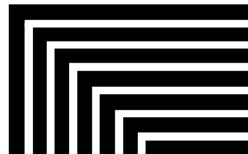
Library Technology
R E P O R T S

Expert Guides to Library Systems and Services

Coding for Librarians: Learning by Example

Andromeda Yelton



American Library Association

Library Technology REPORTS

ALA TechSource purchases fund advocacy, awareness, and accreditation programs for library professionals worldwide.

Volume 51, Number 3

Coding for Librarians: Learning by Example

ISBNs: (print) 978-0-8389-5957-2
(PDF) 978-0-8389-5958-9
(ePub) 978-0-8389-5959-6
(Kindle) 978-0-8389-5960-2

American Library Association

50 East Huron St.
Chicago, IL 60611-2795 USA
alatechsource.org
800-545-2433, ext. 4299
312-944-6780
312-280-5275 (fax)

Advertising Representative

Patrick Hogan
phogan@ala.org
312-280-3240

Editor

Patrick Hogan
phogan@ala.org
312-280-3240

Copy Editor

Judith Lauber

Production

Tim Clifford and Alison Elms

Cover Design

Alejandra Diaz

Library Technology Reports (ISSN 0024-2586) is published eight times a year (January, March, April, June, July, September, October, and December) by American Library Association, 50 E. Huron St., Chicago, IL 60611. It is managed by ALA TechSource, a unit of the publishing department of ALA. Periodical postage paid at Chicago, Illinois, and at additional mailing offices. POSTMASTER: Send address changes to Library Technology Reports, 50 E. Huron St., Chicago, IL 60611.

Trademarked names appear in the text of this journal. Rather than identify or insert a trademark symbol at the appearance of each name, the authors and the American Library Association state that the names are used for editorial purposes exclusively, to the ultimate benefit of the owners of the trademarks. There is absolutely no intention of infringement on the rights of the trademark owners.

ALA TechSource

alatechsource.org

Copyright © 2015 Andromeda Yelton
All Rights Reserved.

About the Author

Andromeda Yelton (http://andromedayelton.com) is a self-employed librarian and software developer who's passionate about promoting coding, collaboration, and diversity in library technology. She has a BS in mathematics from Harvey Mudd College, an MA in classics from Tufts, and an MLS from Simmons. Before her MLS, she taught Latin to middle school boys; after that, she did library outreach, software, and communications at the e-book startup Unglue.it. Her notable honors include winning the 2010 LITA/Ex Libris Student Writing Award; being selected as an ALA Emerging Leader, class of 2011; being a 2013 *Library Journal* Mover & Shaker; and having been a listener contestant on *Wait, Wait, Don't Tell Me*. She is a member of the Ada Initiative advisory board and the LITA board of directors.

Abstract

This issue of *Library Technology Reports* draws from more than fifty interviews with librarians who have written code in the course of their work. Its goal is to help novice and intermediate programmers understand how programs work, how they can be useful in libraries, and how to learn more.

Three chapters discuss use cases for code in libraries. These include data import, export, and cleanup; expanded reporting capability; and patron-facing services such as improvements to catalog and LibGuide usability. Most of the programs discussed are short—under a hundred lines—so that implementing or modifying them is within the reach of relatively novice programmers. Where possible, links to the code itself are provided. Several scripts are explained in depth.

Additional chapters focus on nontechnical aspects of library code. One chapter outlines political situations that have been faced by librarians who code and the solutions they have employed. Another chapter shares interviewees' advice on specific resources and strategies for learning to code.

Get Your *Library Technology Reports* Online!

Subscribers to ALA TechSource's *Library Technology Reports* can read digital versions, in PDF and HTML formats, at http://journals.ala.org/ltr. Subscribers also have access to an archive of past issues. After an embargo period of twelve months, *Library Technology Reports* are available open access. The archive goes back to 2001.

Subscriptions

alatechsource.org/subscribe

Contents

Chapter 1—Introduction 5
- Who This Report Is For 5
- What You'll Find Here 6
- Survey 7
- Acknowledgments 7

Chapter 2—Data Workflows 9
- Examples 9
- Deep Dive: LibALERTS 10
- Notes 12

Chapter 3—Reporting 13
- Examples 13
- Deep Dive: Automated ILS Reporting 14
- Notes 15

Chapter 4—Patron-Facing Services 16
- Overview 16
- Examples 16
- Deep Dive: LibGuides Organizer 19
- Notes 21

Chapter 5—Political and Social Dimensions of Library Code 22
- Library Coders' Job Descriptions and Realities 22
- Buy-in 23
- Institutional Barriers 24
- Notes 25

Chapter 6—Learning to Code 26
- Learning Strategies and Resources That Coders Recommend 26
- Workplace Support 29
- Conclusion 30
- Notes 30

Introduction

There's been a huge amount of interest among librarians in learning to code. Numerous library conferences—including ALA, LITA Forum, Texas Library Association, Access, Code4Lib, and the Ontario Library Association—have featured programming tutorials or hackathons in recent years. These short workshops are wonderful for exposing people to fundamental concepts and creating positive experiences around code, but students don't necessarily know what to do next. For the learning to stick, people need real-world projects to which they can apply their newfound skills, which raises the question: how do librarians use code in their everyday work? This issue of *Library Technology Reports* is an answer to that question.

For this report, I reached out to LITA-L, the Code4Lib and LibTechWomen mailing lists, and my own network to survey librarians on how they use code in their jobs. I particularly looked for people who are not primarily developers and who could share examples of short scripts (under a hundred or so lines) that they'd written. While there are some wonderful large-scale code projects in libraries, I wanted to write about small, useful scripts that don't require expert-level coding skill: projects that are within reach even for fairly new coders. If you've been looking for reasons to learn code, ways to apply developing coding skills, or concrete examples to help you justify professional development support, this report is for you. If you manage aspiring coders you want to support, this report is for you, too.

Who This Report Is For

This report has three main audiences:

- librarians who are considering learning how to code
- librarians who have done some introductory programming study and are looking for next steps
- managers of librarians who code

If you're considering learning how to code, this report will give you lots of ideas about what you can do with code, what might be most helpful to learn, and how you can make your case to management. You'll probably want to skip ahead first to chapter 6, which summarizes survey respondents' advice on learning how to code. Then read chapters 2–4 to see real-world things your peers have done with code. That will help you narrow down your options.

Unless you're aiming to be in a pure development role, it's not practical to learn all the software topics that might be of interest; by contrast, mastering the fundamentals of programming and learning how to make one language do a few useful things is an achievable goal that will give you powerful, flexible new ways to approach your everyday librarian work. The middle chapters describe programs written by librarians with a variety of job descriptions, in different institutions, addressing diverse use cases. Look for projects that would be helpful in the context

of your work, and use those to guide what you need to learn.

If you've already learned programming fundamentals, you may also find chapter 6 helpful, since it includes intermediate learning resources as well as options for review. However, start with chapters 2–4. Writing your own version of one of the programs in these chapters will be an excellent way to practice your skills and find out what you need to learn next. Most of them are under a hundred lines; I explicitly asked for short programs so that they'll be manageable for new programmers. Many of them have source code online; these links are provided in each chapter. Modifying others' code for your own use case is also a great way to improve your skills. Indeed, many library coders get a great deal of mileage out of modifying others' code and rarely if ever write their own from scratch. Finally, chapter 5, which details the political challenges and benefits of library code, will be helpful to you as you start to deploy code at work. Many library coders find that navigating buy-in and advocating for code are every bit as integral to their work as the actual programming. Avoid reinventing the wheel; learn about the problems they've encountered and techniques for handling those problems. And look for inspiration in the ways that people have positively impacted themselves and their coworkers through code.

If you're a manager, whether you code or not, you may be looking for ways to support your technically inclined supervisees. Chapter 6 outlines the support that librarians have received at work for learning to code. Some survey respondents are managers as well as coders, and I also asked them what kinds of support they provide; this is covered in chapter 6 as well. Chapter 5, about the political impacts of code, is also relevant to you. Finally, if you're not a coder, you may be interested in reading chapters 2–4 simply to see what others have done with code in libraries and how similar work might benefit your own library. Of course, if you are a coder, you may be interested in reading these chapters to get ideas for your own projects.

What You'll Find Here

After the introduction, there are three main sections. Chapters 2–4 cover programs that librarians have written to get things done better in their libraries, organized by common use cases: data cleanup, import, and export; reporting; and patron-facing services (mostly, but not always, through the website). Chapter 5 covers the political and social impact of library code, which came up so often in survey responses that it demanded a chapter of its own. Chapter 6 covers resources and advice for learning to code.

Chapters 2–4 each follow the same internal structure. After an introduction outlining the general use case of the code in the chapter, there's an examples section with brief overviews of a half dozen or so scripts. Each overview notes the script's author, language, and purpose. Where available, links to source code are provided, and you're encouraged to consult that code alongside the text.

After the overviews, each of chapters 2–4 has a deep dive into one script. Think of this as a code-reading group we're doing together, aimed at novice programmers. I'll walk you through the scripts line by line, showing you how they're organized and what each part accomplishes. Each of these scripts is online, so please pull up the code in your web browser and follow along. Additionally, the deep dives offer commentary on best practices and suggestions for how to expand or modify each program in case you want concrete ideas for practicing your programming skills.

These deep dives intentionally address different use cases and focus on programs written in different languages (PHP, Python, and JavaScript, all of which are in widespread use in libraries). I want to maximize the chances that one of them is relevant to you, no matter what library role you're in or what programming languages you may speak. I'm also deliberately agnostic throughout this report about what language you should learn. All of them incorporate the same fundamental programming concepts, like variables, control flow, and functions; all of them can be used to tackle a wide variety of problems. While many programmers have strong feelings about what language is best, the best language *for you* depends on which languages have good tools for solving problems you care about; which ones your coworkers, friends, or local community are already using; which have good learning resources you can access readily; and which appeal to your own idiosyncratic sense of aesthetics. I feel strongly that learning to program can be intellectually stimulating, personally empowering, and professionally useful, and you can realize those benefits regardless of what programming language you start with.

A note on mechanics: like many books on programming, this report follows the convention that text written in `monospaced font` represents code. When I am directly quoting lines of code or words from programming languages, I'll use `Courier`. For the most part, however, I refer to code samples online rather than reproducing them in the report.

Finally, there are a lot of survey responses that couldn't fit in this report, and ultimately the web is a more natural home for code than a report. For more information—including both scripts as of this writing, and any changes they may have undergone since—consult the companion website.

