Connected Learning

Exploratory Programming in the Classroom

Angela Chang (Emerson College, Massachusetts Institute of Technology)

Abstract: Exploratory programming is an open-ended learning approach to teach coding in small groups using open-source tools. We describe how students used this hands-on approach to learn programming at a liberal arts college. In 1 semester, students progressed beyond basic programming concepts to independently extend their knowledge. Students used freely available, open-source tools, and online learning resources to create surprisingly complex and varied novel investigations of social, political, and cultural issues. They readily used their skills to appropriate different technological platforms or project outputs. Using creative coding, students learned to engage in programmatic thinking irrespective of specific technology implementations. They also participate in supportive programming communities. This approach costs little to implement, supports nontraditional programming students, and teaches real-world programming techniques.

Introduction

Nontraditional (or casual) programmers are on the rise (Bau, Gray, Kelleher, Sheldon, & Turbak, 2017). Besides computer science classes, many resources (books, online courses, web tutorials, and support forums) are freely available (Shen, 2014). Despite these offerings, students might still have trouble learning to program (Wiedenbeck, 2005). We propose an exploratory programming approach toward coding literacy. Some might be merely curious about programming, not necessarily in becoming professional programmers.

Background

Exploratory programming (Montfort, 2016) combines creative coding and humanistic inquiry. Like traditional STEM teachings, this approach uses hands-on activities and project-based outputs to guide progress. However, exploratory programming differs from existing pedagogies by emphasizing learning together in small groups, interacting with vibrant professional communities, and focusing on open-ended creative output. Students sample diverse technological experiences provided by programming and tinker to craft their own experiences.

Case Study: Code, Culture, and Practice Class at Emerson College

Emerson College is a communication and liberal arts institution situated in metropolitan Boston, Massachusetts. Although no computer science or engineering degrees are offered, three coding classes are available. Two of these classes covered technical platforms: Introduction to Interactive Media (HTML, CSS, JavaScript) and Introduction to Game Design (Unity). Like traditional computer science classes, these classes target a specific language or technology platform.

Code, Culture, and Practice teaches basic programming as a tool for humanistic thought. Although Python is used, students get experience using multiple programming languages and platforms. Offered by the Liberal Arts and Interdisciplinary Studies Department, the class caters to upperclassmen with little or no prior programming experience. We report observations from classes that occurred in Spring 2016, Fall 2017, and Spring 2017.

The class was oversubscribed for two out of three semesters. A total of 66 students enrolled. The average class size was 22 students, with roughly equal female and male attendees (although students at this college may prefer to use gender-neutral descriptions). Figure 1 describes the class in bullet points.

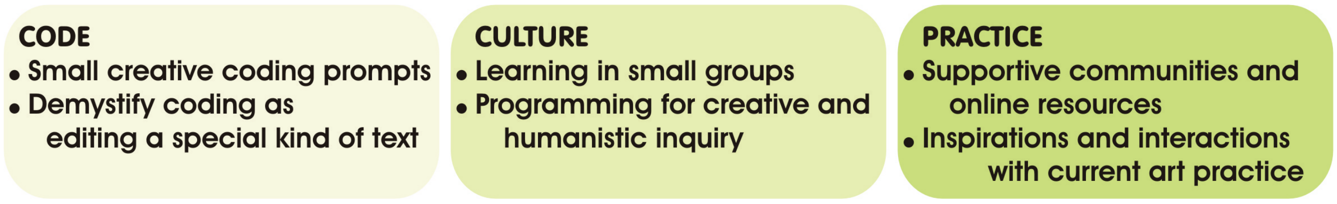

CODE
• Small creative coding prompts
• Demystify coding as editing a special kind of text
CULTURE
• Learning in small groups
• Programming for creative and humanistic inquiry
PRACTICE
• Supportive communities and online resources
• Inspirations and interactions with current art practice

Figure 1. A summary of Code, Culture, and Practice, an exploratory programming approach.

