Critical Computing in the Humanities

We write this chapter as a general reflection on teaching computing fundamentals in the humanities context, and more specifically in the wake of teaching *Computing Foundations for Human(s|ists)* at the Digital Humanities Summer Institute (DHSI), University of Victoria and *Computing in Context* at Columbia University.[[1]](#footnote-2) These courses were intended for humanities researchers with no previous programming experience who wanted to learn how programs work by writing a few simple, useful programs of their own.[[2]](#footnote-3) The syllabus was designed to foster a type of “digital literacy” that goes beyond using prepackaged tools and software and into the foundational skills that can support further self-guided exploration and intellectual growth.

To these ends, the topics covered in our classes included working with files and folders, advanced searching through large collections of texts, algorithmic thinking, data manipulation, and text analysis. The tools we use are few and simple: the command line interface included in most modern computers, the ubiquitous and powerful Python programming language, and a text editor. By the end of 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 evaluates poetry based on its measure of similarity to Byron.

Our aim in this chapter is not to recapitulate the experience of teaching, but to reveal some of the core principles that went into making the courses: to talk about the rationale behind our teaching philosophy, and, more broadly, to suggest an approach to teaching programming in the humanities.

We will elaborate on the above principles in three sections. “Critical Computing in the Humanities” describes the ideas behind our approach to computation, which is premised on extending rather than replacing long-standing critical practices of humanistic inquiry. In the second section, “Digital Humanities Core,” these ideas lead us to a list of core skills for the practicing digital humanist. These are meant to be neither totalizing nor exhaustive. Rather, we outline several necessary prerequisites needed to advance the great variety of work in the field. We conclude with a section that details the “Three Locations of Computing,” which orient the reader to three significant sites of computation: the command line, the text editor, and the programming language interpreter.

Ultimately, our essay gives the authors’ unified take on the skills and competencies required to advance research in the digital humanities and computational culture studies. The outlined program should be used as a guideline, not dogma. We hope it contributes to the broader conversation about curricular development, certification, and student and faculty training.

## 1 Critical Computing Principles

Computational methodologies can complement the rich history of research in the humanities. But to take hold, quantitative approaches to the study of culture must be adapted to more closely align to extant ideals and practices. Inspired by a number of initiatives advancing a similar philosophy, we refer to this approach as “critical computing.”[[3]](#footnote-4) The following seven propositions will connect technological preferences with values intrinsic to humanistic inquiry:

1. Demystify everyday computation.
2. Use few, free, small, universal, and powerful tools that you can alter and understand.
3. Privilege simplicity and human legibility over complexity and machine efficiency.
4. Invest in technologies that align with the values of our community.
5. Identify research objectives prior to selecting the appropriate tools and methodologies.
6. Divide big problems into small, modular components.
7. Be “lazy” by automating mundane tasks, but do it right by commenting your code, taking notes, and sharing with others.

### 1.1 Demystify everyday computation

Contemporary computational devices are a foundation of daily life. They are involved in everything from financial markets, to archival research, to the way many keep in touch with friends and family. Yet for those without technical training, the inner workings of these devices remain a source of mystery and, consequently, frustration. Recognizing this, our courses target the underlying structure of tools that many rely on for their daily computation, teaching our students how these tools work (and not just how to use them). Beyond the principles of programming, we want our students to understand the basics of networking, system administration, and project management. By revealing the innards of opaque computational “black boxes,” we hope to empower our students to take control of their everyday digital practice.

The most universal daily computing task of the humanities, regardless of research interest, is writing. For this reason, we structure our early classes by creating small “experiments” that address our students’ own writing habits. Such exercises might include a lab session in which students analyze their own documents for commonly overused “weasel words,”[[4]](#footnote-5) for example. Working with one’s own documents introduces important technical concepts like “relative” and “absolute” paths, file formats, character encoding, and small shell utilities like grep (used to search through text files), wc (word count), or sed (stream editor for text transformation). These concepts can later be extended into more advanced techniques in data manipulation or natural language processing. Short text-manipulation exercises 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.”

### 1.2 Use few, free, small, universal, and powerful tools that you can alter and understand.

Researchers, librarians, students, and faculty are faced with a bewildering array of software choices. In deciding whether to invest time and resources into learning a new tool or methodology, we are inspired by the free software movement in general and the Unix operating system philosophy in particular. The Unix philosophy of computing prioritizes small, modular pieces of software that “do one thing well.”[[5]](#footnote-6) Such software can then be chained together with other small but powerful tools to achieve complex results. Free software, besides being cost effective (“free as in beer”), also makes the tool itself available to inspection, interpretation, revision, and ultimately critique. Transparency and the ability to modify code to suit one’s own needs is what makes code “free as in speech.”[[6]](#footnote-7) Above all, we seek out universal tools that we can understand and, where needed, customize to fit our own particular applications. These tools can be applied in diverse contexts like library infrastructure, web design, data science, and the production of critical editions.

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. In other words, we privilege skills and concepts that will have the highest impact in the long run by transferring to the greatest number of contexts or tasks. The command line, for example, is useful at first to manage files, and later becomes an important resource for data gathering, cleanup, and analysis. Learning about relative and absolute paths locally makes it easier to understand networking protocols and uniform resource locators (URLs). Familiarity with the command line leads to the ability to administer servers remotely and to encrypt one’s communications—skills needed for journalists, activists, librarians, and scholars alike.