Companion website
https://thatandromeda.github.io/ltr

Survey

I'm writing a Library Technology Report *for ALA TechSource on lightweight, useful programs librarians & archivists have written to get their jobs done better. You need not be a library developer to answer these questions—in fact, I want to hear from people with all sorts of library roles! You need not be an expert coder or have a perfect code sample available, either. If you wrote some code that got something done, no matter how hackish or how elegant, I want to hear from you.*

How much of your job is about coding? Do you have any formal code responsibilities, or is this simply a skill that you bring to your formal responsibilities? *

Have you had support from your employer in learning to code/spending your time on coding? *

If you're a manager, have you provided support to employees who want to learn to code? If so, what sort? If not, why not?

What would you recommend to someone who wanted to learn to write code?

For the following questions, please pick a short program (less than a hundred-ish lines) that you wrote to do some library/archives task. If you have more than one you'd like to talk about, feel free to have multiple answers per question (just make it clear which answer goes with which script).

What language was this code written in? *

What problem did this code solve? *

What was the impact of the code? (time saved, new/improved service to patrons, etc.) *

If the code is publicly available, please link it here.

What did you learn from implementing this code? *

Survey

In the box above are questions and explanatory text from the survey. Asterisks indicate a required question. Not shown are requests for name, contact information, and referrals.

Acknowledgments

Thank you to all of my survey respondents, who shared so generously of your time, insight, and code. You are the giants on whose shoulders I stand.

Hillel Arnold (Lead Digital Archivist, Rockefeller Archive Center); Jason Bengtson (Head of Library Computing and Information Systems, University of Oklahoma); John Blyberg (Assistant Director for Innovation and User Experience, Darien Library); Terry Brady (Applications Programmer Analyst, Georgetown University); Matthew S. Collins (Director of the Library and Associate Professor, Louisville Presbyterian Theological Seminary); Esme Cowles (Lead Product Developer, UC San Diego Library); Jeremy Darrington (Politics Librarian, Princeton University Library); Robin Camille Davis (Emerging Technologies & Distance Services Librarian, John Jay College of Criminal Justice [CUNY]); nina de jesus (Digital Projects Librarian, York University); Misty De Meo (Digital Collections Technician, Canadian Museum for Human Rights); Rachel Donahue (Digital Projects Librarian, Special Collections National Agricultural Library); Mike Drake (Deputy Director, Tulare County Library); Shaun Ellis (User Interface Developer, Princeton University Library); Genny Engel (Webmaster, Sonoma County Library); Chris Fitzpatrick (ArchivesSpace Developer, Lyrasis); Angela Galvan (Digital Reformatting Specialist and Head, Document Delivery, The Ohio State University Health Sciences Library); Mike Giarlo (Digital Library Architect, Penn State University); Ron Gilmour (Web Services Librarian, Ithaca College Library); Annie Glerum (Head of Complex Cataloging, Florida State University Libraries); Jason Griffey (Chief Technology Strategist, University of Tennessee at Chattanooga); Thomas Guignard (École polytechnique fédérale de Lausanne); Cindy Harper (Electronic Services and Serials Librarian, Virginia Theological Seminary); Michael Holt (Reference Librarian/Marketing Coordinator, Valdosta State University Odum Library); Ken Irwin (Reference Librarian, Wittenberg University); Deborah Kaplan (Digital Resources Archivist, Tufts University Digital Collections and Archives); Francis Kayiwa (Assistant Professor and Systems Librarian, Colgate University Libraries); Bohyun Kim (Associate Director for Library Applications and Knowledge Systems, University of Maryland, Baltimore); Sam Kome (Director, Collection Services and Scholarly Communication, Claremont Colleges Library); Tricia Lampron (Metadata Service, University of Illinois at Urbana-Champaign); Joel Marchesoni (Tech Support Analyst, Hunter Library at Western Carolina University); Joe Montibello (Library Systems Manager, Dartmouth College); Christine Moulen (Library Systems Manager, MIT); Chad Nelson (Developer, CollectionSpace); Jeremy Nelson (Metadata and Systems Librarian, Tutt Library, Colorado College); Joy Nelson (Director of Migrations, ByWater Solutions); Eric Phetteplace (Emerging Technologies Librarian, Chesapeake College); Dot Porter (Curator, Digital Research Services, University of Pennsylvania); Carrie Preston (Head of Web Services, Ohio University Libraries); Matthew Reidsma (Web Services Librarian, Grand Valley State University); Sibyl Schaefer (Assistant Director, Head of Digital Programs, Rockefeller Archive Center); Michael Schofield (Librarian of Web Services, Alvin Sherman Library, Research, and

Information Technology Center); Coral Sheldon-Hess (Web Services Librarian, University of Alaska Anchorage); Jason Simon (Technology & Serials Librarian, Fitchburg State University); Owen Stephens (Independent Consultant); Ruth Szpunar (Reference and Instruction Librarian, DePauw University); Scott Turnbull (Lead Software & Systems Engineer, University of Virginia); Esther Verreau (Web Developer, Skokie Public Library); Matt Weaver (Web Librarian, Westlake Porter Public Library); Evviva Weinraub Lajoie (Director, Emerging Technologies & Services, Oregon State University Libraries & Press); Josh Westgard (Graduate Assistant, Digital Programs and Initiatives, University of Maryland Libraries); Amy Wharton (Research & Emerging Technologies Librarian, University of Virginia School of Law Library); Erin White (Web Systems Librarian, Virginia Commonwealth University); and Stuart Yeates (Technical Specialist, Victoria University of Wellington).

I'd like to thank two people particularly for not only responding to the survey but also for giving me feedback on the rough draft: Sarah Simpkin (GIS and Geography Librarian, University of Ottawa) and Becky Yoose (Discovery and Integrated Systems Librarian/Assistant Professor, Grinnell College). Additionally, thank you to the following people for last-minute beta reading: Natalie DeJonghe, Jordan Hale, David Saunders, Padraic Stack, Anne Sticksel, and Deidre Winterhalter.

The listed institutions and titles were accurate when respondents answered the survey; some have since changed jobs. Any other errors are, of course, my own.

Data Workflows

One of the most common use cases for coding in libraries is data processing. Whether it's import/export, quality control, combining data from different sources, or adapting externally provided records to local purposes, data tasks are ubiquitous in library technical services. Many of them are quite repetitive and, as such, lend themselves well to scripting. In addition, computers are often faster and more accurate than humans at repetitive tasks. Therefore, the time spent in developing data processing scripts can pay off manyfold in increased efficiency, freeing librarians to do more creative, sophisticated tasks that require human insight.

In this chapter, I'll provide an overview of eight scripts that simplify various data processing tasks and do a deep dive into a ninth. Their use cases include metadata quality control, import/export workflows, bulk downloading, and data migration.

It's notable that these nine scripts are in seven different programming languages (bash, Python, VBA for Excel, Perl, Ruby, XSLT, PHP). Beginning programmers often want to know the best language to learn, and there truly isn't one. While some languages may be easier or harder for a given student, and more or less suited for a particular use case, they all incorporate the same fundamental programming concepts, and all of them open a lot of doors.

Examples

Facing a need to export data from DSpace, nina de jesus wrote a bash script to do it. This script exports metadata from every handle in a series and dumps it to a CSV file for later processing. Like many programmers, de jesus learned how bash worked in the course of getting this script to work. This made it slow going at first, but she expects it to pay off handsomely over time: "For all that this tiny script took me a long time to write (maybe three or four days to get it working properly), it saved me a lot of tedious hours of slowly (manually) going through database tables and spreadsheets to get the data I needed. And now I can use the script whenever I need to get this kind of data out of DSpace again (which I'm sure will happen)."

nina de jesus's script
http://satifice.com/2014/10/22/exporting-the-metadata-of-a-range-of-handles-in-dspace

Hillel Arnold also needed to export metadata: in his case, EAD files from ArchivesSpace. His short Python script finds all the resource IDs that match a given criterion, gets their EAD, and writes it to a specified destination.

Hillel Arnold's script
https://gist.github.com/helrond/1ef5b5bd47b47bd52f02

Becky Yoose also saved time by automating a tedious workflow. Her library had a trigger file, in Excel format, of books to be acquired under a patron-driven acquisition policy. The library needed to extract MARC records from the database using local control numbers in the file, edit them for consistency with local cataloging rules, and insert codes to make the ILS's purchasing module automatically create an

order request. By hand, this workflow took five to ten minutes per week per record, or almost one to two hours per week of cataloger time; the script reduced processing to two minutes total, for a net savings of one to two weeks per year of cataloger time. Additionally, as she notes, "Each time a human has to touch the record, it's a possible fail point" because of the risk of misspellings and other oversights; machine processing improves accuracy while saving time.

A version of this script (edited for use in the LITA/ALCTS Library Code Year Interest Group Python preconference at ALA Annual 2013) is available online. The README at that link explicitly permits library reuse and adaptation.

Becky Yoose's script (edited)
https://github.com/LibraryCodeYearIG/MARC-record-edit

Tricia Lampron had a text file with bar codes corresponding to files that needed to be downloaded. By hand, this meant she had to "enter in the link, right click to download the file, and then . . . change the file name once downloaded" for up to 190 files—a tremendously tedious process. Her Python script reads the text file, constructs the corresponding URL, downloads the file, and creates an appropriately named XML file locally.

Joy Nelson and Ruth Szpunar both faced metadata cleanup tasks. Szpunar cleaned up and organized metadata from a digitization project using VBA for Excel. Nelson works for an ILS support vendor whose customers often want to move data from one MARC tag to another during ILS migrations; she wrote a Perl script to handle this task. Nelson's use case in particular underscores how tedious, repetitive tasks can be great scripting candidates if you can specify a clear rule for them; as long as you can specify the exact field and subfield that you want data to move from and to, you can write this program with only a handful of lines of code, and it will execute accurately over thousands of records in (almost) no time.

Misty De Meo faced a more complex migration problem. She inherited a controlled vocabulary, "but it became clear that there were a large number of deficiencies in it: inconsistencies, missing terms, duplicate terms, incorrectly-matched relationships, and so on." It's difficult to have a computer fix this kind of problem because there are so many ways the data can be wrong, and making it right could require human judgment calls. However, she was able to write a Ruby script that automatically fixed the simpler errors and flagged others for subsequent human review. Along the way she gained better insight into her data set: "It really helped me understand why the metadata had problems, and helped me reason about what was probably intention vs what was probably an accident. Many patterns that weren't at all obvious when reading metadata by eye instantly became clear once it was being processed by software."

These kinds of data quality problems are common in library coding and can sometimes be so pervasive or frustrating as to make it infeasible to build software on top of the data. However, exposing problems through attempts to write code can suggest opportunities for improvement. Clarifying local cataloging rules, adding input validation (see Eric Phetteplace's script in chapter 4), or writing scripts that run regularly to detect common problems can all improve data quality, for example.

