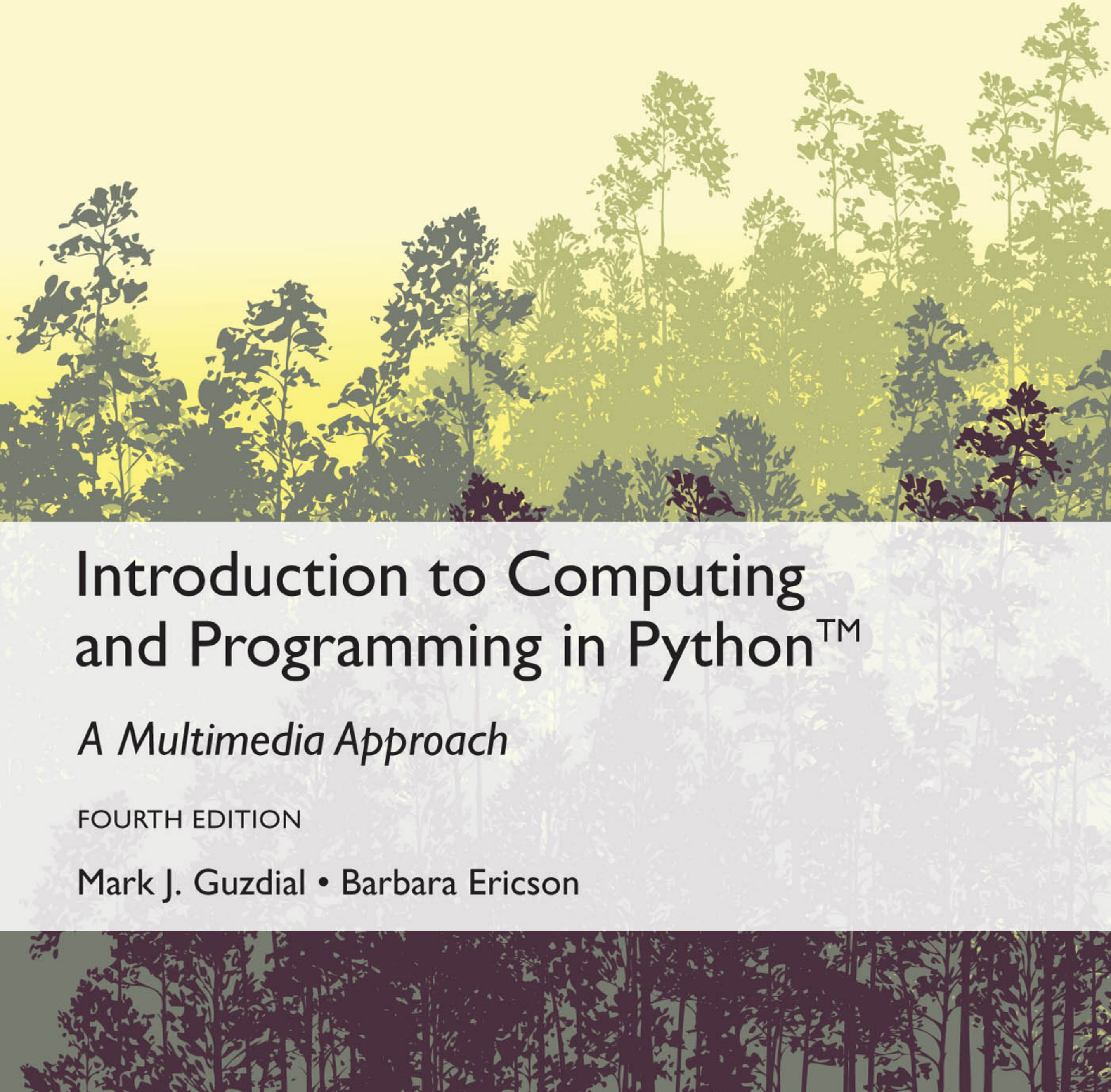
GLOBAL
EDITION



Introduction to Computing and Programming in Python™

A Multimedia Approach

FOURTH EDITION

Mark J. Guzdial • Barbara Ericson

ALWAYS LEARNING

PEARSON

INTRODUCTION TO COMPUTING AND PROGRAMMING IN PYTHON™

A MULTIMEDIA APPROACH

Mark J. Guzdial and Barbara Ericson
College of Computing/GVU
Georgia Institute of Technology

Fourth Edition

Global Edition

PEARSON

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

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Tracy Johnson*
Assistant Acquisitions Editor, Global Edition: *Aditee Agarwal*
Executive Marketing Manager: *Tim Galligan*
Marketing Assistant: *Jon Bryant*
Senior Managing Editor: *Scott Disanno*
Production Project Manager: *Greg Dulles*
Program Manager: *Carole Snyder*
Project Editor, Global Edition: *K.K. Neelakantan*
Senior Manufacturing Controller, Global Edition: *Kay Holman*
Media Production Manager, Global Edition: *Vikram Kumar*
Global HE Director of Vendor Sourcing and Procurement: *Diane Hynes*
Director of Operations: *Nick Sklitsis*
Operations Specialist: *Maura Zaldivar-Garcia*
Cover Designer: *Lumina Datamatics*
Manager, Rights and Permissions: *Rachel Youdelman*
Associate Project Manager, Rights and Permissions: *Timothy Nicholls*
Full-Service Project Management: *Kalpana Arumugam, SPi Global*

MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THE INFORMATION CONTAINED IN THE DOCUMENTS AND RELATED GRAPHICS PUBLISHED AS PART OF THE SERVICES FOR ANY PURPOSE. ALL SUCH DOCUMENTS AND RELATED GRAPHICS ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS HEREBY DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS INFORMATION, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF INFORMATION AVAILABLE FROM THE SERVICES. THE DOCUMENTS AND RELATED GRAPHICS CONTAINED HEREIN COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED HEREIN AT ANY TIME. PARTIAL SCREEN SHOTS MAY BE VIEWED IN FULL WITHIN THE SOFTWARE VERSION SPECIFIED.

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

© Pearson Education Limited 2016

The rights of Mark J. Guzdial and Barbara Ericson to be identified as the authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Introduction to Computing and Programming in Python™: A Multimedia Approach, Fourth Edition, ISBN 9780134025544, by Mark J. Guzdial and Barbara Ericson published by Pearson Education © 2016.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

ISBN 10: 1-292-10986-6
ISBN 13: 978-1-292-10986-2

Typeset in 10.5/13 Times by SPi Global
Printed and bound in Malaysia.

Dedicated to our first teachers, our parents:
Janet, Charles, Gene, and Nancy

Contents

1 INTRODUCTION 25

Preface for the Fourth Edition

We started Media Computation in the of Summer 2002, and taught it for the first time in Spring 2003. It's now over ten years later, which is a good time to summarize the changes across the second, third, and fourth editions.

Media Computation has been used successfully in an undergraduate course at Georgia Tech for the last dozen years. The course continues to have high retention rates (over 85% of students complete the class with a passing grade), and is majority female. Both students and teachers report *enjoying* the course, which is an important recommendation for it.

Researchers have found that Media Computation works in a variety of contexts. The University of Illinois-Chicago had the first Media Computation paper outside of Georgia, and they showed how switching to MediaComp improved their retention rates in classes that were much more diverse than those at Georgia Tech [41]. The University of California-San Diego adopted Media Computation as part of a big change in their introductory course, where they also started using pair-programming and peer instruction. Their paper at the 2013 SIGCSE Symposium showed how these changes led to dramatic improvements in student retention, even measured a year later in the Sophomore year. The paper also won the Best Paper award at the conference [27]. It's been particularly delightful to see Media Computation adopted and adapted for new settings, like Cynthia Bailey Lee's creation of a MATLAB Media Computation curriculum [12].

Mark wrote a paper in 2013, summarizing ten years of Media Computation research. Media Computation does often improve retention. Our detailed interview studies with female students supports the claim that they find the approach to be creative and engaging, and that's what keeps the students in the class. That paper won the Best Paper award at the 2013 International Computing Education Research (ICER) Conference [33].

HOW TO TEACH MEDIA COMPUTATION

Over the last 10 years, we have learned some of the approaches that work best for teaching Media Computation.

- *Let the students be creative.* The most successful Media Computation classes use open-ended assignments that let the students choose what media they use. For example, a collage assignment might specify the use of particular filters and compositions, but allow for the student to choose exactly what pictures are used. These assignments often lead to the students putting in a lot more time to get *just* the look that they wanted, and that extra time can lead to improved learning.

- *Let the students share what they produce.* Students can produce some beautiful pictures, sounds, and movies using Media Computation. Those products are more motivating for the students when they get to share them with others. Some schools provide online spaces where students can post and share their products. Other schools have even printed student work and held an art gallery.
- *Code live in front of the class.* The best part of the teacher actually typing in code in front of the class is that *nobody* can code for long in front of an audience and *not* make a mistake. When the teacher makes a mistake and fixes it, the students see (a) that errors are expected and (b) there is a process for fixing them. Coding live when you are producing images and sounds is fun, and can lead to unexpected results and the opportunity to explore, "How did *that* happen?"
- *Pair programming leads to better learning and retention.* The research results on pair programming are tremendous. Classes that use pair programming have better retention results, and the students learn more.
- *Peer instruction is great.* Not only does peer instruction lead to better learning and retention outcomes, but it also gives the teacher better feedback on what the students are learning and what they are struggling with. We strongly encourage the use of peer instruction in computing classes.
- *Worked examples help with creativity learning.* Most computer science classes do not provide anywhere nearly enough worked-out examples for students to learn from. Students like to learn from examples. One of the benefits of Media Computation is that we provide a lot of examples (we've never tried to count the number of `for` and `if` statements in the book!), *and* it's easy to produce more of them. In class, we do an activity where we hand out example programs, then show a particular effect. We ask pairs or groups of students to figure out which program generated that effect. The students talk about code, and study a bunch of examples.

AP CS PRINCIPLES

The Advanced Placement exam in CS Principles[1] has now been defined. We have explicitly written the fourth edition with CS Principles in mind. For example, we show how to measure the speed of a program empirically in order to contrast two algorithms (Learning Objective 4.2.4), and we explore multiple ways of analyzing CSV data from the Internet (Learning Objectives 3.1.1, 3.2.1, and 3.2.2).