Exploratory Programming in Practice

The first class aimed to convince students that they possessed all the necessary tools to code on their own machines. Before the initial class, students were asked to download and try a coding-friendly text editor. Plain-text editors that display line numbers are not included in operating systems, but they are essential for basic coding. Once installed, they were ready to begin.

Poetry generators (Montfort, 2014) provide a fun way to reveal the inherent coding environment within their laptops. Students read about the historical roots of creative computing (Montfort, 2014). After experiencing these poetry generators, they were shown how to download the JavaScript source. Without any detailed coding instruction and merely by substituting certain words, the content of these computational poems was appropriated. They could open the file using their favorite browser to see the immediate effect of their edits.

Through tinkering, students soon learned what types of changes they could make without breaking the code. They began to recognize patterns in text and functionally distinguish data from code. Overcoming errors was part of the process, and they were shown how to undo their changes, start over, or navigate auto-highlighted text to find errors. There was safety knowing that nothing catastrophic could happen from this process. The satisfaction of being immediately able to test their changes locally within the first class showed them that programming was feasible. After customizing the content, they read the output aloud. Presentation was seen as a way to explore the potential design space together. The diversity of customized computerized poems was noted and appreciated.

This exercise initiated students to the process of iterative development (Figure 2) used by all developers. Students could observe parallels between the familiar process of editing text and modifying code. The class adjourned with a challenge to continue modifying their poems at home.

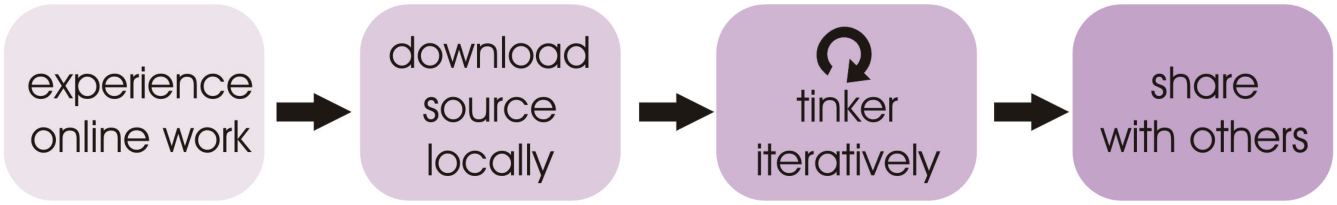


Figure 2. Creative iterative development process.

Promoting a Supportive Environment With Small Groups

One of the instructor’s main tasks was to foster a sense of community so students felt comfortable working closely. Short lectures took the form of live-coding (Rubin, 2013). Small code examples were used to demonstrate coding concepts. Then, students worked in groups (with a maximum of five people) to try the examples together. Each class period required a personal deliverable and group output. These frequent outputs helped to keep students focused on individual and shared productivity.

In small groups, students shared struggles or triumphs more willingly than out in the open. Meanwhile the instructor checked in with each group to suggest approaches or hints. Small groups allow each individual member to contribute more actively than in a larger group. Assisting each other became part of the lesson. Students continued their conversations outside of class (e.g., through messaging, phone, or meeting up in person). They often created shared documents and repositories (Ito, 2013).

Learning to Code by Tinkering With Small Programs

Subsequent classes covered programming fundamentals by manipulating code directly. We purposefully examined different languages to show that programming logic is common across languages. Students compared similarities between JavaScript and Python poetry programs. Then, they explored calculations on data types to get a feel for each language’s syntax and structure. Jupyter notebooks preserved the progression of steps toward developing functions and were annotated so students could proceed through each notebook at his or her own pace (Shen, 2014). These notebooks also enabled the instructor to identify individual student problems.

