. In the discussion that follows, we focus on the hardware and software found in a typical desktop computer system, although similar components are also found in other computer systems, such as smartphones and automatic teller machines (ATMs).

Computer Hardware

The basic hardware components of a computer are **memory**, a **central processing unit (CPU)**, and a set of **input/output devices**, as shown in **Figure 1-1**.

Figure 1-1 Hardware components of a modern computer system



Human users primarily interact with the input and output devices. The input devices include a keyboard, a mouse, a trackpad, a microphone, and a touchscreen. Common output devices include a monitor and speakers. Computers can also communicate with the external world through various **ports** that connect them to **networks** and to other devices such as smartphones and digital cameras. The purpose of most input devices is to convert information that human beings deal with, such as text, images, and sounds, into information for computational processing. The purpose of most output devices is to convert the results of this processing back to human-usable form.

Computer memory is set up to represent and store information in electronic form. Specifically, information is stored as patterns of **binary digits** (1s and 0s). To understand how this works, consider a basic device such as a light switch, which can only be in one of two states, on or off. Now suppose there is a bank of switches that control 16 small lights in a row. By turning the switches off or on, we can represent any pattern of 16 binary digits (1s and 0s) as patterns of lights that are on or off. As you will see later in this book, computer scientists have discovered how to represent any information, including text, images, and sound, in binary form.

Now, suppose there are 8 of these groups of 16 lights. We can select any group of lights and examine or change the state of each light within that collection. We have just developed a tiny model of computer memory. The memory has 8 cells, each of which can store 16 **bits** of binary information. A diagram of this model, in which the memory cells are filled with binary digits, is shown in **Figure 1-2**. This memory is also sometimes called **primary memory** or internal or **random access memory (RAM)**.

Figure 1-2 A model of computer memory

Cell 7	1	1	0	1	1	1	1	0	1	1	1	1	1	1	0	1
Cell 6	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1
Cell 5	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1
Cell 4	1	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
Cell 3	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1
Cell 2	0	0	1	1	1	1	0	1	1	1	0	1	1	1	0	1
Cell 1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1
Cell 0	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	0

The information stored in memory can represent any type of data, such as numbers, text, images, sound, or the instructions of a program. Once the information is stored in memory, we typically want to do something with it—that is, we want to process it. The part of a computer that is responsible for processing data is the central processing unit (CPU). This device, which is also sometimes called a **processor**, consists of electronic switches arranged to perform simple logical, arithmetic, and control operations. The CPU executes an algorithm by fetching its binary instructions from memory, decoding them, and executing them. Executing an instruction might involve fetching other binary information—the data—from memory as well.

The processor can locate data in a computer's primary memory very quickly. However, these data exist only as long as electric power comes into the computer. If the power fails or is turned off, the data in primary memory are lost. Clearly, a more permanent type of memory is needed to preserve data. This more permanent type of memory is called external or **secondary memory**, and it comes in several forms. **Magnetic storage media**, such as tapes and hard disks, allow bit patterns to be stored as patterns on a magnetic field. **Semiconductor storage media**, such as flash memory sticks and universal serial bus (USB) drives, perform much the same function with a different technology, as do **optical storage media**, such as compact disks (CDs) and digital video disks (DVDs). Some of these secondary storage media can hold much larger quantities of information than the internal memory of a computer.

Computer Software

You have learned that a computer is a general-purpose problem-solving machine. To solve any computable problem, a computer must be capable of executing any algorithm. Because it is impossible to anticipate all of the problems for which there are algorithmic solutions, there is no way to hardwire all potential algorithms into a computer's hardware. Instead, some basic operations are built into the hardware's processor and require any algorithm to use them. The algorithms are converted to binary form and then loaded, with their data, into the computer's memory. The processor can then execute the algorithms' instructions by running the hardware's more basic operations.

Any programs that are stored in memory so that they can be executed later are called software. A program stored in computer memory must be represented in binary digits, which is also known as **machine code**. Loading machine code into computer memory one digit at a time would be a tedious, error-prone task for human beings. It would be

convenient if we could automate this process to get it right every time. For this reason, computer scientists have developed another program, called a **loader**, to perform this task. A loader takes a set of machine language instructions as input and loads them into the appropriate memory locations. When the loader is finished, the machine language program is ready to execute. Obviously, the loader cannot load itself into memory, so this is one of those algorithms that must be hardwired into the computer.