Overall, we address the CS Principles learning objectives explicitly in this book as shown below:

- In *Big Idea I: Creativity*:
- LO 1.1.1: . . . use computing tools and techniques to create artifacts.
- LO 1.2.1: . . . use computing tools and techniques for creative expression.

---

[1] `http://apcsprinciples.org`

- LO 1.2.2: . . . create a computational artifact using computing tools and techniques to solve a problem.
- LO 1.2.3: . . . create a new computational artifact by combining or modifying existing artifacts.
- LO 1.2.5: . . . analyze the correctness, usability, functionality, and suitability of computational artifacts.
- LO 1.3.1: . . . use programming as a creative tool.
- In *Big Idea II: Abstraction*:
- LO 2.1.1: . . . describe the variety of abstractions used to represent data.
- LO 2.1.2: . . . explain how binary sequences are used to represent digital data.
- LO 2.2.2: . . . use multiple levels of abstraction in computation.
- LO 2.2.3: . . . identify multiple levels of abstractions being used when writing programs.
- In *Big Idea III: Data and information*:
- LO 3.1.1: . . . use computers to process information, find patterns, and test hypotheses about digitally processed information to gain insight and knowledge.
- LO 3.2.1: . . . extract information from data to discover and explain connections, patterns, or trends.
- LO 3.2.2: . . . use large data sets to explore and discover information and knowledge.
- LO 3.3.1: . . . analyze how data representation, storage, security, and transmission of data involve computational manipulation of information.
- In *Big Idea IV: Algorithms*:
- LO 4.1.1: . . . develop an algorithm designed to be implemented to run on a computer.
- LO 4.1.2: . . . express an algorithm in a language.
- LO 4.2.1: . . . explain the difference between algorithms that run in a reasonable time and those that do not run in a reasonable time.
- LO 4.2.2: . . . explain the difference between solvable and unsolvable problems in computer science.
- LO 4.2.4: . . . evaluate algorithms analytically and empirically for efficiency, correctness, and clarity.
- In *Big Idea V: Programming*:
- LO 5.1.1: . . . develop a program for creative expression, to satisfy personal curiosity or to create new knowledge.
- LO 5.1.2: . . . develop a correct program to solve problems.
- LO 5.2.1: . . . explain how programs implement algorithms.
- LO 5.3.1: . . . use abstraction to manage complexity in programs.

- LO 5.5.1: . . . employ appropriate mathematical and logical concepts in programming.
- In *Big Idea VI: The Internet*:
- LO 6.1.1: . . . explain the abstractions in the Internet and how the Internet functions.

CHANGES IN THE FOURTH EDITION

1. We fixed lots of bugs that our crack bug-finders identified in the third edition.
2. We changed most of the pictures in the book – they were getting stale, and our kids wanted us to not use as many pictures of them.
3. We added more end-of-chapter questions.
4. We added a whole new chapter, on text as a medium and manipulating strings (to make sentences, koans, and codes). This isn't a *necessary* chapter (e.g., we introduce `for` and `if` statements, but we didn't remove the introductions later in the book). For some of our teachers, playing with text with shorter loops (iterating over all the characters in a sentence is typically smaller than the thousands of pixels in a picture) is a more comfortable way to start.
5. We gave up fighting the battle of inventing a Web scraper that could beat out the changes that Facebook made, which kept breaking the one we put in the 3rd edition and then kept updating on the teacher's website[2]. Instead, we wrote examples in this book for processing CSV (Comma-Separated Values), a common format for sharing data on the Internet. We parse the CSV from a file using string processing, then using the CSV library in Python, and then accessing the data by URL.
6. We added some new edge detection code which is shorter and simpler to understand.
7. We added more with turtles: creating dancing turtles (using `sleep` from the `time` module to pause execution) and recursive patterns.
8. We updated the book to use the latest features in JES, which include those that reduce the need to use full pathnames (a problem identified by Stephen Edwards and his students in their SIGCSE 2014 paper [43]).

ACKNOWLEDGMENTS

Our sincere thanks go out to all our reviewers and bug-finders:

- At the top of the list is Susan Schwarz of the US Military Academy at West Point. Susan runs a large course with many instructors, and pays careful attention to what's going in all of the sections of the course. She turned that attention on the third edition of this book. She caught many bugs, and gave us lots of useful feedback. Thanks, Susan!

---

[2] `http://home.cc.gatech.edu/mediaComp` and `http://www.mediacomputation.org`

- Our other bug finders for the book were John Rutkiewicz, U. Massachusetts–Dartmouth; Brian Dorn, U. Nebraska–Omaha; Dave Largent, Ball State University; Simon, University of Newcastle; Eva Heinrich, Massey University; Peter J. DePasquale, The College of New Jersey, and Bill Leahy, Georgia Institute of Technology.
- Matthew Frazier, North Carolina State University, worked with us in the summer of 2014 to create a new version of JES – fixing many bugs, and improving JES considerably.
- We are grateful for the feedback from our book reviewers for the 4th edition: Andrew Cencini, Bennington College; Susan Fox, Macalester College; Kristin Lamberty, University of Minnesota-Morris; Jean Smith, Technical College of the Lowcountry; and William T. Verts, University of Massachusetts-Amherst.
- We are grateful for the input from our book reviewers for the 3rd edition, too: Joseph Oldham, Centre College; Lukasz Ziarek, Purdue University;Joseph O'Rourke, Smith College; Atul Prakash, University of Michigan; Noah D. Barnette, Virginia Tech; Adelaida A. Medlock, Drexel University; Susan E. Fox, Macalester College; Daniel G. Brown, University of Waterloo; Brian A. Malloy, Clemson University; Renee Renner, California State University, Chico.