Specific, short code examples (no more than five lines usually) were carefully chosen to introduce basic programming concepts. These textual toys referenced the work of the Oulipo, an inventive group that pioneered many creative literary experiments (Motte, 1986). Short Python snippets were explained by reading the code verbally. Flow diagrams illustrated data calculations. These programs could also be reused on voluminous data sets. Students could revisit this code to analyze real-world data streams from social media, online news, and government websites for their final projects.

In the last third of the semester, the class covered visual display programming (with Processing IDE). Object-oriented code was covered, allowing multiple programmers to combine classes to create an experience together. Groups modified games, with each member customizing a part of the interface.

Coding as a Tool for Creative Inquiry

Students were repeatedly shown the workings and intentions behind media works. Then, they were prompted to describe how they might modify the works to suit. For example, when learning string manipulation, students were asked to select two pieces out of 100 from *Exercises in Style* (Queneau, 1981). They examined the aesthetics linguistically and programmatically. For example, they might correlate exclamation marks with abruptness. These small challenges aimed to give students a curious context to think about (and experiment) with code outside class. They worked on iteratively creating, just like professional artists (Kleon, 2012).

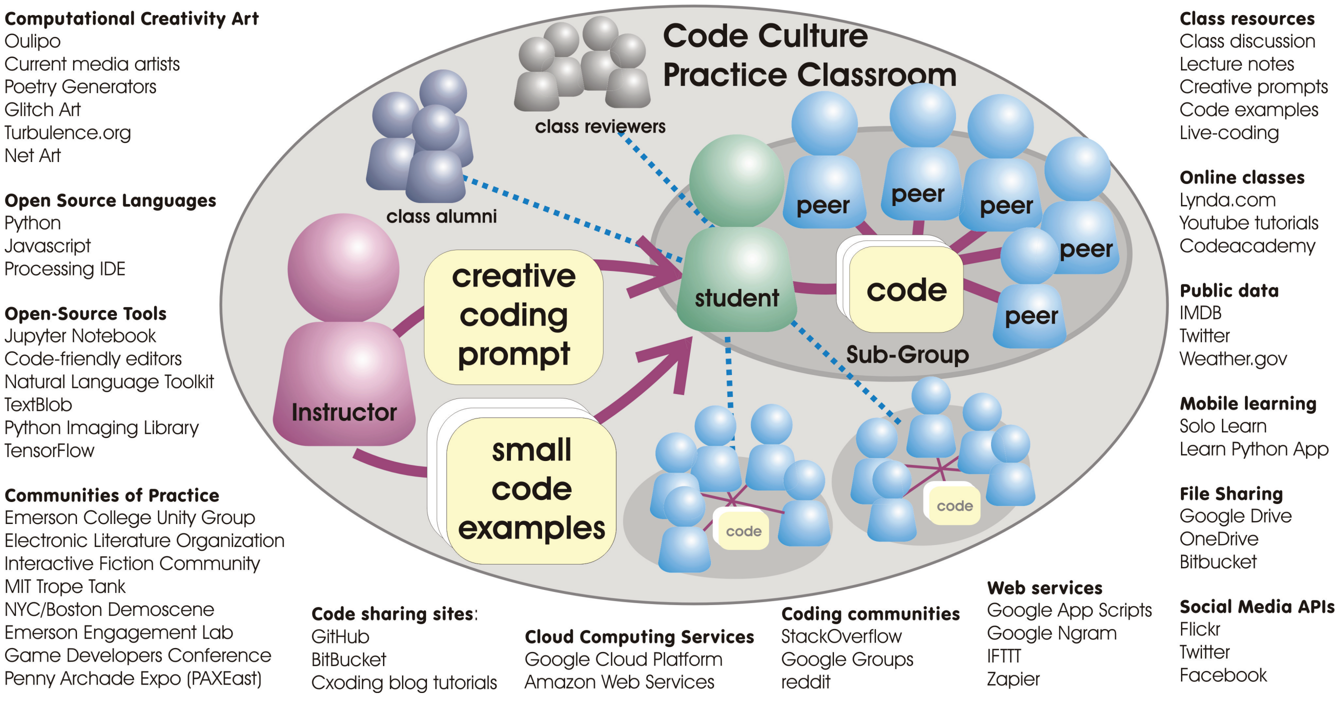
Creative Inspirations From External Resources