### 1.3 Privilege simplicity and human legibility over complexity and machine efficiency.

Whenever possible we privilege simplicity and human legibility over complexity and machine performance. The tools and file formats that we use in our research and archival efforts have serious implications for the long-term accessibility of academic knowledge and resources. The ubiquitous use of Microsoft Word and Adobe Acrobat file formats, for example, makes it difficult to publish, store, and to gather insight even from our own published work. As humanists begin to adopt the use of complex tools and databases, needless complexity becomes even more of a barrier to knowledge sharing.

Simplicity should not be confused with simple-mindedness. As with clarity in writing, clarity in computation comes from painstaking mastery of method and technique. Such mastery is fundamentally difficult, but it is to be preferred to shortcuts that sacrifice clarity for illusory “ease” of use. The student just entering the field of digital humanities and computational culture studies faces the choice of learning to program machines universally, or learning multiple, fragmentary, and non-standardized tools that have limited salience outside of the classroom. The proliferation of tools that obscure fundamental concepts in order to avoid “scaring” beginners adds complexity in the long run. Opaque and artificially hampered tools disfranchise an audience otherwise eager to take on new intellectual challenges. Their use prevents us from communicating with other computationally-minded disciplines and from competing meaningfully in the wider job market.

### 1.4 Invest in technologies that align with the values of our community.

The non-transparent nature of many popular file tools and file storage formats extracts a heavy toll on our community. Each tool that we add to our “tool chain” increases the cognitive burden of training, support, and peer review.

It may be appealing at first to hide computational internals behind “user-friendly” visual interfaces. Yet these interfaces do not share a common visual language; the labor of learning one interface therefore does not not transfer well across other software platforms. Our colleagues in computer science sometimes worry that introducing command line interfaces and raw coding environments may alienate humanists. We believe that limited, “dumbed-down” interfaces do even more harm, further mystifying computing to 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 can motivate the students through any initial difficulties of learning how to code “the hard way,” without shortcuts or artificial limitations.

In considering new tools and methodologies humanists who rarely work with truly large datasets would do well to weigh the risk of rapid obsolescence against any hypothetical gains in speed or performance offered by closed systems like a new note-taking platform, database system, or a proprietary text editor.[[7]](#footnote-8) When selecting a tool or a data format for storage, we ask: Does it need special software to render? How long has it been around? Does the community that support it align with our values? Our choice of the Python programming language, for example, was guided by the fact that Python encodes simplify and human readability into technical specifications.[[8]](#footnote-9) It has broad support from the scientific computing community and in the private sector. It is administered by a non-profit organization, which has articulated a clear diversity statement, has elected a trans woman to its board of directors in 2015, and routinely sponsors efforts, like PyLadies and PyCaribbean. Such efforts increase participation from publics traditionally underrepresented in the technology sector.[[9]](#footnote-10)

### 1.5 Identify research objectives prior to selecting the appropriate tools and methodologies.

Because the tools that we teach are universal, we are able to better tailor our courses to the diverse needs of our students. In formulating their research objectives, beginners often make the mistake of starting with the tool. In this way someone may describe their interests as “using topic modeling on a corpus of nineteenth century literature.” To this we reply: To what ends? Clearly articulated research objectives suggest appropriate tools and methodologies, and not the other way around. Thus the question of “genre formation in the ninetieth century,” for example, might lead to the use of topic modeling, while the study of narrative would perhaps be better served by other means, like event detection or automatic plot summarization. Our goal therefore is to give the students a glimpse of a rich and variegated toolkit that could help advance a variety of research objectives.

### 1.6 Divide big problems into small, modular components.

Our goals in the classroom go beyond the instrumental. The ability to automate machines is merely a side effect of algorithmic, analytical thinking. To learn to think like a programmer is useful in many contexts: it involves dividing big, complex, and seemingly intractable problems into small, modular, solvable components. Writing a grant proposal, for example, a book, or a dissertation may initially seem like a daunting and onerous task. Progress can be made once it is divided into small, doable steps, as though it were a recipe for making a cake (an exercise we use in the classroom). Our coding exercises therefore often begin by having our students describe their research objectives, step by step, in language natural to them and to their task.

### 1.7 Be “lazy” by automating mundane tasks, but do it right by commenting your code, taking notes, and sharing with others.

Pseudocode in plain English becomes the basis for well documented programs and readable code. Modular and well documented code does a service to the community: it is a pleasure to maintain and it communicates the purpose of the program clearly. It teaches others just as it allows them to further adapt the codebase to suit their own needs, to further share and to remix.

Good programmers are lazy in the right way. After doing a task more than a few times, a 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, we can delegate that task to the built-in scheduler called cron. Similar strategies can be used to improve bibliographic management, manuscript preparation, and research and editorial workflows.[[10]](#footnote-11)

Although programmers are lazy, they are lazy in the right way. Doing things badly or in a haphazard fashion accumulates technological, intellectual, and eventually an ethical debt owed to ourselves and to our peers. Code comments (or the lack of them) are a common site of egregious laziness: it is easy to skip documenting your code or to document insufficiently. Since it just works, one might say, why bother? However, a piece of code that makes perfect sense today may seem impenetrable tomorrow. Without comments, code becomes difficult to alter and maintain. For these reasons we advise our students against simply cutting and pasting code snippets from our (or anyone else’s) tutorials. We want them to think independently, to annotate, and to review their notes. In the broader academic context, lazy practices are unethical because they “bank” against the labor of others in the future. They make research more difficult to share and to replicate.

