EXPLORATORY PROGRAMMING FOR THE ARTS AND HUMANITIES

SECOND EDITION

NICK MONTFORT

The open access edition of this work was made possible by generous funding from the MIT Libraries.

This book was set in ITC Stone Serif Std and ITC Stone Sans Std by New Best-set Typesetters Ltd.

Library of Congress Cataloging-in-Publication Data

Names: Montfort, Nick, author.
Title: Exploratory programming for the arts and humanities / Nick Montfort.
Description: Second edition. | Cambridge, Massachusetts : The MIT Press, [2021] | Includes bibliographical references and index. | Summary: "Exploratory Programming for the Arts and Humanities offers a course on programming without prerequisites. It covers both the essentials of computing and the main areas in which computing applies to the arts and humanities"—Provided by publisher.
Identifiers: LCCN 2019059151 | ISBN 9780262044608 (hardcover)
Subjects: LCSH: Computer programming. | Digital humanities.
Classification: LCC QA76.6 .M664 2021 | DDC 005.1—dc23

LC record available at https://lccn.loc.gov/2019059151

10 9 8 7 6 5 4 3 2 1

[Contents]

[List of Figures]

[Acknowledgments]

First, my thanks go to everyone who has taught and encouraged exploratory programming, from at least the 1960s through today.

Many people discussed the concept of this book with me at different stages of the project. Some of these informal conversations were quite important to the final direction I took with the text—for instance, I learned that senior artists and researchers were interested in learning programming using a book of this sort. In response, I developed a book that could be used in a class or by independent learners and tried to improve both of these aspects of the book in the second edition.

In developing the first edition, I greatly appreciate the opportunity to do daylong and multiday workshops on exploratory programming in New York (at NYU), in Mexico City (at UAM-Cuajimalpa), and in the Boston area (at MIT). I also had the opportunity to teach an undergraduate/graduate course based directly on a late draft of the book in New York, at the New School, thanks to Anne Balsamo. My New School students were a great help to me as I worked to complete the first edition. I've benefited from many experiences teaching programming, and also wish to thank my students in semester-long courses at MIT, the University of Pennsylvania, and the University of Baltimore. My thanks go to those at the New York City gallery Babycastles, where a good bit of work on this book was originally done, and particularly to those who helped out there by reading and commenting on parts of the manuscript as I completed it: Del, Emi, Frank, Justin, Lauren, Lee, Nitzan, Patrick, Stephanie, and Todd. As I was in the last stages of work on the first edition, I was able to teach a two-day course based on some of it for the School for Poetic Computation in New York City. I thank my students and those who ran the school's summer session for this opportunity. I have been fortunate to learn about programming and computing throughout my life in many contexts; an important one was at Penn, where I was particularly helped by Michael Kearns, Mitch Marcus, and the researchers at the Institute for Research in Cognitive Science.

My thanks go to several who reviewed the full text of this book in both of its editions. Erik Stayton went through the full first edition manuscript, commenting on it and correcting it, and also completed all the exercises. Patsy Baudoin provided detailed comments on a full draft of that manuscript. The second edition benefited from the close attention of Judy Heflin, who reviewed the text closely and also served as my teaching assistant when I used a draft of this edition. Todd Anderson did a technical edit of the book to help winnow out many mistakes that remained. My inestimable spouse, Flourish Klink, read and commented on the book and supported me in very many other ways as I worked on this project.

As I developed a draft of the second edition and began to refine it, I got to lead a months-long discussion about the draft book, and an in-person workshop based on it, with members of the Society for Spoken Art, and I greatly appreciate the opportunity to refine the book in this creative context. Thank you all, ITs of Full Circle. Thanks, too, to my students at MIT who were the first to use a full draft of the second edition in a regular classroom course: Andrea, Casey, Janina, JJ, and Meng Fu.

The conventional wisdom about a new edition of a textbook seems to be that it's simply a ploy to milk more money out of students. It would be hard to accuse anyone of that this time around, because the second edition will be available both in a print format (which I believe is extremely well-suited to learning, and worth buying or borrowing from a library) and in a free, digital and screen-based open-access format. That's due to the generosity of the MIT Libraries, which offered the financial support needed to make this edition freely available. Beyond that, the conventional wisdom is quite off the mark, as anyone who has worked on a substantial textbook revision like this one will know! I would like to thank everyone who reported errors in the first edition. I also want to give special thanks to two groups of people whose use of the first edition helped me get to this point. One is a group of faculty and staff at Trinity College in Hartford, CT, who undertook extensive study of the book; thanks to an invitation from Jason B. Jones, I was able to talk with the group and hear presentations about where their studies had led them. The other people who helped immensely were instructors who had used the first edition and could report how it worked (or didn't) in their particular class contexts. I particularly want to thank Angela Chang and Zach Whalen for extensive discussions of their teaching experiences.

