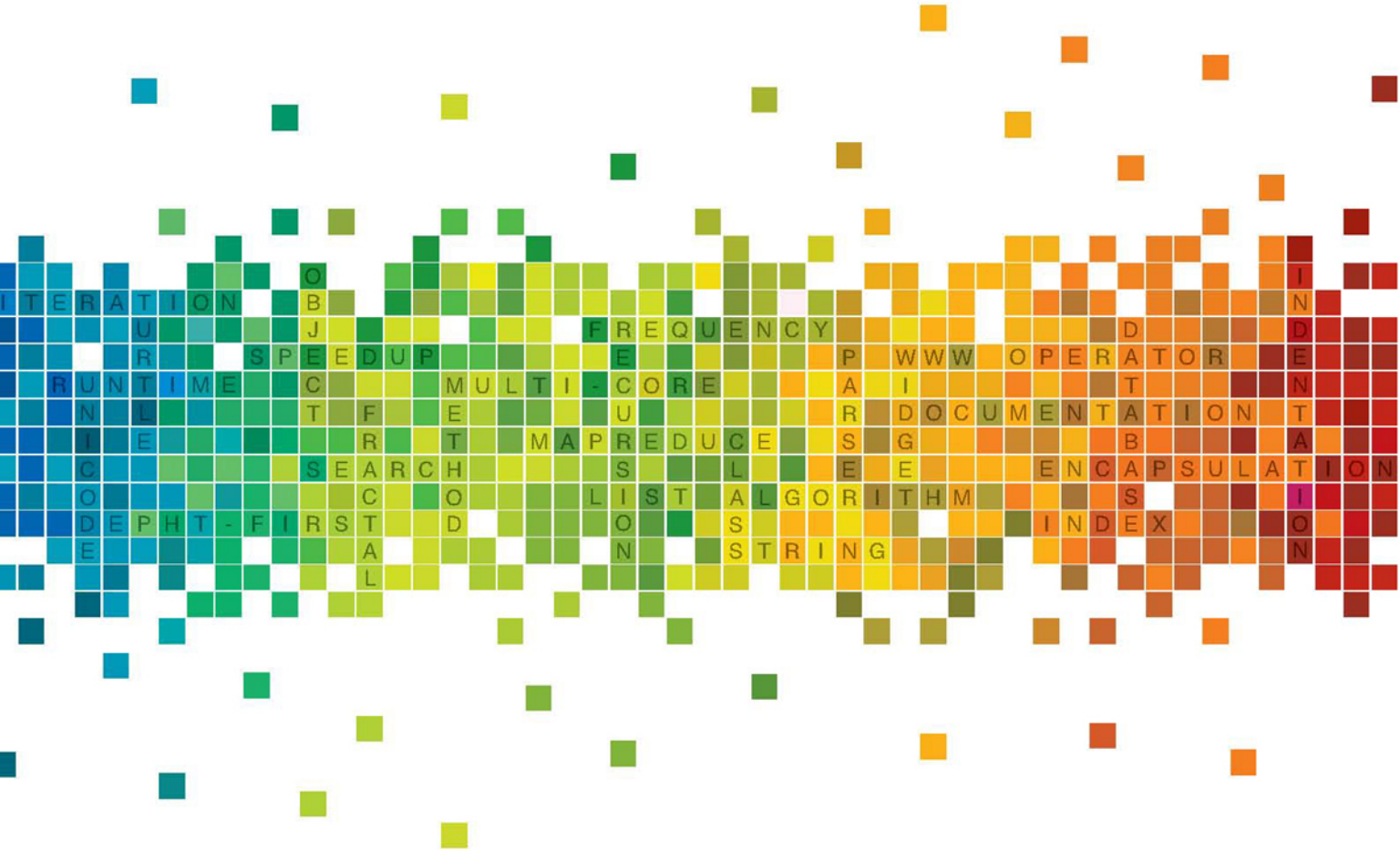
Introduction to Computing Using Python

AN APPLICATION DEVELOPMENT FOCUS



LJUBOMIR PERKOVIC

This page intentionally left blank

Introduction to Computing

Using Python

This page intentionally left blank

Introduction to Computing

Using Python

An Application Development Focus

Ljubomir Perkovic
DePaul University

VP AND EXECUTIVE PUBLISHER	Don Fowley
EXECUTIVE EDITOR	Beth Lang Golub
PROJECT LEAD	Samantha Mandel
EDITORIAL PROGRAM ASSISTANT	Elizabeth Mills
EXECUTIVE MARKETING MANAGER	Christopher Ruel
CREATIVE DIRECTOR	Harry Nolan
SENIOR DESIGNER	Wendy Lai
COVER PHOTO	©simon2579/iStockphoto
SENIOR PRODUCTION EDITOR	Sujin Hong

This book was set in Times New Roman 10 by Ljubomir Perkovic and printed and bound by Courier. The cover was printed by Courier.

This book is printed on acid-free paper. ∞

Founded in 1807, John Wiley & Sons, Inc. has been a valued source of knowledge and understanding for more than 200 years, helping people around the world meet their needs and fulfill their aspirations. Our company is built on a foundation of principles that include responsibility to the communities we serve and where we live and work. In 2008, we launched a Corporate Citizenship Initiative, a global effort to address the environmental, social, economic, and ethical challenges we face in our business. Among the issues we are addressing are carbon impact, paper specifications and procurement, ethical conduct within our business and among our vendors, and community and charitable support. For more information, please visit our website: www.wiley.com/go/citizenship.

ISBN: 978-0-470-61846-2

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.

This page intentionally left blank

Contents

6

Containers and Randomness 171

7

Namespaces 215

8

Object-Oriented Programming 251

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 and to the concepts and 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 use the context of web crawling and search engines to introduce a broad array of topics. These include foundational concepts such as recursion, regular expressions, depth-first search, and Google's MapReduce framework, as well as practical tools such as GUI widgets, HTML parsers, SQL, 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, in particular math, science, and engineering majors.

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, 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 the volume more approachable for today's students.

Most chapters in the text include a *case study* that showcases the concepts and tools covered in the chapter in context. Finally, the textbook contains a *large number of end-of-chapter problems*, many of which are unlike problems typically found in an introductory textbook.

Online Textbook Supplements

The link to the book's online content is `www.wiley.com/college/perkovic`. These supplements are available there:

- Powerpoint slides for each chapter