## 2 Digital Humanities Core

Programming can involve long stretches of frustration (“Why does this not work?”) punctuated by the short bursts of elation that come 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 discipline of problem solving and discovery. The “difficulty” of coding can be made more productive when related to the analogous and more familiar challenges associated with archival research, academic writing, and foreign language acquisition. Understood in this broader context, coding constitutes a small but foundational part of a larger, variegated academic skill set.

Depending on one’s research interests and career path, a DH practitioner will need to acquire a subset of the following core skills (with examples of particular technologies in parenthesis):

* Text markup (plain text, *Markdown*, *Pandoc*, *TEI*)
* Command line interface proficiency (*Bash*, pipes, regular expressions)
* Content management (*Jekyll*, *Wordpress*)
* Version control (*Git*, *GitHub*)
* Programming language (*Python*, *R*, *JavaScript*)
* Networking (cloud computing, Virtual Private Networks)
* Security (Pretty Good Privacy, Secure Socket Shell)
* System administration (*Linux*, *Apache*, *MySql*)
* Project management (*GitHub Issues*, *BaseCamp*)
* Design (data visualization, typography, user experience)
* Probability, statistics, and algorithms

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 the above “dream list.” The ubiquity of these modular components is what classifies them as “core” or “foundational” competencies. Few people, *including* computer scientists and software engineers, would claim mastery over the full range of skills we have delineated above. Rather, individual practitioners are likely to develop proficiency in one or several areas of expertise. An expert in digital publishing, for example, will have drastically different requirements from someone interested in geographic information systems or computational methods.

Critical computing in the humanities begins with text. Whatever our home discipline, we are all involved in the reading and writing of texts. Text gives us intrinsic motivation to explore our own computer environments: to understand how documents are produced and where they reside physically within the machine. Learning to author documents in a **text markup** language like HTML or Markdown naturally leads to more advanced topics like the basics of operating systems, file system hierarchy standards, and version control. For many of these competencies, familiarity with the **command line** is a prerequisite. Using the command line for mundane but necessary academic skills like regular file backup can familiarize students with foundational concepts like relative and absolute file paths or the distinction between plain text and binary formats. Familiarity with Bash and regular expressions extend competency on the command line to text manipulation methods.

The core principle of demystifying everyday computation leads to the topic of **content management** and **version control** early in the curriculum. Although not a simple subject, version control comes naturally to a community used to thinking about drafts, manuscripts, and revisions. Increasingly, version control systems are also serving as content management systems used to host websites, share data, and publish. For example, the editors of *The Programming Historian* use GitHub, a version control system, to publish and to distribute their journal. The team behind *Project GITenberg* “leverages the power of the Git version control system and the collaborative potential of Github to make books more open.” As of 2015, *GITenberg* hosts more than 43,000 books[[11]](#footnote-12). Using a similar model to create personal academic profiles, compile image galleries, or edit critical editions, our students learn while experimenting with aspects of academic production directly relevant to their careers. Furthermore, version control systems improve the quality of academic collaboration. Git and similar tools act as powerful research notebooks. They encourage all researchers involved to keep meticulous notes, which make it possible to document the flow ideas and labor and to attribute work fairly. For these reason we encourage the use version control early on and expect such systems to play an increasing role in academic evaluation.

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 manipulation at the command line using simple shell scripts from more advanced research-oriented programming languages like *Python*, *R*, and *Haskell*. We often “trick” our students into programming by automating simple tasks like word counting and file management at the command line. Thus small tasks like “create a directory,” “move a file,” “count all words in this directory” can eventually grow into text analysis.

Because the internet plays such a key role in transforming academic practice, knowing the basics of **networking** and network **security**—infrastructure, routing, packet switching, protocols, and encryption—is also key to pursuing higher-level goals like preserving free speech online, protecting a journalist’s sources, or bypassing censorship filters, particularly in places where politics and geography impinge on intellectual freedom. The care and maintenance of personal document archives—research papers, article drafts, and book manuscripts—naturally leads into server management. The server is where many of the skills learned earlier come to fruition. Running websites requires a long “stack” of technological components. Advanced **system administration** technologies “in the stack” like the Apache HTTP server, MySQL relational database management system, and the PHP programming language are difficult (if not impossible) to master without prior knowledge of the command line, networking, and programming fundamentals.

No project is complete without some sense of planning and organization. **Project management** is an important part of computation in the private sector, and it is an increasingly formalized part of software engineering. Projects succeed and fail by the measure of their ability to coordinate action across differences in time, temperament, and geography. 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 finally serves as the basis for program design and architecture. We ask our students to submit these documents along with code and consider them an essential part of the project’s output. In addition to getting better results, attention to project management prepares our students for working in groups outside of academia.