The MIT Press of course arranged for anonymous reviewers to consider both the original proposal and the manuscript closely; I am grateful to these reviewers for their support of the project and for their valuable comments and suggestions. At the MIT Press, I also particularly would like to thank Doug Sery, who has discussed, worked

on, and supported my book projects for more than a decade and a half now. I can't imagine having explored as many issues in digital media and creative computing, in the same breath and depth, without his backing over the years. I also appreciate the work MIT Press editor Kathy Caruso (who worked on three previous books of mine) did to improve the manuscript and ready the book for publication, in both editions. Any errors in the published text are of course my responsibility.

The text of appendix A, along with a few paragraphs from the introduction, was also published in *A New Companion to Digital Humanities*, edited by Susan Schreibman, Ray Siemens, and John Unsworth, 98–109 (New York: John Wiley & Sons, 2016), as "Exploratory Programming in Digital Humanities Pedagogy and Research."

[1] Introduction

[1.1]

This is a book about how to think with computation and how to understand computation as part of culture. Programming is introduced as a way to iteratively design both artworks and humanities projects, in a process that enables the programmer to discover, through the process of programming, the ultimate direction that the project will take. My aim is to explain enough about programming to allow a new programmer to explore and inquire with computation, and to help such a person go on to learn more about programming while pursuing projects in the arts and humanities. That is, when someone finishes going through this book, typing in the code as requested and doing the exercises and projects as requested, that person will be a programmer—not particularly an expert, but a person with the ability to use computation in general ways to explore the arts and humanities. The person who completes the work in this book may be a beginning programmer but will be equipped to explore areas of intellectual interest and will be ready to learn more about programming as it becomes necessary.

This book is mainly addressed to people without a programming background—particularly to both individual, self-directed learners and to graduate students in the arts and humanities. I developed this book in part for use in university courses, as a textbook. I also put a lot of effort into developing a book that will be useful outside a standard classroom and course. I provide suggestions for teaching this book, and for learning from it in a class, in appendix B, "Contexts for Learning," which also includes some suggestions for self-directed students.

To some, *programming* is associated with expertise, professional status, and esoteric technical difficulty. I don't think the term *programming* needs to be intimidating, any more than the terms *writing* or *sketching* do. These are simply the conventional words for different activities—creative activities that are also methods for inquiry.

You don't need any background in programming to learn from this book, and courses based on this book do not need to require any such background. If you are already comfortable programming, you may still benefit from using *Exploratory Programming for the Arts and Humanities*, particularly if your work as a programmer has been instrumental: that is, you have mainly worked to implement specifications and solve specific problems rather than using programming to explore. I have assumed, though, that a reader has no previous programming experience.

In my approach to programming, I seek to show its exploratory potential—its facility for sketching, brainstorming, and inquiring about important topics. My discussions, exercises, and explanations of how programming works also include some significant consideration of how computation and programming are culturally situated. This means how programming relates to the manipulation of media of different sorts and, more broadly, to the methods and questions of the arts and humanities.

WARNING! Do not read past this chapter without having your computer next to the book, if you are lucky enough to have the printed version of this book. If you are using the open access digital edition, read it, if at all possible, on a separate device that you place next to your computer.

This book was written and designed to be read alongside a computer, allowing the reader to program while progressing through the book. The book is really meant to be part of a human-book-computer system, one that is set up to help the human learn. After reading this introductory material, I consider it essential to use this book while programming.

I ask at times that readers follow along and simply type code in directly from the book, in part to gain familiarity and comfort with practical aspects of inputting and running programs. At other times, readers are asked to try modifying existing programs, sometimes in minor ways, sometimes in more significant ways. Some carefully chosen exercises, initially simpler and then more substantial, are provided. In some of these cases, a specification is given for a program, describing how it is supposed to work. Although doing such exercises is not an exploratory way to program, they are included to provide a wider range of programming experiences and foster familiarity with code and computing. Particularly when learning the fundamentals of programming, such exercises are important. For these reasons, these exercises are not evenly distributed throughout the book; many of them are provided when fundamentals are introduced and shortly thereafter.