Using a flipped classroom, works by contemporary media artists were presented to initiate students to the active investigative dialogues taking place at the time. These artists would discuss their creative investigations online and perform readings of their computational works. Artists such as Allison Parrish, Ranjit Batnagar, and Leonard Richardson share their source on GitHub. Parrish's word-play tutorials explore bots as artificial intelligent collaborators. Richardson's *In-Dialogue* invites anyone to create custom mash-up novels from *Project Gutenberg*. Batnagar's *Pentametron* generates an endless sonnet from public tweets. Access to source code from these thoughtful artworks inspired students to learn by tinkering. Students adapted the examples and shared their results with the class. Some students even posted their mods for the open-source community or engaged the artists online.

Toward Sustainable Self-Learning

After students mastered basic programming concepts, harder tasks were assigned to give practical applications of these principles. Students delighted in using nested loops to crack each other's ciphers or share custom image filters made using the Python Imaging Library (PIL). These advanced topics also gave them a reason to seek out online forums such as StackOverflow and Reddit. Myriad detailed explanations on how to tackle a particular problem could be found and students compared solutions. Forums addressed issues that were unique to student projects or machine configurations.

Students were motivated to scour online resources and even post questions on these forums. They were surprised to learn that professional programmers participated in these same knowledge-seeking endeavors. If their posts were answered, they relished being considered seriously by these well-respected technical communities. As the class became more challenging, students relied on these external forums even if they did not ask questions. They were now comfortable accessing external learning resources to learn new ideas. Figure 3 depicts a sampling of resources students used.



Computational Creativity Art
Oulipo
Current media artists
Poetry Generators
Glitch Art
Turbulence.org
Net Art

Open Source Languages
Python
Javascript
Processing IDE

Open-Source Tools
Jupyter Notebook
Code-friendly editors
Natural Language Toolkit
TextBlob
Python Imaging Library
TensorFlow

Communities of Practice
Emerson College Unity Group
Electronic Literature Organization
Interactive Fiction Community
MIT Trope Tank
NYC/Boston Demoscene
Emerson Engagement Lab
Game Developers Conference
Penny Archade Expo (PAXEast)

Code sharing sites:
GitHub
BitBucket
Cxoding blog tutorials

Cloud Computing Services
Google Cloud Platform
Amazon Web Services

Coding communities
StackOverflow
Google Groups
reddit

Web services
Google App Scripts
Google Ngram
IFTTT
Zapier

Class resources
Class discussion
Lecture notes
Creative prompts
Code examples
Live-coding

Online classes
Lynda.com
Youtube tutorials
Codeacademy

Public data
IMDB
Twitter
Weather.gov

Mobile learning
Solo Learn
Learn Python App

File Sharing
Google Drive
OneDrive
Bitbucket

Social Media APIs
Flickr
Twitter
Facebook

Figure 3. Some connected learning resources explored by novice programming students.

Discussion: Supporting Diverse Learners on Their Coding Development

The majority of the students had never coded. Students were pursuing degrees in various disciplines, such as Journalism, Interactive/Visual Media and Arts, Media Production, Marketing Communications, and Writing, Literature, and Publishing (WLP). A few students reported that they had previously interacted with code, perhaps in a high school class, but were not regularly coding. Some students had unsuccessfully tried to take online coding courses (such as Code Academy) but did not feel confident about programming. A few reported that they had learned web design but were unable to extrapolate that knowledge to code their own designs. In each class, one or two students could be described as proficient coders in web technologies, but they were not familiar with Python. Unlike traditional computer science students, students were interested in coding but had no intention of becoming professional coders.