Because computational projects in the digital humanities often involve the creation of public-facing tools and archives, they necessarily overlap with the disciplines of **design**, such as data visualization, graphical user interfaces, and user experience design. Johanna Drucker has been a powerful voice in urging our community to encounter design both in practice and as an object of study.[[12]](#footnote-13) The disciplines of human-computer interaction and human factor engineering hold exciting possibilities for a productive collaboration between schools of engineering and the humanities.

Finally, programming fundamentally involves a measure of **probabilistic reasoning, statistical methods, and algorithmic thinking**. Without logic there can be no computation. Ultimately, the art of programming involves the ability to think algorithmically, to atomize complexity into discrete programmable steps, to formalize intuition, and to build models. Logic and statistical reasoning underlie every word cloud, every topic model, and every network visualization tool. Critical computing practice, like critical thought, requires access to deep structure. Those who aspire to become active and equal participants in the formation of computational knowledge (rather than mere passive recipients of tools and methodologies developed elsewhere) must at some point confront the established standards for training required for quantitative work in any field.

An intensive, week-long class, like the one that we teach at DHSI, can only begin to address a small portion of the skills required to run a successful project in the digital humanities. A diverse team of practitioners with complementary expertise will likely comprise any given digital humanities project. For this reason, in our teaching, we concentrate on the following “three locations of computing,” which support nearly all aspects of specialization represented above. Some familiarity with a text editor, the command line, and a programming language improves general digital literacy, useful to the librarian, the information scientist, and the computational humanist alike.

## 3 Three Locations of Computing

We often begin our courses by outlining the above “big picture” principles, challenges, and considerations. Like learning a foreign language, programming takes time and patience to master. And as is the case with any difficult skill, motivation to observe “best practices” correlates to chances of long-term success. Developing the intellectual motivations to stick with the program is therefore one of our paramount goals. For this reason, we strive to address the common “frustration points” of everyday computing. In our experience, even simple tasks like downloading an online file can be rife with anxiety for many students: *Where did that file go? How do I find it again? What type of file is it? Is it safe to open it?* Modern operating systems conceal from the average user the details needed to make informed decisions. Practitioners interested in fields like information science, critical software studies, platform studies, video game studies, or media archeology must learn to extend their inquiry beyond the available surfaces.

Our task as instructors is to reveal the hidden mechanics of computation. Moments of apprehension can be turned into opportunities for discovery. Students, university faculty, and librarians naturally approach documents, file systems, and datasets as matters of critical importance. Such artifacts preserve much precious intellectual labor. In our experience, students respond enthusiastically to the mission of reclaiming these material contexts of their daily intellectual production.

As suggested by the metaphor of the software development “environment,” a site of computation denotes an interface through which humans engage with machines. A “site” is also a conceptual space, containing a logic or a grammar of interaction. We find three such sites in searching for the universal grounds of general digital literacy: the command line, the language interpreter, and the text editor.

### 3.1 Command Line Interface

Because all interaction with the machine on the level of the operating system is in some sense an operation of files, the aspiring coder must develop a firm grasp of the file system topography. Despite its retro appearances, the modern command line offers an intuitive, text-based, “call and response” style of interaction with the file system consistent across a remarkable variety of platforms, from mobile phones to supercomputers, using the “terminal” or terminal emulator.[[13]](#footnote-14) Everything from downloading a sample corpus, to writing a research paper, to debugging code eventually leads to the command line. We therefore embrace it from the beginning of the course as the basis of our operations.

The command line is merely one way among several to communicate with a machine. On the level of hardware, the machine “speaks” in binary code. It “interprets” or translates English language-like commands (in a language called Unix Shell)[[14]](#footnote-15) into binary code. When deleting a file, for example, one instructs the machine to rm filename.txt instead of dragging and dropping it into the trash folder, as one would using a graphical user interface. Note that “dropping into the trash folder” merely offers a visual metaphor for the underlying bitwise operations. The terminal command transforms the metaphor into the more exact command, rm, which stands for “remove.” Similarly, to direct the computer to move a file, we would use the mv or “move” command. Unlike their visual and metaphorical counterparts, the shell commands contain many advanced options, not amenable to visual metaphor. The man command accesses the manual. Thus, man mv displays the manual pages for the move command.

Furthermore, because shell commands are in themselves a type of a programming language, they can be chained together to produce a “script,” or short program. The script can then be used to automate system administration of data analysis tasks. Just to give the reader an example of a command line script we will briefly examine the following lines of code, which ultimately give us a frequency counts for each word in Herman Melville’s *Moby Dick*. Download the [plain text version of the novel](http://www.gutenberg.org/cache/epub/2701/pg2701.txt) from Project Gutenberg and save as moby.txt to follow along.[^ln1-online-supplement]

We encourage the reader to follow along with these exercise by opening their terminal window and using the man command to learn about each of the steps involved. For example, man grep will show the manual pages for the grep utility. More detailed explanations of each step can be found online at the publishers site.

