The Role of Computational Literacy in Computers and Writing

Alexandria Lockett, Elizabeth Losh, David M Rieder, Mark Sample, Karl Stolley, and Annette Vee

(Proceedings from a Town Hall meeting at the 2012 Computers and Writing Conference, Raleigh, North Carolina)

Introduction

If we can measure the significance a new scholarly object by the number of innovative courses, breadth of engaging research, and buzz of online activity, then code has reached a critical moment in writing studies. Scholars such as Katherine Hayles and Espen Aarseth have long focused on the function of code in electronic texts, but ever since Mark Marino described the humanistic reading of code as “critical code studies” in 2006, the field has exploded. Often working at the level of code, the computer scientists Michael Mateas and Noah Wardrip-Fruin are exploring games as narratives. Rhetoricians such as Ian Bogost are similarly concerned with procedurally in new media, particularly game code. And works such as Bradley Dilger and Jeff Rice’s edited collection *From A to <A>* foreground code as a mode of writing. Of course, this work builds on early research by Paul Leblanc, Gail Hawisher, Cynthia Selfe, Jim Kalmbach and Ron Fortune, all of whom endeavored to draw attention to the politics and composition of code in the 1980s and 1990s.

Code has not only made its way into our research, it has also found its way into our classrooms. Kevin Brock’s *Code, Computation and Culture* class at North Carolina State University, Jamie Skye Bianco’s *Composing Digital Media* course at University of Pittsburgh, and James Brown Jr.’s *Writing and Coding* composition course at University of Wisconsin-Madison are all examples of writing classrooms that now include code. As David M Rieder has argued, code is blurring into our textual compositions, such that it’s no longer possible to bracket it off from writing pedagogy.

It may be that composition and rhetoric teachers are discovering what software developers already know. That is, from the perspective of computer science, programming has long looked like writing. Turing Award-winner Donald Knuth has argued for “literate programming” and Frederick Brooks has drawn a parallel between the programmer and the poet. More recent efforts under the name of “computer science for all,” “computer programming for everybody,” or “computational thinking”— promoted by computer science educators such as Mark Guzdial and Jeannette Wing—have sought to teach computing across the university curriculum, just as writing is now. These educators build on decades-old efforts by Seymour Papert (the designer of Logo) and John Kemeny and Thomas Kurtz (the creators of BASIC), who designed languages that would help a wider spectrum of people learn to code.
Beyond academia, popular initiatives and news stories have made the connection between coding and writing. For example, the Code Year initiative made a big splash in January 2012 with some good PR, as well as a promise by New York Mayor Michael Bloomberg that he would learn code this year; as of this writing, over 450,000 people have signed up to receive weekly code lessons via email. Writing for The Guardian in March 2012, John Naughton explains “Why all our kids should be taught to code.” Media theorist Douglas Rushkoff caused a stir in 2011, arguing, “In the emerging, highly programmed landscape ahead, you will either create the software or you will be the software. It's really that simple: Program or be programmed.”

It is against this backdrop that the Town Hall “Program or be Programmed: Do We Need Computational Literacy in Computers and Writing?” was proposed. Reflecting on his idea for the Town Hall, Rieder writes:

The reason I proposed a town hall on code is that it seems like an important issue right now. The growing interest in software studies and critical code studies, the ever-deepening engagement with computational approaches to the digital humanities, and my own local experiences teaching graduate courses on Arduino and Processing and digital humanities, makes it seem like the issue is emergent in our field, too. Also, in the Communication, Rhetoric and Digital Media program at NC State, there is a growing interest in and recognition that some expertise in computational thought is a part of future success as a scholar affiliated with the humanities.

Rieder asked Annette Vee to help plan and recruit panelists for the Town Hall, and both were delighted that Alexandria Lockett, Elizabeth Losh, Mark Sample, and Karl Stolley agreed to speak.