The first time the course was taught, we did not account for math anxiety. Students did well using Python for basic mathematics calculations. However, many struggled when using loops to process images or when following complex manipulations. In subsequent classes, more time was spent on visualizing information flow through a program (e.g., showing how pixels were addressed, or how each position of a string or list was manipulated).

From the onset, the class was designed to heighten students' awareness of current sociocultural, technology, and media arts themes. The instructor strove to connect with students' interests in the course. The instructor shared information about the local interactive fiction, game-development, and demoscene communities. Emerson College students talked about trips to Penny Arcade eXchange, Game Developers Conference, and political rallies. Students were asked to discuss personal projects (such as LGBT activism or applying for citizenship). The instructor used knowledge about student interests to facilitate connections between students and to select interesting data sets to study.

Appealing content motivated students to persist in tackling intricate challenges. Social media application-programming interfaces (APIs) were exciting, and students frequently turned to Flickr and Twitter APIs for their final projects. They readily looped through Twitter data to find nested details. One student used semantic analysis to collect a corpus of sexual harassment postings from the #MeToo movement. Another student documented the pervasiveness of racism online. One student commented,

> I'm a visual learner, so when we got to Processing, I saw how I could use it to create a prototype for my TalkHer App. It's a social network for women to connect with each other about politics, activism, and news. (Spring 2017)

That student used her prototype to pitch a socially motivated startup at MassChallenge, a statewide accelerator program

In every class, there were two to three students who were recovering from medical, emotional, or psychological issues. Students disclosed test anxiety, math phobia, adverse medication reactions, depression, and drug rehabilitation issues. Beyond ensuring that each student received professional support from the college support infrastructure, demystifying coding as "just editing a special kind of text" and focusing on the process of continuous creative output seemed to boost self-confidence. Giving them opportunities to express themselves through code helped them feel supported and valued. Personalized attention and immediate feedback from classmates who met regularly encouraged students to ask questions and sound out their developing ideas.

Recognizing students as individuals was essential to their success. Face-to-face interactions provided avenues for informal feedback to identify stumbling blocks. Smaller groups led students to recognize that their individual contributions were valuable and unique. Students were asked to switch partners each week so that they could interact deeply with many different people. For each assignment, students were encouraged to share their intent. These personal disclosures helped students form final-project groups. Students cared for each other's welfare. When students were absent, group members would reach out and keep each other informed. On return, students often shared with the class any insights while learning outside of class.

Many students remarked that they valued help from fellow students. Experienced students often completed the challenges early. These star students would then guide peers through coding and assisting the teacher. The class was also satisfying for advanced coders because they could share their works regularly with a live peer audience. These students enjoyed showing others how they could extend the basic lesson. This arrangement allowed the instructor to devote more attention to solving harder problems or addressing students who needed individual help beyond what a fellow student could explain.

Appropriate, modify, reuse to iterate quickly. Tinkering with code can yield satisfying results more quickly than building up from scratch. Once students developed an intention to explore, the class took off. Students figured out how to figure things out, and they could identify potential solutions and test them quickly. By experiencing, iterating, and combining resources, students settled on a desired result.

Frequent individualized assessment of coding progress. Each class provided ample opportunity to code together, like in a coding support group. The act of programming anything, even something small, was a good precursor to tackling larger pieces of code. Many assigned tasks required little effort but prepared students for thinking in code. Tangible outputs from lectures and homework provided another avenue for gauging progress. More numerous outputs, interactions, and peer reviews resulted in higher mastery and project quality.

Encourage informal learning. In later classes, the coding prompts were designed to help students break their final project into manageable pieces. The instructor encouraged students to try programming outside of class—for example, while sitting on the train or waiting for a meeting. The idea was to see coding as a casual activity that could happen outside class. The instructor also wanted them to experience coding as cultural currency—which students could do together socially. Students were shown interviews of programmers working together casually, talking about collaboration.