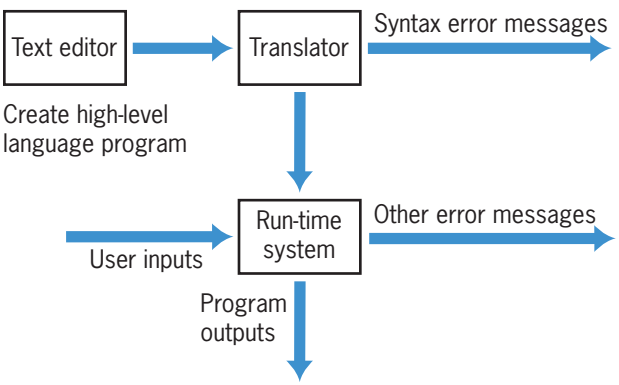
Now that a loader exists, you can load and execute other programs that make the development, execution, and management of programs easier. This type of software is called **system software**. The most important example of system software is a computer's **operating system**. You are probably already familiar with at least one of the most popular operating systems, such as Linux, Apple's macOS, and Microsoft's Windows. An operating system is responsible for managing and scheduling several concurrently running programs. It also manages the computer's memory, including the external storage, and manages communications between the CPU, the input/output devices, and other computers on a network. An important part of any operating system is its **file system**, which allows human users to organize their data and programs in permanent storage. Another important function of an operating system is to provide **user interfaces**—that is, ways for the human user to interact with the computer's software. A **terminal-based interface** accepts inputs from a keyboard and displays text output on a monitor screen. A **graphical user interface (GUI)** organizes the monitor screen around the metaphor of a desktop, with windows containing icons for folders, files, and applications. This type of user interface also allows the user to manipulate images with a pointing device such as a mouse. A **touchscreen interface** supports more direct manipulation of these visual elements with gestures such as pinches and swipes of the user's fingers. Devices that respond verbally and in other ways to verbal commands are also becoming widespread.

Another major type of software is called **applications software**, or simply **apps**. An application is a program that is designed for a specific task, such as editing a document or displaying a web page. Applications include web browsers, word processors, spreadsheets, database managers, graphic design packages, music production systems, and games, among millions of others. As you begin learning to write computer programs, you will focus on writing simple applications.

As you have learned, computer hardware can execute only instructions that are written in binary form—that is, in machine language. Writing a machine language program, however, would be an extremely tedious, error-prone task. To ease the process of writing computer programs, computer scientists have developed **high-level programming languages** for expressing algorithms. These languages resemble English and allow the author to express algorithms in a form that other people can understand.

A programmer typically starts by writing high-level language statements in a **text editor**. The programmer then runs another program called a **translator** to convert the high-level program code into executable code. Because it is possible for a programmer to make grammatical mistakes even when writing high-level code, the translator checks for **syntax errors** before it completes the translation process. If it detects any of these errors, the translator alerts the programmer via error messages. The programmer then has to revise the program. If the translation process succeeds without a syntax error, the program can be executed by the **run-time system**. The run-time system might execute the program directly on the hardware or run yet another program called an **interpreter** or **virtual machine** to execute the program. **Figure 1-3** shows the steps and software used in the coding process.

Figure 1-3 Software used in the coding process



Exercise 1-2

1. List two examples of input devices and two examples of output devices.
2. What does the central processing unit (CPU) do?
3. How is information represented in hardware memory?
4. What is the difference between a terminal-based interface and a graphical user interface (GUI)?
5. What role do translators play in the programming process?

1.3 A Not-So-Brief History of Computing Systems

Now that you have in mind some of the basic ideas of computing and computer systems, let's take a moment to examine how they have taken shape in history. **Figure 1-4** summarizes some of the major developments in the history of computing. The discussion that follows provides more details about these developments.

