Introduction & FAQs

Welcome (again)! This is the introduction to *The JavaScripting English Major*, a course written by me, [Moacir P. de Sá Pereira](http://moacir.com), on teaching humanities undergraduates to write web projects. In this introduction, I provide a few FAQs that explain how I approach the design of this course. If you look at the [About](/about) page, you can learn even more about the origins and technical foundations of the course.

If you’re already committed to becoming a JavaScripting English major, you can skip this chapter and jump straight to the [first chapter](/1-environment), which walks you through setting up your environment.

My goals in the course are straightforward; at the end of 15 sessions, you should be able to:

* host a web project that investigates an academic topic of your choosing;
* consider means by which you can present the topic to a public larger than the class;
* write JavaScript, HTML, Markdown, and CSS;
* use Atom as a development environment for your project;
* create and manipulate Leaflet maps;
* manage GeoJSON data;
* use Git to keep track of your project; and
* think about your literary study as a process that changes over time.

## What are the prerequisites?

If you’re on this website, you’ve got most of the prerequisites covered, as it implies that you have a web browser and the desire to learn.

I believe that anyone can learn to program, but it helps if you are optimistic and fond of puzzles. Programming is, like many intellectual pursuits, a method of creatively solving problems. Like with any puzzle, when you program, you use your mind to realize a goal within a certain set of limitations created ahead of time, be they the shapes of the puzzle pieces of the grammar of the programming language.

The path to your goal can be (and often is) convoluted and unexpected, but that means that reaching the end is that much more satisfying.

However, some familiarity with HTML is useful, as I only give a cursory explanation of it throughout the course. My rationale is straightforward, though. The best way to learn HTML is to see how others use it and retool that for your own purposes. That kind of copy-paste pedagogy won’t work for JavaScript, but it’s a classic method for learning HTML. “View Source” has been a useful, if hidden, command on the browser for two decades for just this reason.

## What if I’m not an English major?

This course is for any student, really. But I was an English major, and I am teaching this in an English department, so I prepare the materials with a specific audience in mind. After all, both mini-projects deal with works of English literature, Geoffrey Chaucer’s [General Prologue](https://en.wikipedia.org/wiki/General_Prologue) and Langston Hughes’s “[Could Be](http://www.songofamerica.net/song/could-be).”

I don’t believe that future computer scientists or students eager to learn about the foundations of programming will find this course particularly illuminating. I am opinionated in how I present the materials, because I am not training programmers. I’m teaching English majors how to program. As a result, the coverage is also thin. Object-oriented programming comes up only implicitly, and this reference right here to ES6 is the only such reference in the entire course. Nevertheless, this course is based on Marijn Haverbeke’s [*Eloquent JavaScript*](http://eloquentjavascript.net), which is written more for a programmer in mind. The order in which I present concepts is, not coincidentally, similar to Haberbeke’s, meaning the book is a useful companion.

My own practice reveals the need I serve with this course. Most programming books I have read fall into one of two categories; they are either introductions to computer science that use a specific programming language for the introduction (Haverbeke, Allen Downey’s [*Think Python*](http://greenteapress.com/wp/think-python-2e/), or even the mindbendingly great [*The Little Schemer*](https://mitpress.mit.edu/books/little-schemer)), or they are introductions to a specific programming language aimed at people who already know how to program (David Black’s [*The Well-Grounded Rubyist*](https://www.manning.com/books/the-well-grounded-rubyist-second-edition)). For my students, the former seem too detailed and get abstract too quickly, while the latter move too quickly (and get abstract almost instantly).

So just as a university might offer different statistics courses depending on the students’ interests, including a sort of “Stat for people who aren’t scientists,” I propose a course that assumes very little advance knowledge, like the first category of books, but also tries to push process and practice over fundamental concepts in computer science. More than anything, I aim to teach by example and by remembering who my students are.

There are books that are more in the vein of this course, such as Matthew Jockers’s [*Text Analysis with* R *for Students of Literature*](http://www.matthewjockers.net/text-analysis-with-r-for-students-of-literature/). Similarly, the title of the course is taken from [*The Programming Historian*](https://programminghistorian.org/), a set of online tutorials that teach basics of programming and digital historical methods to historians with little or no programming background. So if this course can be considered “Jockers, but for JavaScript,” I think I will be very satisfied, and I suspect the same will be true for my students.

## Why JavaScript?

The previous question notes that Matthew Jockers has already written a programming book for English majors, so this question becomes acute. Why learn JavaScript *instead* of R? I believe there are two reasons, and the second follows from the first.

First, JavaScript is *ubiquitous*. Every contemporary laptop or desktop ships with a browser (used to interpret JavaScript) and a text editor (used to write JavaScript). In fact, for simple JavaScript, a browser alone will suffice. One can begin JavaScripting without installing a single new program, though this course does encourage installing the text editor [Atom](http://atom.io).

Second, because JavaScript is ubiquitous, it becomes very easy to see results quickly. The first time I taught programming, I was teaching Python to English majors. But the moment that excited them the most was when I used JavaScript to deface a website. In seconds, I was able to do something they wanted to photograph and share with their friends that would take an entire class period to do in Python.

And that element of sharing is important in this course. A large part of the excitement of engaging in a creative process is sharing the results of that process with others. And progamming is, believe it or not, a very creative process.

