2ND EDITION

Introduction to Computing Using Python

AN APPLICATION DEVELOPMENT FOCUS

Ljubomir Perkovic

WILEY

Introduction to Computing

Using Python

Second Edition

Introduction to Computing Using Python

An Application Development Focus

Second Edition

Ljubomir Perkovic
DePaul University

VICE PRESIDENT AND DIRECTOR	Laurie Rosatone
SENIOR DIRECTOR	Don Fowley
SENIOR ACQUISITIONS EDITOR	Bryan Gambrel
PROJECT SPECIALIST	Marcus Van Harpen
EDITORIAL ASSISTANT	Jessy Moor
EXECUTIVE MARKETING MANAGER	Dan Sayre
SENIOR CONTENT MANAGER	Elle Wagner
SENIOR PRODUCTION EDITOR	John Curley
PROOFREADER	Betty Pessagno
COMPOSITOR	B. Praveen Kumar for SPi Global
COVER PHOTO	©simon2579/iStockphoto

This book was set in TEX Gyre Termes 10 and TEX Gyre Heros 10 by Ljubomir Perković and printed and bound by Quad Graphics/Versailles. The cover was printed by Quad Graphics/Versailles.

This book is printed on acid-free paper. ∞

Library of Congress Cataloging-in-Publication Data

Perkovic, Ljubomir.
Introduction to computing using Python : an application development focus / Ljubomir Perkovic, DePaul University. – Second edition.
pages cm
Includes index.
ISBN 978-1-118-89094-3 (pbk.)
1. Python (Computer program language) 2. Object-oriented programming (Computer science) 3. Computer programming. I. Title.
QA76.73.P98P47 2015
005.1’17–dc23 2015008087

ISBN: 978-1-118-89094-3

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my father, Milan Perković (1937–1970),

who did not get the chance to complete his book.

Contents

Preface xix

Online Textbook Supplements . . . xx
For Students: How to Read This Book . . . xx
Overview of the Book . . . xxi
What Is New in This Edition? . . . xxiv
For Instructors: How to Use This Book . . . xxv

1

Introduction to Computer Science 1

1.1 Computer Science . . . 2
What Do Computing Professionals Do? . . . 2
Models, Algorithms, and Programs. . . . 3
Tools of the Trade . . . 3
What Is Computer Science? . . . 4
1.2 Computer Systems . . . 4
Computer Hardware . . . 4
Operating Systems . . . 5
Networks and Network Protocols . . . 6
Programming Languages . . . 7
Software Libraries . . . 7
1.3 Python Programming Language . . . 8
Short History of Python . . . 8
Setting Up the Python Development Environment . . . 8
1.4 Computational Thinking . . . 9
A Sample Problem . . . 9
Abstraction and Modeling . . . 10
Algorithm . . . 10
Data Types . . . 11
Assignments and Execution Control Structures . . . 12
Chapter Summary . . . 13

2

Python Data Types 15

2.1 Expressions, Variables, and Assignments . . . 16
Algebraic Expressions and Functions . . . 16
Boolean Expressions and Operators . . . 18
Variables and Assignments . . . 20
Variable Names . . . 22
2.2 Strings . . . 23
String Operators . . . 23
Indexing Operator . . . 25
2.3 Lists and Tuples . . . 27
List Operators . . . 27
Lists Are Mutable, Strings Are Not . . . 29
Tuples, or "Immutable Lists" . . . 29
List and Tuple Methods . . . 31
2.4 Objects and Classes . . . 33
Object Type . . . 33
Valid Values for Number Types . . . 35
Operators for Number Types . . . 36
Creating Objects . . . 37
Implicit Type Conversions . . . 38
Explicit Type Conversions . . . 39
Class Methods and Object-Oriented Programming . . . 40
2.5 Python Standard Library . . . 41
Module `math` . . . 41
Module `fractions` . . . 42
Case Study: Turtle Graphics . . . 43
Chapter Summary . . . 43
Solutions to Practice Problems . . . 44
Exercises . . . 45

3

Imperative Programming 51

3.1 Python Programs . . . 52
Our First Python Program . . . 52
Python Modules . . . 54
Built-In Function `print()` . . . 54
Interactive Input with `input()` . . . 55
Function `eval()` . . . 56

3.2 Execution Control Structures . . . 57
One-Way Decisions . . . 57
Two-Way Decisions . . . 60
Iteration Structures . . . 62
Nesting Control Flow Structures . . . 65
Function `range()` . . . 66
3.3 User-Defined Functions . . . 67
Our First Function . . . 67
Function Input Arguments . . . 68
`print()` versus `return` . . . 70
Function Definitions Are “Assignment” Statements . . . 71
Comments . . . 72
Docstrings . . . 72
3.4 Python Variables and Assignments . . . 74
Mutable and Immutable Types . . . 75
Assignments and Mutability . . . 76
Swapping . . . 77
3.5 Parameter Passing . . . 78
Immutable Parameter Passing . . . 79
Mutable Parameter Passing . . . 80
Case Study: Automating Turtle Graphics . . . 81
Chapter Summary . . . 81
Solutions to Practice Problems . . . 82
Exercises . . . 85
Problems . . . 86

4

Text Data, Files, and Exceptions 91
4.1 Strings, Revisited . . . 92
String Representations . . . 92
The Indexing Operator, Revisited . . . 94
String Methods . . . 95
4.2 Formatted Output . . . 98
Function `print()` . . . 98
String Method `format()` . . . 100
Lining Up Data in Columns . . . 102
Getting and Formatting the Date and Time . . . 105
4.3 Files . . . 107
File System . . . 107
Opening and Closing a File . . . 109
Patterns for Reading a Text File . . . 112
Writing to a Text File . . . 115

4.4 Errors and Exceptions 116
Syntax Errors 116
Built-In Exceptions 117
Case Study: Image Files 119
Chapter Summary 119
Solutions to Practice Problems 120
Exercises 121
Problems 124

5

Execution Control Structures 127
5.1 Decision Control and the `if` Statement 128
Three-Way (and More!) Decisions 128
Ordering of Conditions 130
5.2 `for` Loop and Iteration Patterns 131
Loop Pattern: Iteration Loop 131
Loop Pattern: Counter Loop 132
Loop Pattern: Accumulator Loop 134
Accumulating Different Types 135
Loop Patterns: Nested Loop 137
5.3 More on Lists: Two-Dimensional Lists 139
Two-Dimensional Lists 140
Two-Dimensional Lists and the Nested Loop Pattern 141
5.4 `while` Loop 143
`while` Loop Usage 143
5.5 More Loop Patterns 145
Iteration Patterns: Sequence Loop 145
Loop Pattern: Infinite Loop 147
Loop Pattern: Loop and a Half 147
5.6 Additional Iteration Control Statements 149
`break` Statement 149
`continue` Statement 150
`pass` Statement 151
Case Study: Image Processing 151
Chapter Summary 151
Solutions to Practice Problems 152
Exercises 155
Problems 157

6

Containers and Randomness 165

6.1 Dictionaries . . . 166
User-Defined Indexes as Motivation for Dictionaries . . . 166
Dictionary Class Properties . . . 167
Dictionary Operators . . . 169
Dictionary Methods . . . 170
A Dictionary as a Substitute for the Multiway `if` Statement . . . 173
Dictionary as a Collection of Counters . . . 173
`tuple` Objects Can Be Dictionary Keys . . . 176