Figure 1-4 Summary of major developments in the history of computing

Approximate Dates	Major Developments
Before 1800	• Mathematicians discover and use algorithms • Abacus used as a calculating aid • First mechanical calculators built by Pascal and Leibniz
Nineteenth century	• Jacquard's loom • Babbage's Analytical Engine • Boole's system of logic • Hollerith's punch-card machine
1930s	• Turing publishes results on computability • Shannon's theory of information and digital switching
1940s	• First electronic digital computers
1950s	• First symbolic programming languages • Transistors make computers smaller, faster, more durable, and less expensive • Emergence of data-processing applications
1960–1975	• Integrated circuits accelerate the miniaturization of hardware • First minicomputers • Time-sharing operating systems • Interactive user interfaces with keyboard and monitor • Proliferation of high-level programming languages • Emergence of a software industry and the academic study of computer science

(continues)

Figure 1-4 Summary of major developments in the history of computing (continued)

Approximate Dates	Major Developments
1975–1990	• First microcomputers and mass-produced personal computers • GUIs become widespread • Networks and the Internet
1990–2000	• Optical storage for multimedia applications, images, sound, and video • World Wide Web, web applications, and e-commerce • Laptops
2000–present	• Wireless computing, smartphones, and mobile applications • Computers embedded and networked in an enormous variety of cars, household appliances, and industrial equipment • Social networking and use of big data in finance and commerce • Digital streaming of music and video

Before Electronic Digital Computers

Ancient mathematicians developed the first algorithms. The word *algorithm* comes from the name of a Persian mathematician, Muhammad ibn Musa al-Khwarizmi, who wrote several mathematics textbooks in the ninth century. About 2300 years ago, the Greek mathematician Euclid, the inventor of geometry, developed an algorithm for computing the greatest common divisor of two numbers.

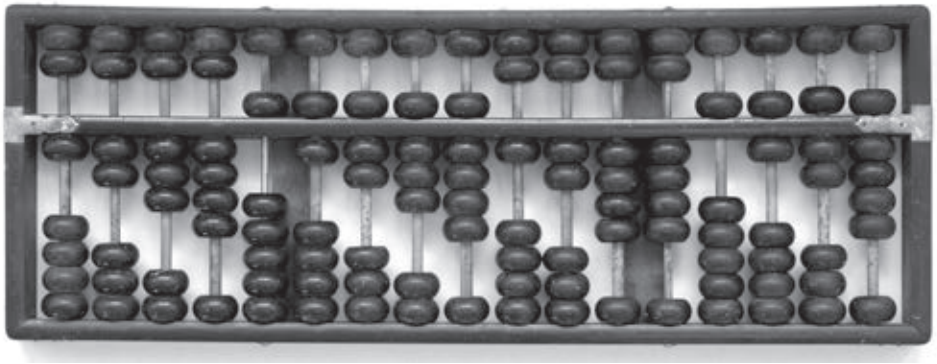
A device known as the **abacus** also appeared in ancient times. The abacus helped people perform simple arithmetic. Users calculated sums and differences by sliding beads on a grid of wires (see **Figure 1-5a**). The configuration of beads on the abacus served as the data.

In the seventeenth century, the French mathematician Blaise Pascal (1623–1662) built one of the first mechanical devices to automate the process of addition (see **Figure 1-5b**). The addition operation was embedded in the configuration of gears within the machine. The user entered the two numbers to be added by rotating some wheels. The sum or output number appeared on another rotating wheel. The German mathematician Gottfried Wilhelm Leibniz (1646–1716) built another mechanical calculator that included other arithmetic functions such as multiplication. Leibniz, who invented calculus concurrently with Newton, went on to propose the idea of computing with symbols as one of our most basic intellectual activities. He argued for a universal language in which one could solve any problem by calculating.

Early in the nineteenth century, the French engineer Joseph-Marie Jacquard (1752–1834) designed and constructed a machine that automated the process of weaving (see **Figure 1-5c**). Until then, each row in a weaving pattern had to be set up by hand, a quite tedious, error-prone process. Jacquard's loom was designed to accept input in the form of a set of punched cards. Each card described a row in a pattern of cloth. Although it was still an entirely mechanical device, Jacquard's loom possessed something that previous devices had lacked—the ability to execute an algorithm automatically. The set of cards expressed the algorithm or set of instructions that controlled the behavior of the loom. If the loom operator wanted to produce a different pattern, he just had to run the machine with a different set of cards.

