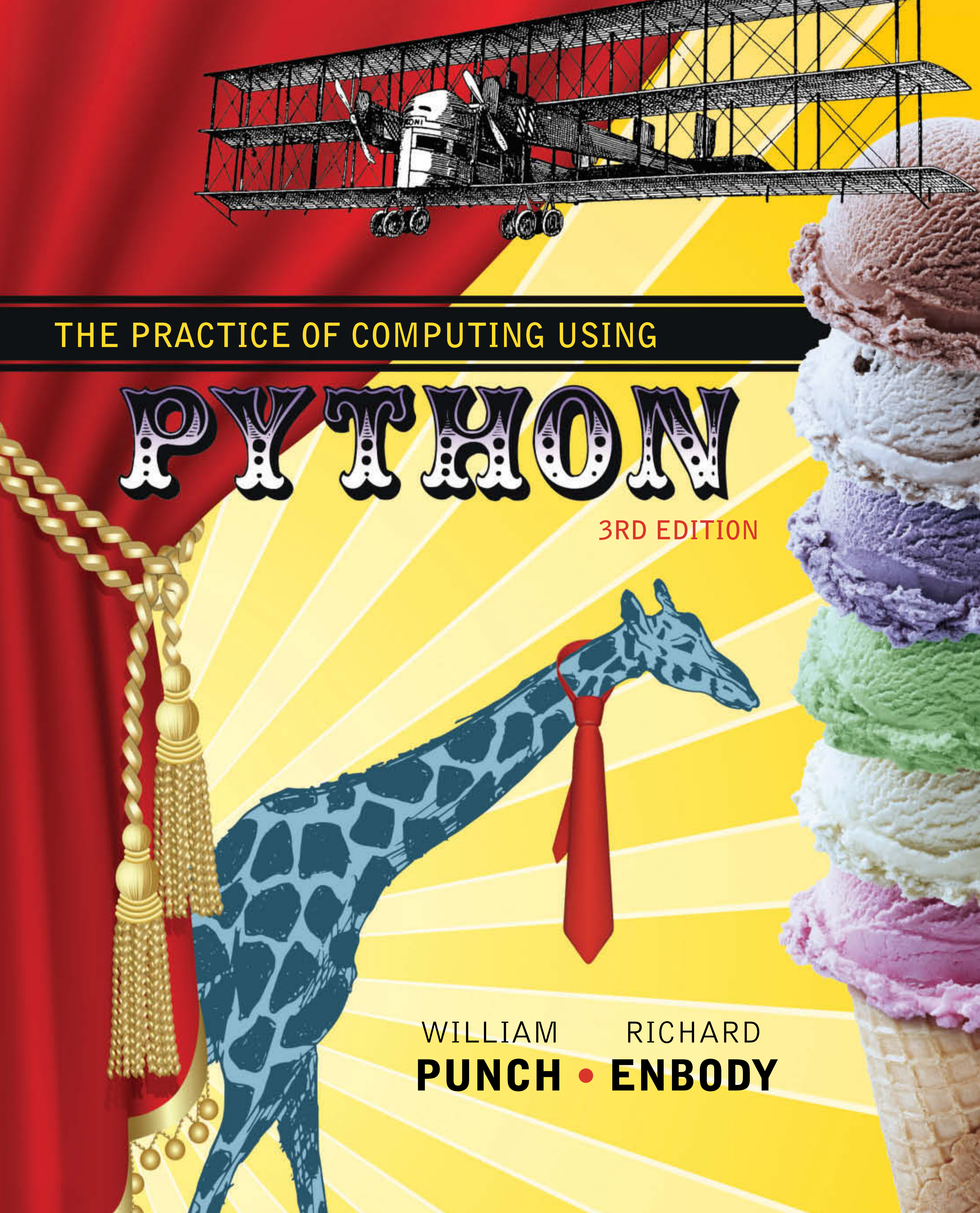

THE PRACTICE OF COMPUTING USING
PYTHON
3RD EDITION
WILLIAM PUNCH • RICHARD ENBODY

THE PRACTICE OF COMPUTING USING

THIRD EDITION

PYTHON

William Punch

Richard Enbody

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President, Editorial Director, ECS: Marcia Horton
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Kristy Alaura
Vice President of Marketing: Christy Lesko
Director of Field Marketing: Tim Galligan
Product Marketing Manager: Bram Van Kempen
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Director of Product Management: Erin Gregg
Team Lead, Program and Project Management: Scott Disanno
Program Manager: Carole Snyder
Senior Specialist, Program Planning and Support: Maura Zaldivar-Garcia
Cover Designer: Joyce Wells
Manager, Rights and Permissions: Rachel Youdelman
Project Manager, Rights and Permissions: William Opaluch
Inventory Manager: Meredith Maresca
Media Project Manager: Dario Wong
Full-Service Project Management: Jogender Taneja, iEnerziger Aptara®, Ltd.
Composition: iEnerziger Aptara®, Ltd.
Printer/Binder: Edwards Brothers Malloy, Inc.
Cover and Insert Printer: Phoenix Color

Cover Photo Credit: Unorobus/Fotolia, Ifong/123RF, Deposit Photos/Glow Images, Onot/Shutterstock, Nataliia Natykach/123RF, Vitezslav Valka/123RF

Library of Congress Cataloging-in-Publication Data available upon request.
Names: Punch, W. F. (William F.), author. | Enbody, Richard J., author.
Title: The practice of computing using Python / W.F. Punch and R.J. Enbody, Department of Computer Science and Engineering, Michigan State University.
Description: 3rd edition. | Boston : Pearson, 2016. | Includes bibliographical references and index.
Identifiers: LCCN 2015050451| ISBN 9780134379760 | ISBN 0134379764
Subjects: LCSH: Python (Computer program language) | Computer programming.
Classification: LCC QA76.73.P98 P92 2016 | DDC 005/13/3–dc23 LC record available at http://lccn.loc.gov/2015050451

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 10: 0-13-437976-4
ISBN 13: 978-0-13-437976-0

To our beautiful wives Laurie and Wendy and our kids Zach, Alex,
Abby, Carina, and Erik,
and our parents.
We love you and couldn't have done this
without your love and support.

This page intentionally left blank

BRIEF CONTENTS

CONTENTS