6.2 Sets . . . 177
Using the `set` Constructor to Remove Duplicates . . . 178
`set` Operators . . . 179
`set` Methods . . . 180

6.3 Character Encodings and Strings . . . 181
Character Encodings . . . 181
ASCII . . . 182
Unicode . . . 183
UTF-8 Encoding for Unicode Characters . . . 185

6.4 Module `random` . . . 186
Choosing a Random Integer . . . 187
Choosing a Random “Real” . . . 188
Shuffling, Choosing, and Sampling at Random . . . 189

Case Study: Games of Chance . . . 190
Chapter Summary . . . 190
Solutions to Practice Problems . . . 190
Exercises . . . 194
Problems . . . 195

7

Namespaces 203

7.1 Encapsulation in Functions . . . 204
Code Reuse . . . 204
Modularity (or Procedural Decomposition) . . . 205
Encapsulation (or Information Hiding) . . . 205
Local Variables . . . 205
Namespaces Associated with Function Calls . . . 206
Namespaces and the Program Stack . . . 207

7.2 Global versus Local Namespaces . . . 211
Global Variables . . . 211
Variables with Local Scope . . . 212
Variables with Global Scope . . . 212
Changing Global Variables Inside a Function . . . 214
7.3 Exceptional Control Flow . . . 215
Exceptions and Exceptional Control Flow . . . 215
Catching and Handling Exceptions . . . 216
The Default Exception Handler . . . 218
Catching Exceptions of a Given Type . . . 218
Multiple Exception Handlers . . . 219
Controlling the Exceptional Control Flow . . . 220
7.4 Modules as Namespaces . . . 223
Module Attributes . . . 223
What Happens When Importing a Module . . . 224
Module Search Path . . . 224
Top-Level Module . . . 226
Different Ways to Import Module Attributes . . . 228
7.5 Classes as Namespaces . . . 230
A Class Is a Namespace . . . 230
Class Methods Are Functions Defined in the Class Namespace . . . 231
Case Study: Debugging with a debugger . . . 231
Chapter Summary . . . 232
Solutions to Practice Problems . . . 232
Exercises . . . 233
Problems . . . 236

8

Object-Oriented Programming 239
8.1 Defining a New Python Class . . . 240
Methods of Class `Point` . . . 240
A Class and Its Namespace . . . 241
Every Object Has an Associated Namespace . . . 242
Implementation of Class `Point` . . . 242
Instance Variables . . . 243
Instances Inherit Class Attributes . . . 244
Class Definition, More Generally . . . 245
Documenting a Class . . . 246
Class `Animal` . . . 247
8.2 Examples of User-Defined Classes . . . 248
Overloaded Constructor Operator . . . 248
Default Constructor . . . 249
Playing Card Class . . . 250

8.3 Designing New Container Classes. 251
Designing a Class Representing a Deck of Playing Cards 251
Implementing the `Deck` (of Cards) Class 252
Container Class `Queue` 254
Implementing a Queue Class 255
8.4 Overloaded Operators 256
Operators Are Class Methods 257
Making the Class `Point` User Friendly 258
Contract between the Constructor and the `repr()` Operator . . . 260
Making the Queue Class User Friendly 262
8.5 Inheritance 264
Inheriting Attributes of a Class 264
Class Definition, in General 267
Overriding Superclass Methods 267
Extending Superclass Methods 270
Implementing a Queue Class by Inheriting from `list` 271
8.6 User-Defined Exceptions 272
Raising an Exception 273
User-Defined Exception Classes 274
Improving the Encapsulation of Class `Queue` 274
Case Study: Indexing and Iterators 275
Chapter Summary 275
Solutions to Practice Problems 276
Exercises 279
Problems 281

9

Graphical User Interfaces 291
9.1 Basics of `tkinter` GUI Development 292
Widget `Tk`: The GUI Window. 292
Widget `Label` for Displaying Text 292
Displaying Images 294
Packing Widgets. 295
Arranging Widgets in a Grid 297
9.2 Event-Based `tkinter` Widgets 299
`Button` Widget and Event Handlers 299
Events, Event Handlers, and `mainloop()` 301
The `Entry` Widget 302
`Text` Widget and Binding Events 305
Event Patterns and the `tkinter` Class `Event` 306
9.3 Designing GUIs 308
Widget `Canvas` 308
Widget `Frame` as an Organizing Widget 311

9.4 OOP for GUIs . . . 313
GUI OOP Basics. . . . 313
Shared Widgets Are Assigned to Instance Variables . . . 315
Shared Data Are Assigned to Instance Variables . . . 317
Case Study: Developing a Calculator . . . 318
Chapter Summary . . . 319
Solutions to Practice Problems . . . 319
Exercises . . . 323
Problems . . . 324

10
Recursion 329
10.1 Introduction to Recursion . . . 330
Functions that Call Themselves . . . 330
Stopping Condition . . . 331
Properties of Recursive Functions . . . 332
Recursive Thinking . . . 332
Recursive Function Calls and the Program Stack . . . 334
10.2 Examples of Recursion. . . . 336
Recursive Number Sequence Pattern . . . 336
Fractals. . . . 338
Virus Scanner . . . 342
Linear recursion . . . 345
10.3 Run Time Analysis . . . 347
The Exponent Function . . . 347
Counting Operations . . . 349
Fibonacci Sequence . . . 349
Experimental Analysis of Run Time . . . 351
10.4 Searching . . . 354
Linear Search. . . . 354
Binary Search . . . 354
Linear versus Binary Search . . . 356
Uniqueness Testing. . . . 357
Selecting the *k*th Largest (Smallest) Item. . . . 358
Computing the Most Frequently Occurring Item . . . 359
Case Study: Tower of Hanoi . . . 359
Chapter Summary . . . 360
Solutions to Practice Problems . . . 360
Exercises . . . 362
Problems . . . 363

11 The Web and Search 371

11.1 The World Wide Web . . . 372
Web Servers and Web Clients . . . 372
"Plumbing" of the WWW . . . 373
Naming Scheme: Uniform Resource Locator . . . 373
Protocol: HyperText Transfer Protocol . . . 374
HyperText Markup Language . . . 375
HTML Elements . . . 376
Tree Structure of an HTML Document . . . 377
Anchor HTML Element and Absolute Links . . . 377
Relative Links . . . 378
11.2 Python WWW API . . . 379
Module `urllib.request` . . . 379
Module `html.parser` . . . 381
Overriding the `HTMLParser` Handlers . . . 383
Module `urllib.parse` . . . 384
Parser That Collects HTTP Hyperlinks . . . 385
11.3 String Pattern Matching . . . 387
Regular Expressions . . . 387
Python Standard Library Module `re` . . . 390
Case Study: Web Crawler . . . 391
Chapter Summary . . . 392
Solutions to Practice Problems . . . 392
Exercises . . . 394
Problems . . . 395

12 Databases and Data Processing 399

12.1 Databases and SQL . . . 400
Database Tables . . . 400
Structured Query Language . . . 402
Statement SELECT . . . 402
Clause WHERE . . . 404
Built-In SQL Functions . . . 406
Clause GROUP BY . . . 406
Making SQL Queries Involving Multiple Tables . . . 407
Statement CREATE TABLE . . . 409
Statements INSERT and UPDATE . . . 409