In this iteration of the Town Hall (which you also can watch on Vimeo, courtesy of Dan Anderson), we’ve provided each panelist’s script, along with a Storify collection that captures the Twitter backchannel during the presentation. We hope our provocations contribute to the continuing conversations about coding and composing in writing studies.
Programming Is the New Ground of Writing
David M. Rieder, North Carolina State University

The image on the screen is by the painter Mark Tansey (see Fig. 1 above). It is titled “Picasso and Braque.” And there are three reasons why I’m leading with it. First, it is reminiscent of Orville and Wilbur Wright’s test flights outside the town of Kitty Hawk, North Carolina, which is just a few hours drive from here. Second, at a conference on writing, many of Tansey’s paintings, including this one, valorize text in one way or another—especially the printed word. In this painting, we can see Picasso trying to fly one of his collages while Braque surveys his progress from the ground, and the point worth noting is that the flying surfaces of Picasso’s collage are made from swaths of newsprint. It’s a multimodal writing machine. Well, really…

It’s a curious combination of kite and plane—a machine made from what look like Hargrave cells wrapped in words. Judging from the second image on the screen (see Fig. 2 above), perhaps Tansey’s painting is a bit of a mash-up of histories of flying, Hargrave’s kites and the Wrights’s plane.
But, getting back on track, in addition to these two points is the third, which includes my main point.

In the painting (see Fig. 1 above), the words printed on the swaths of newsprint—the writing has broken free of its logocentric grounding. It’s a relative break—not an absolute deterritorialization—because we still recognize the writing on those swaths of newsprint as writing, but the words have taken on a value relative to the flying machine that it serves. The words are still readable, but the point here is why read them? Or (I don’t really mean to throw the baby out with the bathwater) why limit their value to their readability? The flying machine, the dynamic medium they now serve, invites us to step away from our conventional stance toward writing, which is writing at a standstill, and unground ourselves. In computational media, writing wants to take a walk, not sit on a couch to be analyzed. Writing doesn’t need to be limited to the faithful representation of glottic thought and expression—it never did, in fact. In computational media, our alphabet is motivated and dynamic. As Richard Lanham explains in *The Economics of Attention*, it is capable of thinking. In today’s digital age, the definition of writing is expanding well beyond the logocentric tradition, but many of us continue to reinforce the static, logocentric tradition. When it comes to computational media and the topic of this town hall, if writing can think (i.e., if it is mobile and transformable) then we should turn our focus away from content, which has diminishing value, to algorithmic forms and functions.

So, my third point about Tansey’s painting, the thing that I like about it, is that we’re all Picasso in flight, whether we know it or not. How many of you are multi-tasking with tablets or laptops during this conference, partly here, partly somewhere else. We’re in flight right now. But here’s the problem, the challenge: while we’re all comfortable in flight, far fewer of us are willing or able to recognize how writing has taken flight, too.

The new ground, the new basis of writing is algorithmic. Today, the power and profit in writing has less to do with representing speech than serving a generative process of creation.
So, if you are teaching and practicing writing as a grounded, representational technology, you are missing the proverbial forest for the trees, the machine for the pages of newsprint. If you can’t write code, if you can’t think with code, if you can’t write algorithmically, you may eventually find yourself stuck in the logocentric sands of the past.

**Coding Values**  
*Annette Vee, University of Pittsburgh*

Today I want to talk about good code. Experienced programmers often think about what good code is. But they rarely agree.

And here’s what I want to say: they don’t agree on what good code is because **there is no good code.** Or, rather, there is no Platonic Ideal of Good Code. Like writing, there is no good code without context.

![There is no good code.](image)

Unfortunately, when good code is talked about, it is often talked about as if there’s no rhetorical dimension to code. It’s talked about as though the context of *software engineering* were the only context in which anyone could ever write code. As if digital humanists, biologists, web hackers, and sociologists couldn’t possibly bring their own values to code.

I’ll give you just a couple of examples of how this happens, and what this means for us in computers and writing.

One of the earlier articulations of the supposed Platonic Ideal of Good Code was Edsger Dijkstra’s infamous “GOTO considered harmful” dictum, from 1968.
Edsger Dijkstra considers go to harmful.

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

Editor:
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More dynamic programming, more textual indie dynamic dependency, let us not repeat A superfluous, recursive procedure elude them.

This article railed against unstructured programming and the GOTO command. Now, many of us first learned the joy of coding through the languages that used the GOTO command. But Dijkstra’s statement suggests that the context of the software engineering work place should override all other possible values for code. This is fine—as far as it goes, which is software engineering and computer science. But this kind of statement of values is often taken outside of those contexts and applied in other places where code operates. When that happens, the values of hacking for fun or for other fields are devalued in favor instead of the best practices of software engineering—that is, proper planning, careful modularity, and unit testing.