VIDEONOTES

PREFACE

A FIRST COURSE IN COMPUTER SCIENCE IS ABOUT A NEW WAY OF SOLVING PROBLEMS computationally. Our goal is that after the course, students when presented with a problem will think, "Hey, I can write a program to do that!"

The teaching of problem solving is inexorably intertwined with the computer language used. Thus, the choice of language for this first course is very important. We have chosen Python as the introductory language for beginning programming students—majors and non-majors alike—based on our combined 55 years of experience teaching undergraduate introductory computer science at Michigan State University. Having taught the course in Pascal, C/C++, and now Python, we know that an introductory programming language should have two characteristics. First, it should be relatively simple to learn. Python's simplicity, powerful built-in data structures, and advanced control constructs allow students to focus more on problem solving and less on language issues. Second, it should be practical. Python supports learning not only fundamental programming issues such as typical programming constructs, a fundamental object-oriented approach, common data structures, and so on, but also more complex computing issues such as threads and regular expressions. Finally, Python is "industrial strength" forming the backbone of companies such as YouTube, DropBox, Industrial Light and Magic, and many others.

We emphasize both the fundamental issues of programming and practicality by focusing on data manipulation and analysis as a theme—allowing students to work on real problems using either publicly available data sets from various Internet sources or self-generated data sets from their own work and interests. We also emphasize the development of programs, providing multiple, worked out, examples, and three entire chapters for detailed design and implementation of programs. As part of this one-semester course, our students have analyzed breast cancer data, catalogued movie actor relationships, predicted disruptions of satellites from solar storms, and completed many other data analysis problems. We have also found that concepts learned in a Python CS1 course transitioned to a CS2 C++ course with little or no impact on either the class material or the students.