Annie Glerum also had metadata quality problems: in her case, inaccurate vendor records. Her XSLT stylesheet "identifies records needing location code edits for the catalog's holdings record, corrections to the MARC coding, edits to bring the record to full level, or human review for special formats and sets." It outputs its report as an Excel spreadsheet, which fits well into subsequent workflows.

Deep Dive: LibALERTS

Patrons at Westlake Porter Public Library (Westlake, OH) wanted to be notified by text message when the library got new books by their favorite authors.[1] While the library's OPAC had similar functionality, it didn't let patrons refine their searches enough to be useful and had been turned off. The library's Drupal website, however, provided many of the building blocks needed: SMS integration, a module to create Drupal nodes from MARC imports, and a module allowing users to subscribe to terms in the site taxonomy. Matt Weaver—"a development team of 1 [with] a budget of 0"[2]—was able to build a prototype alerts service by combining these modules.

However, he quickly found that he had to address data quality issues before the service could be offered to patrons. Publisher-provided MARC records did not consistently handle middle initials and sometimes misspelled author names, meaning that a single author could be represented by a variety of terms. All these terms needed to be combined in order to offer patrons a single term they could subscribe to.[3] This single term is an element of his site's taxonomy, which in turn is generated from MARC records in the Drupal site (which are distinct from MARC records in the catalog). Therefore, Weaver needed to compare his publisher-provided MARC records with his catalog versions and create records with canonical names that he could feed into his Drupal site.

In Weaver's marcreupload repository, the `marc_upload_page` script provides a front end for submitting MARC records, including a Levenshtein distance function that automatically suggests several

close-match spelling options for author names. Records submitted through this page are processed by the `authorchange` script, which we examine here. Follow along at this link:

Matt Weaver's `authorchange` script
https://thatandromeda.github.io/ltr/Chapter2.html

Line 1 simply tells the computer that this is a PHP script. **Line 2** includes the PHP library for processing MARC records, which we'll need later.

Lines 3–14 harvest data from the MARC record submitted through the form on `marc_upload_page` and store it as variables for later use. An important variable here is `$closest`; this is an array of the author names from our catalog that are the closest match to the author names submitted in the form.

Lines 15–19 write HTML; this is the web interface presented to the human using the script, and it's how we'll display feedback. (Keep in mind that this script also has a machine audience: the Drupal site that will be consuming the MARC records it generates.) All subsequent lines that begin with `echo` are also writing HTML, which provides feedback to the user, and will be skipped in this read-through.

Line 21 initializes the `$arraypos` variable; this is how we'll keep track of how many times we've iterated through the upcoming loop.

In **line 22**, we begin the loop that will take up the remainder of the program. Broadly speaking, what we'll do in this loop is look through each submitted author name, compare it to the corresponding closest-match name, and create records for the Drupal site if it seems correct to do so.

In **line 23** we increase the `$arraypos` counter by one (the `++` syntax, meaning "increase this number by 1," is common to many languages). In this loop, we're processing several records. When we generate a new MARC record for the first author name, we want to make sure we're comparing it against the first name in the closest-match array and using MARC field data from the first submitted MARC record (and so on for the second and subsequent records). Keeping track of this counter lets us be sure to look in the right place for all our information.

In **line 28**, we check to see if the author name we're currently examining is the same as the corresponding closest-match name. If it is, we'll write a MARC record; if it's not, we'll skip processing—this is a case that requires human judgment.

Assuming the names match, in **line 32**, we create an empty MARC record, which we'll call `$marc`. **Line 33** sets its leaders to be the same as those in the submitted record. **Lines 34–37** create a new MARC 008 field using the same information as in the 008 field of the submitted MARC record. (Note that we use the `$arraypos` variable to select the first, second, etc., from the array of submitted 008 fields, as appropriate.) We then append that field to `$marc`.

Lines 38–60 proceed in the same manner, copying data from the submitted MARC record to the new one being created. **Line 44** varies this slightly, using the author name from the closest-match array (which, you recall from **line 28**, exactly matches the submitted author name).

In **line 63**, we check to see if we've successfully generated a MARC record. If so, we tell the user we've written a file for it; if not, we inform the user accordingly.

Lines 66–69 actually write the MARC record for our Drupal site to our output file.

Line 72 connects back to the `if` condition that we opened in **line 28**. All the lines since then have been handling the case where the author name and the closest-match name are the same. The `else` in this line switches us to the alternative case. If we don't have matching names, we can't generate an authoritative record, so we simply inform the user of this (**line 74**) and move on. The remaining lines close all our unclosed code blocks to complete the program.

Weaver's script underscores several issues of software development process that numerous respondents commented on. One is the importance of looking for existing code rather than building from scratch. Although Weaver did write several scripts in the process of getting LibALERTS to work, the vast majority of the service resides in Drupal modules already written by others; his code patches them together. Many people, with development budgets similar to Weaver's, will find that this is much more achievable than writing things from scratch. It's often better practice, too, since existing modules benefit from the development expertise and user testing of large communities and get quicker and more thorough bug fixes than in-house code produced with limited labor.

Another issue is iteration. Very few programs work right the first time. Even if they're bug-free (which is rare), developers usually can't envision exactly what users might need to do or all the special cases the code might end up needing to address. In this case, Weaver discovered the data quality issues by writing the prototype version of his code and seeing where it encountered problems. The `authorchange` script is part of how he solved those problems.

This shouldn't be viewed as a failure of the first script, by the way. Fred Brooks, in his classic of software project management, *The Mythical Man-Month*, said you should expect at least half your time to be spent on testing and debugging—and the more components you find yourself integrating, the longer the overall time to completion.[4] Planning for iteration is simply responsible software practice.

So how would you iterate this program from here? Things to try include these:

- As a Pythonista with limited PHP skills, I had trouble reading this program and found myself reformatting it in order to analyze it. In particular, I reflexively applied semantic whitespace—indenting the contents of *for loops* and *if conditions* to make the code blocks stand out more clearly on the page. How could you reorganize and comment the `marc_upload_page` script to make it easier to read?
- Determine: is copying the leaders from the existing record valid? If we've changed author names or failed to preserve any MARC fields between `marc_upload_page` and `authorchange`, the leaders no longer accurately represent the length of the file. The `setLeader` function from PHP's MARC library explicitly does not perform any validation, so we can easily end up with invalid leaders. Figuring out if this is a problem in our case requires analyzing `marc_upload_page` and considering the input data (which may vary in different contexts, depending on local cataloging practices). If we can't safely copy the leaders, what should we do instead? (Alternatively, we could skip the entire analysis if we simply planned to generate leaders rather than copy them. Consult PHP's MARC module source code and documentation to see if it has that functionality.)

`Record.php`, containing the `setLeader` function, from PHP's MARC library
https://github.com/pear/File_MARC/blob/master/File/MARC/Record.php

- The `marc_upload_page` and `authorchange` functions handle authors in the 100 field, but don't handle additional authors from the 700 field. However, patrons who are interested in new works by particular authors may want to see their coauthored works as well. How can we add support for this?

Scripts in This Chapter

nina de jesus's script
http://satifice.com/2014/10/22/exporting-the-metadata-of-a-range-of-handles-in-dspace

Hillel Arnold's script
https://gist.github.com/helrond/1ef5b5bd47b47bd52f02

Becky Yoose's script (edited)
https://github.com/LibraryCodeYearIG/MARC-record-edit

Matt Weaver's `authorchange` script
https://thatandromeda.github.io/ltr/Chapter2.html

Notes

1. See Westlake Porter Public Library, LibALERTS webpage, accessed December 15, 2014, www.westlakelibrary.org/libalerts.
2. Matt Weaver, "LibALERTS: An Author-Level Subscription System," *Code4Lib Journal*, no. 18 (October 3, 2012), http://journal.code4lib.org/articles/7363.
3. See Weaver's *Code4Lib* article (cited in note 2) for sample code handling this issue.
4. Frederick P. Brooks Jr., "The Mythical Man-Month," in *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, 20, (Boston: Addison-Wesley, 1995).

Reporting

In chapter 2, we saw scripts that helped librarians and archivists who were limited by issues of data quality and data portability. In this chapter, we'll take one step closer to front-end users by looking at scripts that aid reporting. These scripts are all about organizing and querying the available data and presenting it to library staff—some of them nontechnical—in ways that aid decision making. They analyze logs to highlight actionable information, simplify workflows, and expand the capabilities of the ILS.

Examples

Two survey respondents wrote scripts to analyze logs. Stuart Yeates found that his web server was under attack from bots that ignored his `robots.txt` file; his bash script (figure 3.1) picked out the relevant lines from his logs and sorted them. This let him find the major traffic sources (i.e., bots) and defend against them, improving server performance. Robin Camille Davis wrote a Python script to analyze EZproxy logs. EZproxy logs an enormous amount of data, which can make it hard to find what you're looking for; her script let her zero in on usage patterns by different patron categories (e.g., students vs. faculty) and graph how usage changed throughout the year.

As Davis noted, "We can get pretty good usage stats from the individual database vendors, but with monthly logs like these, why not analyze them yourself? You could do this in Excel, but Python is *much* more flexible, and much faster, and also, I've already written the script for you."[1] In a blog post, she discusses how and why she wrote the script, changes you may need to make to run it on your own logs, and other questions you could answer by modifying the script.[2] (The need to make changes to get others' scripts working in your own environment is recurrent, and we'll tackle it in more depth later this chapter.)

```
tail -100000 /var/log/httpd/access-
nzetc.log |grep facet |awk '{print
$1}'|sort | uniq -c | sort -n
```

Figure 3.1
The entirety of Yeates's bash script

Robin Camille Davis's script
https://github.com/robincamille/ezproxy-analysis/blob/master/ezp-analysis.py

Three survey respondents wrote scripts that generate reports as part of improving various library workflows. We discussed Annie Glerum's quality control script in the last chapter. Matthew S. Collins wrote a Python script to "check a list of ISBNs from a publisher catalog against our holdings to see what we already have or need to order," saving time in the acquisitions workflow. Joe Montibello helped his preservation librarians, who were dissatisfied that they didn't have good metrics on which parts of their collection had been crawled by LOCKSS. He came up with an algorithm to estimate when crawling would be complete. Like many survey respondents, he had his program write its output to a spreadsheet so that it would be easy for his nonprogramming colleagues to integrate into their existing workflows. By automatically generating this spreadsheet as a shared Google doc, he also saved his own time in reporting out, since his coworkers could check the spreadsheet whenever they were curious rather than needing to ask him.

Joe Montibello's script
https://github.com/joemontibello/update-lockss

Finally, three respondents wrote scripts to fill gaps in ILS reporting capacity. Sam Kome simplified weeding by writing a script that "identified print volumes that met a set of rules for de-accession including a rule that [they] be held at more than one of 50+ libraries in our lending network." Cindy Harper found her serials module "cumbersome," so she wrote her own script that lists what's been sent to the bindery and what needs to be claimed. The third reporting script, below, is the subject of this chapter's deep dive.