<!— Ray, there will be two exercises like these that we would like to publish online using iPython Notebooks. Please insert the correct link above, when you have it. —>

<!— Phil, please test all code on a Mac PRP: confirmed, but we should consider using a cleaned copy of Moby Dick with the Gutenberg front/endmatter removed —>

# find the whale  
grep "whale" moby.txt  
  
# substitute whale for chicken globally  
cat moby.txt | sed 's/whale/chicken/g' > chicken.txt  
  
# see what happened to the whales  
grep "chicken" chicken.txt  
  
# remove punctuation.  
cat moby.txt | tr -d "[:punct:]" > moby-nopunct.txt  
  
# check if it worked  
tail moby-nopunct.txt  
  
# translate all upper case into lower  
cat moby-nopunct.txt | tr '[:upper:]' '[:lower:]' > moby-clean.txt  
tail moby-clean.txt  
  
# sort by word frequency  
cat moby-clean.txt | sed 's/[[:space:]]/\'$'\n/g' | sort | uniq -c | sort -gr -k1 > word-count.txt  
head word-count.txt

We include the above examples to give the reader a compelling example of possibilities at the command line. The online exercises accompanying the present volume will give detailed explanation of the commands involved. For now, note that shell scripting encourages the “data flow” style of text processing. Vertical lines (|) and angle brackets (>) allow us to chain the commands into a system of pipes and redirects, passing the text output of one operation on to the next one. Once saved to disk, these small scripts can be be used to perform text transformations and frequency counts on any file.

Learning the command line is not just a matter interacting with files. With time it becomes possible to use commands like wc and sed to perform sophisticated data cleaning and text analysis operations. The above exercise also introduces the difference between relative and absolute file paths (documents/book/\*.txt vs. /usr/documents/book/\*.txt). It contains the basics of regular expressions ([a-z ]). Finally, the exercise can lead to the basics of remote server management, debugging, networking, security, and encryption.

### 3.2 Python

The second of our foundational sites of computing is the Python interpreter. Where the command line “translates” from shell into machine code, the *Python* interpreter translates from the *Python* language. Shell is a domain-specific command language, designed specifically to interact with the operating system. It traces its roots to the late 1970s, and this longevity makes it stable and ubiquitous. *Python*, on the other hand, was introduced in the early ’90s and became popular in the early 2000s.[[15]](#footnote-16) Unlike *Shell*, it is a general-purpose, high-level programming language. Like *Shell*, it privileges human readability, which fits with our principles.

|  |  |
| --- | --- |
| 1. When to use Shell | 1. When to use Python |
| 1. - automate daily tasks | 1. - data science |
| 1. - manage files & folders | 1. - app development |
| 1. - remote server administration | 1. - natural language processing |
| 1. - data munging[[16]](#footnote-17) | 1. - data visualization |
| 1. - quick & dirty text manipulation | 1. - web scraping |
|  | 1. - corpus analysis |
|  | 1. - everything else |

We chose *Python* among other excellent choices (like *R* and *Haskell*) for several reasons. First, *Python* is popular. According to the TIOBE language popularity index, *Python* holds roughly 3.6% of the market, trailing only behind *Java* and C-family languages (*C*, *C++*, *C#*).[[17]](#footnote-18) Although detailed statistics by field are not available, we infer that in the domain of scientific computing and data science *Python* holds the majority share of the market. This is important, because it means that learning *Python* is a good investment of time. It can lead to jobs inside and outside of academia, and projects using *Python* will have an easier finding collaborators than those using a less popular language.

*Python*’s popularity has an important side-effect. Being a general-purpose language, it has been adapted into a wide variety of contexts, from machine learning to web application development. The Python community packages common design patterns for any given application into ready-made “building blocks.” In aggregate, such building blocks comprise domain-specific software libraries, widely available for reuse. *Python* is consequently comprised of much more than the packages that are bundled in a fresh installation of the interpreter: it also includes the rich variety of software libraries and toolkits that it interacts with. For example, the Natural Language Toolkit contains libraries that perform many common tasks needed for text analysis.

Let us translate the same code we used to explore Melville’s *Moby Dick* in *Shell* into *Python*. As before, feel free to follow along online or on your machine. Do not worry if you do not understand all the code yet. In our class, we cover this material over the course of the week. For now the examples here should give the reader a general feel *Python* programming: grammar, logic, and control structures. (Note: This example uses Python 3. Installations of Python 2.x should omit parenthesis for print commands.)

<!— Phil, please double check each line in iPython —>

Let’s begin by finding all the whales and substituting them for chicken, just for fun.

# open file and read contents into a list of lines  
# mimics the shell behavior in the previous example  
with open('moby.txt', 'r') as f:  
 lines = f.read().splitlines()

Unlike *Shell*, *Python* is a object-oriented language. Just as everything in the Unix world is a file, everything in the *Python* world is an object. Objects have methods associated with them them. You can imagine an object of the type “dog,” for example, to have methods like “sit” and “fetch.” And object of the type “cat” to have methods like “hide” and “hunt.” Methods often return other objects. Thus we may expect the method dog.fetch() to return an object of the type toy. Note that the parenthesis help differentiate the method from the object itself: one is a verb, the other a noun.

But the built-in *Python* objects are limited to just a few primitive types like string, integer, and list. The type of data manipulation we show here depends on knowing what type of object we are working with at all times. When we first open moby.txt we “stuff” it into an object of type *file* that we call, arbitrarily, f. File objects have useful methods like read() which returns an object of type *string*. Strings have the helpful ability to arrange themselves into lines, which are delineated by hidden newline characters. splitlines() uses this feature of the string to return a *list* of strings, which we can assign to an arbitrarily named variable lines. Because splitlines() returns lists, Python attempts to do the right thing by making the lines of the type *list*.