Our goals for the book are as follows:

- Teach problem solving within the context of CS1 to both majors and nonmajors using Python as a vehicle.
- Provide examples of *developing* programs focusing on the kinds of data analysis problems students might ultimately face.
- Give students who take no programming course other than this CS1 course a practical foundation in programming, enabling them to produce useful, meaningful results in their respective fields of study.

WHAT'S NEW, THIRD EDITION

We have taught with this material for over eight years and continue to make improvements to the book as well as adapting to the ever changing Python landscape, keeping up to date with improvements. We list the major changes we have made below.

Anaconda: One of the issues our students ran into was the complexity associated with getting Python packages, along with the necessary pre-requisites. Though tools like `pip` address this problem to some extent, the process was still a bit overwhelming for introductory students.

Thus we switched to the **Anaconda** distribution made freely available from Continuum Analytics. They make available a full distribution with more than 100 modules pre-installed, removing the need for package installation.

Appendix A, newly written for Anaconda, covers the installation process.

SPYDER: Another benefit of the Anaconda distribution is the SPYDER Integrated Development Environment. We have fully adopted SPYDER as our default method for editing and debugging code, changing from the default IDLE editor. SPYDER provides a full development environment and thus has a number of advantages (as listed at the Spyder git page, `https://github.com/spyder-ide/spyder/blob/master/README.md`).

- Integrated editor
- Associated interactive console
- Integrated debugging
- Integrated variable explorer
- Integrated documentation viewer

SPYDER is a truly modern Python IDE and the correct way for students to learn Python programming.

Chapter 1 has been rewritten, incorporating the SPYDER IDE and using SPYDER is sprinkled throughout the book. Appendix A provides a tutorial on SPYDER.

IPython: Anaconda also provides the iPython console as its interactive console. IPython is a much more capable console than the default Python console, providing many features including:

- An interactive history list, where each history line can be edited and re-invoked.
- Help on variables and functions using the "?" syntax.
- Command line completion

Every session of the book was redone for the iPython console and iPython's features are sprinkled throughout the book. Further, a tutorial on the use of iPython is provided in Appendix A.

Debugging help: Debugging is another topic that is often a challenge for introductory students. To address this need, we have introduced a "What's Wrong with My Code" element to the end of chapters 1, 2, 4, 5, 6, 7, 8, 9, and 11. These provide increasingly detailed tips to deal with new Python features as they are introduced. An overall tutorial is also provided in the **new Appendix C**.

As Spyder provides a debugger, use of that debugger is used for all examples.

Pylab updated: We have incorporated graphing through `matplotlib/pylab` into the book since the first edition, but this module has changed somewhat over the years so the "Visual Vignettes" at the end of chapters 1, 2, 5, 7, 9, and 13 have been updated and somewhat simplified. In particular the discussions of NumPy have been removed except for where they are useful for graphing. Appendix D has also been updated with these changes.

Web-based GUIs: Building Graphic User Interfaces (GUIs) is a topic many students are interested in. However, in earlier editions we hesitated to focus on GUI development as part of an introductory text for a number of reasons:

- The extant `tkinter` is cross platform, but old and more complex to work with than we would like for an introductory class.
- Getting a modern GUI toolset can be daunting, especially cross platform.
- Just **which** GUI toolset should we be working with?

A discussion with Greg Wilson (thanks Greg!) was helpful in resolving this problem. He suggested doing Web-based GUIs as a modern GUI approach that is cross-platform, relatively stable and provided in modern distributions like Anaconda.

What that left was the "complexity" issue. We choose to use the package `flask` because it was relatively less complex, but more importantly was easily modularized and could be used in a template fashion to design a GUI.

Thus, we wrote some simple GUI development in `flask` at the end of chapters 4, 6, 7, 9, 10, and 15. We also wrote a **new Appendix E** as a tutorial for development of web-based GUIs.