Finally, throughout the book, "free projects" are described. This term is one I made up, meant to express that these aren't the standard sort of exercises or problems. These are intentionally underspecified projects. The idea is that each free project leaves room for learners to determine their own directions and to write different sorts of programs.

You are free to set your own goals and directions, within a general framework. Most of these projects should be done several times, as I indicate. In these projects, the final program is to be arrived at not simply by implementing a fixed specification, but by undertaking some amount of exploration through programming. Because of the unusual way I ask that learners use this book, I go through the four main methods of using the book in detail at the end of this chapter, in 1.8, Programming and Exploring Together.

In some books and courses on programming, readers learn about different sorting algorithms and about how these algorithms have different complexities in space and in time. These are fine topics, and necessary when building a deep foundation for those who will go on to understand the science of computation very thoroughly. If you know already that you are seeking understanding and skills equivalent to a bachelor's degree in computer science or that you actually wish to pursue such a degree, you should probably find a more appropriate book or take a course that covers that material. Furthermore, if your interest is only in learning how to program, and not in the use of programming for exploration or in making a connection between computation and culture, there are good shorter books worth considering. One example I provided in the first edition is Chris Pine's *Learn to Program,* second edition, which uses Ruby to teach general programming principles. Similarly, if you really don't wish to learn to program, but you are interested in how computation relates to issues in the arts and humanities, there are plenty of books and articles focused on these relationships. One of them is a book I wrote with nine coauthors, *10 PRINT CHR$(205.5+RND(1)); : GOTO 10,* which studies a one-line BASIC program in great depth, considering many of its cultural contexts.

It's fair at this point to be clear about what *Exploratory Programming for the Arts and Humanities* is not: It is not an attempt to have readers completely understand any one particular programming language. It is not meant to show people how to professionally produce products or complete the typical "deliverables" of software engineering. It is, as stated earlier, not meant to provide a complete first course for those who will continue to do significant study of computation itself. This book is certainly compatible with computer science education, but it is an attempt to lay a different foundation, a foundation for artistic and humanistic inquiry with computing.

This is a book about how to think with computation—specifically, how to think about questions in the arts and humanities and how to do so by means of programming. I believe this book will supply a solid enough foundation so that those who read it, and who follow along and do the exercises, will be able to pursue a greater ability in specific programming languages, learn about essential ways of collaborating on

software projects, and understand that computation significantly engages with culture and with intellectual concerns in the arts and humanities. I also hope that working through this book will reveal the potential of programming as a means of inquiry and art-making—at least, that it will reveal *enough* of that potential. As long as it does, readers will likely be eager to continue their programming, and their inquiry, after they are done with the structure I have offered here.

[1.2] Exploration versus Exploitation

Exploration as an activity and an outlook is discussed in many contexts; let's consider a very everyday way in which people might explore.

On a trip to the grocery store, a shopper may not be interested in locating new produce that has just started to be distributed locally or other new foodstuffs that have been introduced. Instead, this shopper may simply *exploit* existing knowledge about what already known food is appropriately nourishing, inexpensive, and pleasing to the taste. Such a shopper would probably be able to remember where such foods are located within the store, allowing a supply of groceries to be quickly gathered. The result is efficient, and I admit, I myself almost always shop in this way. But we should not expect culinary breakthroughs or novelty in diet from a person who only exploits existing knowledge in this way. A shopper who does not wish to look around even within a store, and certainly not beyond that store to a new farmer's market or specialty shop, will be good at continuing past successes (and failures) and will also have a harder time discovering new options. This will particularly be the case if the shopper is not prompted to think of new food choices because of interesting restaurant experiences or by learning about new foods from others.

In organizational behavior, machine learning, and grocery shopping, it is desirable to balance exploration with exploitation. We can imagine a shopper who does nothing but explore—who tries new foods at random but never returns to enjoy a particularly pleasing food again. What is being learned from such exploration? As described, nothing at all is being learned. Not just very little, but nothing: each random selection is completely independent of the previous ones. One day's grocery basket could be improved by remembering some of the best items found so far and selecting those while also continuing to look for new food. Most grocery-seeking individuals balance exploration and exploitation in some way, just as successful companies try to profit from existing, stable lines of business while they also try out new opportunities that might pay off significantly. A robot finding its way around a changing or partially known environment should exploit some known ways to get from place to place while

also devoting some time to exploration, in the hopes that it can find more efficient routes.