- Expanded tutorials on SQL and HTML
- Code examples appearing in the book
- Exercise and problem solutions (for instructors only)
- Project ideas for Chapters 5 to 12 (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, a pen, and paper. 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 consist of 12 chapters that introduce computing concepts and Python programming in a breadth-first manner.

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, and list 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. The case study at the end of the chapter 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.

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 *strings and text 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.

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*.

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.

Chapters 4 through 6 represent the second layer in the breadth-first approach this textbook takes. One of the main challenges students face in a introductory programming course is mastering of 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.

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 and 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.

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 10.5, 11.2, 12.3, and 12.4. These chapters 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, and depth-first search, to practical and contemporary tools like HTML parsers, 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.

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. The different topics covered in this chapter are put to use together with depth-first traversal from the previous chapter in the context of developing a web crawler. 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.

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.

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 Sections 9.4 and 9.5 that illustrate the OOP approach to GUI development, as well as Sections 10.5, 11.2, 12.3, and 12.4, all of which make use of user-defined classes. All these sections 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 this textbook was developed over three years in the context of teaching the CSC241/242 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 anonymous reviewers. 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, 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.

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.

Prof. Perkovic has 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 distributed computing, computational geometry, 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.

This page intentionally left blank

CHAPTER

1

Introduction to Computer Science

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 introduce 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 and 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 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 added a graphical interface to the operating system