Functions Earlier: One of the common feedback points we received was a request to introduce functions earlier. Though we had purposefully done strings first, as a way to start working with data, we had sympathy for those instructors who wanted to change the order and introduce functions before strings. Rather than pick a right way to do this, we **rewrote Chapter 5** so that it had no dependencies Chapter 4, the string chapter. Instructors are now free to introduce functions anytime after Chapter 2 on control. Likewise Chapter 4 on strings has no dependencies on Chapter 5, the functions chapters. Thus the instructor can choose the order they prefer.

Online Project Archive: We have established an online archive for Python CS1 projects, `http://www.cse.msu.edu/~cse231/PracticeOfComputingUsingPython/`. These describe various projects we have used over the years in our classes. This site and its contents has been recognized by the National Center for Women & Information Technology (NCWIT) and was awarded the 2015 NCWIT EngageCSEdu Engagement Excellence Award.

New Exercises: We added 80 new end-of-chapter exercises.

Other changes: We have made a number of other changes as well:

- We updated Chapter 16 with a discussion about Python Numbers and the various representations that are available
- We moved the content of some of the chapters in the "Getting Started" part to the "Data Structures and Functions" part. This is really just a minor change to the table of contents.
- We fixed various typos and errors that were either pointed out to us or we found ourselves as we re-read the book.

PREFACE TO THE SECOND EDITION

The main driver for a second edition came from requests for Python 3. We began our course with Python 2 because Python 3 hadn't been released in 2007 (it was first released in December 2008), and because we worried that it would take some time for important open-source packages such as NumPy and MatPlotLib to transition to Python 3. When NumPy and MatPlotLib converted to Python 3 in 2011 we felt comfortable making the transition. Of course, many other useful modules have also been converted to Python 3—the default installation now includes thousands of modules. With momentum building behind Python 3 it was time for us to rewrite our course and this text.

Why Python 3? The Python community decided to break backward compatibility with Python 3 to fix nagging inconsistencies in the language. One important change was moving the default character encoding to Unicode which recognizes the world-wide adoption of the language. In many ways beyond the introductory level, Python 3 is a better language and the community is making the transition to Python 3.

At the introductory level the transition to Python 3 appears to be relatively small, but the change resulted in touching nearly every page of the book.

One notable addition was:

- We added a set of nine **RULES** to guide novice programmers.

We reworked every section of this text—some more than others. We hope that you will enjoy the changes. Thanks.

BOOK ORGANIZATION

At the highest level our text follows a fairly traditional CS1 order, though there are some differences. For example, we cover strings rather early (before functions) so that we can do more data manipulation early on. We also include elementary file I/O early for the same reason, leaving detailed coverage for a later chapter. Given our theme of data manipulation, we feel this is appropriate. We also "sprinkle" topics like plotting and drawing throughout the text in service of the data manipulation theme.

We use an "object-use-first" approach where we use built-in Python objects and their methods early in the book, leaving the design and implementation of user-designed objects for later. We have found that students are more receptive to building their own classes once they have experienced the usefulness of Python's existing objects. In other words, we motivate the need for writing classes. Functions are split into two parts because of how Python handles mutable objects such as lists as parameters; discussion of those issues can only come after there is an understanding of lists as mutable objects.

Three of the chapters (3, 10, and 13) are primarily program design chapters, providing an opportunity to "tie things together," as well as showing how to design a solution. A few chapters are intended as supplemental reading material for the students, though lecturers may choose to cover these topics as well. For background, we provide a Chapter 0 that introduces some general concepts of a computer as a device and some computer terminology. We feel such an introduction is important—everyone should understand a little about a computer, but this material can be left for reading. The last chapters in the text may not be reached in some courses.

BOOK FEATURES

1.0.1 Data Manipulation

Data manipulation is a theme. The examples range from text analysis to breast cancer classification. The data.gov site is a wonderful source of interesting and relevant data. Along the way, we provide some analysis examples using simple graphing. To incorporate drawing and graphing, we use established packages instead of developing our own: one is built-in (Turtle graphics); the other is widely used (MatPlotLib with NumPy).