Deep Dive: Automated ILS Reporting

Esther Verreau wrote an array of scripts (available on GitHub) that "pull stats from the Sierra ILS and report it to the appropriate destinations, Shoutbomb, Civic Technologies Community Connect and Novelist Select." Some of these had originally been written in Millennium's scripting language, but they were "error prone and difficult to maintain" and became obsolete after an upgrade. The new Python scripts run essentially without human intervention.

Esther Verreau's scripts
https://github.com/everreau/sierra-scripts

These scripts cover an array of uses. One generates RSS feeds of new items (compare with LibALERTS, chapter 2). Another generates an internal report of how many patrons had maxed out their holds; this allowed the library to use real data to decide whether it needed to change its holds policy. Indeed, a striking fact about Verreau's scripts is how many of them are *experimental*—written not to provide a new service, but to let the library gather data on whether a new service or policy would be helpful. Once people are moderately fluent at writing code, one-time-use scripts become reasonable to write. This lets people ask questions it wasn't previously feasible to ask and answer them with hard data.

Verreau's `novelist.py` script exports metadata from the library's entire collection, writes it to a file, FTPs that file to Novelist Select, and notifies someone that it happened. Let's walk through the script.

The latest version of the novelist.py#script
https://thatandromeda.github.io/ltr/Chapter3.html

The comments in **Lines 1–14** describe the script's function and clearly indicate what future users will need to change in order to run the script in their own environments. These sorts of explanatory comments are best practice because future users don't necessarily know what you were thinking and don't want to spend time puzzling through the code to find out. (This includes you, six months from now, when you are guaranteed to have forgotten what you were thinking today.)

Lines 17–25 import other Python functionality the script will rely on. These include interoperating with databases (`psycopg2`), the operating system (`os`), and e-mail (`smtplib`).

Lines 27–31 define a function, `strify`, that the program will use later when writing individual lines of metadata to the output file. The notable thing here is *error handling*—`if obj == None:` recognizes that there may be some empty lines returned by the database query and ensures that they don't result in anything being written to the output file.

Lines 33–41 also define a utility function, `put_file`, which FTPs a file to a given directory. Like `strify`, it shows error handling: if it encounters an exception while trying to FTP the file, it prints the exception to the console rather than letting the program crash. This is a great early step in writing programs because it lets you explore how they might fail. (Again, some degree of failure is the normal case for programs; understanding and handling failures is often more achievable than avoiding them entirely.) More elegant revisions would identify the specific types of exceptions that the program encounters in practice and write thoughtful handling of each. Depending on the nature of the problem, good options might be logging the error, retrying the file transfer in case the error was temporary, or removing broken lines from the file and attempting to send the remainder while maintaining a record of the broken lines for subsequent human intervention.

Lines 43–59 construct the database query in raw SQL, pulling bar codes, titles, and unique identifiers from the entire collection. While the rest of the program could be used in any context, these lines rely on the specific ILS being used. Indeed, Verreau would write them differently today because III's new API would allow her to dramatically simplify this part of the code.

For a sense of how much easier and more readable an API, using the same programming language as its surrounding code, can be, have a look at the documentation for credit card processor Stripe. The right sidebar shows how to charge a user's credit card. On Stripe's end, this information all lives in a database that can be queried in SQL, but the API lets you use simple and Pythonic statements like `stripe.Charge.create()` instead of constructing the SQL query. Indeed, you need not know how the database is structured at all, and your code does not have to change if Stripe decides to change its database schema.

Stripe API documentation: Creating a new charge
https://stripe.com/docs/api#create_charge

Lines 61–66 connect to the database and provide a cursor we'll be able to use to examine the results. This cursor will let us step through the results line by line.

Lines 68–78 remove outdated files from our Novelist directory and create today's filename.

Lines 80–81 connect to the database and fetch the results.

Lines 83–92 open our new file and write a title line. The loop (indicated by `for r in rows:`) then writes one line per record from the database, each on a separate line (`\n` is the newline character). Once all records have been written, it closes the file.

Lines 94–104 attempt to log in to the FTP server and transfer the file. The `message` variable created here is the body of the e-mail we will send in **lines 106–114**.

This program demonstrates several best practices: comments, error handling, and descriptive variable and function names. The functions `strify` and `put_file` also demonstrate the usefulness of breaking logically coherent units of functionality into actual functions. By keeping them separate from the main body of the program, we make the overall logic more readable; the program reads like an outline, and we can dig into the specifics only as needed. The program is also easier to debug when you can isolate functionality and zero in on the parts that may need to be fixed. In a larger program, these functions could also be reused. For instance, if we needed to FTP our output file to several servers instead of just one, we don't need to rewrite all that code—we can just call `put_file` again. And if we found that `put_file` was so useful that we needed to use it in multiple programs, we could grow it into a library that could be imported by other programs, just like this program imports `psycopg2`, `os`, and `smtplib`.

Want to modify this program for use locally, and practice your Python skills while you're at it? Here are some things you might try:

- Replace the SQL with a query suitable to your ILS (ideally using an API).
- Have the program import all the local parameters (like `DB_NAME`) from a separate file, *and make sure you keep that file out of GitHub* so you don't inadvertently share sensitive data. (For instance, name it `parameters.py`, and add `parameters.py` to your `.gitignore`.) Alternately, have it harvest this information from environment variables (and add some error checking so that the program will exit if it doesn't have all the data it needs).
- Find out what kind of exceptions might actually be thrown by the `try/except` clauses, and handle them specifically.
- Add error handling in case the e-mail-sending fails.
- Think through what might happen if `NOVELIST_DIR` contains files you didn't expect (e.g., if it's the directory where text files for some other project are being stored) and what you can do about those risks.
- Identify some other case where you need to harvest data from your ILS and e-mail or FTP the results. Modify this script to do that instead.

Scripts in This Chapter

Robin Camille Davis's script
https://github.com/robincamille/ezproxy-analysis/blob/master/ezp-analysis.py

Joe Montibello's script
https://github.com/joemontibello/update-lockss

Esther Verreau's scripts
https://github.com/everreau/sierra-scripts

Notes

1. Robin Camille Davis, "Analyzing EZproxy Logs with Python," *Emerging Tech in Libraries* (blog), April 22, 2014, http://emerging.commons.gc.cuny.edu/2014/04/analyzing-ezproxy-logs-python.
2. Ibid.

Chapter 4

Patron-Facing Services

Overview

The scripts in chapters 2 and 3 focused on back-of-the-house functions: data quality and technical services workflows. In chapter 4, we'll talk about services patrons interact with directly. One of the wonderful things about coding in libraries is that it inherently breaks down silos: while librarian coders often emerge from technical services, they can be found anywhere, and their work can affect—and link—numerous library functions. A large fraction of library coders are working on library website user experience or other patron-facing services.

Many of these coders are driven by a motivation cited at least as far back as the 1998 version of Eric S. Raymond's landmark essay on open-source software, "The Cathedral and the Bazaar"—"scratching your own itch."[1] Librarians use their library's website, catalog, or other web services frequently, and thus encounter user experience (UX) frustrations that directly affect patrons. Rather than live with inadequate layout or features, they created functionality their system lacked by going outside that system.

Any web product that lets you add some JavaScript—even if you can only add it to the `<head>` and cannot edit the rest of the page—gives you an opportunity to make this sort of tweak. Therefore, JavaScript is the language of choice for most code in this chapter (sometimes augmented and simplified with the wonderful jQuery library).

Examples

The examples in this chapter fall into four categories: UX improvements, presentation of information in new contexts, LibGuides tweaks, and patron services outside of the website.

UX Improvements

Three librarians surveyed wanted to improve the search experience. Amy Wharton wrote a jQuery script to add autocompletion of database names to her search box, allowing users to more quickly find (and correctly spell) what they needed. Joel Marchesoni found that CONTENTdm allowed for Boolean search operators but only through hacking the URL; his ASP script allowed users to enter Boolean queries through conventional search and built the URLs accordingly. Chris Fitzpatrick wrote a CoffeeScript that harvests ISBNs from a page of search results from the Blacklight discovery interface, grabs the corresponding cover image from Google, and enriches the results page with the images—all in a mere twelve lines.

> *Chris Fitzpatrick's script*
> https://gist.github.com/cfitz/5265810
>
> *Blacklight*
> http://projectblacklight.org

Rachel Donohue wanted to simplify the process of creating accessible content. As her employer, the National Agricultural Library, a US government agency, has a section 508 compliance mandate, so accessibility is a must; however, Neatline, the Omeka plug-in it uses to create time lines for digital exhibits, doesn't generate compliant content. Her Ruby script makes it fast and simple to provide an accessible alternative.

> *Rachel Donohue's script*
> https://gist.github.com/sheepeeh/10417852

Two librarians, Matthew Reidsma and Jason Bengtson, wrote scripts to display the library's hours, including whether it is open right now. It's intriguing to look at these side by side because—while they're both JavaScript programs that display open/closed status on a library website—they're written very differently. Bengston's includes special-case handling for holidays, while Reidsma's covers only standard hours. However, Reidsma's encodes schedule information in a significantly simpler format, which enables him to write much more concise, readable code.

Matthew Reidsma's script
https://github.com/gvsulib/Today-s-Hours/blob/master/todayshours.js

Jason Bengtson's script
https://github.com/techbrarian/openchecker/blob/master/openchecker.js

This comparison illustrates how much of software engineering is driven by brainstorming all the special cases code might need to handle and making choices as to which ones are worth handling in your context. It also underscores the role that aesthetic sensibilities play. There's more than one right way to write programs—in a sense, anything that consistently produces correct results is right. However, authors have different stylistic preferences, and their intuitions about elegance can enormously impact the final result.

Repurposing Information in New Contexts

Just as technical librarians' jobs often break institutional silos, technical librarians' work can break content silos. One of the common frustrations of both library software architecture and library user experience is that information is held in separate systems even when it may be used in shared contexts; the three scripts in this section take information from where it's found to where it's needed.

Indeed, Michael Schofield phrased his JavaScript in terms of breaking silos: "I basically wrote a small API that [took] content that was previously silo'd on whatever platform—LibGuides, WordPress, etc.—and let it syndicate itself around our web presence without having to be duplicated." For instance, "if a patron is looking for business resources and we happened to have a scholarly speaker presenting about business, the API would suggest the event to the user."

Coral Sheldon-Hess also wrote something to syndicate content: in her case, a social media aggregator that pulled content from her library's various social media presences into a single home page display area. While they could have used existing WordPress modules, RSScache offered better performance. She modified this tool to be consistent with her library's branding. (Modifying existing code is often easier and less buggy than writing your own from scratch, as well as being a great starting point if writing your own sounds daunting.) This new website feature "helped us kind of institutionalize social media" by featuring it prominently and unifying previously disparate content. Seeing their new items pushing down older entries also gave the social media team an incentive to write more.