It's true that *exploration*, particularly when paired with *exploitation*, has some negative connotations—hinting, for instance, at the history of colonialism. There are some uncomfortable terms and metaphors in computing, as you will know if you have ever read the instructions on how to slave a hard drive to the master. I think some of these terms should be changed, and I find the indifference of some technically oriented individuals to them to be rather disturbing. I don't find that *exploration* packs nearly as much unambiguous historical negativity as *slave*, however. In fact, I am offering it as an alternative to *mastery*, either of the data or of the computer. The point of this book is not to train anyone to be a colonizing power, but to invite people to encounter new ideas and perspectives. The exploration one can do with computing is, to me, more along the lines of how someone from the United States might visit and explore Paris or Mexico City and learn about different architectures, cuisines, organizations of urban spaces, and histories. So perhaps it makes some sense to accuse me of being touristic, but I don't think—whatever the ways in which colonialism is wrapped up with the cities of the world—that my perspective encourages the most negative sense of exploration.

The idea of exploratory programming is not supposed to provide the single solution or "one true way" to approach computing; it's not a suggestion that programmers never develop a system from an existing specification. It's meant, instead, to be one valuable mode in which to think, to encounter computation, and to bring the abilities of the computer to address one's important questions—artistic, cultural, or otherwise.

A problem with programming, as it is typically encountered, is that many people who gain some ability to program hardly learn to explore at all. There are substantial challenges involved in learning how to program and in learning how computing works. If one is interested in fully understanding basic data structures and gaining the type of knowledge that a computer science student needs, it can be hard to discover at the same time, at an early point in one's programming experience, how to use programming as a means of inquiry.

Linked lists and binary trees are essential concepts for those learning the science of computation, but a great deal of exploration can be done without understanding these concepts. Those working in artistic and humanistic areas can apply programming fundamentals to discover how exploration can be done through programming. They can learn how computing allows for abstraction and generalized calculations. They can gain comfort with programming, learn to program effectively, and see how to use programming as a means of inquiry. For those who don't plan on getting a degree in

computer science, it can sometimes be difficult to understand the bigger picture, hard to discern how to usefully work with data and how to gain comfort with programming, while dealing with the more advanced topics that are covered in introductory programming courses. That is, it can be hard to see the forest for the binary trees.

Beyond that, many of those who haven't yet learned about programming get the impression that it is simply a power tool for completing an edifice or a vehicle used to commute from one point to another. While the computer can have impressive results when used in such instrumental ways, it can be used for even more impressive purposes when understood as a sketchpad, sandbox, prototyping kit, telescope, and microscope. As a system for exploration and inquiry, there really isn't anything else with the same capabilities as the computer. Exploratory programming is about using computation in this way, as an artist and humanist.

[1.3] A Justification for Learning to Program?

As I wrote this book, I looked at the beginning of several popular books on learning to play guitar, on learning Spanish, and on introducing artificial intelligence. Interestingly, none of these books seemed to have a section justifying in detail why one should learn to play guitar, or learn Spanish, or understand the basics of artificial intelligence. Sometimes there were a few sentences about how cool it is to play the guitar, or some facts about how many people in the world speak Spanish, or some information about how interesting AI is as a field, but there was nothing like a pep talk, rallying cry, or other exhortation to learn the subject matter of the book. At most, these books just say something like "So you want to learn to play guitar?" or "So you're ready to learn Spanish?"

So—you want to learn to program as a means of creativity and inquiry?

I have found many people interested in learning how to program as a way of exploring computation, texts, language more generally, images, sound, historical and spatial data, and other things. Not everyone feels this way, but when you start looking for such people, you find that a surprising number do. If you are one of these people, this book is for you.

For those who actively fear computers and mathematics, I suggest resolving to overcome this feeling before getting started as a programmer. People who fear reading and conversation and do not believe they are able to learn a foreign language are really not well-suited to begin studying Spanish; they should somehow gain at least a willingness to read and talk before they begin a course of study. People who have convinced themselves that they are inherently nonmusical are probably not in the best

Berners-Lee *didn't* develop the World Wide Web by figuring out all the concepts, applying for a grant, and then dealing with things like programming later. Joseph Weizenbaum *didn't* just design the first chatbot, Eliza, and then get someone else to do the programming later. Douglas Engelbart *didn't* plan all the major advances of his famous system, called oNLine System (NLS)—hypertext, videoconferencing, the mouse, and many others—and then deal with the programming afterwards. Grace Murray Hopper *didn't* come up with the idea of the compiler, and high-level programming languages, without understanding programming to begin with.

