Teaching Philosophy

Andrew Rosen

I consider teaching Computer Science to be the most enjoyable and rewarding thing I do. I have been teaching from the moment I entered my PhD studies and the feedback from my students evaluations over the years record a story of student success. This is because I use a variety of methods to engage with my students and address the challenges of computing.

Computer Science is a large field with many applications and that makes it particularly exciting both to learn and teach. I often tell students that I chose Computer Science for my undergraduate degree because I had no idea what I wanted to do, but I knew that Computer Science and programming would give me the power to choose any field I wanted. This is why I love teaching Computer Science: I can engage with a wide variety of students, each with unique backgrounds and widely varying aspirations.

With this shear breadth comes a challenge - Computer Science can be very difficult to teach and for students to learn. I posit three reasons for this.

The first factor is that Computer Science is a completely new and foreign field for the vast majority of students. Where many degrees teach a system of theory or a practical craft, Computer Science requires both. This means we demand a great deal more from our students, essentially a complete immersion in the field.

The second is stringent requirements of assignments. There is no other subject where the expectation is that homework is completely correct every single time. A single typo can be the difference between resounding success and utter failure. This can be quite intimidating to the novice programmer. Novice students see this and are afraid to try anything; they view making innocuous changes to a program the same way they would view trying to defuse a bomb. Even worse, some students ignore all errors their computers throw at them and trudge blindly on, hoping some simple change at the end will fix all their bug-ridden code.

The third is that you are your own worst enemy when you program. A single misplaced brace or missing equals sign can cause hours of mind-numbingly agonizing debugging. Even the the most experienced programmer is not immune to this; everyone makes mistakes, especially when we read what we want to read, as opposed to what's there. Take the previous sentence for example. Now multiply that out over dozens of lines of code, each capable of crashing your program, on a dull assignment, and we can get a glimpse as to why Computer Science is difficult.

I teach using a wide variety of approaches to address these challenges. Often times, while the curriculum is solid, the textbook and slides are no substitute to actually learning by programming, learning by making mistakes, learning by experimenting and having fun. Reading slides is not sufficient to teach students, especially in lower level courses. This means students need to be given much time as possible to code with their instructor available to guide them. I've done this by taking one of two approaches: live-coding and flipping the classroom.

Live-coding, where you create and run the examples in front of the students, is what I have used in most lectures in the past and is suited for large groups of students. By programming in class and prompting students for input, I am able to use the lecture time to get students coding, albeit vicariously. It makes programming feel more real and less theoretical.

Live-coding means making mistakes, and these mistakes, intentional or unintentional, help students become accustomed to the debugging process they will have to do themselves. I believe that seeing an ostensible expert make mistakes and correct them makes coding much less intimidating. As the course progresses, students become more comfortable with correcting any mistakes they see.

Recently, I've taken to flipping my classrooms. I record my lectures in advance, breaking it up into segments between 5 to 15 minutes depending on the topic (about the limit of being able to maintain attention). The students watch the videos of me live-coding and explaining the material in advance and spend the lecture time working on the homework with their peers and consulting with me for help. This allows me to give students the individual attention they need, when they need it. The student response to my approach has been overwhelmingly positive.

I write all the code the students learn from and make it accessible on github, which also indirectly exposes them to version control, an important concept the doesn't really fit in any specific place in the curriculum. I provide practice exams for my students, so they know what kind of questions I expect. I provide interesting assignments, such as solving mazes, sending secret messages, or making games, that engage students on the content they are learning.

I test how well my students understand content and how they can apply it, not how well they can memorize information. My exams are open notes to reflect the reality that professorial programming is almost never done without reference material readily available. This also allows me to focus on testing my students' abilities to apply information, not just recall it.

I also make myself readily available to any student who needs help. In many cases, when students are unable to make it to office hours or be on campus, I have met with students virtually via a video chat in order to help them understand the material. I have seen many instances where the subject matter "clicks" as a result of meeting one-on-one.

How can I be sure that my teaching methods are effective? The students themselves tell me how much they learn. They tell me how they took what they learned and used it on their other classes or on their own pet projects. They see the my curriculum and lessons aren't a bunch of check boxes that they have to fill so they can go and get their degree, but teach them the skills they need to solve real problems outside of a classroom.