RSScache
www.rsscache.com

Jason Simon wanted to offer subject access to research databases. He had alphabetical lists hard-coded into HTML, but these were time-consuming and error-prone to update; since any given database could appear on multiple subject pages, changes had to be made in each page for every update. Instead, he stored information about which research databases served which subjects in a separate database and wrote a PHP script to pull information from this database and write it automatically into their subject guides.

Subsequently, this project grew organically to encompass new functions, ultimately becoming a "larger back-end homemade ERMS which made it a lot easier to manage subscriptions, statistics, pricing, holdings, etc., for both databases and periodical subscriptions." This points to something that's important to keep in mind if you're a new coder feeling overwhelmed by the size of open-source projects you've looked at—they didn't start out big! In fact, they may not have been started to tackle large problems at all. You can start by writing something small, and along the way you'll learn how you might want to expand it and the skills you need to do so. Conversely, if you know the thing you need to write is large, break it down into the smallest useful pieces and write them one at a time. Other people's software projects didn't spring fully formed like Athena, and yours needn't either.

Springshare Customization

Four survey respondents made changes to their institution's handling of Springshare products, particularly LibGuides. These changes spanned the use cases above—user experience changes and information reuse—but three of them are grouped here to show the variety of possibilities for augmenting this widely deployed product line. The fourth will be the subject of this chapter's deep dive.

Eric Phetteplace's library, Chesapeake College (a hybrid community college/public library), used a wide range of Springshare products for recording library statistics. However, the input forms didn't limit librarians' choices to the recording schema used at Chesapeake; as a result, invalid entries made it hard to analyze the data. Phetteplace installed Tampermonkey on staff computers, which allows for installing additional userscripts—snippets of JavaScript that function as browser plug-ins. He then wrote a userscript that validated form entries before submission, requiring staff to enter valid data. This made it much easier to analyze the data, which in turn helped the library make better choices about how to staff their desks; for example, it could see that it got more computer support questions early in the term and more reference questions near finals and assign desk coverage accordingly. It also helped the library to communicate more effectively with other departments about library usage and impact (see chapter 5).

Tampermonkey
http://tampermonkey.net

At Ohio University Libraries, staff had an array of subject-specific LibGuides and wanted to make sure students looking at the Course Reserves page knew about this option. While the OPAC allowed the library to insert links to LibGuides into those pages, doing so manually would have been prohibitively time-consuming at this school of over 20,000 students. Staff were also concerned that placing links to LibGuides in line with assigned course readings might irk faculty by lessening their control over the content of the course readings area. Instead, Carrie Preston wrote JavaScript (building on the extremely useful jQuery library) that automatically inserted a link to the LibGuides page, including the name of the relevant subject category, in a special block toward the top of the page. You can see it in action on Ohio University's ALICE catalog.

ALICE catalog
http://alice.library.ohiou.edu/search~S7?/rcoms/rcoms/1%2C26%2C29%2CB/frameset&FF=rcoms+4060&1%2C%2C2

Bohyun Kim had the inverse problem. Rather than needing to add LibGuides to course resources, she needed to add course resources to a LibGuide—in her case, hundreds of e-textbooks that were hard to find in the catalog. She had a student worker who was comfortable finding and organizing them but was not comfortable writing HTML, and she wanted to ensure that the end product was compatible with the library's custom LibGuides styling without onerous proofreading on her part. She wrote a web page in HTML and JavaScript (again, taking advantage of jQuery) where her student worker could enter metadata in a human-friendly format. The script then produced appropriately formatted HTML that the student could copy and paste into the LibGuide. You can see the end result, with dozens of e-textbooks alphabetized and properly formatted, on the school's Course E-Books web page. Kim also wrote an ACRL *TechConnect* post that walks through the code.[2] (There are only eleven lines of JavaScript!)

Course E-Books
http://LibGuides.medlib.fiu.edu/courseebooks

Bohyun Kim's script
https://github.com/bohyunkim/examples/blob/master/link.html

Services outside the Web

While most respondents were using code to affect either metadata or the website—that is, strictly computational objects—a few used code to improve services in other domains.

Matt Weaver (whose author name preprocessing script we saw in chapter 2) says, "We purchased a digital signage system from a reseller that didn't really do what we wanted it to do in the first place, and to get it to do something close would have meant a workflow would not have been manageable for one employee." The library wanted the system to display its meeting room schedule, but it could not queue up information to show at set times; staff had to manually change the sign message throughout the day. Weaver's Python script allowed staff to deposit files with event data into a particular folder whenever it was convenient for them to do so. It then pushed the information to the signage software at the appropriate times. Ultimately, "the code rescued a rather expensive, and unpopular project."

In addition, Mike Drake (Deputy Director, Tulare County Library, Tulare, CA) wrote a script to help his children's librarians give better, faster answers to questions like "Do you have any princess books?" In his words, "Some of the most popular books in the children's area are dispersed all over, in different collections. And, most of them are checked out. For example: Disney Princess can be in easy readers, picture books, or juvenile fiction; and filed under several different authors. Our OPAC will only allow us to check the location and status of each title one at a time; and it can be very tedious. I wrote a program that will search the OPAC over the web and return results only for titles that are available, in a single list,

sorted by collection/author. This list can be printed by the librarian, and then the hunt begins!"

While this can be read alongside the UX improvements scripts earlier as another example of transcending the limitations of the OPAC, from a patron perspective it's entirely different. The patrons probably never know that their librarian wasn't using standard library software or that anyone wrote code; most of them probably don't know what code is. They just know that the librarian was able to get them princess books without delay.

Deep Dive: LibGuides Organizer

Jeremy Darrington (Princeton University Library) wrote JavaScript to organize LibGuides to handle an information overload problem. He had some topics for which the library could offer lots of relevant resources, and he wanted to make sure the students could access them all. However, with so many boxes on a page, it was hard for students to navigate the options or get an overall sense of the holdings. He didn't want to clutter the page with too many tabs, either. Instead, he wanted to provide a sidebar table of contents listing all available sections and display only the currently active section so that users didn't feel overwhelmed. Users then could click on the table of contents to selectively reveal sections of interest.

You can see a screencast of the result of the page organizing script on the Princeton website. The page also provides clear, comprehensive instructions on incorporating the script into your own site. (It's based on the older version of LibGuides; LibGuides 2.0 natively incorporates a similar side nav option.)

Page-organizing script screencast
http://LibGuides.princeton.edu/content.php?pid=254621&sid=2824241

The code itself is available on the Princeton site. It's beautifully commented, making clear what each section of the code is doing, as well as specifying the CC BY-NC-SA license. Like a lot of JavaScript, it relies on some understanding of HTML and CSS; if you're rusty on those, pull up a tutorial or reference for them as well. Now, let's dive in!

Jeremy Darrington's script
https://thatandromeda.github.io/ltr/Chapter4.html

Lines 6–8 tell the browser that this function should operate on the document, once it's been fully loaded. They then initialize two variables (that is, create them and assign initial content). These are the lists where we will be storing IDs and titles of the various content boxes (made clear by the excellent variable names `$boxID` and `$boxTitle`). Right now they're empty, but we'll add content over the next few lines.

Lines 10–18 get the titles of the boxes. The CSS selector in **line 10** specifies the header elements of our content boxes. **Line 11** gets the actual text of the header; **lines 12 and 13** test whether it matches a given regular expression. (Regular expressions are ways of specifying patterns of characters; this one means "a number followed by a right parenthesis." This will match the beginning of each line in a list—`1)`, `2)`, and so on.) If the text and the regular expression match, the code strips off the matching part (the `1)`, `2)`, etc.) and adds the remaining text of the header to our list. If there's no match, it adds the entire text. We now have the text of the entries in our table of contents, with unimportant item numbers removed.

Was the regular expression strictly necessary? No; we could have simply added the entire text. Had I been writing this script, the first version would have done just that—and then, after testing it on some LibGuides, I'd have discovered that some table of contents entries had `1)` or `2)` in front of them and some did not. I would have decided that looked weird and added the regular expression to normalize the formatting. This is not necessarily how Darrington proceeded, but this sort of iterative code-test-debug-code process underlies many programs.

Lines 20–21 get the ID attributes of those same boxes and store them in the `$boxID` variable defined earlier.

Lines 24–27 find the box we're going to put the table of contents in. This is a box that was set up during LibGuides configuration, as detailed in the screencast. The HTML IDs that we'll use to find this box (via the selector in **line 24**) are defined by the LibGuides template. It starts out empty, but in **lines 25–27** we loop through `$boxTitle` and `$boxID`—the lists we defined earlier, containing the IDs and titles of our content divs—and add entries to our table of contents. Each time through the loop, we add one line that has a box title as text and that uses the box ID to construct a link. At the end, we have a table of contents. The remaining lines will make it work in the desired manner, hiding and showing page content depending on the currently active link.

Note that these lines assume that the first entry in the `$boxTitle` list corresponds to the first entry in the `$boxID` list —that is, they assume we harvested both from the same place. This happens to be true because the `div[id^="wrapbox"]` selector we use in **lines 10 and 20** to find those boxes always returns them in the same order, and (in those same lines) `.each()` loops through them in the same order each time, *and*

we can count on the lists `$boxTitle` and `$boxID` storing them in the same order we added them. Therefore, we can safely ignore order in this code. However, there are programs where that isn't the case (for example, in Python, lists are stored in order, but dicts are not; when you read a dict, you are not guaranteed to see information in the same order that you wrote it).

Lines 31–51 check the URL the browser is currently pointed at (this is the `location` in **line 31**) to see if it has an anchor at the end (like `#foo`; this is the `.hash` part of **line 31**). We'll do slightly different things in the scenario where it has an anchor and the scenario where it doesn't, using different code blocks (respectively, **lines 32–45** and **lines 47–51**). The curly braces signal to the computer where these code blocks begin and end; the indentation is optional but makes it much easier for humans to keep track.

If there is an anchor in the URL, we'll assume that the user has just clicked on one of the table of contents links (all of which have anchors) and hide or show content accordingly, using the code in **lines 32–39**. **Line 32** gets the ID of the desired box from the anchor link text (ignoring the `#` character at the beginning, which is used by the browser to interpret the URL but is not part of the HTML ID attribute). **Lines 33–34** can be ignored—they're commented out and thus presumably represent failed experiments from the process of writing the code. **Lines 35–39** loop through all the box ID numbers on the page. When they get to our desired `$boxNum`, they show the box, scroll the window to it, and add highlighting to its line in the table of contents to make it clear to the user what the currently active content is. For all other IDs, we hide the div to avoid cluttering up the interface. We're now done processing the scenario where there's an anchor in the URL, and the program will skip down to **line 53**.