To check the type of the object we can run type(lines). print(lines) will show the contents of our list container. Let us now try to replace all whales with chickens, just for fun:

# replace whale for chicken in every line and print results  
for line in lines:  
 if 'whale' in line:  
 print(line.replace('whale', 'chicken'))

Note that although we have not explained control structures like for and if, their use is pretty intuitive. The Pythonic for line in line is not too far from the English “for every line in lines”. The next line says to do something if the word “whale” is in the line. Where lines is a *list*, each individual member of the list is an object of type *string*. Like dogs and cats, strings can do things, as they have methods attached to them. The method replace() works as expected. Unlike self-contained methods like splitlines(), the replace() methods takes two arguments: first, the word to be replaced, and second, the replacement word. Such details can be found in the Python documentation and become more familiar with time.

In conclusion, we come to an operation central to any computational text analysis. To count the unique words as we did before, we need to divide up each line into words. We can then get rid of punctuation, make everything lower case to avoid duplication, and create a list of all words found in the novel. The list of all words is commonly referred to as “tokens” where the list of unique words gives us the “types.” The type–token distinction is incredibly useful in stylistic analysis, for example. It can be used to build more complex models about the quality of writing for example, or about the age range of the intended audience. We expect sophisticated prose by adults to exhibit a high token-to-type ration. Children’s literature uses a more limited vocabulary that repeats more often, giving us a lower ratio value.

from string import punctuation  
from collections import Counter  
  
tokens = []  
  
# split each line into a list of words  
# remove punctuation and make each word lowercase  
# append "cleaned" words to the list "tokens"  
for line in lines:  
 for word in line.split():  
 tokens.append(word.strip(punctuation).lower())  
  
# display 100 most common types  
types = Counter(tokens)  
types.most\_common(100)

Conveniently for us, *Python* strings have the method split(). First we import some helpful libraries that contains some of the logic that we need to perform our operations. Then we declare an empty list and give it an arbitrary name, tokens. We then iterate over each line, and after that over each word in the line. Once the word has been stripped of punctuation (another neat trick given to us by the built-in *Python* functionality) and converted to lower case, we append it to our list of words. The only thing that remains is to count unique word types with Counter(), the second method we imported, and to print the most common words in the novel using the available most\_common() method. Note that even without any preparation, the logic of the code above is readily apparent. Python’s built-in functions sound like English. They are easy to read and therefore easy to share and maintain.

Of course, the above code could be written in a more concise way. We opted for code that is more verbose but also for more expressive. The reader can perhaps already tell that while *Python* is wordier, it offers many more built-in features than *Shell*. This would be even more apparent if we were working with images or binary formats instead of plain text files. For a quick count of words in a novel, we may initially opt to use the Unix shell. As our models and logic grow more complex, though, we are likely to begin writing in *Python*.

### 3.3 Text Editor

The humble text editor is the third and possibly most important site of computing in the digital humanities. In addition to supporting programming, the text editor mediates our research and publication practices. For this reason, we ask our students to reevaluate their relationship with tools like Microsoft Word and Google Docs. These often fail to meet our community’s criteria for usability and they do not align well with humanistic values. At the very least, to write code we need a *plain text* editor that does not add any hidden formatting characters to our code. We also need an editor that we can modify and extend without being hampered by proprietary licenses or restrictions. Many text editors meet these criteria. Among them are *Atom*, *Emacs*, *Leafpad*, *Notepad++*, and *Vim*.

Where command prompts and *Python* interpreters allow for an “interactive,” back-and-forth dialog style of programming that happens in real time, the text editor gives a measure of permanence to the conversation. When working with data sets, we often begin with exploratory data analysis at the command line or an interactive Python interpreter to familiarize ourselves with the data and form intuitions about its explanatory potential. Once those intuitions are formed, we can move to writing and debugging code in the text editor. The code helps describe our formal models. It lets us test our intuitions against the dataset as a whole. The Python interpreter and the shell remain open in the background. We will use them them to manage our files and to test snippets of code. But the text editor is where we can begin to think algorithmically. It is where our projects gain a more permanent shape.

About halfway through the session, the students are ready to formulate a project of their own. Rather than using prepackaged exercises, we encourage our students to formulate small research projects related to their own work or interests. In our last class, a group of librarians built a program to copy selected metadata from one .csv file to another while checking for errors in data formatting (like an improperly formatted date, for example). Another group built an automatic essay grader. Yet another analyzed poetry for its similarity to Byron. A fourth group wrote a script that automates the download of a film script corpus.

All of these projects begin with a set of step-by-step instructions in English. Thus, a simple essay grader may be expressed as follows:

# Open and read a file.  
# Calculate variation of sentence lengths.  
  
# Assign a score based on variation.  
  
# Calculate a measure of linguistic variety.  
  
# Assign a vocabulary score.  
  
# Find common spelling mistakes and "weasel words."  
  
# Average the scores to come up with a total grade.

Once formalized, we can begin to convert the logic from the English language pseudocode into Python, expanding and filling in the code under the comments.