These major advances, not simply technical advances but also ones that were resonant with many cultural implications, were *not* made by treating programming as instrumental or as a detail to be handled later. These and dozens of other major breakthroughs were made by programmers who used programming to think, doing creative computing and programming as inquiry. These innovators actually developed their breakthrough ideas while programming.

Douglas Rushkoff, in his book *Program or Be Programmed*, notes that while "we see actual coding as some boring chore, a working-class skill like bricklaying, which may as well be outsourced to some poor nation while our kids play and even design video games . . . the programming—the code itself—is the place from which the most significant innovations emerge" (Rushkoff 2010, 137). The examples I provided are only a few of the historically significant cases in which innovation was inseparable from programming.

The problem with separating high-level digital media design from programming is that, in many cases, they are inseparable. Even if you do plan to enlist others to do the heavy lifting of coding, you will somehow have to figure out what to do in the first place, which requires an understanding of computation, knowing about the capabilities of the computer in the way that programmers do. An article in *Mother Jones* explained this:

> The happy truth is, if you get the fundamentals about how computers think, and how humans can talk to them in a language the machines understand, you can imagine a project that a computer could do, and discuss it in a way that will make sense to an actual programmer. Because as programmers will tell you, the building part is often not the hardest part: It's figuring out what to build. "Unless you can think about the ways computers can solve problems, you can't even know how to ask the questions that need to be answered," says Annette Vee, a University of Pittsburgh professor who studies the spread of computer science literacy. (Raja 2014)

The book you are reading seeks to enable new programmers to do the type of sketching, exploration, and iterative development that were done in the influential projects just listed—and it focuses on humanistic and artistic inquiry. It's my hope that

it will help the reader acquire some of the sense for programming, and some of the willingness to explore, that was exhibited by the people who made these important breakthroughs.

[1.6] Programming Languages Used in This Book

Python and Processing are the main languages used in this book, although the first encounter with programming is via JavaScript—in case you thought that all programming languages had to begin with the letter *p*. In chapter 16, "Sound, Bytes, and Bits," we'll write not entire programs, but the arithmetic expressions (canonically used in the C programming language) that are central to a curious and compelling type of sound generation.

Python is a powerful, standardized, widely used language; I find that it is very good for exploration and also very suitable for new programmers. With additional modules, it can be used for image processing, to develop games, to do extensive statistical work, and for all types of purposes. Without installing anything additional, it can serve very well for simple text processing and to introduce computing. Guido van Rossum began developing Python in 1989. Python is actually included in OS X and GNU/Linux distributions, but I ask that everyone using this book install the same standard distribution of Python 3, Anaconda, which works on OS X, GNU/Linux, and Windows and provides important and very useful features.

Processing is a language, based on and similar to Java in many ways, that provides excellent facilities for computational visual art and design. It was created by Ben Fry and Casey Reas and first released in 2001. Processing (we will use version 3) includes an elegant, simple integrated development environment (IDE) that makes sketching with code easy, and was designed with artistic exploration in mind. For those interested in continuing to explore using Processing, there are several good books and extensive online resources, which appear in the references and are listed, with discussion, at the end of chapter 9. Processing is available at no cost for GNU/Linux, Mac, and Windows.

The language commonly known as JavaScript has a very obvious virtue: it can be incorporated into HTML and can run in practically any Web browser, locally or over the Web. JavaScript as it existed for many years, in the context of HTML and CSS on the Web, may be thought of as something of an affliction, attested to by the fact that *JavaScript: The Good Parts* is a 172-page book, while *JavaScript: The Complete Reference, Third Edition* weighs in at 976 pages. (As an exercise, I invite you to compute the percentage of JavaScript that is not good.) As recently as a few years ago, producing

JavaScript code that worked consistently across browsers required time, effort, and expertise. Even then, one could quickly understand that there are benefits to using this language. Understanding and modifying some existing JavaScript programs would show that they can be very easily shared online. The language, which was released in 1995, is suitable for artistic explorations and for sharing creative projects with others on the Web. It was developed (originally by Brendan Eich) for use by those who weren't computer scientists or professional programmers. Only a text editor and a Web browser are needed to write and run JavaScript, although the Firefox browser offers a scratchpad for writing, running, and editing code along with other facilities. There are online options for doing similar things, too.