If there isn't an anchor in the URL, we'll skip **lines 32–39** and instead process **lines 47–51**. In this scenario, we assume the user has just loaded the page (using its base URL) and should be shown the first content box with the first line of the table of contents highlighted. **Lines 47–49** hide all content divs except the first (LibGuides displayed them all by default). **Lines 50–51** highlight the first line in the table of contents.

To summarize, at this point we've done the following:

- collected information from our page that we'll need to build the table of contents and connect its entries to content areas on the page
- checked the URL to see what the user's currently selected content area is
- made sure the corresponding line in the table of contents is highlighted so users know where they are
- made sure the corresponding content block is shown and the rest are hidden to keep the screen from being cluttered with irrelevant content

Scripts in This Chapter

Chris Fitzpatrick's script
https://gist.github.com/cfitz/5265810

Rachel Donohue's script
https://gist.github.com/sheepeeh/10417852

Matthew Reidsma's script
https://github.com/gvsulib/Today-s-Hours/blob/master/todayshours.js

Jason Bengtson's script
https://github.com/techbrarian/openchecker/blob/master/openchecker.js

Bohyun Kim's script
https://github.com/bohyunkim/examples/blob/master/link.html

Jeremy Darrington's script
https://thatandromeda.github.io/ltr/Chapter4.html

Matthew Reidsma and Kyle Felker's 360Link Reset
https://github.com/gvsulib/360Link-Reset

Matthew Reidsma's snippets on GitHub
https://gist.github.com/mreidsma

Grand Valley State University Libraries scripts on GitHub
https://github.com/gvsulib

Matthew Reidsma's repositories on GitHub
https://github.com/mreidsma

All we have to do now is ensure that, if the user selects a new line in the table of contents, the highlighting shifts to that line, the old content box is hidden, and the new one is revealed; we accomplish this in **lines 53–69**. **Line 53** specifies that this code block is a function that is triggered whenever the user clicks an element whose class is `boxNav`. (This is the class name that Darrington applied to his table of contents entries in **line 26**.) In **lines 54–55**, we find the currently highlighted entry and remove the `currentNav` class (thereby removing the highlight styling). **Lines 56–57** find `this`—a special JavaScript keyword that here represents the element the user clicked—and add the styling that indicates it's the currently active nav entry. **Lines 58–69** then loop through the content areas in the LibGuide, showing (and scrolling to the top) the one that corresponds to the active table of contents entry and hiding the remainder.

And now we're done! When users load the LibGuide, it will show only the first (or active) content area, with the table of contents highlighted accordingly; when they click on the table of contents, the corresponding content will be displayed and the rest hidden. Now Darrington can add quite a lot of content to a LibGuide without overwhelming or confusing the user, as long as he organizes it into logical chunks.

What are some key takeaways from the code? First, clear comments are a great service. Because this code is organized into logical sections and each has an explanatory comment, it's clear what each section of the code is doing even if you don't speak JavaScript. This also makes it much easier to figure out where to look if you'd like to write similar code or change one aspect while keeping the remaining functionality.

Several lines of this code (e.g., 10, 20, 53) also illustrate that JavaScript is often very tightly bound to the HTML of the page it operates on. Changing a single class name or displaying content inside a different element breaks many JavaScripts, as they can no longer find the content they were meant to operate on. On the other hand, if you can control the HTML of a page, or at least have high confidence it won't change, JavaScript gives you a great deal of power. Once you know where to find the information you need (using CSS selectors), you can hide, show, move, and reformat it on the fly. By writing a custom stylesheet and using JavaScript to add or remove classes from that stylesheet as needed, you can (re)define a page's layout, appearance, and usability.

Better yet, you can do this even if you're working with a product that doesn't let you edit the `<body>` of the HTML but does let you insert CSS and JavaScript into the `<head>`. If you read the HTML thoroughly, you can generally construct CSS selectors that uniquely identify parts of the page you'd like to change; you can then write JavaScript to target those parts. Using this technique, Matthew Reidsma and Kyle Felker entirely redesigned Grand Valley State University's 360Link implementation. This let them not only improve design but also address concerns that had arisen during usability testing. For this and other examples of improving user experience through JavaScript, explore the GVSU Libraries' and Matthew Reidsma's personal GitHub repositories.

Matthew Reidsma and Kyle Felker's 360Link Reset
https://github.com/gvsulib/360Link-Reset

Want to modify Darrington's program for use locally and practice your JavaScript (and jQuery and CSS) skills while you're at it? Here are some things you might try:

- **Lines 32–39** assume that any anchor in the URL actually corresponds to an element on the page; they don't defend against the possibility that a user has edited the URL. What happens if the URL has an invalid anchor? If the outcome is bad, can you check for validity before deciding whether to run the code?
- Change the appearance of the highlighting applied to table of contents entries. This actually isn't a JavaScript question at all; the styling comes from the CSS rules defined for the `currentNav` class (in a separate file). Merely changing the CSS, without touching the JavaScript, can give you very different results.
- Write something inspired by this script that works with LibGuides 2.0.

Notes

1. Eric S. Raymond, "The Cathedral and the Bazaar," *First Monday* 3, no. 3 (March 2, 1998), http://firstmonday.org/article/view/578/499; Eric S. Raymond, The Cathedral and the Bazaar website, February 18, 2010, www.catb.org/~esr/writings/cathedral-bazaar.
2. Bohyun Kim, "Playing with JavaScript and JQuery—The Ebook Link HTML String Generator and the EZproxy Bookmarklet Generator," *TechConnect Blog,* April 8, 2013, http://acrl.ala.org/techconnect/?p=3098.

Political and Social Dimensions of Library Code

The original concept of this report encompassed only code samples and analyses and learn-to-code resources. However, survey responses discussed the political and social dimensions of library code so often as to make them inseparable from the technical dimensions.

Sometimes, this was positive. Matt Weaver's digital signage code (chapter 4) "rescued a rather expensive, and unpopular project"; along the way, he "learned a lot about the emotional impact a technology project can have across staff." Coral Sheldon-Hess's RSS-cache code (chapter 4), which enabled the library to display its diverse social media presences on its home page, incentivized staff members to write more social media content because they were excited to see their new material on top. Hillel Arnold's Captain's Log was written specifically to solve a communication problem, giving staff from different reading rooms an easy way to leave each other notes.

Hillel Arnold's Captain's Log
https://github.com/RockefellerArchiveCenter/captains-log

Respondents wrote of positive emotional impacts on themselves, too. Evviva Weinraub Lajoie discovered "I was capable of building something that thousands of people across the world use to access electronic resources, which was really quite powerful and empowering for me." Several people wrote of their pleasure when their code or documentation helped coworkers to advance their own skills. Jeremy Darrington (chapter 4) said, "I like that coding makes me feel that I'm not helpless, that I can solve some of the problems I face with tools at my disposal."

On the other hand, not all emotional responses were so positive. Many library coders spend a significant amount of time trying to cultivate buy-in, educate their colleagues about technology, or work against siloed organizational structures as they produce inherently cross-departmental work. Code can challenge hierarchies and change workflows, leading to resistance. And, as one librarian writes, "there are folks out there who will hold on to their assumptions about how patrons use library tools no matter what data you show them. (And a corollary, if your data goes against assumptions that are necessary for the survival of a way of thinking or a business, look out. Folks will get NASTY.)"

Coding in libraries often requires the political skills to generate buy-in, surmount institutional barriers, and navigate relationships with management who don't understand what you do. Managers who do understand, or are sympathetic to, coding may face similar challenges on their supervisees' behalf. This chapter outlines issues respondents faced and techniques they used to support and advocate for their projects.

Library Coders' Job Descriptions and Realities

One complication for many library coders is that their job descriptions don't necessarily involve coding. They may have duties that can be achieved far more quickly and effectively with code than by traditional means, or indeed that require at least occasional code editing to be accomplished, but coding is nowhere in the job description. As Carrie Preston puts it, "Certainly my supervisors in my earliest positions never conceived of

my job as being 'about coding,' and I think my activities remained largely mysterious and unfathomable in their eyes."

In some cases, this can make professional development and managerial support hard to come by, even when management recognizes the quality of employees' output. Other librarians, like Angela Galvan, find that "My job description and what I actually do all day are increasingly disconnected things." This may result in a tacit, laissez-faire kind of support, as long as the required work is getting done somehow. On the other hand, a substantial minority of librarians surveyed found that coding became an official part of their jobs, incorporated into subsequent job descriptions, as management recognized its value. For example, Carrie Preston found that "eventually some other members of the cataloging department began to use some of the scripts I wrote, and batch editing and batch loading of bibliographic data (which often involves some coding) did become a formal job responsibility." Josh Westgard is now in a job that is about half coding because he "advocated for the automation of many previously manual tasks."

Across the board, librarians with tech-savvy managers had an easier time getting support for their coding activities (whether formal, like courses, or informal, like time to code at the office as long as the work got done). While many librarians did not indicate whether their managers also had coding skills, 100 percent of those who said their managers were tech-savvy also said they had received some professional development support. Similarly, 100 percent of the coding librarians who are also managers mentioned offering professional development support for coding skills to their supervisees. Indeed, several respondents who are not managers create and run technology workshops for their coworkers.

Buy-in

One issue that came up frequently was buy-in. Although library coders are often solo, and individuals can do a lot with code, it's hard to turn code into a useful service for the library without cooperation. Access to testing and deployment servers, authority over website content, and time for developing and maintaining projects all need institutional support. Numerous respondents talked about both strategies for gaining that support and limitations when they didn't get it.

Bohyun Kim recommended Tito Sierra's exceptionally useful Project One-Pager. This is a document written collaboratively by stakeholders in order to come to a shared understanding of a project. It specifies key information like project scope (including what's out of scope), deadlines, and participants. Not only does this shared understanding promote buy-in, but it also helps everyone see when a project is finished and get the morale boost that comes along with successful project completion.

> *Project One-Pager*
> www.slideshare.net/tsierra/the-projectonepager

Coral Sheldon-Hess has also achieved buy-in through documentation. She worked with the web team to write up guiding principles for web design, content, and process.[1] Through researching this document, her team reached a shared understanding of best practices; by writing them down, they generated a reference point for the library as a whole. Sheldon-Hess shared her thoughts on this process in a 2013 LITA Forum presentation.[2]

Documentation can be useful for buy-in throughout a project life cycle, too. Terry Brady notes that it "can allow users to learn at their own pace and to revisit the documentation as often as needed. This is a great approach to achieving buy-in for a solution."

Other respondents achieved buy-in through directly demonstrating the value of library code. Robin Camille Davis did a live coding demonstration of her EZproxy script (chapter 3), and "the people I was with at that demonstration (the systems librarian and the systems manager) were very impressed and got that 'We can do ANYTHING with Python!' gleam in their eyes." Other respondents recommended pilot projects. Often it's hard to talk about what code can do in the abstract, but people respond strongly to prototypes.