12.2 Database Programming in Python . . . 410
Database Engines and SQLite . . . 410
Creating a Database with `sqlite3` . . . 411
Committing to Database Changes and Closing the Database . . . 412
Querying a Database Using `sqlite3` . . . 413
12.3 Functional Language Approach . . . 415
List Comprehension . . . 415
MapReduce Problem-Solving Framework . . . 417
MapReduce, in the Abstract . . . 420
Inverted Index . . . 421
12.4 Parallel Computing . . . 423
Parallel Computing . . . 423
Class `Pool` of Module `multiprocessing` . . . 424
Parallel Speedup . . . 427
MapReduce, in Parallel . . . 428
Parallel versus Sequential MapReduce . . . 429
Case Study: Data Interchange . . . 431
Chapter Summary . . . 432
Solutions to Practice Problems . . . 432
Exercises . . . 435
Problems . . . 436

Case Studies 441
CS.2 Turtle Graphics . . . 442
Classes `Screen` and `Turtle` . . . 442
Solution to the Practice Problem . . . 446
Problems . . . 446
CS.3 Automating Turtle Graphics . . . 448
Function `jump()` . . . 448
Solution to the Practice Problem . . . 450
Problems . . . 450
CS.4 Processing Image Files . . . 452
Class `Image` in Module `PIL.Image` . . . 452
Image Size, Format, and Mode . . . 453
`Image` Class Methods . . . 454
Creating and Saving a New Image . . . 455
Solution to the Practice Problem . . . 456
Problems . . . 457
CS.5 Image-Processing Algorithms . . . 458
Accessing Pixels . . . 458
Copying an Image . . . 459
Rotating an Image by 90 Degrees . . . 459
Applying an Image Filter . . . 461
Solutions to Practice Problems . . . 463
Problems . . . 464

CS.6 Games of Chance 465
Blackjack 465
Creating and Shuffling the Deck of Cards 466
Dealing a Card 467
Computing the Value of a Hand 467
Comparing the Player's and the House's Hands 468
Main Blackjack Function 468
Problems 469
CS.7 Debugging with a Debugger 471
Debugging Commands 471
Analyzing the Program Stack 474
Solution to the Practice Problem 476
Problems 477
CS.8 Indexing and Iterators 479
Overloading the Indexing Operators 479
Iterators and OOP Design Patterns 481
Solutions to Practice Problems 484
Problems 484
CS.9 Developing a Calculator 486
The Calculator Buttons and Passing Arguments to Handlers . . . 486
Implementing the "Unofficial" Event Handler `click()` 488
Solution to the Practice Problem 490
Problems 490
CS.10 Tower of Hanoi 492
The Recursive Solution 493
Classes `Peg` and `Disk`. 495
Problems 497
CS.11 Web Crawlers 498
Recursive Crawler, Version 0.1 498
Recursive Crawler, Version 0.2 500
The Web Page Content Analysis 502
Solution to the Practice Problem 504
Problems 505
CS.12 Data Interchange 506
Serialization and Data Interchange Formats. 506
JSON (JavaScript Object Notation) 506
Data Compression 508
I/O Streams 510
Solution to the Practice Problems 512
Problems 513

Index 514

Preface

This textbook is an introduction to programming, computer application development, and the science of computing. It is meant to be used in a college-level introductory programming course. More than just an introduction to programming, the book is a broad introduction to computer science concepts and to the tools used for modern computer application development.

The computer programming language used in the book is Python, a language that has a gentler learning curve than most. Python comes with powerful software libraries that make complex tasks—such as developing a graphics application or finding all the links in a web page—a breeze. In this textbook, we leverage the ease of learning Python and the ease of using its libraries to do more computer science *and* to add a focus on modern application development. The result is a textbook that is a broad introduction to the field of computing and modern application development.

The textbook's pedagogical approach is to introduce computing concepts and Python programming in a breadth-first manner. Rather than covering computing concepts and Python structures one after another, the book's approach is more akin to learning a natural language, starting from a small general-purpose vocabulary and then gradually extending it. The presentation is in general problem oriented, and computing concepts, Python structures, algorithmic techniques, and other tools are introduced when needed, using a "right tool at the right moment" model.

The book uses the imperative-first and procedural-first paradigm but does not shy away from discussing objects early. User-defined classes and object-oriented programming are covered later, when they can be motivated and students are ready. The last three chapters of the textbook and the associated case studies use the context of web crawling, search engines, and data mining to introduce a broad array of topics. These include foundational concepts such as recursion, regular expressions, depth-first search, data compression, and Google's MapReduce framework, as well as practical tools such as GUI widgets, HTML parsers, SQL, JSON, I/O streams, and multicore programming.

This textbook can be used in a course that introduces computer science and programming to computer science majors. Its broad coverage of foundational computer science topics as well as current technologies will give the student a broad understanding of the field *and* a confidence to develop "real" modern applications that interact with the web and/or a database. The textbook's broad coverage also makes it ideal for students who need to master programming and key computing concepts but will not take more than one or two computing courses.

The Book's Technical Features

The textbook has a number of features that engage students and encourage them to get their hands dirty. For one, the book makes heavy use of *examples that use the Python interactive shell*. Students can easily reproduce these one-liners on their own. After doing so, students will likely continue experimenting further using the immediate feedback of the interactive shell.

Throughout the textbook, there are inline *practice problems* whose purpose is to reinforce concepts just covered. The solutions to these problems appear at the end of the corresponding chapter or case study, allowing students to check their solution or take a peek in case they are stuck.

The textbook uses Caution boxes to warn students of potential pitfalls. It also uses Detour boxes to briefly explore interesting but tangential topics. The large number of boxes, practice problems, figures, and tables create visual breaks in the text, making reading the volume more approachable for students.

Finally, the textbook contains a *large number of end-of-chapter problems*, many of which are unlike problems typically found in an introductory textbook.

The *E-Book Edition* of the textbook includes additional material consisting of 11 case studies. Each case study is associated with a chapter (2 through 12) and showcases the concepts and tools covered in the chapter in context. The case studies include additional problems, including practice problems with solutions.

Online Textbook Supplements

These textbook supplements are available on the textbook web site:

- PowerPoint slides for each chapter
- Learning objectives for each section
- Code examples appearing in the book
- Exercise and problem solutions (for instructors only)
- Exam problems (for instructors only)

For Students: How to Read This Book

This book is meant to help you master programming and develop computational thinking skills. Programming and computational thinking are hands-on activities that require a computer with a Python integrated development environment as well as a pen and paper for "back-of-the-envelope" calculations. Ideally, you should have those tools next to you as you read this book.

The book makes heavy use of small examples that use Python's interactive shell. Try running those examples in your shell. Feel free to experiment further. It's very unlikely the computer will burst into flames if you make a mistake!

You should also attempt to solve all the practice problems as they appear in the text. Problem solutions appear at the end of the corresponding chapter. If you get stuck, it's OK to peek at the solution; after doing so, try solving the problem without peeking.

The text uses Caution boxes to warn you of potential pitfalls. These are very important and should not be skipped. The Detour boxes, however, discuss topics that are only tangentially related to the main discussion. You may skip those if you like. Or you may go further and explore the topics in more depth if you get intrigued.

At some point while reading this text, you may get inspired to develop your own app, whether a card game or an app that keeps track of a set of stock market indexes in real time. If so, just go ahead and try it! You will learn a lot.