We have tried to focus on non-numeric examples in the book, but some numeric examples are classics for a good reason. For example, we use a rational numbers example for creating classes that overload operators. Our goal is always to use the best examples.

1.0.2 Problem Solving and Case Studies

Throughout the text, we emphasize problem solving, especially a divide-and-conquer approach to developing a solution. Three chapters (3, 10, and 13) are devoted almost exclusively to program development. Here we walk students through the solution of larger examples. In addition to design, we show mistakes and how to recover from them. That is, we don't simply show a solution, but show a *process of developing* a solution.

1.0.3 Code Examples

There are over 180 code examples in the text—many are brief, but others illustrate piecemeal development of larger problems.

1.0.4 Interactive Sessions

The Python interpreter provides a wonderful mechanism for briefly illustrating programming and problem-solving concepts. We provide almost 250 interactive sessions for illustration.

1.0.5 Exercises and Programming Projects

Practice, practice, and more practice. We provide over 275 short exercises for students and nearly 30 longer programming projects (many with multiple parts).

1.0.6 Self-Test Exercises

Embedded within the chapters are 24 self-check exercises, each with five or more associated questions.

1.0.7 Programming Tips

We provide over 40 special notes to students on useful tips and things to watch out for. These tips are boxed for emphasis.

SUPPLEMENTARY MATERIAL ONLINE

- For students
  - All example source code
  - Data sets used in examples

The above material is freely available at www.pearsonhighered.com/cs-resources/

- For instructors
  - PowerPoint slides
  - Laboratory exercises
  - Figures (PDF) for use in your own slides
  - Exercise solutions

Qualified instructors may obtain supplementary material by visiting www.pearsonhighered.com/irc. Register at the site for access. You may also contact your local Pearson Education sales representative

W. F. Punch
R. J. Enbody

BREAKTHROUGH
To improving results

MyProgrammingLab™

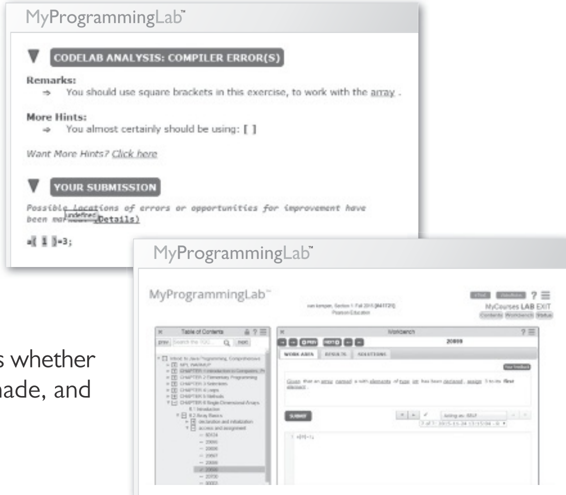
Through the power of practice and immediate personalized feedback, MyProgrammingLab helps improve your students' performance.

PROGRAMMING PRACTICE

With MyProgrammingLab, your students will gain first-hand programming experience in an interactive online environment.

IMMEDIATE, PERSONALIZED FEEDBACK

MyProgrammingLab automatically detects errors in the logic and syntax of their code submission and offers targeted hints that enables students to figure out what went wrong and why.



GRADUATED COMPLEXITY

MyProgrammingLab breaks down programming concepts into short, understandable sequences of exercises. Within each sequence the level and sophistication of the exercises increase gradually but steadily.

DYNAMIC ROSTER

Students' submissions are stored in a roster that indicates whether the submission is correct, how many attempts were made, and the actual code submissions from each attempt.

PEARSON eTEXT

The Pearson eText gives students access to their textbook anytime, anywhere.

STEP-BY-STEP VIDEONOTE TUTORIALS

These step-by-step video tutorials enhance the programming concepts presented in select Pearson textbooks.

