Think about what you've done today. If it's early in the morning, maybe you opened up Outlook to check emails or caught up on Instagram. If it's the afternoon, you may have used Spotify during your commute or done work on your Mac. If it's late at night, you might be unwinding with a TV show on Netflix or ordering food through Uber Eats. Whatever it is you're doing, whatever time of day it may be, there's a very likely chance that you are interacting with some kind of technology—more specifically, some kind of software. "But I'm six years old and I don't have a phone or laptop!" you say. Think about the digital billboards you see in the train station, or the self-checkout your parents use at Target, or even any elevators—how does the elevator know which floor to go to when two people call it simultaneously? These are topics and questions that fascinated me as a kid and sparked my interest in the common denominator between them all: software.

Growing up in the city with two working parents, I often found myself being much too young to go out alone yet a bit too old to be playing with toy cars and staring at the TV. One day, my dad came home with an old laptop that his friend had no use for anymore, and thus began my fascination with computers and the internet. Although I would eventually go on to study computer science in college, I didn't actually touch a single line of code until about the age of 16 or 17. Instead, I spent a lot of my time playing video games, customizing various aspects of my operating system (i.e. custom icons, cursors, taskbars, etc), and surfing the web. These experiences served as the backbone of my interest in software, and by around highschool, I knew that I did not want to just consume digital content but also help create and refine it. This is around the time where I began watching videos and reading various articles about how exactly computers work and how applications are created. Having a friend who was also interested in the field, one of the first things I explored on his recommendation was Linux, which I first read up on through a Wikipedia article [1]. I was always fascinated by the prospect of customizing the things I used (as I'm sure many other CS majors can relate to), so this exploration of Linux led me into a spiraling hole of testing different Linux distributions, rooting my Android, etc.

Past this point of unnecessarily customizing every single piece of software I interacted with, I was getting closer to college and began thinking about possible majors that I could pursue. One of the resources I decided to consult was the College Board Book of Majors; I already knew I wanted to do something with technology, so I read through all the relevant sections in search of something that I may be interested in pursuing. Keep in mind that at this

point in my career, I knew *how* to use various pieces of technology but I still wasn't quite sure how any of it worked from the inside. It's also important to note that growing up in Queens, New York, all of my friends' parents (including my own) were first-generation immigrants who all worked relatively basic, white collar jobs—that is to say, they were aware that majors like "computer science" existed, but they were not quite sure what it required or entailed. After reading through the various sections, I found that there were two majors that caught my eye in particular: computer engineering and computer science.

I began reading about what exactly it was that computer engineers and scientists did, and, arguably most importantly, how much they made. I consulted various online articles, YouTube videos, and even spoke with my (at the time) current math teacher, who worked as an electrical engineer before becoming a teacher. Although I did not end up making up my mind about which major to choose until only a few months out from college, I knew that software was an integral part of both majors and so I began to look into programming. While I didn't quite have the resources or knowledge to access electrical components and mess with the hardware side of computers, I did have my own computer which was all I needed to get started with the software side of things. The very first programming language I decided to learn was Python, and thus began the rabbit hole of exploring a new programming language. I constantly combed through Python documentations and forum discussions to learn new language features, as well as to debug any issues I ran into with my code. Within the first week or two I was up and running, writing code at extended lengths and creating tiny programs. After only a month or two of learning, I was able to create simple programs that were fully functional.

One of the first programs I created was a simple countdown program that, given a certain date, would tell you the days, hours, minutes, and seconds until that given date, and would also allow you to assign a name to the event. After creating this program, I ended up taking some time to document what exactly I had created, the concepts I utilized, and some new things I learned along the way. While the program was pretty simple in terms of functionality, it was monumental in both exercising the concepts I had recently learned as well as instilling some degree of confidence in my programming abilities. This had been a test run of sorts which helped confirm that, while definitely not easy, programming was something that I enjoyed doing and I could see myself pursuing in college.