Mark Guzdial and Barbara Ericson
Georgia Institute of Technology

Preface to the First Edition

Research in computing education makes it clear that one doesn't just "learn to program." One learns to program *something* [8, 19], and the motivation to do that something can make the difference between learning and not learning to program [5]. The challenge for any teacher is to pick a *something* that is a powerful enough motivator.

People want to communicate. We are social creatures and the desire to communicate is one of our primal motivations. Increasingly, the computer is used as a tool for communication even more than a tool for calculation. Virtually all published text, images, sounds, music, and movies today are prepared using computing technology.

This book is about teaching people to program in order to communicate with digital media. The book focuses on how to manipulate images, sounds, text, and movies as professionals might, but with programs written by students. We know that most people will use professional-grade applications to perform these type of manipulations. But, knowing *how* to write your own programs means that you *can* do more than what your current application allows you to do. Your power of expression is not limited by your application software.

It may also be true that knowing how the algorithms in a media applications work allows you to use them better or to move from one application to the next more easily. If your focus in an application is on what menu item does what, every application is different. But if your focus is on moving or coloring the pixels in the way you want, then maybe it's easier to get past the menu items and focus on what you want to say.

This book is not just about programming in media. Media-manipulation programs can be hard to write or may behave in unexpected ways. Natural questions arise, like "Why is the same image filter faster in Photoshop?" and "That was hard to debug—Are there ways of writing programs that are *easier* to debug?" Answering questions like these is what computer scientists do. There are several chapters at the end of the book that are about *computing*, not just programming. The final chapters go beyond media manipulation to more general topics.

The computer is the most amazingly creative device that humans have ever conceived. It is completely made up of mind-stuff. The notion "Don't just dream it, be it" is really possible on a computer. If you can imagine it, you can make it "real" on the computer. Playing with programming can be and *should* be enormous fun.

OBJECTIVES, APPROACH AND ORGANIZATION

The curricular content of this book meets the requirements of the "imperative-first" approach described in the ACM/IEEE *Computing Curriculum 2001* standards document [2]. The book starts with a focus on fundamental programming constructs: assignments, sequential operations, iteration, conditionals, and defining functions. Abstractions (e.g., algorithmic complexity, program efficiency, computer organization, hierarchical decomposition, recursion, and object-oriented programming) are emphasized later, after the students have a context for understanding them.

This unusual ordering is based on the findings of research in the learning sciences. Memory is associative. We remember new things based on what we associate them with. People can learn concepts and skills on the premise that they will be useful some day but the concepts and skills will be related only to the premises. The result has been described as "brittle knowledge" [25]—the kind of knowledge that gets you through the exam but is promptly forgotten because it doesn't relate to anything but being in that class.

Concepts and skills are best remembered if they can be related to many different ideas or to ideas that come up in one's everyday life. If we want students to gain *transferable* knowledge (knowledge that can be applied in new situations), we have to help them to relate new knowledge to more general problems, so that the memories get indexed in ways that associate with those kinds of problems [22]. In this book, we teach with concrete experiences that students can explore and relate to (e.g., conditionals for removing red-eye in pictures) and later lay abstractions on top of them (e.g., achieving the same goal using recursion or functional filters and maps).

We know that starting from the abstractions doesn't really work for computing students. Ann Fleury has shown that students in introductory computing courses just don't buy what we tell them about encapsulation and reuse (e.g., [7]). Students prefer simpler code that they can trace easily and they actually think that such code is *better*. It takes time and experience for students to realize that there is value in well-designed systems. Without experience, it's very difficult for students to learn the abstractions.

The **media computation** approach used in this book starts from what many people use computers for: image manipulation, exploring digital music, viewing and creating Web pages, and making videos. We then explain programming and computing in terms of these activities. We want students to visit Amazon (for example) and think, "Here's a catalog Web site—and I know that these are implemented with a database and a set of programs that format the database entries as Web pages." We want students to use Adobe Photoshop and GIMP and think about how their image filters are actually manipulating red, green, and blue components of pixels. Starting from a relevant context makes transfer of knowledge and skills more likely. It also makes the examples more interesting and motivating, which helps with keeping students in the class.

The media computation approach spends about two-thirds of the time on giving students experiences with a variety of media in contexts that they find motivating. After that two-thirds, though, they naturally start to ask questions about *computing*. "Why is it that Photoshop is faster than my program?" and "Movie code is slow—How slow do programs get?" are typical. At that point, we introduce the abstractions and the valuable insights from computer science that answer *their* questions. That's what the last part of this book is about.

A different body of research in computing education explores why withdrawal or failure rates in introductory computing are so high. One common theme is that computing courses seem "irrelevant" and unnecessarily focus on "tedious details" such as efficiency [21, 1]. A communications context is perceived as relevant by students (as they tell us in surveys and interviews [6, 18]). The relevant context is part of the explanation for the success we have had with retention in the Georgia Tech course for which this book was written.