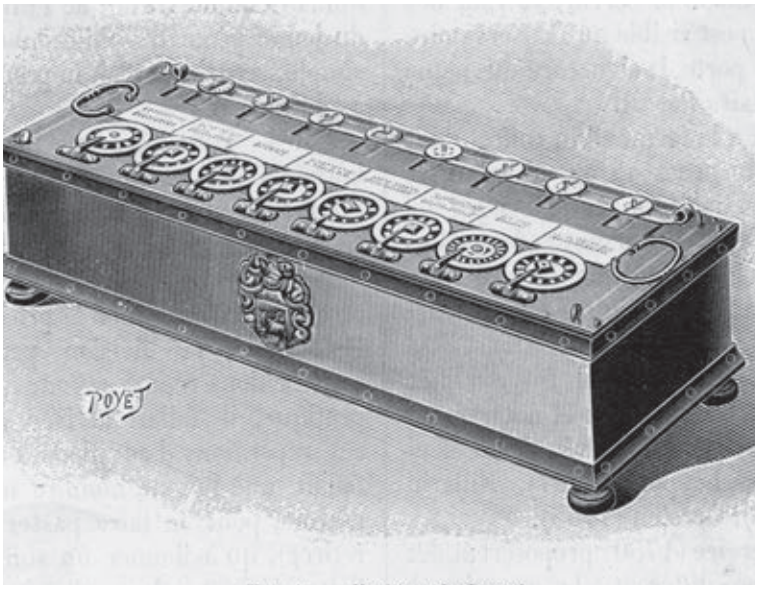
The British mathematician Charles Babbage (1792–1871) took the concept of a programmable computer a step further by designing a model of a machine that, conceptually, bore a striking resemblance to a modern general-purpose computer. Babbage conceived his machine, which he called the Analytical Engine, as a mechanical device. His design called for four functional parts: a mill to perform arithmetic operations, a store to hold data and a program, an operator to run the instructions from punched cards, and an output to produce the results on punched cards. Sadly, Babbage's computer was never built. The project perished for lack of funds near the time when Babbage himself passed away.

Figure 1-5 Some early computing devices



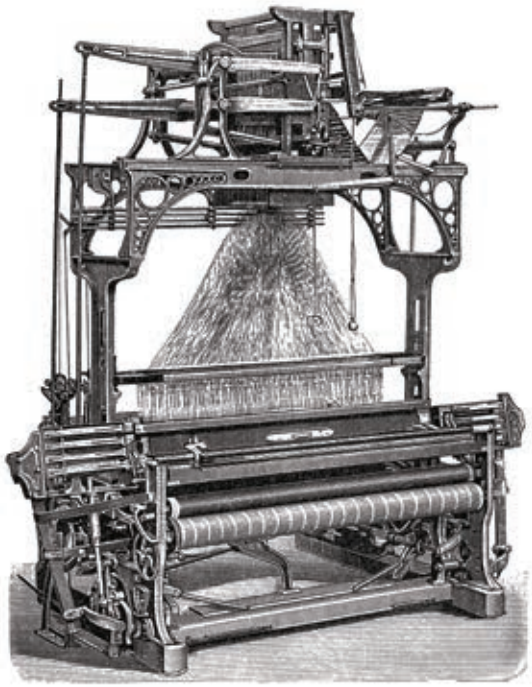
ChewHow/Shutterstock.com

(a) Abacus



Archivist/Adobe Stock Photos

(b) Pascal's Calculator



Nastasic/Getty Images

(c) Jacquard's Loom

In the last two decades of the nineteenth century, a U.S. Census Bureau statistician named Herman Hollerith (1860–1929) developed a machine that automated data processing for the U.S. Census. Hollerith's machine, which had the same component parts as Babbage's Analytical Engine, simply accepted a set of punched cards as input and then tallied and sorted the cards. His machine greatly shortened the time it took to produce statistical results on the U.S. population. Government and business organizations seeking to automate their data processing quickly adopted Hollerith's punched card machines. Hollerith was also one of the founders of a company that eventually became International Business Machines (IBM).

Also in the nineteenth century, the British secondary school teacher George Boole (1815–1864) developed a system of logic. This system consisted of a pair of values, TRUE and FALSE, and a set of three primitive operations on these values, AND, OR, and NOT. Boolean logic eventually became the basis for designing the electronic circuitry to process binary information.

A half century later, in the 1930s, the British mathematician Alan Turing (1912–1954) explored the theoretical foundations and limits of algorithms and computation. Turing's essential contributions were to develop the concept of a universal machine that could be specialized to solve any computable problems and to demonstrate that some problems are unsolvable by computers.