My teaching philosophy is intuitive. For as long as I can remember, I have been teaching. In early elementary, when I finished work early, my teachers would assign me to help other students. I was too young to remember first learning the difference between telling someone the answer and teaching it, but I have been teaching ever since. I open with this not as an anecdote of passion or aptitude for teaching, rather because it informs my approach and comprehension of how to teach. It is instinctual, it is empathetic. My goals for students stem from this as well. I fundamentally want others to learn. I derive a genuine joy for learning new things, and as a teacher I am successful if I can impart that passion onto students as well. Success is going beyond simple knowledge or comprehension of material but to develop an intuition for it, capable of using what was learned as the foundation for synthesis of new ideas. Success is motivating people to care about what they are learning in a way that drives them to explore beyond what is taught of their own volition and drives retention of the core concepts long past a single course.

In computer science education, many complex concepts such as operating systems, networking, or artificial intelligence leverage "learning by doing" with supporting projects that help students to develop a deep intuition of what they are creating. In my early career as a teaching aide, however, I observed a troubling trend in these courses. The mechanics of software engineering – how to author, edit, track, build, test, and debug code – were standing in the way of students developing projects for some students. As a result, learning comprehension was impeded and achievement (grade) on projects was in many cases mismatched to student comprehension of the material. Students missing these mechanics were trapped in some bizarre space between Imposter Syndrome and Norman's taught helplessness – they grasped the concepts with the same degree of comprehension but could not express them in software as effectively as their peers. The root cause of this knowledge gap was surprisingly simple; these mechanics were simply not part of the curriculum.

To address this gap, I designed and developed a new course, Computing for Computer Scientists (C4CS). The design of C4CS achieves the direct goal by structuring itself around the tools used by computer scientists to make software development easier. The course goes further, however, by decomposing these tools, exploring how they are built from first principles in computer science, simultaneously developing a foundational understanding for their operation and demonstrating the capabilities and utilities for core concepts being learned across the curriculum.

"Lectures" in C4CS are unique. Once a new concept or tool is introduced abstractly, we explore its usage live and interactively. This involves real-time examples, questions, and critically real-time mistakes. One of the most consistent and positive pieces of feedback from the course is the learning that comes from watching a mistake, how it is debugged, and how it is corrected and working through that thought process as a group.

A final, unfortunate element of computer science is the comparably extreme ratio of underrepresented minorities. Part of this stems from a perception of computer science as a field with a high barrier to entry. The course explicitly calls out the false barrier to entry created by non-intuitive tools that are rarely explained and simply used (bash, git, make, etc). We break into discussions on dangers of certain programmer archetypes like the "rockstar developer" or the very personal nature of developer environments to help understand and enforce that developing an identity as a computer scientist will take time and that encountering a strong, and different, identity does not necessarily make you any less capable than them. C4CS has exhibited early success in supporting underrepresented minorities by targeting some of the issues and challenges facing these populations without overtly targeting those demographics, but rather using these discussions to shift perception among all computer scientists.