Here’s a more recent example, which I pulled from the Hacker News forum. Here the values of software engineering are more tacit, and more problematic:

Ender7 is replying here to a thread about a recent ScientificAmerican story that suggested scientists were reluctant to release the code they used to reach their conclusions, in part because
they were “embarrassed by the ‘ugly’ code they write for their own research.” According to Ender7, they *should* be ashamed of their code. Ender7 goes on to say:

> So, I don’t blame them for being embarrassed to release their code. However, to some degree it’s all false modesty since all of their colleagues are just as bad.

Why is academic code an “unmitigated nightmare” to Ender7? Because it’s not properly following the rules of software engineering. Again, the rules of software engineering presumably work well for them. I’m not qualified to comment on that. But that doesn’t mean that those values work for other contexts as well, such as biology.

So, in this example, software engineering’s values of modularity, security, and maintainability might be completely irrelevant to the scientist writing code for an experiment. If scientists take care to accommodate these irrelevant values, they may never finish the experiment, and therefore never contribute to the knowledgebase of their own field. The question, then, isn’t about having good values in code; it’s about which values matter.

We often hear how important it is to have proper grammar and good writing skills, as if these practices had no rhetorical dimension, as if they existed in a right or wrong space. But we know from writing studies that context matters.

Put another way: like grammar, code is also rhetorical. What is good code and what is bad code should be based on the context in which the code operates. Just as rhetorical concepts of grammar and writing help us to think about the different exigencies and contexts of different populations of writers, a rhetorical concept of code can help us think about the different values for code and different kinds of coders.

And this is how coding values are relevant to us in computers and writing. The contingencies and contexts for what constitutes good code isn’t always apparent to someone just beginning to learn to code, in part because the voices of people like Ender7 can be so loud and so insistent. We know from studies on teaching grammar and writing that the overcorrective tyranny of the red pen can shut writers down. Empirical studies indicate it’s no different with code. Sure there are certain ways of writing code that won’t properly communicate with the computer. But circle of valid expressions for the computer is much, much larger than Ender7 or Dijkstra insist upon.

To close, I want to share with you a bit of what might be considered very ugly code, a small Logo program I call, tongue-in-cheek, “codewell”:

> Code is rhetorical.
This is bad code because:

- it is uncommented and hard to read
- it's in an old, seldom-used language
- it is baggy and has repeated statements that should be rewritten as functions
- it is not modular or reusable
- it’s an “unmitigated nightmare”

If you run the code [on github here] in a LOGO interpreter, it looks like this:
So, in addition to saying my code sucks, you could also say this:

- it could be used to teach people some things about functions and code
- it’s a start for a LOGO library of letters that might be kindof cool
- it does what I want it to do, namely, make my argument in code form.

Let’s imagine a world where coding is more accessible, where more people are able to use code to contribute to public discourse or solve their own problems, or just say what they want to say. For that to happen, we need to widen the values associated with the practice of coding. To Edsger Dijkstra, I’d say: coding values that ignore rhetorical contexts and insist on inflexible best practices or platonic ideals of code should be CONSIDERED HARMFUL – at least to computers and writing.
5 BASIC Statements on Computational Literacy

Mark Sample, George Mason University

I want to briefly run through five basic statements about computational literacy. These are literally 5 statements in BASIC, a programming language developed at Dartmouth in the 1960s. As some of you might know, BASIC is an acronym for Beginner’s All-Purpose Symbolic Instruction Code, and the language was designed in order to help all undergraduate students at Dartmouth—not just science and engineering students—use the college’s time-sharing computer system.

Each BASIC statement I present here is a fully functioning 1-line program. I want to use each as a kind of thesis—or a provocation of a thesis—about the role of computational literacy in computers and writing, and in the humanities more generally.

10 PRINT 2+3

I’m beginning with this statement because it’s a highly legible program that nonetheless highlights the mathematical, procedural nature of code. But this program is also a piece of history: it’s the first line of code in the user manual of the first commercially available version of BASIC, developed for the first commercially available home computer, the Altair 8800. The year was 1975 and this BASIC was developed by a young Bill Gates and Paul Allen. And of course, their BASIC would go on to be the foundation of Microsoft. It’s worth noting that although Microsoft BASIC was the official BASIC of the Altair 8800 (and many home computers to follow), an alternative version, called Tiny BASIC, was developed by a group of programmers in San Francisco. The 1976 release of Tiny BASIC included a “copyleft” software license, a kind of predecessor to contemporary open source software licenses. Copyleft emphasized sharing, an idea at the heart of the original Dartmouth BASIC.