The late entrance of abstraction isn't the only unusual ordering in this approach. We start using arrays and matrices in Chapter 3, in our first significant programs. Typically, introductory computing courses push arrays off until later, because they are obviously more complicated than variables with simple values. A relevant and concrete context is very powerful [19]. We find that students have no problem manipulating matrices of pixels in a picture.

The rate of students withdrawing from introductory computing courses or receiving a D or F grade (commonly called the *WDF rate*) is reported in the 30–50% range or even higher. A recent international survey of failure rates in introductory computing courses reported that the average failure rate among 54 U.S. institutions was 33% and among 17 international institutions was 17% [24]. At Georgia Tech, from 2000 to 2002, we had an average WDF rate of 28% in the introductory course required for all majors. We used the first edition of this text in our course *Introduction to Media Computation*. Our first pilot offering of the course had 121 students, no computing or engineering majors, and two-thirds of the students were female. Our WDF rate was 11.5%.

Over the next two years (Spring 2003 to Fall 2005), the average WDF rate at Georgia Tech (across multiple instructors, and literally thousands of students) was 15% [29]. Actually, the 28% prior WDF rate and 15% current WDF rate are incomparable, since all majors took the first course and only liberal arts, architecture, and management majors took the new course. Individual majors have much more dramatic changes. Management majors, for example, had a 51.5% WDF rate from 1999 to 2003 with the earlier course, and had a 11.2% failure rate in the first two years of the new course [29]. Since the first edition of this book was published, several other schools have adopted and adapted this approach and evaluated their result. All of them have reported similar, dramatic improvements in success rates [4, 42].

Ways to Use This Book

This book represents what we teach at Georgia Tech in pretty much the same order. Individual teachers may skip some sections (e.g., the section on additive synthesis, MIDI, and MP3), but all of the content here has been tested with our students.

However, this material has been used in many other ways.

- A short introduction to computing could be taught with just Chapters 2 (introduction to programming) and 3 (introduction to image processing), perhaps with some material from Chapters 4 and 5. We have taught even single-day workshops on media computation using just this material.
- Chapters 6 through 8 basically replicate the computer science concepts from Chapters 3 through 5 but in the context of sounds rather than images. We find the replication useful—some students seem to relate better to the concepts of iteration and conditionals when working with one medium than with the other.

Further, it gives us the opportunity to point out that the same **algorithm** can have similar effects in different media (e.g., scaling a picture up or down and shifting a sound higher or lower in pitch are the same algorithm). But it could certainly be skipped to save time.

- Chapter 12 (on movies) introduces no new programming or computing concepts. While motivational, movie processing could be skipped to save time.
- We recommend getting to at least some of the chapters in the last unit, in order to lead students into thinking about computing and programming in a more abstract manner, but clearly not *all* of the chapters have to be covered.

Python and Jython

The programming language used in this book is Python. Python has been described as "executable pseudo-code." We have found that both computer science majors and non majors can learn Python. Since Python is actually used for communications tasks (e.g., Web site development), it's a relevant language for an introductory computing course. For example, job advertisements posted to the Python Web site (`http://www.python.org`) show that companies like Google and Industrial Light & Magic hire Python programmers.

The specific dialect of Python used in this book is *Jython* (`http://www.jython.org`). Jython *is* Python. The differences between Python (normally implemented in C) and Jython (which is implemented in Java) are akin to the differences between any two language implementations (e.g., Microsoft vs. GNU C++ implementations)—the basic language is *exactly* the same, with some library and details differences that most students will never notice.

TYPOGRAPHICAL NOTATIONS

Examples of Python code look like this: `x = x + 1`. Longer examples look like this:

```
def helloWorld():
  print "Hello, world!"
```

When showing something that the user types in with Python's response, it will have a similar font and style, but the user's typing will appear after a Python prompt (>>>):

```
>>> print 3 + 4
7
```

User interface components of JES (Jython Environment for Students) will be specified using a small caps font, like Save menu item and the Load button.

There are several special kinds of sidebars that you'll find in the book.

Computer Science Idea: An Example Idea

Key computer science concepts appear like this.

Common Bug: An Example Common Bug

Common things that can cause your program to fail appear like this. ■

Debugging Tip: An Example Debugging Tip

If there's a good way to keep a bug from creeping into your programs in the first place, it's highlighted here. ■

Making It Work Tip: An Example How to Make It Work

Best practices or techniques that really help are highlighted like this. ■

INSTRUCTOR RESOURCES

The instructor resources are available on the Pearson Education's Instructor Resource Center at `www.pearsonglobaleditions.com/guzdial`:

- PowerPoint® Presentation slides

ACKNOWLEDGMENTS

Our sincere thanks go out to the following:

- Jason Ergle, Claire Bailey, David Raines, and Joshua Sklare, who made the initial version of JES with surprising quality in an amazingly short amount of time. Over the years, Adam Wilson, Larry Olson, Yu Cheung (Toby) Ho, Eric Mickley, Keith McDermott, Ellie Harmon, Timmy Douglas, Alex Rudnick, Brian O'Neill, and William Fredrick (Buck) Scharfnorth III have made JES into the useful and still understandable tool that it is today.
- Adam Wilson built the MediaTools that are so useful for exploring sounds and images and processing video.
- Andrea Forte, Mark Richman, Matt Wallace, Alisa Bandlow, Derek Chambless, Larry Olson, and David Rennie helped build course materials. Derek, Mark, and Matt created many example programs.
- There were several people who really made the effort come together at Georgia Tech. Bob McMath, Vice-Provost at Georgia Tech, and Jim Foley, Associate Dean for Education in the College of Computing, invested in this effort early on. Kurt Eiselt worked hard to make this effort real, convincing others to take it seriously. Janet Kolodner and Aaron Bobick were excited and encouraging about the idea of media computation for students new to computer science. Jeff Pierce reviewed

and advised us on the design of the media functions used in the book. Aaron Lanterman gave me lots of advice on how to convey the digital material content accurately. Joan Morton, Chrissy Hendricks, David White, and all the staff of the GVU Center made sure that we had what we needed and that the details were handled to make this effort come together. Amy Bruckman and Eugene Guzdial bought Mark time to get the final version completed.

- We are grateful to Colin Potts and Monica Sweat who have taught this class at Georgia Tech and given us many insights about the course.
- Charles Fowler was the first person outside of Georgia Tech willing to take the gamble and trial the course in his own institution (Gainesville College), for which we're very grateful.
- The pilot course offered in Spring 2003 at Georgia Tech was very important in helping us improve the course. Andrea Forte, Rachel Fithian, and Lauren Rich did the assessment of the pilot offering of the course, which was incredibly valuable in helping us understand what worked and what didn't. The first teaching assistants (Jim Gruen, Angela Liang, Larry Olson, Matt Wallace, Adam Wilson, and Jose Zagal) did a lot to help create this approach. Blair MacIntyre, Colin Potts, and Monica Sweat helped make the materials easier for others to adopt. Jochen Rick made the CoWeb/Swiki a great place for CS1315 students to hang out.
- Many students pointed out errors and made suggestions to improve the book. Thanks to Catherine Billiris, Jennifer Blake, Karin Bowman, Maryam Doroudi, Suzannah Gill, Baillie Homire, Jonathan Laing, Mireille Murad, Michael Shaw, Summar Shoaib, and especially Jonathan Longhitano, who has a real flair for copyediting.
- Thanks to former *Media Computation* students Constantino Kombosch, Joseph Clark, and Shannon Joiner for permission to use their snapshots from class in examples.
- The research work that led to this text was supported by grants from the National Science Foundation—from the Division of Undergraduate Education, CCLI program, and from the CISE Educational Innovations program. Thank you for the support.
- Thanks to computing students Anthony Thomas, Celines Rivera, and Carolina Gomez for allowing us to use their pictures.
- Finally but most important, thanks to our children Matthew, Katherine, and Jennifer Guzdial, who allowed themselves to be photographed and recorded for Mommy and Daddy's media project and who were supportive and excited about the class.

MARK GUZDIAL AND BARBARA ERICSON
Georgia Institute of Technology

Pearson would like to thank and acknowledge Dheeraj D., REVA Institute of Technology and Management Kattigenahalli, Somitra Sanadhya, Indian Institute of Technology Delhi, and Simon, University of Newcastle, for contributing to the Global Edition, and Shaligram Prajapat, Devi Ahilya University Indore, Ng Hu, Multimedia University, Rohit Tahiliani, Renjith S., and Shivani Pandit, for reviewing the Global Edition.

About the Authors

Mark Guzdial is a professor in the School of Interactive Computing in the College of Computing at Georgia Institute of Technology. He is one of the founders of the ACM's International Computing Education Research workshop series. Dr. Guzdial's research focuses on learning sciences and technology, specifically, computing education research. His first books were on the programming language Squeak and its use in education. He was the original developer of "Swiki" (Squeak Wiki), the first wiki developed explicitly for use in schools. He is a Fellow and a Distinguished Educator of the ACM. He is on the editorial boards of the *Journal of the Learning Sciences* and *Communications of the ACM*. He was a recipient of the 2012 IEEE Computer Society Undergraduate Teaching Award.

Barbara Ericson is a research scientist and the director of Computing Outreach for the College of Computing at Georgia Tech. She has been working on improving introductory computing education since 2004.

She has served as the teacher education representative on the Computer Science Teachers Association board, the co-chair of the K-12 Alliance for the National Center for Women in Information Technology, and as a reader for the Advanced Placement Computer Science exams. She enjoys the diversity of the types of problems she has worked on over the years in computing including computer graphics, artificial intelligence, medicine, and object-oriented programming.

Mark and Barbara received the 2010 ACM Karl V. Karlstrom Award for Outstanding Computer Educator for their work on Media Computation including this book. They led a project called "*Georgia Computes!*" for six years, which had a significant impact in improving computing education in the US state of Georgia [31]. Together, they Mark and Barbara are leaders in the *Expanding Computing Education Pathways* (ECEP) alliance[3]

---

[3] `http://www.ecepalliance.org`

PART 1

INTRODUCTION

CHAPTER

1

Introduction to Computer Science and Media Computation

Chapter Learning Objectives

- To explain what computer science is about and what computer scientists are concerned with.
- To explain why we digitize media.
- To explain why it's valuable to study computing.
- To explain the concept of an **encoding**.
- To explain the basic components of a computer.