I have to note that when the first edition of this book was published, my feelings about JavaScript were rather negative—and I think justified! Since then, I have done more work in a more recent version of JavaScript, officially called ECMAScript 6 (ES6). This version introduces some pleasing improvements; also, current browsers are providing much more consistent support for ES6 than has been the case in previous years and with earlier versions of the language. This means that while a JavaScript programmer's time used to be consumed in dealing with special cases for different browsers, it's now possible—assuming you are programming for current-generation browsers—to spend more of one's time actually dealing with the core computational issues, exploring and creating. More concretely, it's great to see some of the particular constructs I find intuitive in Python, such as looping over every element of a list, implemented in an intuitive way in ES6.

[1.7] Free Software and No-Cost Software

In describing what you will be asked to download and set up, I have mentioned that any software needed to pursue *Exploratory Programming for the Arts and Humanities* and not already included with your system is available at no cost. I'll add to that now: everything required to follow along in this book is *also* free software. The distinction is not obvious, but it's an important one, particularly for those concerned with the cultural implications of computing and how computing can be used for inquiry and creative work.

Some software that doesn't cost anything, also called *freeware*, can be downloaded and used without a financial transaction. Freeware can still be encumbered in various ways, however. The license that allows use of the software may say that it can only be used for noncommercial purposes, for instance, or only by students, or only by people who have signed a loyalty oath, or only by men. Furthermore, you may be given the

software only in executable form, without source code, making it impossible for you or anyone else to fix bugs in it, to expand it, or to adapt it to different needs, including new computers and operating systems.

Because free software comes with access to source code, some people use *open source* as a similar term, sometimes as a synonym. I prefer the term *free software*. The openness of the source code is not the only freedom in free software. There is also, for instance, the freedom for anyone to be able to use the software for any purpose, even if they are not students or have not signed a loyalty oath. In other words, you can be given "open" source code and still have restrictions placed on how you use it. If you believe that computation is a way of thinking and that code should be treated in the best ways that we treat ideas, your real concern is not only with code being open, but with software freedom.

The term *free software* was first used in its current sense in 1985 by Richard Stallman, who had, shortly before, founded the GNU project to develop an operating system that was to be available as free software. (The *G* in *GNU* is pronounced when speaking of the project, although that's not the case when naming the animal.) In March 1985, Stallman's "GNU Manifesto" was published, and he founded the Free Software Foundation (FSF). Of course, many people desired software liberty before 1985, and they acted to promote it in various ways. And people after 1985 have sometimes wanted the software they create and use to be free but haven't explicitly used the term *free software*. When people write very small programs or snippets of code, for instance, they often don't include a lengthy license or an official declaration, even if they wish their work to be freely available and freely shared. (I don't add licenses to very tiny programs that I write, even though I'm glad for them to be shared, modified, and reused in any way.) Stallman, in 1985, brought together several useful principles to form the first concept of free software, one that has continued to evolve and to be refined—although the principles remain the same.

The FSF's definition of *software freedom* at gnu.org/philosophy/free-sw.html.en includes four points, numbered 0 to 3. (A moment of foreshadowing: As we begin working with lists and arrays, we'll see that it is conventional in computing to begin numbering a sequence with 0. There are reasons that this convention was established and persists, too, which will be covered later.) The four freedoms are as follows:

[Freedom 0] The freedom to run the program as you wish, for any purpose.

[Freedom 1] The freedom to study how the program works, and change it so it does your computing as you wish. Access to the source code is a precondition for this.

[Freedom 2] The freedom to redistribute copies so you can help others.

> **[Freedom 3]** The freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

A version of, for instance, some commercial illustration software that is licensed for educational use only, by full-time students, does not offer any of these freedoms. When the user of this software finishes school, they are not legally allowed to use the software to further edit or export illustrations done while in school—denying the student access to their own creative work. A supposedly "free" app from Apple, Inc.'s App Store, even if it allows use for any purpose, still lacks freedoms 1, 2, and 3. A user cannot redistribute copies to a neighbor, family member, or even to himself or herself; it's a requirement that one go to the App Store. If the app disappears from there and the user gets a new phone, too bad. Also, freedoms 1 and 3 are missing because they rely on access to source code. For these reasons, it doesn't make a great deal of sense to me to call such purportedly free apps "free." I think of them as currently priced at zero dollars and as locked down and restricted.