Overview of the Book

This textbook consists of 12 chapters that introduce computing concepts and Python programming in a breadth-first manner. The E-Book Edition also includes case studies that showcase concepts and tools covered in the chapters in context.

Tour of Python and Computer Science

Chapter 1 introduces the *basic computing concepts and terminology*. Starting with a discussion of what computer science is and what developers do, the concepts of modeling, algorithm development, and programming are defined. The chapter describes the computer scientist's and application developer's toolkit, from logic to systems, with an emphasis on programming languages, the Python development environment, and computational thinking.

Chapter 2 covers *core built-in Python data types*: the integer, Boolean, floating-point, string, list, and tuple types. To illustrate the features of the different types, the Python interactive shell is used. Rather than being comprehensive, the presentation focuses on the purpose of each type and the differences and similarities between the types. This approach motivates a more abstract discussion of objects and classes that is ultimately needed for mastering the proper usage of data types. Case Study CS.2 takes advantage of this discussion to introduce Turtle graphics classes that enable students to do simple, fun graphics interactively.

Chapter 3 introduces *imperative and procedural programming, including basic execution control structures*. This chapter presents programs as a sequence of Python statements stored in a file. To control how the statements are executed, basic conditional and iterative control structures are introduced: the one-way and two-way `if` statements as well as the simplest `for` loop patterns of iterating through an explicit sequence or a range of numbers. The chapter introduces functions as a way to neatly package a small application; it also builds on the material on objects and classes covered in Chapter 2 to describe how Python does assignments and parameter passing. Case Study CS.3 uses the visual context of Turtle graphics to motivate automation through programs and abstraction through functions.

The first three chapters provide a *shallow* but *broad* introduction to Python programming and computers science. Core Python data types and basic execution control structures are introduced so students can write simple but complete programs early. Functions are introduced early as well to help students conceptualize what a program is doing, that is, what inputs it takes and what output it produces. In other words, abstraction and encapsulation of functions is used to help students better understand programs.

Focus on Algorithmic Thinking

Chapter 4 covers *text and file processing in more depth*. It continues the coverage of strings from Chapter 2 with a discussion of string value representations, string operators and methods, and formatted output. File input/output (I/O) is introduced as well and, in particular, the different patterns for reading text files. Finally, the context of file I/O is used to motivate a discussion of exceptions and the different types of exceptions in Python. Case Study CS.4 discusses how image files (typically stored as binary files rather than text files) are read and written and how images can be processed using Python.

Chapter 5 covers *execution control structures and loop patterns in depth*. Basic conditional and iteration structures were introduced in Chapter 3 and then used in Chapter 4 (e.g., in the context of reading files). Chapter 5 starts with a discussion of multiway conditional statements. The bulk of the chapter is spent on describing the different loop patterns: the various ways `for` loops and `while` loops are used. Multidimensional lists are introduced as well, in the context of the nested loop pattern. More than just covering Python loop structures, this core chapter describes the different ways that problems can be broken down. Thus, the chapter fundamentally is about *problem solving and algorithms*. Case Study CS.5 looks underneath the hood of image processing and describes how classic image processing algorithms can be implemented.

Chapter 6 completes the textbook's coverage of *Python's built-in container data types and their usage*. The dictionary, set, and tuple data types are motivated and introduced. This chapter also completes the coverage of strings with a discussion of character encodings and Unicode. Finally, the concept of randomness is introduced in the context of selecting and permuting items in containers. Case Study CS.6 makes use of concepts introduced in this chapter to show how a blackjack application can be developed.

Chapters 4 through 6 represent the second layer in the breadth-first approach this textbook takes. One of the main challenges students face in an introductory programming course is mastering conditional and iteration structures and, more generally, the computational problem-solving and algorithm development skills. The critical Chapter 5, on patterns of applying execution control structures, appears *after* students have been using *basic* conditional statements and iteration patterns for several weeks and have gotten somewhat comfortable with the Python language. Having gained some comfort with the language and basic iteration, students can focus on the algorithmic issues rather than less fundamental issues, such as properly reading input or formatting output.

Managing Program Complexity

Chapter 7 shifts gears and focuses on the software development process itself and the problem of managing larger, more complex programs. It introduces *namespaces as the foundation for managing program complexity*. The chapter builds on the coverage of functions and parameter passing in Chapter 3 to motivate the software engineering goals of code reuse, modularity, and encapsulation. Functions, modules, and classes are tools that can be used to achieve these goals, fundamentally because they define separate namespaces. The chapter describes how namespaces are managed during normal control flow and during exceptional control flow, when exceptions are handled by exception handlers. Case Study CS.7 builds on this chapter's content to show how to use a debugger to find bugs in a program or, more generally, to analyze the execution of the program.

Chapter 8 covers the *development of new classes in Python and the object-oriented programming (OOP) paradigm*. The chapter builds on Chapter 7's uncovering of how Python classes are implemented through namespaces to explain how new classes are developed. The chapter introduces the OOP concepts of operator overloading—central to Python's design philosophy—and inheritance—a powerful OOP property that will be used in Chapters 9 and 11. Through abstraction and encapsulation, classes achieve the desirable software engineering goals of modularity and code reuse. The context of abstraction and encapsulation is then used to motivate user-defined exception classes. Case Study CS.8 goes one step further and illustrates the implementation of iterative behavior in user-defined container classes.

Chapter 9 introduces *graphical user interfaces (GUIs) and showcases the power of the OOP approach for developing GUIs*. It uses the Tk widget toolkit, which is part of the Python Standard Library. The coverage of interactive widgets provides the opportunity to discuss the event-driven programming paradigm. In addition to introducing GUI development, the chapter also showcases the power of OOP to achieve modular and reusable programs. Case Study CS.9 illustrates this in the context of implementing a basic calculator GUI.

The broad goal of Chapters 7 though 9 is to introduce students to the issues of program complexity and code organization. They describe how namespaces are used to achieve functional abstraction and data abstraction and, ultimately, encapsulated, modular, and reusable code. Chapter 8 provides a comprehensive discussion of user-defined classes and OOP. The full benefit of OOP, however, is best seen in context, which is what Chapter 9 is about. Additional contexts and examples of OOP are shown in later chapters and specifically in Sections 11.2, 12.3, and 12.4 as well as in Case Study CS.10. Chapters 7 though 9 provide a foundation for the students' future education in data structures and software engineering methodologies.

Crawling through Foundations and Applications

Chapters 10 through 12, the last three chapters of the textbook, cover a variety of advanced topics, from fundamental computer science concepts like recursion, regular expressions, data compression, and depth-first search, to practical and contemporary tools like HTML parsers, JSON, SQL, and multicore programming. The theme used to motivate and connect these topics is the development of web crawlers, search engines, and data mining apps. The theme, however, is loose, and each individual topic is presented independently to allow instructors to develop alternate contexts and themes for this material as they see fit.

Chapter 10 introduces foundational computer science topics: *recursion, search, and the run-time analysis of algorithms*. The chapter starts with a discussion of how to think recursively. This skill is then put to use on a wide variety of problems from drawing fractals to virus scanning. This last example is used to illustrate depth-first search. The benefits and pitfalls of recursion lead to a discussion of algorithm run-time analysis, which is then used in the context of analyzing the performance of various list search algorithms. This chapter puts the spotlight on the theoretical aspects of computing and forms a basis for future coursework in data structures and algorithms. Case Study CS.10 considers the Tower of Hanoi problem and shows how to develop a visual application that illustrates the recursive solution.

