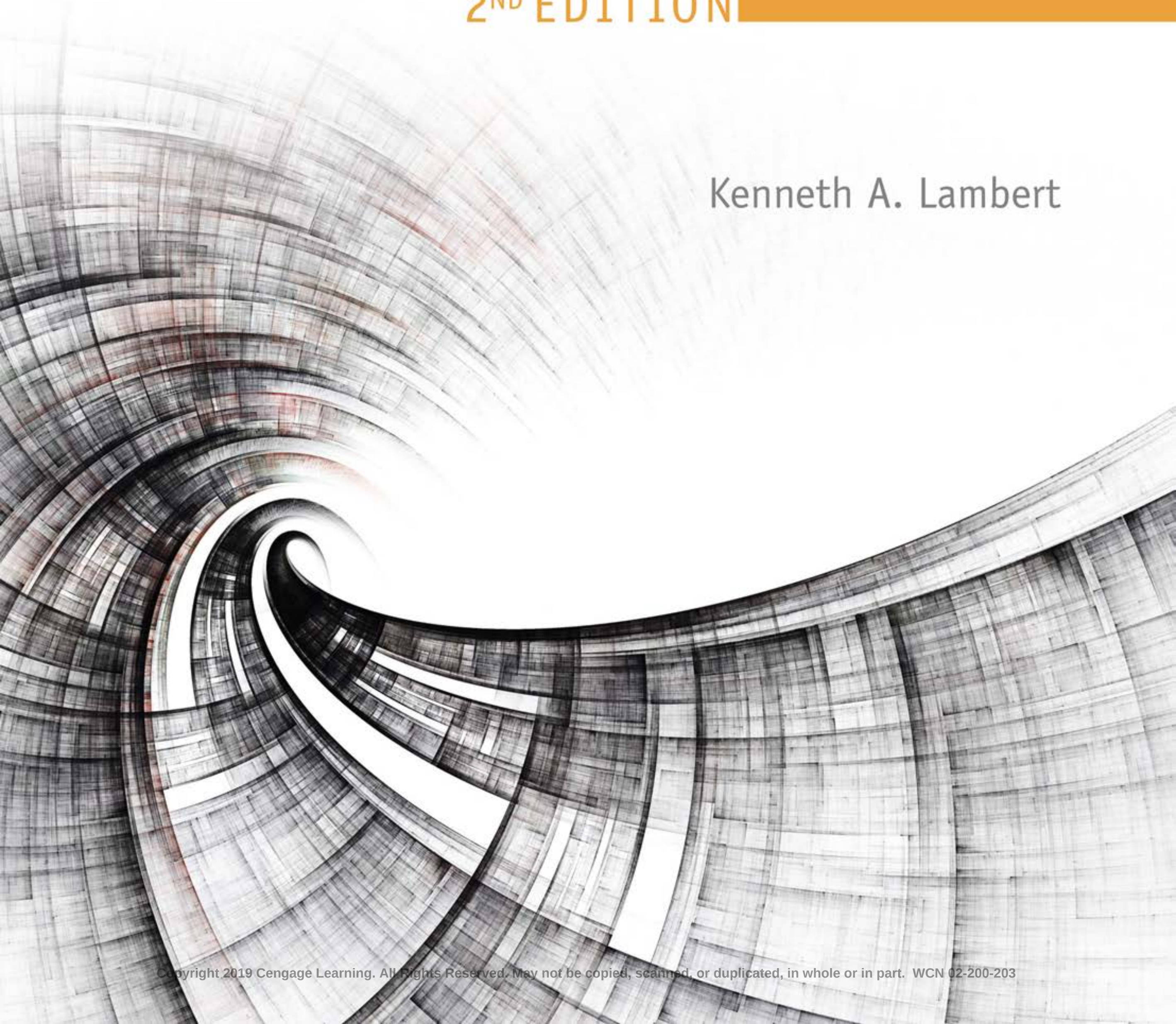
Fundamentals of

PYTHON FIRST PROGRAMS

2ND EDITION

Kenneth A. Lambert



SECOND EDITION

FUNDAMENTALS OF PYTHON: FIRST PROGRAMS

KENNETH A. LAMBERT

MARTIN OSBORNE,
CONTRIBUTING AUTHOR

CENGAGE

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

CENGAGE

Fundamentals of Python: First Programs, Second Edition

Kenneth A. Lambert

SVP, GM Skills: Jonathan Lau

Product Team Manager: Kristin McNary

Associate Product Manager: Kate Mason

Executive Director of Development: Marah Bellegarde

Senior Content Development Manager: Leigh Hefferon

Content Development Manager: Jill Gallagher

Senior Content Developer: Natalie Pashoukos

Product Assistant: Jake Toth

Marketing Director: Michele McTighe

Marketing Manager: Stephanie Albracht

Senior Content Project Manager: Jennifer Feltri-George

Senior Designer/Art Director: Diana Graham

Cover image: Digital_Art/Shutterstock.com

Production Service/Composition: SPi Global

Library of Congress Control Number: 2017952738

Softbound ISBN: 978-1-337-56009-2

Loose Leaf ISBN: 978-1-337-69934-1

Cengage
20 Channel Center Street
Boston, MA, 02210
USA

Notice to the Reader

Printed in the United States of America
Print Number: 01 Print Year: 2017

Table of Contents

Preface xiii

CHAPTER 1 Introduction 1

Two Fundamental Ideas of Computer Science:
 Algorithms and Information Processing. 2
  Algorithms 2
  Information Processing 4
Exercises. 5
The Structure of a Modern Computer System 6
  Computer Hardware 6
  Computer Software 7
Exercises. 9
A Not-So-Brief History of Computing Systems 9
  Before Electronic Digital Computers. 11
  The First Electronic Digital Computers (1940–1950) . . . 13
  The First Programming Languages (1950–1965) 14
  Integrated Circuits, Interaction,
   and Timesharing (1965–1975) 16
  Personal Computing and Networks (1975–1990) 17
  Consultation, Communication,
   and E-Commerce (1990–2000) 19
  Mobile Applications and Ubiquitous
   Computing (2000–present) 21
Getting Started with Python Programming 22
  Running Code in the Interactive Shell 22
  Input, Processing, and Output 24
  Editing, Saving, and Running a Script 27
  Behind the Scenes: How Python Works 28
Exercises. 29
Detecting and Correcting Syntax Errors 29
Exercises. 30
Suggestions for Further Reading 30
Summary 31