You may hear that the *free* in *free software* is free as in *free speech*, not as in *free beer*. That can be helpful to understanding the concept. People also refer to free software as free/libre/open source, or FLOSS, to emphasize that the relevant sense of free is "libre" as in freedom, not "gratis" as in given away without cost. Another good way to put it is that freedoms in *software freedom* are not really the freedoms of the software itself, but the freedoms of the people using the software; they are user and programmer freedoms (Hill 2011).

Many good speeches have been given, and many articles and books have been written, about the concept of free software and the virtues of this idea. I will offer just a brief comment here. When people innovate and develop new ways of using computation, this can be treated as a contribution to the world of ideas—$E = mc^2$ or the polio vaccine—or it can be treated like the song "Happy Birthday to You," the copyright to which was claimed by Warner/Chappell Music Inc. more than one hundred years after the song was written. If people view the computer as a way to make money fast, to reallocate resources from other people to themselves, they will prefer the latter option. If they believe that computation's most significant use is in enlarging the human intellect and making the world a better place, it will be more reasonable to establish a framework of sharing, freedom to use software for any purpose, and allowing people to build on one another's successes. No-cost software in general, those programs that sport price tags of zero dollars, do not inherently embody this idea. This is, however, the basic concept of free software.

Whether you are working in a group or not, I must emphasize again that you should definitely work through this book—all of the rest of the book—*with a computer in front of you*. And it should be a computer you are actively using! Typing code from the book into your programming environment will give you a feel for typing in valid code and running it and will reassure you that short programs can be entered from scratch and run. As you work through this book, I suggest you avoid setting aside some fixed, sacrosanct time that is the *only* time when you program or (even worse) going to some sterile computer lab that you would never otherwise visit and making this the exclusive place to do your work. Just set up the systems you need on the computer that you regularly use and make time the way you would for other types of study, practice, and activity.

You don't need to be online as you work through this book, except to download software. The Processing reference pages, for instance, are part of the Processing download—you can access them through the Help > Reference menu item. You also *can* access them online, but you have them installed on your system, and you can be sure that the ones you have installed are the right version. In the Anaconda Navigator, you can click on the Learning tab on the left and you'll see plenty of Python documentation, including Python Reference. This happens to link out to the Web and seems to always point the latest version of the Python 3 documentation, even if you have an earlier version of Python 3 installed. This book shows you how to get basic information about how Python works within the interpreter we will use, Jupyter Notebook. The point is, it's not necessary to be online to learn more about the specifics of Python or Processing—certainly not to do the exercises. Reading through such reference material is not even essential for exploration and for doing the free projects.

Thus, if you find it distracting to be online, you might take your notebook computer to one of the dwindling places in the world where you won't be online. If you find yourself on a plane or train without Wi-Fi, or otherwise in an environment in which you lack Internet access, don't let that stop you from working through *Exploratory Programming for the Arts and Humanities*.

However, if it isn't a distraction and net access is available, there are some good reasons to be online. In addition to the Processing reference pages you have on disk and your ability to find out how Python works, you can also check online documentation. Once you understand the essential concepts, the documentation for Python and Processing can help you determine how to accomplish what you conceptually understand in the specific syntax of the programming language. For Python 3, documentation for the current version can be installed locally and is also available online at docs.python.org/3/. The Processing reference, again, gets installed locally with Processing itself, but also be found online at processing.org/reference/.

For instance, the pattern `Khaaaa?a?a?a?n!` works exactly the same way as that previous pattern. It's messier, but if you want to write something like that to start—and you can get your pattern working—you can then refactor it, as with any other code, and end up with a tidier regular expression later in the process. A final note about this complicated quantifier is you can leave off either of the two numbers. The pattern `Kha{3,}n!` covers all exclamations of Khan's name with three or more occurrences of *a* included, no matter how far the name is extended. If you use `Kha{,7}n!`, the pattern will match from "Khn!" up to "Khaaaaaaan!" because zero to seven occurrences of *a* will be fine.

[9.7.5] Grouping Parts of Patterns

Although we aren't diving into this in this book, it's possible to group parts of a pattern together using parenthesis and this can be very useful to do. There are lots of reasons to group parts of a pattern when you're using regular expressions within a program, but I will focus on what's probably the simplest use of grouping. It allows you to apply a quantifier to whatever part of a pattern you prefer—not particularly a single literal, a single special sequence, or a single character class.