Eric Phetteplace (chapter 4) found that his code let his library do a better job of demonstrating its value on campus. Once his form validation code ensured that they were collecting sound reference statistics, they could see that 60 percent of their questions were about technical help. This helped the library advocate for its role in computer literacy and challenge assumptions that it dealt only with books.

Several respondents, particularly in technical services, were able to make strong arguments about the time-saving value of code. We saw in chapter 2 that Becky Yoose saved one to two weeks of cataloger time every year by scripting a repetitive task. Similarly, Carrie Preston noted that "as my department's then-only regular user of [OCLC Macro Language] scripts, I had several times the cataloging productivity of any other cataloger in that department, even while spending a smaller percentage of my time on cataloging." And Annie Glerum (chapter 2) found "that even with reduced staffing, it is possible to achieve both quality and timeliness."

And, when all else fails, some coders go rogue. One noted, "I have learned intentionally breaking systems

known to be fragile is a good way for me to gain the permissions I need to do the work I'd like." Of course, it's always better if the library administration and IT are on board! But coders are by definition inclined to make (and break) things; they tend to find places to exercise their skills or grow deeply frustrated if they can't. One respondent was irked that his code, which made it easier for website users to access digital content, had limited impact because of inadequate support for digitization. He "learned that the impact of code can be limited by administrative lack of resolve, understanding, and focus." And at least eight of the fifty-three survey respondents have changed jobs between answering the survey in spring 2014 and this writing in November 2014. While their reasons vary, this does point to how hard it is to keep coders satisfied if they don't have scope for building things.

Finally, several respondents raised the issue of mission-criticality, but without agreement. Some said that coding mission-critical projects is a good way to achieve buy-in and sustain motivation; others noted that working on key projects is a good way to justify professional development support. However, as Becky Yoose says, "Do not start coding on a project that's mission-critical because that is a good way to fail." She and others recommended building small pilot projects to demonstrate value and build skills before tackling critical services.

Institutional Barriers

Many librarians were missing some important kind of institutional support for learning and writing code. These missing pieces fell into three broad categories:

- lack of support for learning
- lack of support for doing the work
- lack of collaboration

One librarian who hadn't received support for learning to code said, "Coding is really useful, but you're just supposed to know it." Many respondents reported learning to code on their own time, outside of work. Some librarians had difficulty convincing employers to let them spend professional development funds on code learning; indeed, one manager could not secure support for a supervisee because the higher-ups "didn't want her to learn because that would mean that they would have to bump her up a classification level." Other librarians simply didn't have enough professional development funds to cover high-impact learning opportunities like formal classes or conferences. In many cases, the best form of support described was benign neglect—managers who didn't know what these librarians were doing but wouldn't stop them from coding as long as things got done somehow.

> "The institution, however, only gives me $500 in professional development funds per year so although the resources are here to learn whatever I want, any structured learning I want to do comes out of pocket. As it is, the institution is not paying for me to speak at conferences related to my job directly unless they are planned for 12+ months in advance, and I do not have the time to play institutional Calvin Ball with a budget office that doesn't know how libraries work."

Other librarians who already have the skills to code described environments that were hostile to doing that sort of work. Lack of access to servers or permission to install software is a recurrent problem; one librarian says, "I mean, seriously, there is one section where I parse XML with regular expressions. But at the time I didn't have access to install libxml on the system!" Another librarian, whose resume is code-heavy and who was hired in a systems role, found that his managers expected him to use only proprietary software, even when open source options (which he had the expertise to implement) would have been better or cheaper. They also expected him to call vendor support rather than figuring out problems on his own. In one extreme case, a librarian who spends well over half his time on coding and related tasks is at an institution where most units (including his) are explicitly banned from touching code. His middle management recognizes how valuable his work is and finds ways to protect the time while keeping upper management in the dark.

Unsurprisingly, isolation is a major issue for many coding librarians. They may be the only ones in their department, or even their library, who know how to code. Organizational and cultural barriers may prevent them from collaborating with IT or with librarians in other institutions. This is particularly unfortunate because, contrary to popular stereotypes, coding is a profoundly social occupation. Most programs of any size are written by teams; most learning takes place through shoulder-surfing, code review, and other forms of pair programming or mentorship. This is especially true for advanced programming skills, like making good decisions about the overall organization of programs, and for everyday craft knowledge, like discovering good editing and debugging tools.

One librarian wrote, "We had a systems librarian who was very much the fabled hardcore geek of yore, who had basically single-handedly programmed much of the infrastructure we depended on (e.g., ERMS, website CMS, etc.) but was known to only work on a problem if he believed it to be important (not many external suggestions—even from the [University Librarian]—passed this test)." There are good reasons for people to be territorial about code—it's important to have high standards of quality and maintainability for

mission-critical applications—but at this extreme, the whole institution is held hostage because only one person understands the code. The respondent taught himself enough code to solve some problems that the systems librarian wasn't interested in fixing, but this was an enormous lost opportunity for knowledge transfer. Furthermore, since he is self-taught, he recognizes that he doesn't "have any of the best practices that make code sharing easier." This, in turn, will make it harder to collaborate with any future coding coworkers.

Of course, many librarians who code do not have even one coworker they can talk to about code. In their case, the ability to share code and participate in open-source projects is critical for skills development. Many libraries, however, do not have formal policies on whether code can be shared and may not have an informal consensus; some are actively hostile to open source. Dale Askey outlined diverse reasons for this hostility, including perfectionism, fear of ongoing support responsibilities, and misunderstanding of open source.[3] The upshot, however, is untold wasted hours of duplicated work and limits on librarians' ability to increase their own skills.

Bohyun Kim (who ran into this challenge herself) recommends thinking about open source and intellectual property from the very start. Coders are often in fairly junior roles and may not have the ability to negotiate with their institutions; however, it's good to identify what approvals you would need to release your code and who owns it. If you can identify, or create, a release procedure, your code will be more useful and personally rewarding.

Notes

1. Anna Bjartmarsdottir et al., "Plan for the Web Presence," UAA/APU Consortium Library, November 10, 2013, http://connect.ala.org/node/213992.
2. Coral Sheldon-Hess, "Getting Buy-in on User Centricity," presentation. LITA Forum, Louisville, KY, November 7–10, 2013, www.slideshare.net/csheldonhess/lita-forum.
3. Dale Askey, "Column: We Love Open Source Software. No, You Can't Have Our Code," *Code4Lib Journal*, no. 5 (December 15, 2008), http://journal.code4lib.org/articles/527.

Chapter 6

Learning to Code

Whenever I speak on library code issues, one of the first questions I get is, "How can I learn to code?" If that was your question, this chapter is for you. I'll discuss respondents' recommendations for learning strategies and resources. I'll also cover the various forms of workplace support librarians have received in learning to code so that you know what to ask of your manager, or what to provide if you are a manager.

Learning Strategies and Resources That Coders Recommend

I asked survey respondents what they would recommend to people who'd like to learn to code. The recurring themes were these:

- find a project
- rely on Google and existing code
- write documentation
- persevere
- find a mentor

Of these, *finding a project* is the most important. It doesn't matter if it's for work or for fun, though it will be easier to get professional development support for work projects; it just has to be important to you. Having a goal you're committed to will help you persevere through the inevitable challenges (see below). It will give you a sense of accomplishment when you make progress; it may even have real-world impact, which is tremendously motivational for many coders. It can also provide natural answers to questions like "What programming language should I learn?" and "What do I need to learn next?"

What sort of project? You may already have one in mind, in which case, start there! If not, automate a repetitive task, simplify a bothersome workflow, or improve some element of user experience. Or, of course, take on one of the projects in this report! Most of them can be accomplished in under a hundred lines of code; you'll need a solid grasp of programming fundamentals, but you don't need a deep grounding in computer science or years of experience. Write one from scratch, rewrite one in your preferred language, or modify one to work better for you; the scripts in this report are intended to be a springboard for you. Whatever you choose, make it as small as possible (or break it down into small parts) so it doesn't get too overwhelming, and feel free to incorporate working code snippets you find online. The sooner you can get something interesting working, the sooner you'll feel rewarded and capable.

This brings us to the second piece of advice, *rely on Google and existing code*. Modifying existing code is not cheating! There's a good chance someone else has already written code to do most of what you want; the ability to read and edit others' code can get you a long way, even if you never write your own programs from scratch. Even experienced programmers regularly look up syntax details and copy and paste code snippets from around the web. Googling for something like "[programming language] [problem keyword] example" will often turn up helpful code samples and StackOverflow advice. Spending some quality time browsing library coders' GitHub repositories can yield lots of useful code and inspiration, too. The Code4Lib wiki page "Libraries Sharing Code" is a good starting place. Many of the people cited in this report have GitHub repositories as well.

StackOverflow
http://stackoverflow.com

Libraries Sharing Code
http://wiki.code4lib.org/Libraries_Sharing_Code

Not familiar with GitHub? You don't need an account to browse and download code. However, it's more useful once you have an account so that you can fork repositories (i.e., make your own copy to edit) and master a few basic commands. The LITA Library Code Year Interest Group has a hands-on tutorial available.

Learn GitHub tutorial
https://github.com/LibraryCodeYearIG/Codeyear-IG-Github-Project

Google, StackOverflow, and (to a lesser extent) GitHub work as learning tools because people have invested time in *documentation*. Pay it forward! Writing up your own learning process can be helpful to those who come after you—notably including yourself in six months, when you've forgotten everything you were thinking today. Organizing your thoughts well enough to write them is a good self-teaching tool. Additionally, many open-source projects want help with documentation as well as code, and this can be an easier route than code to begin contributing. Read the project guidelines, look for a bug tracker with open documentation bugs, and make things better while your memory is fresh. Finally, writing documentation increases the chances that others will build on your work; seeing others succeed because of your work can be motivational and rewarding.

Step four: *persevere*. Learning to code is hard! You must devote a lot of time to it. Also, you'll make mistakes, and some of them will be hard to debug. Beginners often think this means they don't have the aptitude, but they're wrong; coders at all levels constantly run into challenging bugs. As Kate Roy says, "There is no mastery, there is no final level. The anxiety of feeling lost and stupid is not something you learn to conquer, but something you learn to live with."[1] Or, as Cecily Carver notes, in an outstanding Medium article on what she wishes she'd known as a new coder, "I've found that a big difference between new coders and experienced coders is faith: faith that things are going wrong for a logical and discoverable reason, faith that problems are fixable, faith that there is a way to accomplish the goal."[2]

People don't talk enough about emotion in learning to code. They talk about languages and tools and MOOCs and books, but not about feelings: about the intense ways code learning can push us into impostor syndrome, can make us feel we don't belong (particularly if we're not a 19-year-old white male in a hoodie), can make us feel frustrated and anxious and overwhelmed. You probably will feel that way if you learn to code, and that's okay. One of the biggest things, in fact, that learning to code will give you is a toolbox for handling those feelings and the knowledge that you can do the work even if you're intimidated.