Review Questions 32
Projects 33

CHAPTER 2 Software Development, Data Types, and Expressions 34

The Software Development Process. 35
Exercises. 37
Case Study: Income Tax Calculator 38
Strings, Assignment, and Comments 41
Data Types 41
String Literals. 42
Escape Sequences 43
String Concatenation 43
Variables and the Assignment Statement 44
Program Comments and Docstrings. 45
Exercises. 46
Numeric Data Types and Character Sets. 47
Integers 47
Floating-Point Numbers 47
Character Sets 48
Exercises. 49
Expressions 49
Arithmetic Expressions. 50
Mixed-Mode Arithmetic and Type Conversions 52
Exercises. 53
Using Functions and Modules. 54
Calling Functions: Arguments and Return Values 54
The `math` Module 55
The Main Module 56
Program Format and Structure 57
Running a Script from a Terminal Command Prompt . . . 57
Exercises. 59
Summary 59
Review Questions 61
Projects 62

CHAPTER 3 Loops and Selection Statements 64

Definite Iteration: The `for` Loop 65
Executing a Statement a Given Number of Times. 65
Count-Controlled Loops 66
Augmented Assignment 67

Loop Errors: Off-by-One Error. 68
Traversing the Contents of a Data Sequence 68
Specifying the Steps in the Range 69
Loops That Count Down 69
Exercises. 70
Formatting Text for Output 70
Exercises. 72
Case Study: An Investment Report 73
Selection: if and if-else Statements 77
The Boolean Type, Comparisons, and Boolean Expressions 77
if-else Statements 78
One-Way Selection Statements 79
Multi-Way if Statements 80
Logical Operators and Compound Boolean Expressions . . 82
Short-Circuit Evaluation 84
Testing Selection Statements. 84
Exercises. 85
Conditional Iteration: The while Loop. 86
The Structure and Behavior of a while Loop. 86
Count Control with a while Loop 87
The while True Loop and the break Statement 88
Random Numbers 90
Loop Logic, Errors, and Testing 91
Exercises. 92
Case Study: Approximating Square Roots 92
Summary 96
Review Questions 97
Projects 99

CHAPTER 4 Strings and Text Files **102**

Accessing Characters and Substrings in Strings 103
The Structure of Strings 103
The Subscript Operator 104
Slicing for Substrings 105
Testing for a Substring with the in Operator. 105
Exercises. 106
Data Encryption 106
Exercises. 109
Strings and Number Systems. 109
The Positional System for Representing Numbers . . . 110
Converting Binary to Decimal. 111