For more information and titles available with **MyProgrammingLab**, please visit **www.myprogramminglab.com**.

ALWAYS LEARNING

PEARSON

PART 1

Thinking About Computing

This page intentionally left blank

CHAPTER 0

The Study of Computer Science

> Composing computer programs to solve scientific problems is like writing poetry. You must choose every word with care and link it with the other words in perfect syntax.
>
> James Lovelock

0.1 WHY COMPUTER SCIENCE?

It is a fair question to ask. Why should anyone bother to study computer science? Furthermore, what is "computer science"? Isn't this all just about programming? All good questions. We think it is worth discussing them before you forge ahead with the rest of the book.

0.1.1 Importance of Computer Science

Let's be honest. We wouldn't be writing the book and asking you to spend your valuable time if we didn't think that studying computer science is important. There are a couple of ways to look at why this is true.

First, we all know that computers are everywhere, millions upon millions of them. What were once rare, expensive items are as common place as, well, any commodity you can imagine (we were going to say the proverbial toaster, but there are many times more computers than toasters. In fact, there is likely a small computer *in* your toaster!). However, that isn't enough of a reason. There are millions and millions of cars, and universities don't require auto mechanics as an area of study.

Second, Computers are not only common, but they are also more universally applicable than any other commodity in history. A car is good for transportation, but a computer can be used in so many situations. In fact, there is almost no area one can imagine where a computer would *not* be useful. That is a key attribute. No matter what your area of interest, a computer could be useful there as a *tool*. The computer's universal utility is unique, and learning how to use such a tool is important.

0.1.2 Computer Science Around You

Computing surrounds you, and it is computer science that put it there. There are a multitude of examples, but here are a few worth noting.

Social Networking The tools that facilitate social networking such as Facebook or Twitter are, of course, computer programs. However, the tools that help study the interactions within social networks involve important computer science fields such as *graph theory*. For example, the Iraqi dictator Saddam Hussein was located using the graph theoretic analysis of his social network.

Smartphones Smartphones are small, very portable computers. Apps for smartphones are simple computer programs written specifically for smartphones.

Your Car Your car probably hosts dozens of computers. They control the engine, the brakes, the audio system, the navigation, and the climate control system. They determine if a crash is occurring and trigger the air bags. Some cars park automatically or apply the brakes if a crash is imminent. Fully autonomous cars are being tested, as are cars that talk to each other.

The Internet The backbone of the Internet is a collection of connected computers called *routers* that decide the best way to send information to its destination.

0.1.3 Computer "Science"

> Any field that has the word science in its name is guaranteed thereby not to be a science.
>
> Frank Harary

A popular view of the term "computer science" is that it is a glorified way to say "computer programming." It is true that computer programming is often the way that people are introduced to computing in general, and that computer programming is the primary reason many take computing courses. However, there is indeed more to computing than programming, hence the term "computer science." Here are a few examples.

Theory of Computation

Before there were the vast numbers of computers that are available today, scientists were thinking about what it means to do computing and what the limits might be. They would ask questions, such as whether there exist problems that we can conceive of but cannot compute. It turns out there are. One of these problems, called the "Halting Problem,"[1] cannot be solved by a program running on any computer. Knowing what you can and cannot solve on a computer is an important issue and a subject of study among computer scientists that focus on the theory of computation.

Computational Efficiency

The fact that a problem is computable does not mean it is easily computed. Knowing roughly how difficult a problem is to solve is also very important. Determining a meaningful measure of difficulty is, in itself, an interesting issue, but imagine we are concerned only with time. Consider designing a solution to a problem that, as part of the solution, required you to sort 100,000 items (say cancer patient records, or asteroid names, or movie episodes, etc.). A slow algorithm, such as the sorting algorithm called the Bubble Sort, might take approximately 800 seconds (about 13 minutes); another sorting algorithm called Quick Sort might take approximately 0.3 seconds. That is a difference of around 2400 times! That large a difference might determine whether it is worth doing. If you are creating a solution, it would be good to know what makes your solution slow or what makes it fast.