1.1 WHAT IS COMPUTER SCIENCE ABOUT?

Computer science is the study of **process**: how we or computers do things, how we specify what we do, and how we specify what the stuff is that we're processing. That's a pretty dry definition. Let's try a metaphorical one.

Computer Science Idea: Computer Science Is the Study of Recipes

''Recipes'' here are a special kind—one that can be executed by a computational device, but this point is only of importance to computer scientists. The important point overall is that a computer science recipe defines *exactly* what has to be done.

More formally, computer scientists study *algorithms* which are step-by-step procedures to accomplish a task. Each step in an algorithm is something that a computer already knows how to do (e.g., add two small integer numbers) or can be taught how to do (e.g., adding larger numbers including those with a decimal point). A recipe that can run on a computer is called a *program*. A program is a way to communicate an algorithm in a representation that a computer can execute.

To use our metaphor a bit more—think of an algorithm as the step-by-step way that your grandmother made her secret recipe. She always did it the same way, and had a reliably great result. Writing it down so that you can read it and do it later is like turning her algorithm into a program for you. You *execute* the recipe by *doing* it—following the recipe step-by-step in order to create something the way that your grandmother did. If you give the recipe to someone else who can read the language of the recipe (maybe English or French), then you have communicated that process to that other person, and the other person can similarly execute the recipe to make something the way that your grandmother did.

If you're a biologist who wants to describe how migration works or how DNA replicates, then being able to write a recipe that specifies *exactly* what happens, in terms that can be completely defined and understood, is *very* useful. The same is true if you're a chemist who wants to explain how equilibrium is reached in a reaction. A factory manager can define a machine-and-belt layout and even test how it works—before physically moving heavy things into position—using computer **programs**. Being able to exactly define tasks and/or simulate events is a major reason why computers have radically changed so much of how science is done and understood.

In fact, if you *can't* write a recipe for some process, maybe you don't really understand the process, or maybe the process can't actually work the way that you are thinking about it. Sometimes, trying to write the recipe is a test in itself. Now, sometimes you can't write the recipe because the process is one of the few that cannot be executed by a computer. We will talk more about those in Chapter 14.

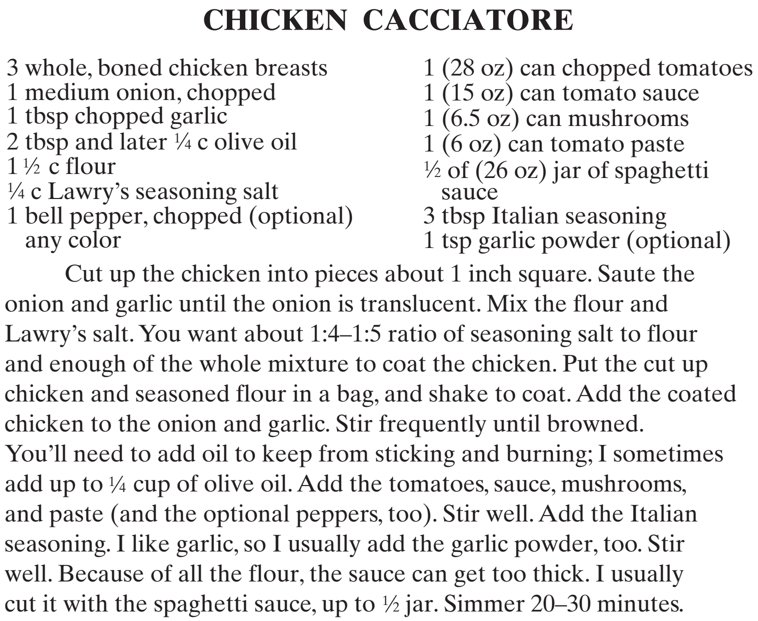
It may sound funny to call *programs* a recipe, but the analogy goes a long way. Much of what computer scientists study can be defined in terms of recipes.

- Some computer scientists study how recipes are written: Are there better or worse ways of doing something? If you've ever had to separate egg whites from yolks, you realize that knowing the right way to do it makes a world of difference. Computer science theoreticians think about the fastest and shortest recipes, and the ones that take up the least amount of space (you can think about it as counter space—the analogy works), or even use the least amount of energy (which is important when running on low-power devices like cell phones). *How* a recipe works, completely apart from how it's written (e.g., in a program), is called the study of algorithms. Software engineers think about how large groups can put together recipes that still work. (Some programs, like the ones that keep track of credit card transactions, have literally millions of steps!) The term **software** means a collection of computer programs (recipes) that accomplish a task.
- Other computer scientists study the units used in recipes. Does it matter whether a recipe uses metric or English measurements? The recipe may work in either case, but if you don't know what a pound or a cup is, the recipe is a lot less understandable to you. There are also units that make sense for some tasks and not others, but if you can fit the units to the tasks, you can explain yourself more easily and get things done faster—and avoid errors. Ever wonder why ships at sea measure their speed in *knots*? Why not use something like meters per second? Sometimes, in certain special situations—on a ship at sea, for instance—the more common terms aren't appropriate or don't work as well. Or we may invent new kinds of units, like a unit that represents a whole other program or a computer, or

a network like your friends and your friends' friends in Facebook. The study of computer science units is referred to as **data structures**. Computer scientists who study ways of keeping track of lots of data (in lots of different kinds of units) and figuring out how to access the data quickly are studying **databases**.