Converting Decimal to Binary 112
Conversion Shortcuts 112
Octal and Hexadecimal Numbers 113
Exercises. 114
String Methods 115
Exercises. 118
Text Files 118
Text Files and Their Format. 118
Writing Text to a File. 119
Writing Numbers to a File 119
Reading Text from a File 120
Reading Numbers from a File 121
Accessing and Manipulating Files and Directories on Disk 122
Exercises. 125
Case Study: Text Analysis 126
Summary 130
Review Questions 131
Projects 132

CHAPTER 5 Lists and Dictionaries **134**

Lists 135
List Literals and Basic Operators 135
Replacing an Element in a List 138
List Methods for Inserting and Removing Elements . . . 138
Searching a List. 140
Sorting a List 140
Mutator Methods and the Value `None` 141
Aliasing and Side Effects. 141
Equality: Object Identity and Structural Equivalence 143
Example: Using a List to Find the Median of a Set of Numbers 143
Tuples 144
Exercises. 145
Defining Simple Functions 146
The Syntax of Simple Function Definitions 146
Parameters and Arguments. 147
The `return` Statement. 147
Boolean Functions. 148
Defining a `main` Function. 148
Exercises. 149
Case Study: Generating Sentences 150

Dictionaries 153
  Dictionary Literals 153
  Adding Keys and Replacing Values 154
  Accessing Values 154
  Removing Keys 155
  Traversing a Dictionary 155
  Example: The Hexadecimal System Revisited. 156
  Example: Finding the Mode of a List of Values 157
Exercises. 158
Case Study: Nondirective Psychotherapy 159
Summary 163
Review Questions 164
Projects 165

CHAPTER 6 Design with Functions **167**

A Quick Review of What Functions Are and How They Work. 168
  Functions as Abstraction Mechanisms 169
  Functions Eliminate Redundancy 169
  Functions Hide Complexity 170
  Functions Support General Methods with Systematic Variations 170
  Functions Support the Division of Labor 171
Exercises. 171
Problem Solving with Top-Down Design 172
  The Design of the Text-Analysis Program 172
  The Design of the Sentence-Generator Program 173
  The Design of the Doctor Program 174
Exercises. 176
Design with Recursive Functions 176
  Defining a Recursive Function 176
  Tracing a Recursive Function 177
  Using Recursive Definitions to Construct Recursive Functions 178
  Recursion in Sentence Structure 179
  Infinite Recursion 179
  The Costs and Benefits of Recursion 180
Exercises. 182
Case Study: Gathering Information from a File System . . 183
Managing a Program's Namespace 190
  Module Variables, Parameters, and Temporary Variables 190
  Scope 191

Lifetime 192
Using Keywords for Default and Optional Arguments . . 193
Exercises. 194
Higher-Order Functions 195
Functions as First-Class Data Objects 195
Mapping 196
Filtering 197
Reducing 197
Using `lambda` to Create Anonymous Functions 198
Creating Jump Tables 199
Exercises. 199
Summary 200
Review Questions 202
Projects 203

CHAPTER 7 Simple Graphics and Image Processing. . . **205**

Simple Graphics. 206
Overview of Turtle Graphics 206
Turtle Operations 207
Setting Up a turtle.cfg File and Running IDLE. 209
Object Instantiation and the `turtle` Module 210
Drawing Two-Dimensional Shapes 212
Examining an Object's Attributes 213
Manipulating a Turtle's Screen 214
Taking a Random Walk 214
Colors and the RGB System 215
Example: Filling Radial Patterns with Random Colors 216
Exercises. 218
Case Study: Recursive Patterns in Fractals 218
Image Processing 222
Analog and Digital Information 223
Sampling and Digitizing Images. 223
Image File Formats 224
Image-Manipulation Operations 224
The Properties of Images 225
The `images` Module 225
A Loop Pattern for Traversing a Grid 228
A Word on Tuples 229
Converting an Image to Black and White. 230
Converting an Image to Grayscale 231
Copying an Image 232
Blurring an Image 233

Edge Detection 234
Reducing the Image Size 235
Exercises. 237
Summary 237
Review Questions 238
Projects 240

