Coding Fundamentals for Human(s|ists)

0 Introduction

We write this chapter in reflection of teaching computing fundamentals in the humanities context in general, and more specifically, in the wake of teaching two instances of **Computing Foundations for Human(s|ists)** at the Digital Humanities Summer Institute (DHSI), University of Victoria.[1] This week-long course was intended for humanities-based researchers with no previous programming experience, and, as we wrote in the course description, for those who would like to understand how programs work by writing a few simple, but useful programs of their own.[2] The topics covered included working with files and folders at the command line, text stream manipulation with Bash Unix Shell, regular expressions, along with Python basics like native data types, variables, functions, and control structures. At the end of the course our students worked on their own and in small groups to create a small web scraper, an "essay grader," a comma separated value file manipulator, and a program that evaluated poetry for its measure of similarity to Byron.

Our aim in this chapter is not so much to recapitulate the experience of teaching (we would not have the space to do it here, in any case) but to reveal some of the core principles that went into making the course, to talk about the rationale behind our teaching philosophy, and, more broadly, to suggest an approach to teaching programming in the humanities environment.

<!— JS: Should we say something about coding vs programming here? DT: I don't have strong feelings about the distinction. Up to you. —>

[1] "Human(s|ists)" is actually a regular expression, a way to search text for specified patterns. In this case it picks out anything starting with "Human" and ending in *either* "s" or "ists". So, it acts as a stand-in for both "Humans" and "Humanists". This little flourish in the title is important because it signals a number of the principles that guide the course and signal what is to come.

[2] An archived version of the course can be accessed at http://web.archive.org/web/20150614161609/https://github.com/denten-workshops/dhsi-coding-fundamentals/blob/master/README.md

1 Critical Computing in the Humanities, Core Principles

It is our firm belief that the teaching of computational principles in the humanities must be grounded in the practice of humanities-based research and answer to the values held by the academic community. Inspired by a number of initiatives advancing a similar philosophy, we refer to this approach as "critical computing."[3] The following ten principles connect key ideas in computation to values intrinsic to humanistic inquiry:

1.0 Demystify the black box (as much as possible).

<!— Let's limit these to one paragraph each. We can go into greater detail in later sections. —>

Contemporary computational devices are a fact of daily life. They are involved in everything from financial markets, to archival research, and to the way many keep in touch with friends and family. Yet, for those without computational background, the inner workings of these devices remain a source of mystery and, consequently, frustration. Recognizing this, our course targets daily computation. We want our students to understand not only the principles of programming, but the basics of networking, security, and operating systems. As much as possible, we would like to reveal the innards of opaque computational "black boxes," empowering our students to take control of their everyday computational practice.

1.1 Use few, free, small, simple, universal, and powerful tools that you can hack and understand.

Researchers, librarians, students, and faculty are faced with a bewildering array of software choices. In making the decision to invest time and resources into learning a new tool or methodology, we are guided by "Unix Philosphy" and by the Free and Open-source Software movement (FOSS). The Unix way of computing privileges small, modular pieces of software that "do one thing well."[4] Such software can then be chained together with other small but powerful tools to achieve complex results. FOSS software works in the academic humanities

[3] We are not the first nor the only instructors to think about things this way, nor are we the only ones to be offering a course like this. For example, Software Carpentry has been advocating a similar approach since its inception. Similarly, the *Programming Historian* is "an online, open-access, peer-reviewed suite of tutorials that help humanists learn a wide range of digital tools, techniques, and workflows to facilitate their research." See also D. Fox Harrell, "Toward a Theory of Critical Computing; The Case of Social Identity Representation in Digital Media Applications," *CTHEORY* 0.0 (2015), online, Internet, 9 Sep. 2015., Available: http://journals.uvic.ca/index.php/ctheory/article/view/14683.

[4] M.D. McIlroy, E.N. Pinson, and B.A. Tague, "UNIX time-sharing system: Foreword," *Bell System Technical Journal, The* 57.6 (1978): 1899–1904.

context because free and open access code is available for inspection, and therefore, for interpretation, critique, and modification. More than anything, we seek out tools that we can understand and, if needed, to customize to fit our own particular needs and contexts. When deciding what to use and what to teach, we seek out universal tools that underlie a variety of computing practices, from library infrastructure, to web design, to data science and critical-edition making.