Chapter 11 introduces the *World Wide Web as a central computing platform and as a huge source of data* for innovative computer application development. HTML, the language of the web, is briefly discussed before tools to access resources on the web and parse web pages are covered. To grab the desired content from web pages and other text content, regular expressions are introduced. A benefit of touching HTML parsing and regular expressions in an introductory course is that students will be familiar with their uses in context before rigorously covering them in a formal languages course. Case Study CS.11 makes use of the different topics covered in this chapter to show how a basic *web crawler* can be developed.

Chapter 12 covers *databases and the processing of large data sets*. The database language SQL is briefly described as well as a Python's database application programming interface in the context of storing data grabbed from a web page. Given the ubiquity of databases in today's computer applications, it is important for students to get an early exposure to them and their use (if for no other reason than to be familiar with them before their first internship). The coverage of databases and SQL is introductory only and should be considered merely a basis for a later database course. This chapter also considers how to leverage the multiple cores available on computers to process big data sets more quickly. Google's MapReduce problem-solving framework is described and used as a context for introducing list comprehensions and the functional programming paradigm. This chapter forms a foundation for further study of databases, programming languages, and data mining. Case Study CS.12 uses this last context to discuss *data interchange* or how to format and save data so that it is accessible, easily and efficiently, to any program that needs it.

What Is New in This Edition?

The big change between the first and second editions of the textbook is a structural one. A clear separation now exists between the foundational material covered in a chapter and the case study illustrating the concepts covered in the chapter. The case studies have been moved out of the chapters and are now grouped together in the E-Book Edition of the textbook. There are two benefits from this structural change. First, the coverage of the textbook chapters is now more focused on foundational material. The streamlined content, together with a switch to a Black&White format, allows the new Print Edition of the textbook to be priced less than the previous one. The second benefit of moving the case studies to the E-Book Edition is that the move gives more space for the case studies to be enriched. Four new case studies appear in the new edition, and there is now a case study associated with every chapter of the textbook (except the "non-technical" introductory chapter).

In addition to this structural change, new material has been added, some material has been moved, errata have been corrected, and the presentation has been improved. We outline these changes next.

In Chapter 2, we have added coverage of the tuple type (covered in Chapter 6 in the first edition). This move is justified because the tuple type is a key built-in type in Python that is used by many Standard Library modules and Python applications. For example, tuple objects are used by the image processing modules discussed in the case studies associated with Chapters 4 and 5. Because the tuple type is very similar to the list type, this additional content adds very little to the time needed to cover Chapter 2.

In Chapter 3, the presentation of functions has been improved. In particular, there are more examples and practice problems to help illustrate the passing of different numbers and types of function parameters. The Chapter 4 case study has been replaced with a new one on processing image files. The new case study gives students an exciting opportunity to see the textbook material in the context of visual media. Also, the material on processing and formating date and time strings has been moved to Section 4.2. The important Chapter 5 has, in the second edition, an associated case study on implementing image processing algorithms. This material again uses the attractive context of visual media to illustrate fundamental concepts such as nested loops.

Chapter 6 no longer includes coverage of the tuple type (moved to Chapter 2). Chapter 7 has, in the second edition, an associated case study on debugging and the use of a debugger. It effectively uses the concepts covered in the chapter to provide students with a new tool that will help them with debugging. Chapters 8 and 9 have changed only slightly. Chapter 10 has a deeper and more explicit coverage of linear recursion and its relationship to iteration. Chapter 11 has few changes. Finally, Chapter 12 has, in the second edition, an associated case study on data interchange which will help students gain practical experience working with data sets.

Finally, about 60 practice and end-of-chapter problems have been added to the book.

For Instructors: How to Use This Book

The material in this textbook was developed for a two quarter course sequence introducing computer science and programming to computer science majors. The book therefore has more than enough material for a typical 15-week course (and probably just the right amount of material for a class of well-prepared and highly motivated students).

The first six chapters of the textbook provide a comprehensive coverage of imperative/procedural programming in Python. They are meant to be covered in order, but it is possible to cover Chapter 5 before Chapter 4. Furthermore, the topics in Chapter 6 may be skipped and then introduced as needed.

Chapters 7 through 9 are meant to be covered in order to effectively showcase OOP. It is important to cover Chapter 7 before Chapter 8 because it demystifies Python's approach to class implementation and allows the more efficient coverage of OOP topics such as operator overloading and inheritance. It is also beneficial, though not necessary, to cover Chapter 9 after Chapter 8 because it provides a context in which OOP is shown to provide great benefits.

Chapters 9 through 12 are all optional, depend only on Chapters 1 through 6—with the few exceptions noted—and contain topics that can, in general, be skipped or reordered at the discretion of the course instructor. Exceptions are Section 9.4, which illustrates the OOP approach to GUI development, as well as Sections 11.2, 12.3, and 12.4, all of which make use of user-defined classes. All these should follow Chapter 8.

Instructors using this book in a course that leaves OOP to a later course can cover Chapters 1 through 7 and then choose topics from the non-OOP sections of Chapters 9 through 12. Instructors wishing to cover OOP should use Chapters 1 through 9 and then choose topics from Chapters 10 through 12.

Acknowledgments

The material for the first edition of this textbook was developed over three years in the context of teaching the CSC 241/242 course sequence (Introduction to Computer Science I and II) at DePaul University. In those three years, six separate cohorts of computer science freshmen moved through the course sequence. I used the different cohorts to try different pedagogical approaches, reorder and reorganize the material, and experiment with topics usually not taught in a course introducing programming. The continuous reorganization and experimentation made the course material less fluid and more challenging than necessary, especially for the early cohorts. Amazingly, students maintained their enthusiasm through the low points in the course, which in turn helped me maintain mine. I thank them all wholeheartedly for that.

I would like to acknowledge the faculty and administration of DePaul's School of Computing for creating a truly unique academic environment that encourages experimentation and innovation in education. Some of them also had a direct role in the creation and shaping of this textbook. Associate Dean Lucia Dettori scheduled my classes so I had time to write. Curt White, an experienced textbook author, encouraged me to start writing and put in a good word for me with publishing house John Wiley & Sons. Massimo DiPierro, the creator of the web2py web framework and a far greater Python authority than I will ever be, created the first outline of the content of the CSC241/242 course sequence, which was the initial seed for the book. Iyad Kanj taught the first iteration of CSC241 and selflessly allowed me to mine the material he developed. Amber Settle is the first person other than me to use this textbook in her course; thankfully, she had great success, though that is at least as much

due to her excellence as a teacher. Craig Miller has thought more deeply about fundamental computer science concepts and how to explain them than anyone I know; I have gained some of his insights through many interesting discussions, and the textbook has benefited from them. Finally, Marcus Schaefer improved the textbook by doing a thorough technical review of more than half of the book.

My course lecture notes would have remained just that if Nicole Dingley, a Wiley book rep, had not suggested that I make them into a textbook. Nicole put me in contact with Wiley editor Beth Golub, who made the gutsy decision to trust a foreigner with a strange name and no experience writing textbooks to write a textbook. Wiley senior designer Madelyn Lesure, along with my friend and neighbor Mike Riordan, helped me achieve the simple and clean design of the text. Finally, Wiley senior editorial assistant Samantha Mandel worked tirelessly on getting my draft chapters reviewed and into production. Samantha has been a model of professionalism and good grace throughout the process, and she has offered endless good ideas for making the book better.