10 PRINT "HELLO WORLD"

If BASIC itself was a program that invited collaboration, then this—customarily one of the first programs a beginner learns to write—highlights the way software looks outward. Hello, world. Computer code is writing in public, a social text. Or, what Jerry McGann calls a “social private text.” As McGann explains, “Texts are produced and reproduced under specific social and institutional conditions, and hence…every text, including those that may appear to be purely private, is a social text” (McGann 21).

10 PRINT “GO TO STATEMENT CONSIDERED HARMFUL”: GOTO 10

My next program is a bit of an insider’s joke. It’s a reference to a famous 1968 diatribe by Edsger Dijkstra called “Go To Statement Considered Harmful.” Dijkstra argues against using the goto command, which leads to what critics call spaghetti code. I’m not interested in that specific debate, so much as I like how this famous injunction implies an evaluative audience, a set of norms, and even an aesthetic priority. Programming is a set of practices, with its own history and tensions. Any serious consideration of code—any
serious consideration of computers—in the humanities must reckon with these social elements of code.

10 REM PRINT “GOODBYE CRUEL WORLD”
The late German media theorist Frederich Kittler has argued that, as Alexander Galloway put it, “code is the only language that does what it says” (Galloway 6). Yes, code does what it says. But it also says things it does not do. Like this one-line program which begins with REM, short for remark, meaning this is a comment left by a programmer, which the computer will not execute. Comments in code exemplify what Mark Marino has called the “extra-functional significance” of code, meaning-making that goes beyond the purely utilitarian commands in the code (Marino).

Without a doubt, there is much even non-programmers can learn not by studying what code does, but by studying what it says, and what it evokes.

10 PRINT CHR$(205.5+RND(1));:GOTO 10
Finally, here’s a program that highlights exactly how illegible code can be. Very few people could look at this program for the Commodore 64 and figure out what it does. This example suggests there’s a limit to the usefulness of the concept of literacy when talking about code. And yet, when we run the program, it’s revealed to be quite simple, though endlessly changing, as it creates a random maze across the screen.

[Video]
http://www.youtube.com/watch?feature=player_embedded&v=n7QrX2EwYO4

So I’ll end with a caution about relying on the word literacy. It’s a word I’m deeply troubled by, loaded with historical and social baggage and it’s often misused as a gatekeeping concept, an either/or state; one is either literate or illiterate.
In my own teaching and research I’ve replaced my use of literacy with the idea of competency. I’m influenced here by the way teachers of a foreign language want their students to use language when they study abroad. They don’t use terms like literacy or fluency, they talk about competency. Because the thing with competency is, it’s highly contextualized, situated, and fluid. Competency means knowing the things that are required in order to do the other things you need to do. It’s not the same for everyone, and it varies by place, time, and circumstance.

Translating this experience to computers and writing, competency means reckoning with computation at the level appropriate for what you want to get out of it—or put into it.

Notes


**I am Not A Computer Programmer**

*Alexandria Lockett, Pennsylvania State University*

I am not a computer programmer. My knowledge of programming language extends as far as hypertext mark-up language (html), which I learned in the days of crude computing better known as Web 1.0. Yet, for the past couple of years, I have been using Ubuntu, the user-friendly Linux desktop environment founded by Mark Shuttleworth. (Ubuntu is a Bantu word which means “humanity unto others” and a concept I’ll return to later in this piece) Sick of my slow, barely functioning computer, I asked my neighbor—a robotics software engineer—what new computer I should buy. Yes, just so happens that the moment I conspired to murder my machine that a computer programmer happened upon my front door to help me move a heavy piece of furniture. Indeed, it seems Open Source thrives on synchronicity. I never wanted to use Windows again, and I couldn’t afford the expensive membership fees of the iCult. He suggested that I stop using proprietary software altogether and consider switching to Linux. I heard the word Linux and I automatically felt crushed. I remembered my high school dork friends with their terminals and their occasional visits with the FBI, on account of their cracking credit cards. How was I to use Linux?