CHAPTER 8 Graphical User Interfaces 244

The Behavior of Terminal-Based Programs and GUI-Based Programs 245
The Terminal-Based Version 246
The GUI-Based Version. 246
Event-Driven Programming 248
Exercises. 249
Coding Simple GUI-Based Programs. 249
A Simple “Hello World” Program 249
A Template for All GUI Programs 251
The Syntax of Class and Method Definitions 251
Subclassing and Inheritance as Abstraction Mechanisms 252
Exercises. 253
Windows and Window Components 253
Windows and Their Attributes. 253
Window Layout 254
Types of Window Components and Their Attributes . . . 256
Displaying Images. 257
Exercises. 259
Command Buttons and Responding to Events 260
Exercises. 262
Input and Output with Entry Fields. 262
Text Fields 262
Integer and Float Fields for Numeric Data 264
Using Pop-Up Message Boxes 265
Exercises. 267
Defining and Using Instance Variables 267
Exercises. 269
Case Study: The Guessing Game Revisited. 269
Other Useful GUI Resources 273
Using Nested Frames to Organize Components. 273
Multi-Line Text Areas. 275
File Dialogs 277
Obtaining Input with Prompter Boxes 280
Check Buttons 281

Radio Buttons 282
Keyboard Events 284
Working with Colors 285
Using a Color Chooser 287
Summary 289
Review Questions 289
Projects 290

CHAPTER 9 Design with Classes **293**

Getting Inside Objects and Classes 294
A First Example: The `Student` Class 295
Docstrings 297
Method Definitions. 297
The `__init__` Method and Instance Variables 298
The `__str__` Method. 299
Accessors and Mutators 299
The Lifetime of Objects 299
Rules of Thumb for Defining a Simple Class 300
Exercises. 301
Case Study: Playing the Game of Craps 301
Data-Modeling Examples 309
Rational Numbers 309
Rational Number Arithmetic and Operator Overloading . 311
Comparison Methods 312
Equality and the `__eq__` Method 313
Savings Accounts and Class Variables. 314
Putting the Accounts into a Bank 317
Using `pickle` for Permanent Storage of Objects . . . 319
Input of Objects and the `try-except` Statement . . . 320
Playing Cards 321
Exercises. 324
Case Study: An ATM 324
Building a New Data Structure: The Two-Dimensional Grid . 330
The Interface of the `Grid` Class 330
The Implementation of the `Grid` Class: Instance Variables for the Data 332
The Implementation of the `Grid` Class: Subscript and Search 333
Case Study: Data Encryption with a Block Cipher. 333
Structuring Classes with Inheritance and Polymorphism . . 337
Inheritance Hierarchies and Modeling 338
Example 1: A Restricted Savings Account 339
Example 2: The Dealer and a Player in the Game of Blackjack 340

Polymorphic Methods 344
The Costs and Benefits of Object-Oriented Programming 345
Exercises. 346
Summary 347
Review Questions 348
Projects 349

CHAPTER 10 Multithreading, Networks, and Client/Server Programming 352

Threads and Processes 353
Threads 354
Sleeping Threads 357
Producer, Consumer, and Synchronization 358
Exercises. 364
The Readers and Writers Problem. 364
Using the `SharedCell` Class 365
Implementing the Interface of the `SharedCell` Class . . 366
Implementing the Helper Methods of the `SharedCell` Class 368
Testing the `SharedCell` Class with a Counter Object . . 369
Defining a Thread-Safe Class 370
Exercises. 371
Networks, Clients, and Servers 371
IP Addresses 372
Ports, Servers, and Clients. 373
Sockets and a Day/Time Client Script 373
A Day/Time Server Script 375
A Two-Way Chat Script 377
Handling Multiple Clients Concurrently. 378
Exercises. 380
Case Study: Setting Up Conversations between Doctors and Patients 381
Summary 386
Review Questions 387
Projects 388

CHAPTER 11 Searching, Sorting, and Complexity Analysis 390

Measuring the Efficiency of Algorithms 391
Measuring the Run Time of an Algorithm. 391
Counting Instructions 394
Exercises. 396

Complexity Analysis 397
Orders of Complexity 397
Big-O Notation 399
The Role of the Constant of Proportionality 400
Measuring the Memory Used by an Algorithm 400
Exercises. 401
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
Exercises. 405
Basic Sort Algorithms 405
Selection Sort 406
Bubble Sort. 407
Insertion Sort 408
Best-Case, Worst-Case, and Average-Case Performance Revisited 410
Exercises. 410
Faster Sorting 411
Quicksort. 411
Merge Sort 415
Exercises. 418
An Exponential Algorithm: Recursive Fibonacci 419
Converting Fibonacci to a Linear Algorithm 420
Case Study: An Algorithm Profiler. 421
Summary 427
Review Questions 428
Projects 429