The final version of the book is similar to the original draft in surface only. The vast improvement over the initial draft is due to the dozens of reviewers, many of them anonymous. The kindness of strangers has made this a better book and has given me a new appreciation for the reviewing process. The reviewers have been kind enough not only to find problems but also offer solutions. For their careful and systematic feedback, I am grateful. Some of the reviewers, including David Mutchler (Rose-Hulman Institute of Technology), who offered his name and email for further correspondence, went beyond the call of duty and helped excavate the potential that lay buried in my early drafts. Jonathan Lundell also provided a technical review of the last chapters in the book. Because of time constraints, I was not able to incorporate all the valuable suggestions I received from them, and the responsibility for any any omissions in the textbook are entirely my own.

I would like to thank, in particular, the following faculty who made use of the first edition in their courses and gave me invaluable feedback: Ankur Agrawal (Manhattan College), Albert Chan (Fayetteville State University), Gabriel Ferrer (Hendrix College), David G. Kay (University of California, Irvine), Gerard Ryan (New Jersey Institute of Technology), Sridhar Seshadri (University of Texas at Arlington), Richard Weiss (Evergreen State College), and Michal Young (University of Oregon). I have tried my best to incorporate their suggestions in this second edition.

Finally, I would like to thank my spouse, Lisa, and daughters, Marlena and Eleanor, for the patience they had with me. Writing a book takes a huge amount of time, and this time can only come from "family time" or sleep since other professional obligations have set hours. The time I spent writing this book resulted in my being unavailable for family time or my being crabby from lack of sleep, a real double whammy. Luckily, I had the foresight to adopt a dog when I started working on this project. A dog named Muffin inevitably brings more joy than any missing from me... So, thanks to Muffin.

About the Author

Ljubomir Perkovic is an associate professor at the School of Computing of DePaul University in Chicago. He received a Bachelor's degree in mathematics and computer science from Hunter College of the City University of New York in 1990. He obtained his Ph.D. in algorithms, combinatorics, and optimization from the School of Computer Science at Carnegie Mellon University in 1998.

Professor Perkovic started teaching the introductory programming sequence for majors at DePaul in the mid-2000s. His goal was to share with beginning programmers the excitement that developers feel when working on a cool new app. He incorporated into the course concepts and technologies used in modern application development. The material he developed for the course forms the basis of this book.

His research interests include computational geometry, distributed computing, graph theory and algorithms, and computational thinking. He has received a Fulbright Research Scholar award for his research in computational geometry and a National Science Foundation grant for a project to expand computational thinking across the general education curriculum.

CHAPTER

1

Introduction to Computer Science

1.1 Computer Science 2
1.2 Computer Systems 4
1.3 Python Programming Language 8
1.4 Computational Thinking 9
Chapter Summary 13

IN THIS INTRODUCTORY CHAPTER, we provide the context for the book and introduce the key concepts and terminology that we will be using throughout. The starting point for our discussion are several questions. What is computer science? What do computer scientists and computer application developers do? And what tools do they use?

Computers, or more generally computer systems, form one set of tools. We discuss the different components of a computer system including the hardware, the operating system, the network and the Internet, and the programming language used to write programs. We specifically provide some background on the Python programming language, the language used in this book.

The other set of tools are the reasoning skills, grounded in logic and mathematics, required to develop a computer application. We introduce the idea of computational thinking and illustrate how it is used in the process of developing a small web search application.

The foundational concepts and terminology introduced in this chapter are independent of the Python programming language. They are relevant to any type of application development regardless of the hardware or software platform or programming language used.

1.1 Computer Science

This textbook is an introduction to programming. It is also an introduction to Python, the programming language. But most of all, it is an introduction to computing and how to look at the world from a computer science perspective. To understand this perspective and define what computer science is, let's start by looking at what computing professionals do.

What Do Computing Professionals Do?

One answer is to say: they write programs. It is true that many computing professionals do write programs. But saying that they write programs is like saying that screenwriters (i.e., writers of screenplays for movies or television series) write text. From our experience watching movies, we know better: screenwriters invent a world and plots in it to create stories that answer the movie watcher's need to understand the nature of the human condition. Well, maybe not all screenwriters.

So let's try again to define what computing professionals do. Many actually do *not* write programs. Even among those who do, what they are really doing is developing computer applications that address a need in some activity we humans do. Such computing professionals are often called *computer application developers* or simply *developers*. Some developers even work on applications, like computer games, that are not that different from the imaginary worlds, intricate plots, and stories that screenwriters create.

Not all developers develop computer games. Some create financial tools for investment bankers, and others create visualization tools for doctors (see Table 1.1 for other examples.)

What about the computing professionals who are *not* developers? What do they do? Some talk to clients and elicit requirements for computer applications that clients need.

Table 1.1 The range of computers science. Listed are examples of human activities and, for each activity, a software product built by computer application developers that supports performing the activity.

Activity	Computer Application
Defense	Image processing software for target detection and tracking
Driving	GPS-based navigation software with traffic views on smartphones and dedicated navigation hardware
Education	Simulation software for performing dangerous or expensive biology laboratory experiments virtually
Farming	Satellite-based farm management software that keeps track of soil properties and computes crop forecasts
Films	3D computer graphics software for creating computer-generated imagery for movies
Media	On-demand, real-time video streaming of television shows, movies, and video clips
Medicine	Patient record management software to facilitate sharing between specialists
Physics	Computational grid systems for crunching data obtained from particle accelerators
Political activism	Social network technologies that enable real-time communication and information sharing
Shopping	Recommender system that suggests products that may be of interest to a shopper
Space exploration	Mars exploration rovers that analyze the soil to find evidence of water

Others are managers who oversee an application development team. Some computing professionals support their clients with newly installed software and others keep the software up to date. Many computing professionals administer networks, web servers, or database servers. Artistic computing professionals design the interfaces that clients use to interact with an application. Some, such as the author of this textbook, like to teach computing, and others offer information technology (IT) consulting services. Finally, more than a few computing professionals have become entrepreneurs and started new software businesses, many of which have become household names.

Regardless of the ultimate role they play in the world of computing, all computing professionals understand the basic principles of computing, how computer applications are developed, and how they work. Therefore, the training of a computing professional always starts with the mastery of a programming language and the software development process. In order to describe this process in general terms, we need to use slightly more abstract terminology.

Models, Algorithms, and Programs

To create a computer application that addresses a need in some area of human activity, developers invent a *model* that represents the "real-world" environment in which the activity occurs. The model is an abstract (imaginary) representation of the environment and is described using the language of logic and mathematics. The model can represent the objects in a computer game, stock market indexes, an organ in the human body, or the seats on an airplane.

Developers also invent *algorithms* that operate in the model and that create, transform, and/or present information. An algorithm is a sequence of instructions, not unlike a cooking recipe. Each instruction manipulates information in a very specific and well-defined way, and the execution of the algorithm instructions achieves a desired goal. For example, an algorithm could compute collisions between objects in a computer game or the available economy seats on an airplane.

The full benefit of developing an algorithm is achieved with the *automation* of the execution of the algorithm. After inventing a model and an algorithm, developers implement the algorithm as a *computer program* that can be executed on a *computer system*. While an algorithm and a program are both descriptions of step-by-step instructions of how to achieve a result, an algorithm is described using a language that we understand but that cannot be executed by a computer system, and a program is described using a language that we understand *and* that can be executed on a computer system.