## Why not Python or R?

[Python](http://www.python.org) is great. [R](http://r-project.org) is great. [Ruby](http://ruby-lang.org) is my favorite language. And so on. If you have a chance to take a course in any of those languages, please do so. However, all of them require more initial setup than JavaScript does, which thwarts my students from making web projects as quickly as possible while also understanding every line of code they type.

JavaScript has a peculiar origin story. It was written in a ten-day frenzy by [Brendan Eich](https://en.wikipedia.org/wiki/Brendan_Eich), and then Netscape decided to name it “JavaScript,” which makes it sound like Java’s little brother, even though JavaScript has nothing to do with Java. Nevertheless, these peculiarities have contributed to JavaScript’s reputation as a “toy” language, a reputation that persisted for over a decade.[[1]](#footnote-41) More recently, however, JavaScript has emerged as a vital language for doing anything web-based, both on the client (browser) side and on the server side.

Because of its importance to the web, JavaScript will appear even when writing web applications in Python or Ruby. So why not learn the language that is unavoidable? After all, once you have learned the basics of programming in one language, jumping to another becomes easier.

## Atom!?! But Vim is the one true editor!

Any text editor in the world can be used to write JavaScript. TextEdit, Notepad, [Vim](http://www.vim.org), and even [Emacs](http://www.gnu.org/s/emacs). This course encourages students to use [Atom](http://atom.io), however. Atom brings three benefits that the others do not.

1. Atom is available (for free) for Windows, Mac, and Linux, and it looks more or less the same on all of the operating systems. I don’t want to be platform specific when teaching, because I’m already relying on the students’ having some access to some/any computer. I’d rather not rely on it being a specific kind of computer.
2. Atom has nice Git and [GitHub](http://github.com) support preinstalled. Part of the skills taught in this course include thinking about a creative and intellectual process that includes versioning, so Git-in-the-box is a great feature.
3. Atom’s configuration files are written in the same languages as used on the web, namely JavaScript (CoffeeScript) and CSS, meaning learning JavaScript also sets the student up for being able to hack Atom.

If you would like to take this course using Vim and Git from the command line, that is entirely up to you. Of course, if you’re already using Vim, then you might be too advanced for this course, anyway.

## There’s a lot of “how to make a website” stuff here, but not a lot of analysis

Correct. This course is not designed to teach you to be a better literary scholar. It teaches you how to use JavaScript in order to be a better literary scholar. I believe that process- and project-based learning are useful engines of interpretive practice, meaning that submitting a literary text to the kind of work we do in this course can only help but augment the experience of reading the text.

Jockers’s book, mentioned above, is about using R as a tool for literary criticism. This is about using JavaScript as a tool to make more (and different) literary criticism possible down the line. That is why the focus is on handling and presenting information, not (geo) statistical analysis, say. The “analysis” comes later. I don’t provide, by the end of the [demo](/examples/could-be.html), a new reading of Langston Hughes’s “Could Be.” What I do, instead, is provide an opportunity for scholars to increase their knowledge of the text and look at it in a different way, thereby enhancing that reading.

## Teaching people how to code is just a Silicon Valley ploy to drive down developer wages

Of course. However, the goal of this course is not to teach students how to become coders so that they can dilute the programmer pool in California. Instead, I am teaching process- and project-based creative thinking when it comes to literary study through programming. If students are inspired to continue learning to program as a result of this course, then they can consider that sector of the job market down the line.

That said, each instance of bias coming out of Silicon Valley reveals how much they need to hire more people with humanities training. As a result, learning to code will help a humanist get hired in Silicon Valley—but probably not as a developer. Instead, the company will likely want to leverage the student’s humanities skills. In such a way, their familiarity with programming is a bonus for the company’s plans to make the world a better place. English majors have roles in Silicon Valley, but it won’t be among the engineers.

## But isn’t digital humanities a neoliberal scheme?

Please see the multiple volumes of [*Debates in the Digital Humanities*](http://dhdebates.gc.cuny.edu/). However, it’s likely that even those books won’t convince you otherwise.

## Why should I trust you?

This course is called “The JavaScripting English Major.” I was an English major for many years. I then became a PhD student in English, and now I’ve leveled up to being an Assistant Professor / Faculty Fellow in English at NYU. In other words, in contrast with literary-minded programming books like Angus Croll’s [*If Hemingway Wrote JavaScript*](https://www.nostarch.com/hemingway) or the peculiar Shakespeare epigraphs in every chapter of Douglas Crockford’s otherwise superb [*JavaScript: The Good Parts*](http://shop.oreilly.com/product/9780596517748.do), this work considers students of literature first and programming second.

At the same time, I have been programming for three decades—so longer than I have been an English major. I moved from writing fantasy games in BASIC on an Apple IIe to writing HyperCard on my Mac to programming my TI-81 to learning Perl and PHP for web applications to having a part-time student job as a Ruby developer. I find programming to be a rewarding exercise because there’s always at least something that almost sort of works at the other end. Programming is, of course, also frustrating. Irritating. Infuriating. But then it is also, again, so rewarding.

But the main reason you should trust me is that I teach this course, making adjustments where necessary. And I hope to keep teaching it. JavaScript and the web are going nowhere, and the humanities are as important now as they have ever been.

So happy JavaScripting, and I’ll see you in [Chapter 1](/1-environment)!

## Footnotes

1. I discuss the history of JavaScript in a bit more detail in [Chapter 2](/2-calculator). [↑](#footnote-ref-41)