Since I could burn ISO files, I was thrilled I could test out Ubuntu on my sluggish machine. Not only was it super easy to use, but I was pissed that I didn’t know this option was available to me. I asked my neighbor, who would soon become one of my best friends, why everyone didn’t drop their costly, barely functioning Windows OS’s. After hundreds of thousands of hours, over many long days and long nights, I understood concepts like modularity and proprietary lock-in. I came into contact with sites like Slashdot, ReadWriteWeb, and Reddit. I wasn’t interested in “becoming” a member of this community, but I did want to know more about the people who
wrote the code--their personal experiences, how they interacted. I wanted to visualize their values, beliefs, and ways of seeing the world. I eagerly read Richard Stallman’s Manifesto, learned more about Linus Torvalds, Eric Raymond, Alan Turing, Steven Wozniak, Larry Page, Sergey Brin, and Ward Cunningham. When my printer and scanner hard drives weren’t compatible with Ubuntu, I googled how to manually install them. Soon, I began to recognize the characteristics of the Free and Open Source Software Community and its massive impact on other communities. It had a complex history, social languages, intertextuality (i.e. GNU’s not Unix), discourses, conversations. All the ‘stuff’ of a discourse community. Software developments that facilitate(d) web 2.0, our dependence on ‘free’ applications, and Linux servers running major businesses and the Interwebz is striking evidence of its power.

Indeed, I am not a computer programmer, but I began to recognize several commonalities between my perspective as a writing teacher and my friend’s perspective as a programmer. The objectives of having access to code, tinkering with the code, running programs for any purpose, and sharing improvements with the community so everyone benefited were tantamount to my pedagogical approaches to rhetoric and composition. I renewed connections with those “dork” friends from high school and recognized what we had in common all along: a love of wit and play, an insatiable curiosity for learning about how things (and people) work, and a knack for solving problems.

I do not feel as if I need to be a programmer to faithfully represent and perform values held by practitioners of Computers and Writing. I have repeatedly stated that I am not a computer programmer, but I will assert now that I am a hacker. I am able to recognize how bureaucratic linguistic practices inhibit me from “tinkering” with language. I see how the humanities’ obsession with authors and owners inhibits many people from collaborating, dialoguing about, and innovating scholarly research. In fact, I’ve been on the border my whole life---switching back and forth from Standard White English and African American Vernacular English, Midwest Plain Style and Decorous Southern Speech. I’ve used the invisibility and visibility of my identity to push the limits of argumentation beyond the limits of ‘normality.’ I’ve been translating language as long as I can remember, trying to understand these rules, poking fun at them, playing with them, succeeding or failing at remixing and subverting them to open up new paths for language use. Computational literacy should include a wider range of competencies besides just ‘technical’ literacy. Coding is a dynamic performance, a demonstration of various levels of competency: how and why I do code or chose to learn about the significance and value of coding influences what programs I run--in both human and machine interactions. For example, working with multilingual writers in the writing center for the past year has enabled me to recognize the benefits of using computing as a metaphor for helping them understand grammar. I’ve learned to understand the operational limits and potentials of this code, and recognize the inseparable relationship between the technical, cultural, and political. This talk, for instance, is a program I’m running to facilitate trust between us so that we can acknowledge the value of anyone who wants to tinker with these discourses for the benefit of helping others quench their thirst for knowledge or find out which beverage motivates them to pursue their drink.

I am not a computer programmer, but should I desire to close the wide gap that exists between what I need to know and what I can possibly learn to become one, I have access to every
resource available to me to do so vis-a-vis open educational resources. I don’t know C++, Ruby, Python, or Perl, but I was able to get a sense of the politics surrounding their developments and implementations by visiting their websites, reading discussion forums, current events, trade and scholarly journal articles, and talking to individuals teaching and learning these languages. I recognize how I benefit from the architecture of these collaborative feedback systems, which include countless distributed autonomous communities constituting intertwingularity, as Internet pioneer Ted Nelson aptly describes. I do not teach with technology without discussing the power distributions of socio-technical systems, the ethical responsibilities they inflict upon human users, and the ways in which our grammatical code and linguistic arrangements make it difficult for us to talk about emergence. We need hackers of all gradients--from the computer programmer to the radical pragmatist instructor to the DJ to the comic--to resolve the broader issues of helping students develop enough confidence and generosity to hack language, improve their writing, and self-consciously participate in a much broader effort to do humanity unto others.