At the end of this chapter, in Section 1.4, we will take up a sample task and go through the steps of developing a model and an algorithm implementing the task.

Tools of the Trade

We already hinted at a few of the tools that developers use when working on computer applications. At a fundamental level, developers use logic and mathematics to develop models and algorithms. Over the past half century or so, computer scientists have developed a vast body of knowledge—grounded in logic and mathematics—on the theoretical foundations of information and computation. Developers apply this knowledge in their work. Much of the training in computer science consists of mastering this knowledge, and this textbook is the first step in that training.

The other set of tools developers use are computers, of course, or more generally computer systems. They include the hardware, the network, the operating systems, and also the programming languages and programming language tools. We describe all these systems in more detail in Section 1.2. While the theoretical foundations often transcend changes in technology, computer system tools are constantly evolving. Faster hardware, improved operating systems, and new programming languages are being created almost daily to handle the applications of tomorrow.

What Is Computer Science?

We have described what application developers do and also the tools that they use. What then is computer science? How does it relate to computer application development?

While most computing professionals develop applications for users outside the field of computing, some are studying and creating the theoretical and systems tools that developers use. The field of computer science encompasses this type of work. Computer science can be defined as the study of the theoretical foundations of information and computation and their practical implementation on computer systems.

While application development is certainly a core driver of the field of computer science, its scope is broader. The computational techniques developed by computer scientists are used to study questions on the nature of information, computation, and intelligence. They are also used in other disciplines to understand the natural and artificial phenomena around us, such as phase transitions in physics or social networks in sociology. In fact, some computer scientists are now working on some of the most challenging problems in science, mathematics, economics, and other fields.

We should emphasize that the boundary between application development and computer science (and, similarly, between application developers and computer scientists) is usually not clearly delineated. Much of the theoretical foundations of computer science have come out of application development, and theoretical computer science investigations have often led to innovative applications of computing. Thus many computing professionals wear two hats: the developer's and the computer scientist's.

1.2 Computer Systems

A computer system is a combination of hardware and software that work together to execute application programs. The hardware consists of physical components—that is, components that you can touch, such as a memory chip, a keyboard, a networking cable, or a smartphone. The software includes all the nonphysical components of the computer, including the operating system, the network protocols, the programming language tools, and the associated application programming interface (API).

Computer Hardware

The computer *hardware* refers to the physical components of a computer system. It may refer to a desktop computer and include the monitor, the keyboard, the mouse, and other external devices of a computer desktop and, most important, the physical "box" itself with all its internal components.

The core hardware component inside the box is the *central processing unit* (CPU) . The CPU is where the computation occurs. The CPU performs computation by fetching program instructions and data and executing the instructions on the data. Another key internal component is *main memory*, often referred to as *random access memory* (RAM). That is where program instructions and data are stored when the program executes. The CPU fetches instructions and data from main memory and stores the results in main memory.

The set of wirings that carry instructions and data between the CPU and main memory is commonly called a *bus*. The bus also connects the CPU and main memory to other internal components such as the hard drive and the various *adapters* to which external devices (such as the monitor, the mouse, or the network cables) are connected.

The *hard drive* is the third core component inside the box. The hard drive is where files are stored. Main memory loses all data when the computer is shut down; the hard drive, however, is able to store a file whether the computer is powered on or off. The hard drive also has a much, much higher capacity than main memory.

The term *computer system* may refer to a single computer (desktop, laptop, smartphone, or pad). It may also refer to a collection of computers connected to a network (and thus to each other). In this case, the hardware also includes any network wiring and specialized network hardware such as *routers*.

It is important to understand that most developers do not work with computer hardware directly. It would be extremely difficult to write programs if the programmer had to write instructions directly to the hardware components. It would also be very dangerous because a programming mistake could incapacitate the hardware. For this reason, there exists an *interface* between application programs written by a developer and the hardware.

Operating Systems

An application program does not directly access the keyboard, the computer hard drive, the network (and the Internet), or the display. Instead it requests the *operating system* (OS) to do so on its behalf. The operating system is the software component of a computer system that lies between the hardware and the application programs written by the developer. The operating system has two complementary functions:

1. The OS protects the hardware from misuse by the program or the programmer and
2. The OS provides application programs with an interface through which programs can request services from hardware devices.

In essence, the OS manages access to the hardware by the application programs executing on the machine.

DETOUR

Origins of Today's Operating Systems

The mainstream operating systems on the market today are Microsoft Windows and UNIX and its variants, including Linux and Apple OS X.

The UNIX operating system was developed in the late 1960s and early 1970s by Ken Thompson at AT&T Bell Labs. By 1973, UNIX was reimplemented by Thompson and Dennis Ritchie using C, a programming language just created by Ritchie. As it was free for anyone to use, C became quite popular, and programmers *ported* C and UNIX to various computing platforms. Today, there are several versions of UNIX, including Apple's Mac OS X.

The origin of Microsoft's Windows operating systems is tied to the advent of personal computers. Microsoft was founded in the late 1970s by Paul Allen and Bill Gates. When IBM developed the IBM Personal Computer (IBM PC) in 1981, Microsoft provided the operating system called MS DOS (Microsoft Disk Operating System). Since then Microsoft has added a graphical interface to the operating system and renamed it Windows. The latest version is Windows 7.

Linux is a UNIX-like operating sytem developed in the early 1990s by Linus Torvalds. His motivation was to build a UNIX-like operating system for personal computers since, at the time, UNIX was restricted to high-powered workstations and mainframe computers. After the initial development, Linux became a community-based, *open source* software development project. That means that any developer is welcome to join in and help in the further development of the Linux OS. Linux is one of the best examples of successful open-source software development projects.

Networks and Network Protocols

Many of the computer applications we use daily require the computer to be connected to the Internet. Without an Internet connection, you cannot send an email, browse the web, listen to Internet radio, or update your software. In order to be connected to the Internet, though, you must first connect to a network that is part of the Internet.

A computer network is a system of computers that can communicate with each other. There are several different network communication technologies in use today, some of which are wireless (e.g., Wi-Fi) and others that use network cables (e.g., Ethernet).

An *internetwork* is the connection of several networks. The *Internet* is an example of an internetwork. The Internet carries a vast amount of data and is the platform upon which the World Wide Web (WWW) and email are built.

DETOUR

Beginning of the Internet

On October 29, 1969, a computer at the University of California at Los Angeles (UCLA) made a network connection with a computer at the Stanford Research Institute (SRI) at Stanford University. The ARPANET, the precursor to today's Internet, was born.

The development of the technologies that made this network connection possible started in the early 1960s. By that time, computers were becoming more widespread and the need to connect computers to share data became apparent. The Advanced Research Projects Agency (ARPA), an arm of the U.S. Department of Defense, decided to tackle the issue and funded network research at several American universities. Many of the networking technologies and networking concepts in use today were developed during the 1960s and then put to use on October 29, 1969.

The 1970s saw the development of the TCP/IP network protocol suite that is still in use today. The protocol specifies, among other things, how data travels from one computer on the Internet to another. The Internet grew rapidly during the 1970s and 1980s but was not widely used by the general public until the early 1990s, when the World Wide Web was developed.

Programming Languages

What distinguishes computers from other machines is that computers can be programmed. What this means is that instructions can be stored in a file on the hard drive, and then loaded into main memory and executed on demand. Because machines cannot process ambiguity the way we (humans) can, the instructions must be precise. Computers do exactly what they are told and cannot understand what the programmer "intended" to write.