Using this process, students work alone or in groups to define the scope of their program. Even at this stage we can begin a critical discussion about the models implicit in the proposed program. For example, does the above logic for an essay “grader” accurately capture what we mean by “writing well” or “writing poorly?” Is it enough to reduce notions of value to “a measure of linguistic variety?” What can we do to make our model more robust and to make it better correspond to our native intuitions about literary value? In another recent course at Columbia University, students building an automatic essay-grader had to explain and defend the basis of their grading criteria. In doing so, they confronted their own implicit criteria for good writing and initiated a spirited debate about algorithmic judgments of clarity and style. Some students rewarded rich vocabularies. Others looked for variation in sentence and paragraph length. In this way, we use the algorithm to challenge intuitions about academic writing.

When the scope and logic of the program have been determined, we work with individual groups to help translate the English-language heuristic into workable Python code. Inevitably, the programs grow more sophisticated. In the above example, students used published work to test their grading algorithm. It was interesting to see how Ernest Hemingway fared against David Foster Wallace, for example. In a longer course, we may have introduced supervised learning techniques to classify essays for quality based on similarity to work that has already been evaluated. The difficulty of any specific project may be tailored to the length of the course and to the level of individual expertise. During such free-form “laboratory” sessions we encourage students to help each other and to share expertise with their peers.

The command line, the Python interpreter, and the text editor provide the foundations of critical computing in the humanities. We do not expect all of our students to become programmers. But at the very least, they become exposed to a powerful problem-solving method and to operating system internals used widely in all aspects of computation, from sending email to writing papers.[[18]](#footnote-19)

## Further Reading

* Brian W. Kernighan[[19]](#footnote-20)
* Jeroen Janssens[[20]](#footnote-21)
* DH Notes
* [Digital Humanities Research Portal](https://www.computecanada.ca/research-portal/digital-humanities-working-group/), Compute Canada
* Christopher D Manning and Hinrich Schütze[[21]](#footnote-22)
* Steven Bird and Ewan Klein[[22]](#footnote-23)
* Charles Petzold[[23]](#footnote-24)
* Scott Chacon[[24]](#footnote-25)
* Project Jupyter
* PyLadies
* Python Software Foundation
* The Programming Historian
* Allen Dawney[[25]](#footnote-26)

## References

Bird, Steven, and Ewan Klein. *Natural Language Processing with Python*. vols. 1 edition. Beijing ; Cambridge Mass.: O’Reilly Media, 2009.

Bourne, S. R. “UNIX Time-Sharing System: The UNIX Shell.” *Bell System Technical Journal* 57.6 (1978): 1971–1990. Online. Internet. 7 Nov. 2015. Available: <http://onlinelibrary.wiley.com.ezproxy.cul.columbia.edu/doi/10.1002/j.1538-7305.1978.tb02139.x/abstract>.

Chacon, Scott. *Pro Git*. vols. 2nd ed. 2014 edition. Apress, 2014.

Dawney, Allen. *Think Python: How to Think Like a Computer Scientist*. vols. 2 edition. O’Reilly Media, 2015.

Drucker, Johanna, and Emily McVarish. *Graphic Design History*. vols. 2 edition. Boston: Pearson, 2012.

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>.

Hellman, Eric, Raymond Yee, and Seth Woodworth. “Project GITenberg,” 2012. Online. Internet. 10 Nov. 2015. Available: <https://gitenberg.github.io/>.

Janssens, Jeroen. *Data Science at the Command Line: Facing the Future with Time-Tested Tools*. vols. 1 edition. O’Reilly Media, 2014.

Kernighan, Brian W. *D is for Digital: What a well-informed person should know about computers and communications*. CreateSpace Independent Publishing Platform, 2011.

Lutz, Mark. *Programming Python*. Nutshell Handbook. Sebastopol, Calif: O’Reilly, 1996. Online. Internet. Available: <http://ezproxy.cul.columbia.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1378&site=ehost-live&scope=site>.

Manning, Christopher D, and Hinrich Schütze. *Foundations of statistical natural language processing*. Cambridge, Mass.: MIT Press, 1999.

McDaniel, W. Caleb. “Wcaleb (W. Caleb McDaniel).” *GitHub*, Nov. 2015. Online. Internet. 10 Nov. 2015. Available: <https://github.com/wcaleb>.

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.

Petzold, Charles. *Code: The Hidden Language of Computer Hardware and Software*. vols. 1 edition. Microsoft Press, 2000.

Python Software Foundation, The. “Diversity.” *Python.org*, 2015. Online. Internet. 9 Nov. 2015. Available: <https://www.python.org/community/diversity/>.

———. “PSF Board Resolutions.” *Python.org*, Oct. 2015. Online. Internet. 10 Nov. 2015. Available: <https://www.python.org/psf/records/board/resolutions/>.

———. “Python Software Foundation.” *Python.org*, 2001. Online. Internet. 9 Nov. 2015. Available: <https://www.python.org/psf/>.

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>.

Software, TIOBE. “Tiobe Index,” Sep. 2015. Online. Internet. 12 Sep. 2015. Available: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

Stallman, Richard. “Why Open Source misses the point of Free Software,” 2007. Online. Internet. 11 Sep. 2015. Available: <http://www.gnu.org/philosophy/open-source-misses-the-point.en.html>.