1.2 Wherever possible store data in human-readable text streams.

The file formats that we use to store data have serious implications for anyone's ability to make use of that data in the future. As the formats chosen tend toward obscurity or opaqueness it becomes increasingly difficult for new programmers to read during development and for others to draw on them in the future. Choosing appropriate file formats relates closely to the proliferation of closed tools and platforms. It is most acutely felt by archivists faced with preserving short-lived data structures from obsolete platforms from recent past. For many humanists, who rarely work with truly large datasets or collections, the real risk of rapid obsolescence offsets any hypothetical gains in speed or performance offered by a new note-taking platform or a complex database.[5] Similarly, when coding, we begin by having our students describe their program, step by step, in language natural to them and to their task. Pseudo code in plain English becomes the basis for well documented programs. Such programs are easy to read and to maintain. If forced to make a compromise, we privilege legibility and longevity over tersity of expression or performance.

<!— I moved the plain English point here. —>

1.3 If you have to do something more than ~~once~~ a hundred times, automate.

Programmers are smart and lazy. After doing a task more than a few times, a good programmer's intuition will be to automate the task. For example, we often use the `rsync` command to back up our documents; however, after a few months of running it manually, the user can delegate that task to the built-in scheduler called `cron`. The saying normally goes that if you do it more than *once*, automate. However, one must know exactly what to automate. When

[5] Conveniently for us, the Unix philosophy privileges inputs and outputs in plain text format, which can be used to store everything from personal notes, to article drafts, to huge datasets of metadata. Unix provides many powerful utilities that operate on plain text. In fact, a notion of human readability is encoded at the operating system level. When faced with a list of compromises, in the name of design, performance, efficacy, or legibility, we consistently prioritize legibility. That choice informs our practice throughout. When working with audio or visual material, for example, we similarly prefer widely-supported, non-proprietary standards. When selecting a data format, we ask: does it need special software to render? How long has it been around? and What organization is responsible for maintaining the standard?

backing up your files, do you want to back up the whole system or a few select folders? How often should the backup script run? The answers become apparent only after extensive manual use. As we introduce automated "daemons" that run tasks on our behalf, we want to make sure we think through any unintended side-effects: technological, personal, and political.

1.4 Do it right—*the first time.*

Although programmers are lazy, they are lazy in the right way. Doing things badly, in a haphazard fashion, accumulates technological, intellectual, and eventually an ethical debt, to oneself and to the community. Code comments are the simplest example of this. It is easy to skip documenting one's code. "It just works, why bother?" However, a piece of code that makes perfect sense today, becomes obscure tomorrow. Without comments, time needs to be spent to recreate the reasoning behind the original implementation. Similarly, we advise our students against simply cutting and pasting code snippets from our tutorials. We want them to follow our thinking, to annotate, and to review their notes regularly. In the social context, lazy practices are unethical because they "bank" against the labor of others, in the future, literally borrowing someone else's time. Doing things the right way now saves unnecessary effort later.

1.5 Target daily, real-world computation.

Programming classes in the sciences often begin with coding for coding's sake, intended an audience inherently interested in logic, math, and engineering. The humanities must find its own intrinsic motivations for learning to code, broad enough to appeal to the community at large. For this reason, we begin our class with exercises that target daily computation and writing related tasks that are common to humanities researchers whatever particular research interest they may have. We create small "experiments" that address one's writing, for example. These include a lab session in which students analyze their own documents for commonly over-used "weasel words,"[6] for example. Working with one's own documents introduces important concepts like "relative" and "absolute" paths, file formats, and small shell utilities like `grep` (used to search through text files), `wc` (word count), `sed` (stream editor for text transformation), that can later be extended into more advanced concepts in system administration or natural language processing. When put together, these small utilities form the students' first programs, performing tasks like "safely rename all the files in this folder according to such-and-such rule," or "keep a daily log of my writing progress."

[6] Weasel words are words that sound very meaningful, but instead of adding, diminish the impact of persuasive writing. The "very" in the previous sentence, for example.

1.7 Bootstrap and time well spent.

When thinking of what to teach or where to invest our time, we look for "bootstrapping" effects that come from using powerful, universally available, and extensible software. The command line, for example, useful at first for file management and operating system exploration, later becomes an important resource for remote server administration, web design, and data science. Skills learned in the command line transfer to physical computing, fabrication, web scraping, and text analysis. Learning about relative and absolute paths locally, eventually helps to explain internet infrastructure, domain names, and resource allocation. It leads to Secure Shell and Pretty Good Privacy (PGP), both used by activists and journalists to protect their communications from surveillance.