Coming into my introductory computer science classes with a bit of Python knowledge under my belt, I found that some of the concepts introduced in lectures were relatively easy to pick up. This advantage, however, was extremely short-lived. Within the first few weeks of class I was inundated with various readings on good code design, data structures, and language-specific features and limitations. On top of all these readings came arduous assignments that tested our knowledge daily and allowed for very few mistakes. A notable assignment that was particularly difficult was when we had to design the game Tetris from scratch using Racket. This was the first large project that we had been assigned, and unfortunately in this situation my partner was not a very helpful resource. I spent hours upon hours writing code, debugging it, reading up on documentation, and restructuring my code to follow good design. After many long days, the time finally came to turn in the assignment and I breathed a sigh of relief as I pressed the "Submit" button. Thus marked the first proper challenge I experienced in CS (with many, many more to come), and although it was a difficult process it was also an extremely rewarding one. When it came to writing our reflections afterwards and describing how the project worked, I was easily able to reference the various design patterns I followed and clearly form a write up that was easy for professors and TAs to read. Because the people I was designing the write up for were very proficient in the field, I was able to write with a ton of detail and assume that the reader knew many of the more niche terms I was using. I went through this experience several more times as I proceeded through the fundamental computer science classes and beyond, and each time I went through the process I came out with a deeper understanding of the principles I was learning and a much stronger vocabulary with which I could describe those concepts. It wasn't until my first co-op at which I learned about the importance of writing and communicating computer science concepts at varying levels of detail.

After starting my first co-op as a software developer, one of the biggest lessons I learned is the importance of being able to write and communicate technical ideas to non-technical people. While working in a software team that was part of a broader financial team, I would oftentimes interact and exchange ideas with people who had varying levels of familiarity with programming. For example, when starting out with a project I would engage in discussions with the Product Manager, who essentially served as the designer and architect for the project. While they were intimately familiar with what had to be done and how parts of the final product had to be designed, they were oftentimes very unfamiliar with the nuances of how the task would be

executed. On the other hand, if I were to hash things out with a fellow software developer, I could be a lot less cognizant of the terms I was using and go into much deeper detail about what I was working on. Going through this experience significantly improved my abilities as a writer in this discipline, as I was able to more effectively convey ideas in a way that didn't bog down my explanation with technical terms, yet was detailed enough to satisfy people who were intimately familiar with the subject matter. Throughout my co-op, I also continued with my process of reading significant amounts of documentation, blogs, and help forums in order to get a better understanding of what technologies I was working with and its features and limitations.

My second co-op followed a similar cycle of reading and writing as the first one, and by the end of it I had not only become a much better reader and writer in my discipline, but I also had a slightly clearer picture of what exactly I wanted to do and accomplish. My first co-op focused mostly on DevOps tasks (think setting up cloud infrastructure, automating a pipeline for developer projects, etc.) while my second co-op was mostly focused on front-end (building out the UI, making visual changes to the application) and a bit of back-end development (editing the database, changing and adding logic to the "behind the scenes" code). All in all, these two co-op opportunities gave me experience will almost all facets of software engineering (not including Quality Assurance, which I don't find to be very fulfilling personally), and after testing out all three I can say with some confidence that backend development is what I want to do in the future. While frontend development is a very exciting and popular avenue for many software developers (especially when a company is just starting out or building a brand new feature), the work can get a bit tedious and monotonous when it comes to maintaining and editing an existing UI. There were many tasks I worked on that involved several hours of fiddling with HTML and CSS, two powerful yet sometimes unintuitive languages, and a lot of back and forth with the team's product manager. On the other side of things, DevOps is the complete opposite of frontend design (as everything is behind the scenes), but setting up infrastructure often involves reading a lot of documentation and is not very creatively fulfilling. Backend development, however, strikes a nice balance between allowing some creative input while also being a bit more complex and therefore more fulfilling to complete.

All in all I've come very far from the early days of my interest in this discipline, and my reading and writing skills in relation to computer science have evolved tremendously. On the reading front, I have become much more comfortable with navigating documentation,

understanding blog posts and scholarly articles, and consulting with online forums to address bugs in my code. In terms of documentation, I have become much more comfortable with finding what exactly is relevant to my task at hand while ignoring everything else that is unrelated. For blog posts and scholarly articles, my familiarity with the discipline and some of its nuanced terms has improved significantly, allowing me to minimize my knowledge gaps and actually understand many of the blog posts and articles that I read now. This has also carried over to my experiences with online forums, where I can better assess if a response is relevant to my issue and subsequently learn how to mitigate the issue in the future. In terms of being a writer in my discipline, I have become proficient in effectively documenting things such as ideas, code changes, or even explanations of how my code works. I have also become much more comfortable with documenting these ideas with varying levels of detail depending on the audience I am writing for, which is an essential skill for any writer regardless of their discipline. I'm much more comfortable with many of the topics in my field as well, and in my past co-ops I would oftentimes write up explanations or tutorials of certain tools or concepts used at the company. As I continue to grow and adapt to the ever-changing landscape of my career, I look forward to using my reading and writing skills to help me and those around me become better developers.

[1] Wikipedia Contributors, "Linux," Wikipedia, Sep. 14, 2019. https://en.wikipedia.org/wiki/Linux