- Can recipes be written for anything? Are there some recipes that *can't* be written? Computer scientists know that there are recipes that can't be written. For example, you can't write a recipe that can absolutely tell whether some other recipe will actually work. How about *intelligence*? Can we write a recipe such that a computer following it would actually be *thinking* (and how would you tell if you got it right)? Computer scientists in **theory**, **intelligent systems**, **artificial intelligence**, and **systems** worry about things like this.
- There are even computer scientists who focus on whether people *like* what the recipes produce, almost like restaurant critics for a newspaper. Some of these are **human–computer interface** specialists who worry about whether people can understand and make use of the recipes ("recipes" that produce an *interface* that people use, like windows, buttons, scrollbars, and other elements of what we think about as a running program).
- Just as some chefs specialize in certain kinds of recipes, like crepes or barbecue, computer scientists also specialize in certain kinds of recipes. Computer scientists who work in *graphics* are mostly concerned with recipes that produce pictures, animations, and even movies. Computer scientists who work in *computer music* are mostly concerned with recipes that produce sounds (often melodic ones, but not always).
- Still other computer scientists study the *emergent properties* of recipes. Think about the World Wide Web. It's really a collection of *millions* of recipes (programs) talking to one another. Why would one section of the Web get slower at some point? It's a phenomenon that emerges from these millions of programs, certainly not something that was planned. That's something that **networking** computer scientists study. What's really amazing is that these emergent properties (that things just start to happen when you have many, many recipes interacting at once) can also be used to explain noncomputational things. For example, how ants forage for food or how termites make mounds can also be described as something that just happens when you have lots of little programs doing something simple and interacting. There are computer scientists today who study how the Web allows for new kinds of interactions, particularly in large groups (like Facebook or Twitter). Computer scientists who study *social computing* are interested in how these new kinds of interactions work and the characteristics of the software that are most successful for promoting useful social interactions.

The recipe metaphor also works on another level. Everyone knows that some things in a recipe can be changed without changing the result dramatically. You can always increase all the units by a multiplier (say, double) to make more. You can always add more garlic or oregano to the spaghetti sauce. But there are some things that you cannot change in a recipe. If the recipe calls for baking powder, you may not substitute baking



CHICKEN CACCIATORE

3 whole, boned chicken breasts
1 medium onion, chopped
1 tbsp chopped garlic
2 tbsp and later ¼ c olive oil
1 ½ c flour
¼ c Lawry's seasoning salt
1 bell pepper, chopped (optional) any color

1 (28 oz) can chopped tomatoes
1 (15 oz) can tomato sauce
1 (6.5 oz) can mushrooms
1 (6 oz) can tomato paste
½ of (26 oz) jar of spaghetti sauce
3 tbsp Italian seasoning
1 tsp garlic powder (optional)

Cut up the chicken into pieces about 1 inch square. Saute the onion and garlic until the onion is translucent. Mix the flour and Lawry's salt. You want about 1:4–1:5 ratio of seasoning salt to flour and enough of the whole mixture to coat the chicken. Put the cut up chicken and seasoned flour in a bag, and shake to coat. Add the coated chicken to the onion and garlic. Stir frequently until browned. You'll need to add oil to keep from sticking and burning; I sometimes add up to ¼ cup of olive oil. Add the tomatoes, sauce, mushrooms, and paste (and the optional peppers, too). Stir well. Add the Italian seasoning. I like garlic, so I usually add the garlic powder, too. Stir well. Because of all the flour, the sauce can get too thick. I usually cut it with the spaghetti sauce, up to ½ jar. Simmer 20–30 minutes.

FIGURE 1.1
A cooking recipe—you can always double the ingredients, but throwing in an extra cup of flour won't cut it, and don't try to brown the chicken *after* adding the tomato sauce!

soda. The order matters. If you're supposed to brown the chicken and then add tomato sauce, you won't get the same result if you add tomato sauce and then (somehow) try to brown the chicken (Figure 1.1).

The same holds for software recipes. There are usually things you can easily change: the actual names of things (though you should change names consistently), some of the **constants** (numbers that appear as plain old numbers, not as variables), and maybe even some of the data **ranges** (sections of the data) being manipulated. But the order of the commands to the computer, however, almost always has to stay exactly as stated. As we go on, you'll learn what can be safely changed, and what can't.

1.2 PROGRAMMING LANGUAGES

Computer scientists write a recipe in a **programming language** (Figure 1.2). Different programming languages are used for different purposes. Some of them are wildly popular, like Java and C++. Others are more obscure, like Squeak and Scala. Some others are designed to make computer science ideas very easy to learn, like Scheme or Python, but the fact that they're easy to learn doesn't always make them very popular or the best choice for experts building larger or more complicated recipes. It's a hard balance in teaching computer science to pick a language that is easy to learn *and* is popular and useful enough to experts that students are motivated to learn it.

Why don't computer scientists just use natural human languages, like English or Spanish? The problem is that natural languages evolved the way they did to enhance