Use final projects to connect students with professional tools and people. In the last three weeks, a final project was proposed. Students had to craft an interesting experience using computation. Outside reviewers were invited to critique students' final presentations and demos. A mix of academics and professional programmers attended. Some visitors examined code and discussed implementation choices with students. Visitors asked questions about the cultural implications or reflected on artistic merits. Many students felt camaraderie with these visiting practitioners. It bolstered student self-assurance to be considered as colleagues.

Considerations That Foster Collaborative Feedback and Sharing

Internet access and a projector were required for teaching the class. Students brought their own laptops. All the learning resources were freely available. However, the following considerations helped foster collaboration and learning:

Furniture mobility had a large effect on group collaboration. One semester used fixed seats. People were reluctant to change partners because it was inconvenient to hop over seats. Flexible-use classrooms allowed students to transition between working independently and together. Participants started off in a circle for face-to-face discussions, working separately on their laptops. Then, when one person wanted to share a solution or needed help, all would cluster behind one screen. Movable chairs and desks enabled students to quickly shift their shared visual reference. Physical proximity when working together heightened the level of intimacy and increased ease in informal discussions. Control over their space enabled others to retreat from the group when needed.

Class size affected the feedback that people were able to give each other. In small classes, people had the opportunity to work together multiple times. More frequent interactions were desirable, and students in small classes benefited from a frequent history of working together. It was easier to give students personal guidance when there were fewer students. Approximately 15 students per class was manageable and yet allowed for enough interesting variety in projects, programming abilities, and cognitive diversity. In peer reviews, students in larger classes reported more interpersonal conflicts and were more selective about working with others.

Physical presence enabled immediate informal discussion and sharing. By troubleshooting problems in situ, progress was efficient. Student examples often spurred people to riff off one another. When students faced an issue, they could share their findings without the need to properly compose a solution. Others working at different paces may have missed the sharing. To preserve these teachable moments, the instructor kept a “Hall of Fame” of solutions to share at the next class.

Students were coached on how to work effectively in groups. Students were assigned new partners every two classes so everyone could practice working with diverse working styles. At midsemester, confidential peer surveys were given to address expectations. Students were counseled on being responsive and communicating clearly. Because of attendance issues or large class sizes, some people may not have worked together. Peer reviews from people who had not worked together were not useful. However, multiple positive experiences working in groups helped struggling students improve.

Sustainable Self-Learners

Many students went from never having seen code to being proficient using libraries and repurposing code. Students tackled APIs even the teacher had not used (such as TensorFlow or Unity’s Twitter API). Some used the Natural Language Toolkit sentiment analysis to visualize political tweets, sexual harassment, and mental health stigmas. Another group scraped news sites to identify partisan bias, scoured the Internet Movie Database to suggest movies, or searched GameSutra to create a visualization of game reviews. One student made MemeCaptionBot, a Twitterbot, to explore crowdsourced voting. They circulated a photo and calculated which captions received the most retweets. The bot superimposed the winning caption onto the photo and posted the meme. Another student wrote a motivation bot to retweet encouragements from Twitter. They learned to create a cron job to automate posts.

Yet another student proposed to redesign the Google Ngram visualizer using particle systems. They found a Python API to download the data. Then, they learned about particle systems by viewing Daniel Schiffman’s tutorials on YouTube. The student ported sample Java code into Python to prototype his design. “At first I thought I had to code it from scratch, but now I realize I can just combine things other people have done," said the student. At the final presentation, this student was thrilled to show his work to a programmer from Google.

This creative-inquiry process has many benefits. These search and appropriation skills are transferable outside of the class. Students of this method contributed to the greater knowledge available. Many uploaded their work to GitHub to be reused by others.

An Explosion of Creativity Versus Chaos

The final projects varied widely in focus and aims. Students investigated cultural topics such as racism, gender rights, and politics. Other students pursued artistic explorations: for example, a dramatic play generator, a text-adventure based in their school locale, or a shared remote backup system for roommates. They explored relaxation, color identity, or more holistic topics. This explosion of ideas demonstrated that the exploratory programming model supported student interests and creativity. The final projects addressed so many different themes and employed such a wide assortment of technologies that reviewers commented, "Did they all take the same class?"