Source Literacy: A Vision of Craft
Karl Stolley, Illinois Institute of Technology

Introduction

For the things we have to learn before we can do them, we learn by doing them. E.g., people become builders by building.

- Aristotle, Nicomachean Ethics

“Program, or be Programmed.” That’s a dire warning. And being a firm believer in the importance of computational literacy, or source literacy as I prefer to call it, I’ve issued more than my fair share of dire warnings: What happens if we only write in other people’s apps, other people’s text boxes. What happens if we think ourselves so privileged as a field that we can pick and choose from the digital buffet of what will and will not be our concerns.

But today, I don’t want to issue any dire warnings. I don’t want to make any appeals as to why you should learn to program the moment you get up from lunch this afternoon. I don’t even want to argue with anyone about whether we should pursue source literacy. For me, the answer is an obvious and unqualified Yes. But debate or even sharing abstract ideas alone won’t persuade you.

What I do want to share today, then, is my vision for the field of Computers and Writing. I want to share what it is that we could do and be if we all learn to program, starting right after lunch. And I will conclude with four things that all of us can do immediately to make this vision a reality.

My Vision

Do not automate the work you are engaged in, only the materials.
Because this is a vision, I’m going to use the present tense. Not the future tense.

My vision for Computers and Writing places craft at the center of what we do. And what we do is digital production. We make things from raw digital materials: open-source computer languages and open formats. Which is to say, we write digital things. To write digital things, we rely on a strong command of source literacy.

There are no language tricks with the verb write in this source-literate field I envision. Writing is not a metaphor to explain how writing is accomplished by clicking through a WYSIWYG that generates and automates our work for us. The verb write is literal.

Everyone, of course, writes with web standards: HTML, CSS, and JavaScript. Above all else, the field values digital projects that are accessible from any browser, on any device. In my vision, that value has become pervasive; it’s rare anymore to hear of a writing assignment that doesn’t require web standards.

And while debates rage over different markup and design patterns, the level of discourse and collaboration regarding digital craft has never been more sophisticated. In fact, in the field as I envision it, several members of the computers and writing community are now active participants in the working groups that oversee the development of web standards specifications.

But the really remarkable change has been the number and diversity of open-source software projects initiated and maintained by members of the field I envision. It’s unusual to find someone in Computers and Writing who doesn’t have a GitHub account and a favorite language, or who doesn’t attend the special interest group devoted to that language at the 4Cs each year.

From all of this digital production activity, there has been a renaissance in publishing on digital craft in our field. Numerous digital book titles by researchers in Computers and Writing routinely appear on must-read lists outside the field, especially for programmers and developers whose own professional practice has been greatly augmented by a rhetorical approach to thinking about and doing the work of digital craft.

How did this sense of craft emerge? By rejecting a model of computing that is suited to office cubicles and deskill writers. By embracing, instead, a deep appreciation for the raw materials, the languages, of the digital medium, and seeing digital writing as more than the on-screen result of the machinations of commercial software.

“Craft,” in the words of Malcolm McCullough, “is commitment to the worth of personal knowledge.” It is commitment to research and learning over technical support and intuitive interfaces. Craft is ultimately the sense of taking responsibility for the digital writing that we unleash on the world. It is the thrill and wonder of watching our work emerge from the thousands of lines of source code that make up the eBooks and web applications that we, in the field I envision, write every single year.
How to Make this Vision a Reality

Here’s how this vision can become reality; these aren’t really steps. It’s an entire shift in digital production that centralizes craft, a shift that you either buy into, or you don’t.

1. Learn a Unix-like operating system at its command line.

   Whether you have a shiny new Mac with the latest version of OS X or run Ubuntu Linux on an old computer you have laying around, the first step to source literacy is to get comfortable with relying on the keyboard to interact with the computer.

2. Commit to writing in a text editor with good syntax highlighting, and start writing HTML and CSS.

   For everything. Old habits hold people back more than the challenges of learning something new. Even if ultimately you have to submit something, like a journal article, in Microsoft Word, do your composing in HTML.

3. Learn the distributed version control system Git, and establish a GitHub account.

   Experimentation, expansive learning, and even simple revision are all impossible if we’re shackled to the regular file system of a computer, where versioning happens by renaming files. If the computer is, as Steve Jobs said, a bicycle for the mind, then Git is a time machine and guardian angel for the mind.