APPENDIX A Python Resources. **432**

APPENDIX B Installing the images and breezypythongui Libraries **434**

APPENDIX C The API for Image Processing **436**

APPENDIX D Transition from Python to Java and C++ . . **438**

Glossary. **439**

Index **455**

Preface

"Everyone should learn how to code." That's my favorite quote from Suzanne Keen, 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 edition of *Fundamentals of Python: First Programs*, and why I now offer you this second 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 course (CS1) 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, and network communications. 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 they are then expected to move quickly into topics like software development, complexity analysis, and data structures that, 30 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 Book

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, stepwise refinement, and recursive design 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. A section on functional programming with higher-order functions shows how to exploit functional design patterns to simplify solutions.

Chapter 7 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—areas in which object-based programming is particularly useful.

Chapter 8 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 creation and layout of GUI components are explored, 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 9 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 10 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. Chapter 10 introduces multithreaded programs and the construction of simple network-based client/server applications.

Chapter 11 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.

Special Features

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:

- Case studies—These 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 projects.
- Chapter objectives and chapter summaries—Each chapter begins with a set of learning objectives and ends with a summary of the major concepts covered in the chapter.
- Key terms and a glossary—When a technical term is introduced in the text, it appears in boldface. Definitions of the key terms are also collected in a glossary.
- 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. Each chapter ends with a set of review exercises.
- Programming projects—Each chapter ends with a set of programming projects of varying difficulty.

- A software toolkit for image processing—This book comes with an open-source Python toolkit for the easy image processing discussed in Chapter 7. The toolkit can be obtained from the student downloads page on *www.cengage.com*, or at *http://home.wlu.edu/~lambertk/python/*
- A software toolkit for GUI programming—This book comes with an open-source Python toolkit for the easy GUI programming introduced in Chapter 8. The toolkit can be obtained from the student downloads page on *www.cengage.com*, or at *http://home.wlu.edu/~lambertk/breezypythongui/*
- Appendices—Four appendices include information on obtaining Python resources, installing the toolkits, and using the toolkits' interfaces.

New in This Edition

The most obvious change in this edition is the addition of full color. All program examples include the color coding used in Python's IDLE, so students can easily identify program elements such as keywords, program comments, and function, method, and class names. Several new figures have been added to illustrate concepts, and many exercises and programming projects have been reworked. The brief history of computing in Chapter 1 has been brought up to date. A discussion of a **`Grid`** type has been included to give students exposure to a two-dimensional data structure. The book remains the only introductory Python text with a thorough introduction to realistic GUI programming. The chapter on GUIs (Chapter 8) now uses the **`breezypythongui`** toolkit to ease the introduction of this topic. The chapter on GUIs has also been placed ahead of the chapter on design with classes (Chapter 9). This arrangement allows students to explore the customizing of existing classes with GUI programming before they tackle the design of entirely new classes in the following chapter. Finally, a new section on the readers and writers problem has been added to Chapter 10, to illustrate thread-safe access to shared resources.

Instructor Resources

MindTap

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

All MindTap activities and assignments are tied to defined unit 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. Use the content and learning path as-is or pick-and-choose how our materials will wrap around yours. You control what the students see and when they see it. Learn more at http://www.cengage.com/mindtap/.

Instructor Companion Site

The following teaching tools are available for download at the Companion Site for this text. Simply search for this text at www.cengagebrain.com and choose "Instructor Downloads." An instructor login is required.

- **Instructor's Manual:** The Instructor's 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. A sample syllabus is also available.
- **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.
- **Solutions:** Solutions to all programming exercises are available. If an input file is needed to run a programming exercise, it is included with the solution file.
- **Source Code:** The source code is available at *www.cengagebrain.com.* If an input file is needed to run a program, it is included with the source code.

We Appreciate Your Feedback

We have tried to produce a high-quality text, but should you encounter any errors, please report them to *lambertk@wlu.edu or http://support.cengage.com.* A list of errata, should they be found, as well as other information about the book, will be posted on the Web site *http://home.wlu.edu/~lambertk/python/* and with the student resources at *www.cengagebrain.com*.

Acknowledgments

I would like to thank my contributing author, 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 colleagues at Washington and Lee University who have used this book and given helpful feedback on it over the life of the first edition.

In addition, I would like to thank the following reviewers for the time and effort they contributed to *Fundamentals of Python*: Steven Robinett, Great Falls College Montana State University; Mark Williams, University of Maryland Eastern Shore; Andrew Danner, Swarthmore College; Susan Fox, Macalester College; Emily Shepard, Central Carolina Community College.