Algorithms and Data Structures

Algorithms and data structures are the currency of the computer scientist. Discussed more in Chapter 3, algorithms are the methods used to solve problems, whereas data structures are the organizations of data that the algorithms use. These two concepts are distinct: a general approach to solving a problem (such as searching for a particular value, sorting a list of objects and encrypting a message) differs from the organization of the data that is being processed (as a list of objects, as a dictionary of key-value pairs, as a "tree" of records). However, they are also tightly coupled. Furthermore, both algorithms and data structures can be examined independently of how they might be programmed. That is, one designs algorithms and data structures and then actually implements them in a particular computer program. Understanding abstractly how to design both algorithms and data structures independent of the programming language is critical for writing correct and efficient code.

Parallel Processing

It may seem odd to include what many consider an advanced topic, but parallel processing, using multiple computers to solve a problem, is an issue for everyone these days. Why? As it turns out, most computers come with at least two processors or CPUs (see Section 0.6), and many come with four or more. The Playstation4(TM) game console uses a special AMD chip with a total of eight processors, and Intel has released its new Phi card with more than 60 processors! What does this mean to us, as both consumers and new computer scientists?

[1] http://www.wired.com/2014/02/halting-problem/

The answer is that new algorithms, data structures, and programming paradigms will be needed to take advantage of this new processing environment. Orchestrating many processors to solve a problem is an exciting and challenging task.

Software Engineering

Even the process of writing programs itself has developed its own subdiscipline within computer science. Dubbed "software engineering," it concerns the process of creating programs: from designing the algorithms they use, to supporting testing and maintenance of the program once created. There is even a discipline interested in representing a developed program as a mathematical entity so that one can *prove* what a program will do once written.

Many Others

We have provided but a taste of the many fields that make computer science such a wonderfully rich area to explore. Every area that uses computation brings its own problems to be explored.

0.1.4 Computer Science Through Computer Programming

We have tried to make the point that computer science is not just programming. However, it is also true that for much of the book we will focus on just that aspect of computer science: programming. Beginning with "problem solving through programming" allows one to explore pieces of the computer science landscape as they naturally arise.

0.2 THE DIFFICULTY AND PROMISE OF PROGRAMMING

If computer science, particularly computer programming, is so interesting, why doesn't everybody do it? The truth is that it can be hard. We are often asked by beginning students, "Why is programming so hard?" Even grizzled programming veterans, when honestly looking back at their first experience, remember how difficult that first programming course was. Why? Understanding why it might be hard gives you an edge on what you can do to control the difficulty.

0.2.1 Difficulty 1: Two Things at Once

Let's consider an example. Let us say that, when you walk into that first day of Programming 101, you discover the course is not about programming but French poetry. French poetry?

Yes, French poetry. Imagine that you come in and the professor posts the following excerpt from a poem on the board.

A une Damoyselle malade

Ma mignonne,
Je vous donne
Le bon jour;
Le séjour
C'est prison.

Clément Marot

Your assigned task is to translate this poetry into English (or German, or Russian, whatever language is your native tongue). Let us also assume, for the moment, that:

(a) You do not know French.
(b) You have never studied poetry.

You have two problems on your hands. First, you have to gain a better understanding of the syntax and semantics (the form and substance) of the French language. Second, you need to learn more about the "rules" of poetry and what constitutes a good poem.

Lest you think that this is a trivial matter, an entire book has been written by Douglas Hofstadter on the very subject of the difficulty of translating this one poem ("Le Ton beau de Marot").

So what's your first move? Most people would break out a dictionary and, line by line, try to translate the poem. Hofstadter, in his book, does exactly that, producing the crude translation in Figure 0.1.

My Sweet/Cute [One] (Feminine)	**A une Damoyselle malade**
My sweet/cute [one] (feminine) I [to] you (respectful) give/bid/convey The good day (i.e., a hello, i.e., greetings). The stay/sojourn/ visit (i.e., quarantine) {It} is prison.	Ma mignonne, Je vous donne Le bon jour; Le séjour C'est prison.