To demonstrate this, I'll provide a pattern that will match any sequence of five or more three-letter words in a text. This is a pattern of specific professional interest to me, as I've been writing a poetry book called *All the Way for the Win* which consists entirely of three letter words. Here is the pattern:

```
\b(\w\w\w\W+){7,}
```

I've specified that whatever we match needs to start at a word boundary (the \b) and then have five or more occurrences of the next part of the pattern, which I've grouped using parenthesis. This group has three word characters in it followed by one or more nonword characters, which will include spaces but also punctuation. I then use the fancy curly bracket quantifier to specify that I am looking for seven or more of what's in the parentheses, occurring in a sequence.

I could improve this pattern in a few ways to make it more general. Right now, it will match "words" like *007* that I don't happen to be interested in because they aren't three-letter words; in fact, that one doesn't have any letters at all in it. But that's a first example of how to use grouping and why it might be of use to you. I was quite taken aback when I used this pattern to search through the Enron corpus of corporate emails, finding that someone had excitedly written the phrase "YOU ARE THE MAN FOR THE JOB" (just like that, in all caps) to extend an employment offer.

Groups can be used for many other purposes, not just finding sentences like this one. For instance, you can specify several and, in your Python program, access what

is matched in each group, manipulating the data separately. Do start applying those parentheses when you feel ready to try it out. In many cases, though, you can effectively use several simpler regular expressions in different parts of your program. So don't let the complexity of grouping keep you from exploring what you can do with regular expressions.

[9.7.6] More on Regular Expressions in Python

Hopefully this chapter, and this section of it, provide a running start. So far, I have not mentioned a few of some of the more often-used aspects of regular expressions. There are additional metacharacters. The ^ allows a match only at the beginning of a line and the $ only at the end of a line. There is also an "or" operator | such that the pattern `A|B` will match A as well as B—that is, A *or* B are both fine. These and other aspects of regular expressions are all very useful, but there are always limits to what can be explained alongside the rest of the artistic and humanistic computing in this book. For another explanation of how regular expressions work in Python—one that is longer and more comprehensive—I suggest the helpful how-to in the Python documentation for the `re` library, which should be easier to digest with this chapter as a foundation:

docs.python.org/3.8/howto/regex.html

[9.8] Essential Concepts

[Concept 9-1] Explore Regular Expressions in an Editor

You should understand the basic, practical way to develop regular expressions (by trying them out in a text editor) and then understand how to include the ones you have developed in a program.

[Concept 9-2] Patterns Go Far beyond Literals

Regular expressions open up a wide range of possibilities that go beyond a simple *search string*—a pattern that is a literal string. At the same time, they are a simpler formalism than computer programming in general. Once you have understood the way regular expressions extend the traditional idea of finding a plain old string, and once you are comfortable experimenting with regular expressions, you can develop your skills and strengths further as you explore.

[10] Image I: Pixel by Pixel

[10.1]

In this chapter we'll look at very simple ways of modifying and analyzing image files in a standard, widely used format. This chapter and the next deal with low-level image manipulation, showing how it can be scaled up to work on large numbers of files (using iteration). The manipulations covered are the same as some of the ones implemented in Photoshop and the Glimpse editor (a free software program for photo editing). In chapter 11, "Image II: Pixels and Neighbors," there is a further opportunity for the analysis of collections of images. In chapter 14, "Image III: Visual Design and Interactivity," different techniques for drawing lines and shapes, and for producing simple animations, are covered. The work in chapter 14 is done using Processing, an ideal language for computational visual design. Processing is introduced in chapter 12, "Statistics, Probability, and Visualization."

Images can be represented in different ways, but the ones we'll consider in this chapter are represented as grids or rectangles of pixels. This *bitmap* representation is a very common one for images. While there are also vector-based representations and other ways of representing images, everything that is displayed on a modern-day computer screen is represented in this bitmapped way, at least at the final stage of display and often earlier.

The advantage of focusing on low-level, Photoshop-like manipulations is that the programming needed to accomplish these is very much like that needed to analyze certain important properties of images. For instance, we will write a short program to redden images—to add red to every pixel. We will also write a short function to determine the redness of images so one can be compared to another. This means we can iterate through large numbers of images and find the reddest one. This is a technique that can be built upon to do other types of meaningful image analysis.

[Index]

Page numbers followed by *f* refer to figures.