The instructions that are actually executed are *machine language* instructions. They are represented using binary notation (i.e., a sequence of 0s and 1s). Because machine language instructions are extremely hard to work with, computer scientists have developed programming languages and language translators that enable developers to write instructions in a human readable language and then translate them into machine language. Such language translators are referred to as *assemblers*, *compilers*, or *interpreters*, depending on the programming language.

There are many programming languages out there. Some of them are specialized languages meant for particular applications such as 3D modeling or databases. Other languages are *general-purpose* and include C, C++, C#, Java, and Python.

While it is possible to write programs using a basic text editor, developers use *Integrated Development Environments* (IDEs) that provide a wide array of services that support software development. They include an editor to write and edit code, a language translator, automated tools for creating binary executables, and a *debugger*.

DETOUR

Computer Bugs

When a program behaves in a way that was not intended, such as crashing, freezing the computer, or simply producing erroneous output, we say that the program has a *bug* (i.e., an error). The process of removing the error and correcting the program is called *debugging*. A *debugger* is a tool that helps the developer find the instructions that cause the error.

The term "bug" to denote an error in a system predates computers and computer science. Thomas Edison, for example, used the term to describe faults and errors in the engineering of machines all the way back in the 1870s. Interestingly, there have also been cases of *actual* bugs causing computer failures. One example, as reported by computing pioneer Grace Hopper in 1947, is the moth that caused the Mark II computer at Harvard, one of the earliest computers, to fail.

Software Libraries

A general-purpose programming language such as Python consists of a small set of general-purpose instructions. This core set does not include instructions to download web pages, draw images, play music, find patterns in text documents, or access a database. The reason why is essentially because a "sparser" language is more manageable for the developer.

Of course, there are application programs that need to access web pages or databases. Instructions for doing so are defined in *software libraries* that are separate from the core language, and they must be explicitly *imported* into a program in order to be used. The description of how to use the instructions defined in a library is often referred to as the *application programming interface* (API).

1.3 Python Programming Language

In this textbook, we introduce the Python programming language and use it to illustrate core computer science concepts, learn programming, and learn application development in general. In this section, we give some background on Python and how to set up a Python IDE on your computer.

Short History of Python

The Python programming language was developed in the late 1980s by Dutch programmer Guido van Rossum while working at CWI (the Centrum voor Wiskunde en Informatica in Amsterdam, Netherlands). The language was not named after the large snake species but rather after the BBC comedy series *Monty Python's Flying Circus*. Guido van Rossum happens to be a fan. Just like the Linux OS, Python eventually became an open source software development project. However, Guido van Rossum still has a central role in deciding how the language is going to evolve. To cement that role, he has been given the title of "Benevolent Dictator for Life" by the Python community.

Python is a general-purpose language that was specifically designed to make programs very readable. Python also has a rich library making it possible to build sophisticated applications using relatively simple-looking code. For these reasons, Python has become a popular application development language *and* also the preferred "first" programming language.

CAUTION

Python 2 versus Python 3

There are currently two major versions of Python in use. Python 2 was originally made available in 2000; its latest release is 2.7. Python 3 is a new version of Python that fixes some less-than-ideal design decisions made in the early development of the Python language. Unfortunately, Python 3 is not backward compatible with Python 2. This means that a program written using Python 2 usually will not execute properly with a Python 3 interpreter.

In this textbook, we have chosen to use Python 3 because of its more consistent design. To learn more about the difference between the two releases, see:

```
http://wiki.python.org/moin/Python2orPython3
```

Setting Up the Python Development Environment

If you do not have Python development tools installed on your computer already, you will need to download a Python IDE. The official list of Python IDEs is at

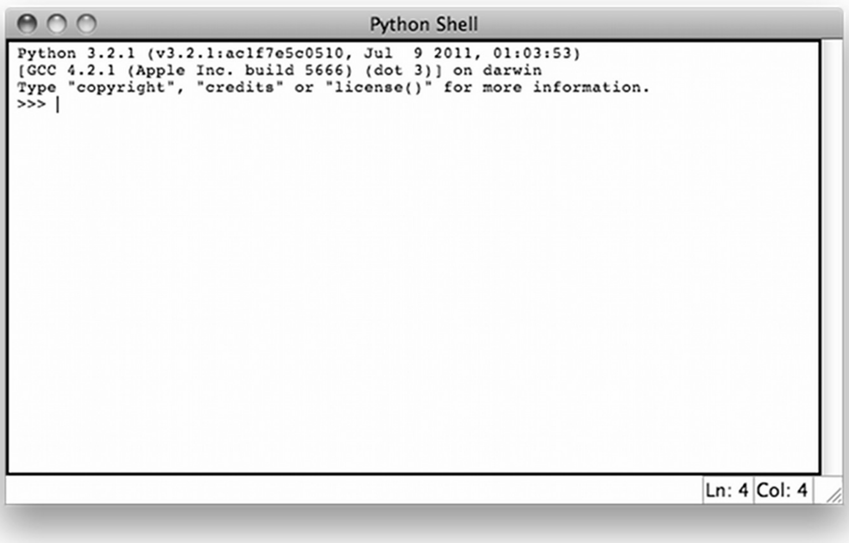
```
http://wiki.python.org/moin/IntegratedDevelopmentEnvironments
```

We illustrate the IDE installation using the standard Python development kit that includes the IDLE IDE. You may download the kit (for free) from:

```
http://python.org/download/
```

Listed there are installers for all mainstream operating systems. Choose the appropriate one for your system and complete the installation.

To get started with Python, you need to open a Python *interactive shell* window. The IDLE interactive shell included with the Python IDE is shown in Figure 1.1.



```
Python 3.2.1 (v3.2.1:ac1f7e5c0510, Jul  9 2011, 01:03:53)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> |
```

Figure 1.1 The IDLE IDE. The IDLE Integrated Development Environment is included in the standard implementation of Python. Shown is the IDLE interactive shell. At the >>> prompt, you can type single Python instructions. The instruction is executed by the Python interpreter when the Enter/Return key is pressed.

The interactive shell expects the user to type a Python instruction. When the user types the instruction `print('Hello world')` and then presses the Enter/Return key on the keyboard, a greeting is printed:

```
Python 3.2.1 (v3.2.1:ac1f7e5c0510, Jul  9 2011, 01:03:53)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> print('Hello world')
Hello world
```

The interactive shell is used to execute single Python instructions like `print('Hello world')`. A program typically consists of multiple instructions that must be stored in a file before being executed.

1.4 Computational Thinking

In order to illustrate the software development process and introduce the software development terminology, we consider the problem of automating a web search task. To model the relevant aspects of the task and describe the task as an algorithm, we must *understand* the task from a "computational" perspective. *Computational thinking* is a term used to describe the intellectual approach through which natural or artificial processes or tasks are understood and described as computational processes. This skill is probably the most important one you will develop in your training as a computer scientist.

A Sample Problem

We are interested in purchasing about a dozen prize-winning novels from our favorite online shopping web site. The thing is, we do not want to pay full price for the books. We would rather wait and buy the books on sale. More precisely, we have a target price for each book and will buy a book only when its sale price is below the target. So, every couple of days, we visit the product web page of every book on our list and, for each book, check whether the price has been reduced to below our target.