4. Learn a couple of web-oriented, open-source programming languages.

   JavaScript and Ruby are the two languages I wish everyone knew, on top of HTML and CSS. Despite its reputation, JavaScript has matured into an important language, as evidenced by server-side applications such as Node.js. Ruby, courtesy of development frameworks such as Rails and Sinatra, is quickly becoming the language of choice to power web applications. But Ruby has utility in many other domains, including for programming command-line applications to extend and deepen command-line literacy beyond basic shell scripting.

I don’t expect any of this to be easy. But I also reject ‘ease’ as a measure of anything valuable or worthwhile. So, let’s get to it. Right after lunch.

Response to Program or Be Programmed

Elizabeth Losh, University of California, San Diego

The central concept of this panel “Program or Be Programmed” might immediately bring up performance anxiety issues for many people in this audience. As Stephen Ramsay put it recently, the very notion of the tech-savvy digital humanities as the newest “hot thing” tends to bring up “terrible, soul-crushing anxiety about peoples’ place in the world.” For those in composition, the
anxiety might be even more acutely soul-crushing in light of existing labor politics. Every time the subject of learning code comes up, one can almost see the thought balloons appearing: “How can I learn Python in my spare time when I can’t even see over the top of the stack of first-year papers that I have to grade?” And for those who care about inclusion, what does it mean to choose the paradigm of computer programming culture, where women and people of color so frequently feel marginalized? Furthermore, if all these powerful feelings are being stirred up, what questions should we be asking about ideology as an object of study. For example, Wendy Chun has argued that a desire for mastery over blackboxed systems or access to originary source code shows how a particular dialectic of freedom and control makes it difficult for us to have meaningful discussions about technology and to acknowledge our own limited access to totalizing understanding, even if you are a software engineer.

Fortunately, after hearing these talks, people in the audience should feel a little less anxious. They should know that doing-it-yourself means doing-it-with-others, whether it is imagining Picasso and Braque building a flying machine, as David Rieder suggests, or installing Ubuntu with the help of a neighbor, as Alexandria Lockett describes. The message to instructors in this panel is comforting: relax, be confident in your own abilities to learn new things, ask questions, facilitate the questions of others, and network in ways that help you make new friends.

However, if you are an administrator as well as an instructor, don’t get too relaxed just yet. These talks also bring up some very thorny questions about disciplinary turf. After all, who defines how digital literacy should be taught and who will teach it? Computer scientists? Media artists? Librarians? Us?

Although he uses the word “craft,” Karl Stolley asserts that “source literacy” doesn’t require an elaborate apprenticeship. All it takes is moving toward a set of everyday common-sense practices involving command lines and file structures. Mark Sample suggests the term “code competency” as an alternative to “code literacy,” because of all the cultural baggage associated with the word “literacy” itself. Trebor Scholz has suggested “fluency” as a better characterization of what we are trying to teach, but Sample notes the limitations of that term.

In a 2010 essay called “Whose Literacy Is It Anyway?” Jonathan Alexander and I pointed to Michael Mateas’s work on “procedural literacy” as a way for compositionists to begin to engage with these issues. Mateas worries that universities are often too eager to adopt the training regimes of computer science departments, which is great for graduating computer science majors but not so great for teaching students in other majors or with other passions to use code. So what should be the relationship between writing studies and computer science in the academy?

People at this conference are probably more likely to be able to say that line about “some-of-my-best-friends-are-computer-scientists” than those at the MLA. (I personally carpooled with computer science faculty for ten years when I worked at UC Irvine and learned something about discrete math and number theory in the process.) But what does that collegiality get us? Both Sample and Vee mention Edsger Dijkstra, who was also the author of “On the Cruelty of Really Teaching Computer Science,” a decidedly anti-humanistic diatribe on the superiority of formal logic and mathematics as the keys to supposedly real knowledge.
Given the fact that badly written code can overdose patients with radiation or knock out someone’s retirement savings, I’m not sure that I always agree with Vee that clean, legible, rationalized code is not worth teaching to everyone, but I’ll put in my own GOTO command for writing studies to keep this spaghetti-like discussion going with our colleagues elsewhere long after this panel concludes.