We were surprised with the breadth of student projects and yet pleased by variety. Even technically conservative students contributed interesting pieces worthy of discussion and peer interest. With only one instructor, it was challenging to give every student in-depth guidance. Additional technical support could provide more directed advice to scaffold each student's creative journey. One student posted his Unity questions online, with little success. He finally got help from a previous student who advised him on creating a virtual reality state machine in Unity. Such questions were beyond the instructor's firsthand knowledge and would have required much time to research. Creating academic partnerships with external programmers or institutions would accelerate students' learning potential. These partners could consult on specific student projects as needed and might not require much time from the experts.

Conclusion

Although there are many approaches to learning programming, we believe the exploratory approach is ideal for helping casual programmers start their programming journey. Although this approach does not provide a thorough computer science education, it gives students programming knowledge that can lead to more serious study. Learners get hands-on experience with popular resources (languages, open-source repositories, and tools) used by many career programmers. They are taught a creative inquiry process that transcends any particular technological platform or predefined commercial output. Most important, developing these skills gives them the ability to connect with programming professionals via supportive online communities.

So, what does it mean to teach everyone to program? To suit a diverse population requires many different approaches. Exploratory programming emphasizes a technically and emotionally supportive local environment. The classroom helps students take fuller advantage of freely available learning resources and support from networked communities. We believe this approach promotes sustainable self-directed learning because it allows students investigative freedom to guide their work. This yields surprising and satisfying discoveries for the learner. In our study, students sought out challenging topics independently driven by their own curiosity.

We found that coding literacy is directly useful for nonprogramming majors. They use it to shine a unique lens on the world. Programming has cultural and ethical impact (Rushkoff, 2010). Exploratory programming can be useful for the general population, not just liberal arts students. Cognitively diverse programmers can make novel contributions to social, political, and cultural conversations.

References

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM*, *60*(6), 72–80.

Ito, M., Gutiérrez, K., Livingstone, S., Penuel, B., Rhodes, J., Salen, K., … Watkins, S. C. (2013). *Connected learning: An agenda for research and design*. Irvine, CA: Digital Media and Learning Research Hub.

Kleon, A. (2012). *Steal like an artist: 10 things nobody told you about being creative*. New York, NY: Workman.

Montfort, N. (2014, November 14). *Memory slam.* Retrieved from http://nickm.com/memslam/

Montfort, N. (2016). *Exploratory programming for the arts and humanities.* Cambridge, MA: The MIT Press.

Motte, W. F. (Ed.). (1986). *Oulipo: A primer of potential literature* (pp. 143–152). Lincoln & London, England: University of Nebraska Press.

Queneau, R. (1958). *Exercises in style* (B. Wright, Trans., 1981, pp. 72–73). New York, NY: New Directions.

Rubin, M. J. (2013, March). The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM technical symposium on computer science education* (pp. 651–656). New York, NY: ACM.

Rushkoff, D. (2010). *Program or be programmed: Ten commands for a digital age.* New York, NY: Or Books.

Shen, H. (2014). Interactive notebooks: Sharing the code. *Nature News*, *515*(7525), 151.

Wiedenbeck, S. (2005, October). Factors affecting the success of non-majors in learning to program. In *Proceedings of the first international workshop on computing education research* (pp. 13–24). New York, NY: ACM.

Acknowledgments

We thank Nick Montfort, the MIT Trope Tank, and the ideas in his book *Exploratory Programming for the Arts and Humanities*. This work was supported by the Liberal Arts and Interdisciplinary Studies Department at Emerson College, past students, and invited reviewers of *Code, Culture, and Practice*. We also wish to thank programmers Zahra Kanji and James Gouldstone for their editorial counsel.