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”. [↑](#footnote-ref-2)
2. An archived version of the DHSI course can be accessed at http://web.archive.org/web/20150614161609/https://github.com/denten-workshops/dhsi-coding-fundamentals/blob/master/README.md [↑](#footnote-ref-3)
3. We are not the first nor the only instructors to think about digital pedagogy this way, nor are we the only ones to be offering a course based on these principles. Software Carpentry, for example, 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>. [↑](#footnote-ref-4)
4. Weasel words are words that sound very meaningful, but diminish instead of adding to the impact of persuasive writing. The “very” in the previous sentence, for example, is a weasel word. [↑](#footnote-ref-5)
5. 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. [↑](#footnote-ref-6)
6. Richard Stallman, “Why Open Source misses the point of Free Software,” 2007, online, Internet, 11 Sep. 2015., Available: <http://www.gnu.org/philosophy/open-source-misses-the-point.en.html>. [↑](#footnote-ref-7)
7. 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 work with plain text. In fact, the notion of human readability is even encoded at the operating system level. [↑](#footnote-ref-8)
8. Python Enhancement Proposal 20 reads: “Simple is better than complex. Complex is better than complicated Sparse is better than dense. Readability counts. [↑](#footnote-ref-9)
9. See The Python Software Foundation, “Diversity,” *Python.org*, 2015, online, Internet, 9 Nov. 2015., Available: <https://www.python.org/community/diversity/>, The Python Software Foundation, “PSF Board Resolutions,” *Python.org*, Oct. 2015, online, Internet, 10 Nov. 2015., Available: <https://www.python.org/psf/records/board/resolutions/>, and The Python Software Foundation, “Python Software Foundation,” *Python.org*, 2001, online, Internet, 9 Nov. 2015., Available: <https://www.python.org/psf/> for further detail on the Python Software foundation and its initiatives. [↑](#footnote-ref-10)
10. The public GitHub code repository of W. Caleb McDaniel, a historian at Rice University, is exemplary in this regard (W. Caleb McDaniel, “Wcaleb (W. Caleb McDaniel),” *GitHub*, Nov. 2015, online, Internet, 10 Nov. 2015., Available: <https://github.com/wcaleb>). [↑](#footnote-ref-11)
11. Eric Hellman, Raymond Yee, and Seth Woodworth, “Project GITenberg,” 2012, online, Internet, 10 Nov. 2015., Available: <https://gitenberg.github.io/>. [↑](#footnote-ref-12)
12. See, for example, her work with Emily McVarish on *Graphic Design History* (Johanna Drucker and Emily McVarish, *Graphic Design History*, vols., 2 edition. (Boston: Pearson, 2012)). [↑](#footnote-ref-13)
13. While some platforms default to a text-based command line (the “terminal), most modern graphical machines use a”terminal emulator" to achieve the same results: Windows, Mac, and Linux have built-in terminal emulators and support many third-party applications that serve the same function. [↑](#footnote-ref-14)
14. See S. R. Bourne’s overview for more detail on the Unix shell S. R. Bourne, “UNIX Time-Sharing System: The UNIX Shell,” *Bell System Technical Journal* 57.6 (1978): 1971–1990, online, Internet, 7 Nov. 2015., Available: <http://onlinelibrary.wiley.com.ezproxy.cul.columbia.edu/doi/10.1002/j.1538-7305.1978.tb02139.x/abstract>. [↑](#footnote-ref-15)
15. Mark Lutz, *Programming Python*, Nutshell Handbook (Sebastopol, Calif: O’Reilly, 1996), online, Internet., Available: <http://ezproxy.cul.columbia.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1378&site=ehost-live&scope=site>. [↑](#footnote-ref-16)
16. Data munging is a recursive computer acronym that stands for “Mung 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>. [↑](#footnote-ref-17)
17. TIOBE Software, “Tiobe Index,” Sep. 2015, online, Internet, 12 Sep. 2015., Available: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. [↑](#footnote-ref-18)
18. A detailed history of author contributions can be found on our GitHub page at <https://github.com/denten-workshops/dhsi-coding-fundamentals/commits/master/book-chapter/main.md> [↑](#footnote-ref-19)
19. *D is for Digital: What a well-informed person should know about computers and communications* (CreateSpace Independent Publishing Platform, 2011). [↑](#footnote-ref-20)
20. *Data Science at the Command Line: Facing the Future with Time-Tested Tools*, vols., 1 edition. (O’Reilly Media, 2014). [↑](#footnote-ref-21)
21. *Foundations of statistical natural language processing* (Cambridge, Mass.: MIT Press, 1999). [↑](#footnote-ref-22)
22. *Natural Language Processing with Python*, vols., 1 edition. (Beijing ; Cambridge Mass.: O’Reilly Media, 2009). [↑](#footnote-ref-23)
23. *Code: The Hidden Language of Computer Hardware and Software*, vols., 1 edition. (Microsoft Press, 2000). [↑](#footnote-ref-24)
24. *Pro Git*, vols., 2nd ed. 2014 edition. (Apress, 2014). [↑](#footnote-ref-25)
25. *Think Python: How to Think Like a Computer Scientist*, vols., 2 edition. (O’Reilly Media, 2015). [↑](#footnote-ref-26)