It may be appealing at first to hide computational complexity behind "simple" visual interfaces. Yet, these interfaces do not share a common visual language and they do not transfer well across software platforms. Our colleagues in computer science sometimes worry[7] that introducing command line interfaces and raw coding environments may serve to alienate humanists. We believe that limited, "dumbed-down" interfaces do even more harm. They further alienate an audience that already feels removed from the material contexts of their daily knowledge production. In building the foundations, we want our students to spend their time well: to learn tools and skills that can support a wide variety of activity within diverse cultural contexts. The extra care we take in explaining the reasoning behind our technological choices works to motivate the students through any initial difficulties of learning how to code properly, without artificially hampered and patronizing simplification.

1.8 Divide big problems into small, modular components.

[Seems this could be collapsed into 1.2 for the sake of simplicity.]

Our goals in the classroom go beyond the instrumental. The ability to automate machines is merely a side effect of programmatic thinking. To learn to think like a programmer is useful in all contexts: it involves dividing big, complex, and seemingly intractable problems into small, modular, solvable components. Making a cake, for example, seems hard. But one can read, write, and follow a recipe—an exercise that we use in our teaching. Similarly, large projects from library administration to dissertation writing can benefit from the power of programmatic thinking.

1.9 Keep poking, get help, take notes, comment, annotate, share, remix, and train others.

[Could we collapse this into 1.0?][Possibly add in the importance of teaching

[7] Mention the CS chair observation [???].

them how to help themselves…]

Finally, and this was noticed by several course observers over the years, students in the humanities are sometimes curiously hesitant to explore their machines. In the absence of a tinkering upbringing and in the presence of cultural divides that regularly reinforce norms like "Those in the Arts and Humanities can't do programming, math, science, etc.", they might be worried of breaking expensive equipment or losing sensitive data. Demystifying the magical black box and learning some habits that prevent irrevocable loss of data addresses that fear, encouraging the students to tinker and to explore. Researchers at the later stages of their career may be particularly reluctant to ask for help or to express their questions in public. Our teaching approach therefore encourages shared expertise. We model not being afraid to expose our own gaps in knowledge. We ask students to keep copious notes, to annotate their code, to share and improve on each other's work. Notes grow into tutorials, tutorials into courses and workshops. Eventually, in following these principles, we hope for our students to become catalysts of change in their own communities of practice.

2 Digital Humanities Core

Programming can involve long stretches of frustration (Why does this not work?) punctuated by short bursts of elation that comes with accomplishing something difficult (It works!). Rather than allowing students to view their initial lack of results as failures, we attempt to channel feelings of hindrance into a practice of problem solving and discovery, related to similarly difficult, but more familiar tasks of archival research and long-form analytical writing. Just like writing, coding skills fall on a spectrum of proficiency that constitutes a small, but foundational part, of a larger variegated skill set.

Depending on one's research interests and career path, a DH practitioner will have some mixture of the following core skills (with examples in parenthesis):

- Text markup (plain text, *Markdown*, *Pandoc*, *TEI*)
- POSIX command line proficiency (*Bash*, regular expressions)
- Content management (*Jekyll*, *Wordpress*)
- Version control (*Git*, *GitHub*)
- Programming language (*Python*, *R*, *JavaScript*)
- Networking (SSH, VPN, hosting)
- Security (PGP)
- System administration (*AWS*, *Linux*, *Apache*, *MySql*)
- Project management (*GitHub Issues*, *BaseCamp*)
- Probability, statistics, algorithms

Computational practice in the humanities begins with text. Whatever our home discipline, we are all involved in the reading and writing of texts. It is natural

then to commence training with textual transformations: to understand how text is produced and where it resides physically, on the machine. Learning to author in a mark up language like HTML or Markdown is often a first step into the world of critical computing. Text transformations lead to the operating system and the file hierarchy. Tasks like regular backups and desktop organization can be used to familiarize the student with foundational concepts like relative and absolute file paths, or the distinction between plain text and binary formats. The idea of starting with daily computation also suggests covering version control and content management early in the curriculum. Although not an easy subject, version control comes naturally to a community used to thinking about drafts, manuscripts, and revisions. Increasingly, version control systems can also serve to host websites, to stream data, and to create full-blown publishing platforms. For example, *The Programming Historian* journal uses *Jekyll*, a static website generator, and GitHub Pages for publishing and distribution. Using that model to create personal academic profiles, image galleries, or critical editions is another good way to engage the humanities community. Git has the additional benefit of encouraging students to take notes and to journal alongside their work. Similar to the scientist's lab notebook, the Git journal fixes the flow of ideas and labor, helping teams keep track of work and attribution.