"Very recently, a cataloging support staff member presented me with a printout of one of my old OCLC Macro Language cataloging scripts. The script produced a template MARC record for a title from a specific e-book collection, and she had edited it, largely correctly, to make the record it produced comply with new Resource Description and Access cataloging practice. She had 'discovered' programming by way of one of my scripts—this was very thrilling to me!"
—Carrie Preston

This, however, is a big reason that it's good to *find a mentor*. Mentors are great for answering technical questions and for telling you about tools and best practices that may not be written in books. But they're also great for holding your hand, cheering you up, and bolstering your self-confidence.

"You have to keep persisting. This is very different from writing a LibGuide or a handout."
—Bohyun Kim

Where do you find one? If you have a friendly, technically skilled colleague at work or a nearby institution, that's ideal. Some institutions (e.g., the George Washington University and the University of Maryland) have even started regular code-learning groups for their librarians. If you can't find a nearby colleague, the numerous technology-focused library conferences are great places to meet people. Nonlibrary technology can also be a good place to look. Many technical groups organize on Meetup.com; look for nearby meetups focused on your technology of choice. Be aware, though, that not all are beginner-friendly, and some can be downright hostile to women or people of color; look for groups that have outreach events, codes of conduct, or other clear commitments to hospitality. There are also technical groups focusing on outreach to specific populations that may be relevant to you, like PyLadies, PyStar, RailsBridge, and Trans*H4CK. All of these groups (plus ones focused on outreach to children, like Black Girls Code) are constantly looking for meeting space; if your library can offer some, that's a great way to build bridges to your local technical community, too.

Meetup.com
www.meetup.com

PyLadies
www.pyladies.com

PyStar
http://pystar.org

RailsBridge
www.railsbridge.org

*Trans*H4CK*
www.transhack.org

Black Girls Code
www.blackgirlscode.com

Finally, while in-person mentors are generally better, it's okay if you don't have access to them; the mailing lists and IRC channels for Code4Lib, LITA-L, LibTechWomen, and the like can expose you to current thinking and give you a place to ask questions. LibTechWomen has been running Code Club discussion groups; it's easy to set one up yourself by following Saron Yitbarek's advice.

Saron Yitbarek, "Reading Code Good"
http://bloggytoons.com/code-club

You may have noticed there's one question many beginners ask that I didn't answer here; to wit: "What language should I learn?" That's because there's no one answer to this question. Survey respondents wrote library code in fourteen different languages. The best language for you to learn depends on your personal taste, whether you have ready access to a community of experts, and above all the project you want to write. If you're modifying existing code or participating in an established open-source project, the choice of language is already made. If you're starting from scratch, your choice of project still influences your choice of language; for instance, web development probably means JavaScript, and MARC processing wants a language with an established MARC library, like ruby-marc, Python's pymarc, or PHP's File_MARC. Look for projects similar to the one that you want to do (including the projects in this report) and use their language choices as a guideline.

Finally, what tools should you use for learning? Google and Codecademy came up frequently in survey respondents' recommendations. While they have value (and Google is indispensable), as a teacher of code to librarians, I'm skeptical of unstructured and unsupported learning experiences. Because there's so little formal pipeline for teaching librarians to code, those librarians who do are, almost definitionally, the ones who do well with self-teaching, and their recommendations demonstrate a certain survivorship bias. I believe many librarians who aren't already coding, but want to, are more likely to succeed with a more structured, social experience. I've also been more generally impressed with the curricula in O'Reilly books than in free online courses; whatever your language of choice is, O'Reilly almost certainly publishes an introduction.

ruby-marc
https://github.com/ruby-marc/ruby-marc

Python's pymarc
https://github.com/edsu/pymarc

PHP's File_MARC
https://github.com/pear/File_MARC

Other specific resources recommended by respondents include:

- *The Art of UNIX Programming*, by Eric S. Raymond, https://openlibrary.org/works/OL6036022W/The_art_of_UNIX_programming. Many librarians find that command-line tools are even more useful than programming languages.
- *_why's (Poignant) Guide to Ruby*, a *sui generis*, part-cartoon introduction available free online, http://mislav.uniqpath.com/poignant-guide/book.
- *Python Programming in Context*, by Bradley N. Miller and David L. Ranum.
- The Pragmatic Studio, "Ruby Programming," online course, $132 with discounts and free trial available, http://pragmaticstudio.com/ruby.
- Lynda.com courses, www.lynda.com. In my experience, these are somewhat advanced for beginners, but excellent if you have a bit of prior experience, or good mentors; many libraries have a subscription.
- Google's Python Class, https://developers.google.com/edu/python. This resource is also best suited for people with some background; it is free, with good practice exercises.
- Formal courses available at your institution or in your area. These will probably be more theoretical than many librarians want and will likely not address library use cases, but taking even one will make it much easier to get mileage out of free resources.

It's also worth noting that several respondents said you should *not* try learning to code—or, at least,

"For the, the big thing was *find the right introduction*. There are a lot of guides for learning to code around, many of whom assume this or that reason why you might want to program, or start with the assumption that you have pre-existing knowledge of how to program. I learned to program from *_why's (Poignant) Guide to Ruby*, and I think this sentence is the very moment it clicked: 'You will be writing stories for a machine.' Coding as creative act, as artwork. Not algorithms or math or business rules. That caught my attention, and that got me going."
—Misty De Meo

that you should do it only if you're genuinely passionate about it and not just to check off a line on your resume. They indicated that people without this passion either would not succeed or would not become very good coders (and they felt that the world does not need more low-skilled coders). I agree in part and disagree in part. Coding is challenging enough that commitment is necessary; if you don't have that commitment, by all means spend your time on other things—there's no shortage of skills that will enrich your life and work! And becoming a deft, insightful coder is a full-time pursuit, and thus out of scope for most librarians. On the other hand, as we've seen in this report, you don't need to write large-scale, polished, reusable software in order to get big benefits from learning to code. Automating a task with a few dozen lines of code can save you many hours in a year. Even if you're a barely adequate coder, you can spend those extra hours being a fabulous original cataloger or research consultant or department head, employing human judgment and doing tasks the computer can't.

Workplace Support

Because learning to code can be time-consuming—and because librarians' code skills can be so beneficial to their institutions—it is both helpful and relevant for librarians to receive professional development support in learning to code. I asked respondents what, if any, workplace support they had received; I also asked managers what, if any, they had provided or would provide.

Answers varied significantly. While managers who code understand uniformly the value in supporting this skill, not everyone is lucky enough to have such a manager. Among institutions that do support code learning, funding and policy vary. Among survey respondents, the gold standard was set by Evviva Weinraub Lajoie at Oregon State University Libraries & Press. She provides employees with twenty hours per month of learning time, at least one conference per year, access to paid online tutorials, and even structured internships. Other libraries can't offer this level of support, but at least provide informal mentorships, code review, and the like.

Unfortunately, some librarians have no support, or even face active hostility. Some institutions simply don't have pertinent checkboxes on their paperwork, and it's hard to pertinent the relevance of these skills to a faceless bureaucracy. Two managers were unable to secure coding skills development for interested supervisees because their institutions did not want to reclassify them into higher salary categories reflecting those skills. And, as we saw in chapter 5, one librarian who spends a significant amount of time coding is doing so without upper management's knowledge; in that institution, only people belonging to other, explicitly technical, units are allowed to code. (Middle management "works pretty hard to keep me writing code as much as possible, even letting me out of some regular meetings because they know I can contribute more if I'm tickling a keyboard," says this librarian, who will remain anonymous for obvious, though distressing, reasons.)

Tech-savvy managers uniformly recognize the value of these skills and are willing to support them. Not all of them have supervisees who are interested, and the availability of funds varies, so the specific support provided does also. However, types of support that managers provide include:

- time: finding ways for planned projects to include learning new technologies, setting aside time for learning and experimentation, defending this time to upper management
- books
- software licenses
- root privileges, development sandboxes, testing servers, quality hardware: in short, the ability to install and experiment with software
- conference attendance: supported in time, money, or both
- workshops: some paying for attendance, others teaching them personally
- regular study groups, such as the one at the University of Maryland libraries or the George Washington University code reading group
- courses: online (such as Lynda.com, Code School, RailsCasts, Treehouse) or face-to-face, through tuition remission in the case of academic libraries
- code review
- mentorship
- formal internship programs
- making coding skills part of supervisees' performance goals, which helps justify other forms of support

University of Maryland Libraries Coding Workshop
https://github.com/umd-coding-workshop/website/wiki

Lynda.com
www.lynda.com

Code School
https://www.codeschool.com

RailsCasts
http://railscasts.com

Treehouse
http://teamtreehouse.com

Conclusion

Throughout this report, you've seen how librarians use short programs to make their work lives better in concrete ways, the opportunities (and obstacles) posed by code, and strategies you can use to start learning or to upgrade your skills.

Now it's your turn! Pick a project, find a class, put together a study group: whatever your next steps are, get started.

Whatever you do, you can always find source code for the projects discussed in this report, plus others that didn't fit—including the source code for the Django app I wrote to keep track of my own survey data—on the companion website. If you have a project you'd like to share—particularly one you wrote as a result of reading this report!—I'd love to feature it there as well; instructions are on the site.

Companion website
https://thatandromeda.github.io/ltr

Notes

1. Kate Ray, "Don't Believe Anyone Who Tells You Learning to Code Is Easy," TechCrunch, May 24, 2014, http://techcrunch.com/2014/05/24/dont-believe-anyone-who-tells-you-learning-to-code-is-easy.
2. Cecily Carver, "Things I Wish Someone Had Told Me When I Was Learning How to Code: And What I've Learned from Teaching Others," Medium, November 22, 2013, https://medium.com/@cecilycarver/things-i-wish-someone-had-told-me-when-i-was-learning-how-to-code-565fc9dcb329.

Notes

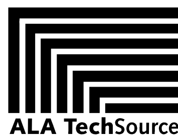
Keep up with

Library Technology REPORTS

Upcoming Issues	
May/June 51:4	**Library Services Platforms** by Marshall Breeding
July 51:5	**Altmetrics** by Robin Chin Roemer and Rachel Borchardt
August/ September 51:6	**Open Access Journals** by Walt Crawford

Subscribe
alatechsource.org/subscribe

Purchase single copies in the ALA Store
alastore.ala.org



alatechsource.org

ALA TechSource, a unit of the publishing department of the American Library Association