FIGURE 0.1 Crude translation of excerpt.

The result is hardly a testament to beautiful poetry. This translation does capture the syntax and semantics, but not the poetry, of the original. If we take a closer look at the poem, we can discern some features that a good translation should incorporate. For example:

- Each line consists of three syllables.
- Each line's main stress falls on its final syllable.
- The poem is a string of rhyming couplets: *AA, BB, CC, …*
- The semantic couplets are out of phase with the rhyming couplets: *A, AB, BC, …*

Taking some of these ideas (and many more) into account, Hofstadter comes up with the translation in Figure 0.2.

My Sweet Dear	**A une Damoyselle malade**
My sweet dear, I send cheer – All the best! Your forced rest Is like jail.	Ma mignonne, Je vous donne Le bon jour; Le séjour C'est prison.

FIGURE 0.2 Improved translation of excerpt.

Not only does this version sound far more like poetry, but it also matches the original poem, following the rules and conveying the intent. It is a pretty good translation!

Poetry to Programming?

How does this poetry example help? Actually, the analogy is pretty strong. In coming to programming for the first time, you face exactly the same issues:

- You are not yet familiar with the syntax and semantics of the language you are working with—in this case, of the programming language Python and perhaps not of *any* programming language.
- You do not know how to solve problems using a computer—similar to not knowing how to write poetry.

Just like the French poetry neophyte, you are trying to solve two problems simultaneously. On one level, you are just trying to get familiar with the syntax and semantics of the language. At the same time, you are tackling a second, very difficult task: creating poetry in the previous example and solving problems using a computer in this course.

Working at two levels, the meaning of the programming words and then the intent of the program (what the program is trying to solve) are the two problems the beginning programmer has to face. Just like the French poetry neophyte, your first programs will be a bit clumsy as you learn both the programming language and how to use that language to solve problems. For example, to a practiced eye, many first programs look similar in nature to the literal translation of Hofstadter's in Figure 0.1. Trying to do two things simultaneously is difficult for anyone, so be gentle on yourself as you go forward with the process.

You might ask whether there is a better way. Perhaps, but we have not found it yet. The way to learn programming is to program, just like swinging a baseball bat, playing the piano, and winning at bridge; you can hear the rules and talk about the strategies, but learning is best done by doing.

0.2.2 Difficulty 2: What Is a Good Program?

Having mastered some of the syntax and semantics of a programming language, how do we write a good program? That is, how do we create a program that is more like poetry than like the mess arrived at through literal translation?

It is difficult to discuss a good program when, at this point, you know so little, but there are a couple of points that are worth noting even before we get started.

It's All About Problem Solving

If the rules of poetry are what guides writing good poetry, what are the guidelines for writing good programs? That is, what is it we have to learn in order to transition from a literal translation to a good poem?

For programming, it is *problem solving*. When you write a program, you are creating, in some detail, how it is that *you* think a particular problem or some class of problems, should be solved. Thus, the program represents, in a very accessible way, your thoughts on problem solving. Your thoughts! That means, before you write the program you must *have* some thoughts.

It is a common practice, even among veteran programmers, to get a problem and immediately sit down and start writing a program. Typically that approach results in a mess, and, for the beginning programmer, it results in an unsolved problem. Figuring out how to solve a problem requires some initial thought. If you think before you program, you better understand what the problem requires as well as the best strategies you might use to solve that problem.

Remember the two-level problem? Writing a program as you figure out how to solve a problem means that you are working at two levels at once: the problem-solving level and the programming level. That is more difficult than doing things sequentially. You should sit down and think about the problem and how you want to solve it *before* you start writing the program. We will talk more about this later, but rule 1 is:

> **Rule 1:** Think before you program!

A Program as an Essay

When students are asked "What is the most important feature a program should have?" many answer, "It should run." By "run," they mean that the program executes and actually does something.