A programming language occupies a central place in computational practice. All forms of digitality pass through some form of encoding and automation. Only a small step separates text transformations, command line shells, and content management systems from a Turing-complete programming language. We often "trick" our students into programming by automating simple tasks like word substitution from the command line. The same task could then be repeated using *Python* or *R*, reinforcing skills learned earlier in the process. Because the internet plays such a key role in transforming academic practice, knowing the basics of networking: infrastructure, routing, packet switching, protocols, security, encryption are also key to higher level activity like preserving free speech online, protecting a journalist's sources, or bypassing censorship filters. The care and maintenance of personal document archives—research papers, article drafts, and book manuscripts—grows into server management. The server is where many of the skills learned earlier come to fruition. The running of websites requires a long "stack" of technological components. These are almost impossible to use well without knowledge of the command line, a programming language, and computer networks.

No project is complete without some sense of planning and organization. Project management is an important part of computation in the private sector, and, an increasingly formalized part of software engineering. Projects fail and succeed by the measure of their ability to coordinate action across time zones and continents. When teaching programming, we ask our students to start with "scoping" their projects in plain English first, then to transform these technical specifications into pseudocode, which then serves as the basis for program design and architecture. We ask our students to submit these documents along with code and consider them as important as a functioning program.

Finally, programming fundamentally involves a measure of algorithmic thinking. On some abstract level, the specific languages, tools, and implementations are secondary to the logical structures that support all higher level activity. This may be the most difficult obstacle to tackle. Every word cloud, every topic model, and network visualization tool hides a number of assumptions driven by sophisticated logic that comes from the fields of statistics and computer science. Without training in the methods we are bound to remain mere consumers of technology. Critical computing practice, like critical thought, requires a measure of logical and mathematical literacy.

We do not mean for this list to represent a comprehensive statement about computation in the humanities. Rather, we would argue that most projects, however large or small, employ at least *some* aspects from most of the above categories. Their ubiquity is what classifies them as "core" or "foundational" competencies. Few people apart from professional computer scientists and software engineers would claim mastery over the full stack of what is mentioned here. It is much more likely for digital humanists to develop proficiency in one or several areas of practice. Yet any one of the above foundational competencies have "spillover" effects in "leveling up" the rest of the list. Our intensive, week-long class can only begin to address a small part of the larger, complicated puzzle.

3 Three Locations of Computing

<!— JS: Our approach owes a small debt to Software Carpentry since this is what I have based a good chunk of the content on the first year and some of this got rolled into the second. This is likely the place to acknowledge it. DT: No problem. Let's do it in footnotes though. —>

We often begin our courses in outlining the above "big picture" principles, challenges, and considerations. There is no hiding the fact the programming is difficult and that people who do it well often have much "soft" knowledge (like command line expertise or game mod development), not formally taught in the university environment. Instead of offering quick but shallow victories, our approach is to engage the students intellectually. For this reason, we begin with the "frustration points" of everyday computing. In our experience, even simple tasks like saving a file from the internet browser is rife with anxiety and frustration. Where did that file go? How do I find it again? What type of file is it? Modern operating systems purposefully hide such information from average users. Yet our users are not average. Rather, they are students, faculty, and librarians who rely on documents, file systems, and datasets as a matter of critical importance. They have typically put years, if not decades, into acquiring expertise in academic disciplines and can certainly rise to the challenge of computation. In our experience, they respond extremely well to the mission of reclaiming the material contexts of their daily intellectual practice and learning how to control more directly the computers through which they carry out much of their research.