Also, thank you to the individuals at Cengage who helped to assure that the content of all data and solution files used for this text were correct and accurate: John Freitas, MQA Project Leader, and Danielle Shaw, MQA Tester.

Finally, thanks to several other people whose work made this book possible: Kate Mason, Associate Product Manager, Cengage; Natalie Pashoukos, Senior Content Developer, Cengage; and Jennifer Feltri-George, Senior Content Project Manager, Cengage. I also want to thank Scarlett Lindsay for her superb copyediting of the book and Chandrasekar Subramanian for an excellent job managing the paging of the project.

Dedication

To my good friends, Lesley and David Novack
Kenneth A. Lambert
Lexington, VA

CHAPTER 1

Introduction

After completing this chapter, you will be able to

- Describe the basic features of an algorithm
- Explain how hardware and software collaborate in a computer's architecture
- Summarize a brief history of computing
- 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 digital movie after preparing a snack in a microwave oven. 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 cell phone. You and your friends have most likely used a desktop computer or a laptop computer to do some significant coursework in high school or college.

These activities rely on something in common: computer technology. Computer technology is almost everywhere, not only in our homes but also in our schools and in the places where we work and play. Computer technology plays an important role in entertainment, education, medicine, manufacturing, communications, government, and commerce. It has been said that we have digital lifestyles and that we live in an information age with 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.

It's difficult to imagine our world without computation, although we don't think about the actual computers very much. It's also hard to imagine that the human race did without computer technology for thousands of years, and that computer technology has pervaded the world as we know it for only the past 30 years or so.

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.

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. We 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 computing 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 probably took you some time to learn how to do it, and 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, when added to $2.39, make $10.00.

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 = 3 dimes, 1 nickel, and 4 pennies), then dollars are selected to bring the price to the next 5-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 we write an algorithm to describe a particular type of computation, we can build a machine to do the computing. Put another way, if we can develop an algorithm to solve a problem, we 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 oven. 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 we have ever 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 human beings 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.

Exercises

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 or operating a DVD player.
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.)

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 handheld devices and ATMs (automatic teller machines).

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.

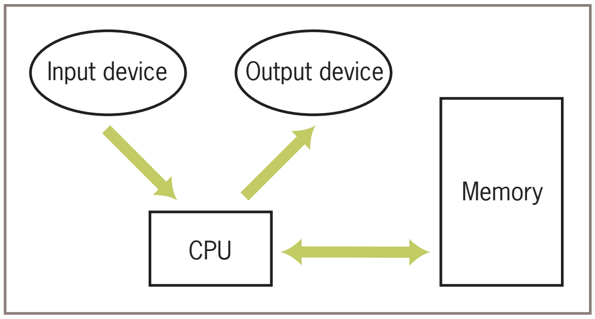


Figure 1-1 Hardware components of a modern computer system

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 we 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** or **internal** or **random access memory (RAM)**.

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

Figure 1-2 A model of computer memory

The information stored in memory can represent any type of data, such as numbers, text, images, or sound, or the instructions of a program. We can also store in memory an algorithm encoded as binary instructions for the computer. 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, perform much the same function with a different technology, as do **optical storage media**, such as CDs and 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, we build some basic operations 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, we 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.

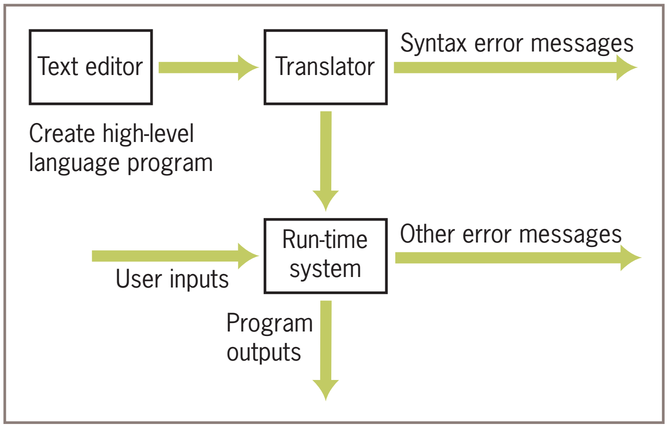


Figure 1-3 Software used in the coding process

Exercises

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?
5. What role do translators play in the programming process?

A Not-So-Brief History of Computing Systems

Now that we 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.