Guided by this approach, we have identified two core competencies from our "wish list"—ones that we believe can have a significant impact on "launching" a new student into the field of digital humanities and computational culture studies. The first of these is the Bash, a Unix shell and a command language for text-based "dialog" based interaction between humans and machines. The second is Python, a widely-used modern Programming language, that, like Bash, privileges simplicity and human readability.[8]

More directly, there are six properties that Bash and Python embody that make them particularly suitable for an introduction to coding class:

1. Simplicity. The idiosyncrasies of programming names within Bash/*nix aside, the syntax used within both Bash and Python is relatively straightforward. In Bash almost every task can be broken down into a series of single-line commands that could alternatively be chained together into a single line through pipes (|) and redirects (> and >>). Python does away with much of the frustrating syntax embedded in languages that came before it, saving users from needing to remember to close anywhere near so many brackets in exchange for a format that improves readability.
2. Power. Both Bash and Python are able to do a lot, pretty much any programming task really. Granted, they may not be well-suited to any programming task but together they can form a universal swiss army knife of coding, allowing users to get done what they need done.
3. Durability. Neither Bash nor Python are going away anytime soon, making it the case that solutions that are written in either will continue to be functional beyond the foreseeable future.
4. Universality. Both Bash and Python are widely used. While it is the case that Bash is confined to /*nix-style operating systems there are emulators that reproduce its functionality on most systems that a student in the class is likely to encounter. Python, and tools that help write Python code, is likewise available on all systems that a student of the class is likely to encounter. As a consequence of the ability to use both tools in a wide range of environments in conjunction with the other properties listed here both are used in a wide range of environments by very large user communities. This makes it the case that help is widely available.
5. Hackability. It is relatively straightforward to write one's own commands/methods within both Bash and Python *and to see how those that currently exist were written*. While this is not something that we cover in our courses the possibility for end users to investigate and expand

[8]Given that we have looked to Software Carpentry for some of the methodology that we employ in the course it should be noted that we do not spend any time on version control via tools such as Git or Mercurial. This was done initially so that more time could be spent on programming concepts, hands-on coding work, and unpacking the black box that is the command line. The importance of version control for efficient and effective coding via protecting against loss and enabling collaboration with others is recognized and future versions of the course may include it as a consequence. As with all training that is already time constrained down to the essentials, the challenge is what to take out to add this in.

the languages that they are working on goes directly to the heart of empowering users by unlocking what would otherwise be black boxes.

6. Fun. Packaging up the previous properties as they have been in both Bash and Python has resulted in the ability to have coding experiences that are not only effective but also fun. Both Bash and Python invite a sort of call and response methodology that is at once engaging and interesting, drawing students forward in their learning.

3.0 Bash?

> Bash is the GNU Project's shell. Bash is the Bourne Again SHell. Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.[9]

3.1 Python?

What is a programming language?

control structures + data types + built-in functions + syntax + interpreter Why Python

3.2 Text Editor

| When to use Bash | When to use Python | +————————————|————————————+ | - automate daily tasks | - data science | | - manage files & folders | - app development | | - remote server administration | - NLTK | | - data munging[10] | - data visualization | | - quick & dirty text manipulation | - glue code | | | - everything else |

Live coding. Conversational programming. Interactive. vs.

| Interactive Programming | Text Ed | +————————————|————————————+

[9] http://www.gnu.org/software/bash/

[10] Data munging is a recursive computer acronym that stands for "Munge Until No Good," referring to a series of discrete and potentially destructive data transformation steps Eric S. Raymond, "Mung," *The Jargon File*, 2004, online, Internet, 15 Jun. 2015., Available: http://web.archive.org/web/20150615165058/http://www.catb.org/jargon/html/M/mung.html.

Conclusion

These three locations build solid foundations for critical practice in the digital humanities. All more advanced stuff rests on these foundations. In starting here we dispell anxiety and illusion.

Harrell, D. Fox. "Toward a Theory of Critical Computing; The Case of Social Identity Representation in Digital Media Applications." *CTHEORY* 0.0 (2015). Online. Internet. 9 Sep. 2015. Available: http://journals.uvic.ca/index.php/ctheory/article/view/14683.

McIlroy, M.D., E.N. Pinson, and B.A. Tague. "UNIX time-sharing system: Foreword." *Bell System Technical Journal, The* 57.6 (1978): 1899–1904.

Raymond, Eric S. "Mung." *The Jargon File*, 2004. Online. Internet. 15 Jun. 2015. Available: http://web.archive.org/web/20150615165058/http://www.catb.org/jargon/html/M/mung.html.