Programming Professional Development at the Knowles Teacher Initiative: How We Made a Code Camp

In my previous post, I talked about how teachers could use programming not just as the target of instruction, but also as a tool to teach the content of their courses. At the Knowles Teacher Initiative, we aim to support teachers learning programming for many purposes: programming as the content of a course (such as teaching computer science), programming to support the teacher's work and/or personal interests (such as writing gradebook automation), and using programming to teach or assess course content (such as using Python to build simulations that students can run or modify in the service of learning physics content). Different teachers have different needs from professional development on programming, but we aim to support many of these modes of interaction.

When I was first approached by Knowles Fellows about providing a "learn how to program" professional development, I reviewed what was available both online and in-person, and it seemed that there were no teacher-centered courses that focused on both skill-building in a language and thinking about different ways to use programming in the classroom. That is why I undertook to design and teach Knowles Code Camp, a professional development for teachers learning to program.

In order to maximize attendance, Code Camp 2016 was presented as a three-day summer workshop: this allowed us to reach the largest number of Knowles Fellows, as they were already going to be traveling to the meeting site for our annual summer meeting. Each day of Code Camp included about eight hours of instruction, split between learning about programming techniques, learning about computational thinking, and working on individual projects. Sixteen Knowles Fellows participated: 14 were regular participants and two (Nate Pinsky and Allison Honold) were recruited as co-facilitators. The co-facilitators had prior programming experience, and helped plan and implement Code Camp instruction. As a computer science teacher, Nate was able to provide support in planning and implementing the programming instruction (and indeed led much of that instruction). Allison was a participant in a prior Code Camp and is a physics teacher: she provided a much-needed content teacher perspective in the design of Code Camp as well about the strengths and weaknesses of the earlier Code Camp's design.

There were a number of constraints on the design of Code Camp. We had a limited amount of time, a broad range of prior programming experience, and teachers who wanted to learn programming for all the different reasons I enumerated above. It would be hard to cover the basics of programming, computational thinking, and incorporating coding into curriculum in a month, and impossible in three days. The overall outcome for Code Camp was therefore getting the participants exposed to the principles of programming and computational thinking, and helping them develop the sense of self-efficacy needed to continue learning programming on their own time.

There are a number of different programming languages we could have used in Code Camp (such as R, Python, Java, and Scratch), but we chose to teach our participants to program using JavaScript. Although probably most known as the language that makes web pages interactive, JavaScript can also be used to develop mobile phone apps, desktop apps, command-line or server code, and even is the macro scripting language for Google Docs, Sheets and Sites. It is a high-leverage programming language: other tools may be better suited for specific tasks, but JavaScript is a general-purpose tool that can be used to do just about anything.

As a content area, programming is particularly amenable to project- and problem-based approaches. The instructor can set up a simple challenge ('draw a square on the screen wherever the user clicks') and provide a number of extensions ('now draw a regular n-sided polygon for any integer from 1 to 20, based on an input') that can challenge teachers at many different levels. Programming challenges are also often self-assessing: when you look at the output of a potential solution to a challenge, a novice can easily decide if it works ("no, you aren't drawing hexagons there", or "yes, it's a square, but it is in the wrong place"). There are many correct solutions to most programming problems, but incorrect solutions are pretty easy to spot.

A well-crafted programming challenge can also branch out in a number of directions: starting from that simple square challenge, for example, we could move in the regular n-sided polygon direction, or could work more with screen graphics to animate changing numbers of sides, or could do something like maintain only the last three figures drawn on the screen, erasing the rest. Different challenges are not necessarily harder or easier, but represent different

modes of thinking and different techniques needed for solving the problem.

Asking teachers to work in small groups on these challenges also mirrors common industry practices that software engineers use (pair programming, for example). When small groups work collaboratively on a single computer program, it is easier for group members to take turns asking higher-level questions, such as "what are we trying to accomplish here, and will this code do that?" or simply, "why did you make that decision?" These questioning practices are central to effective pair programming, but more generally are quite useful in making small group, project-based learning effective, especially among groups of different prior ability levels in the content area (White & Frederiksen, 1998, p. 46).

In this situation, we had existing assets to build on: as Knowles Fellows, the participating teachers were already quite used to working effectively together in small groups, owing to the professional development they participate in as part of being a Knowles Fellow.

Code Camp Days were split 2-1-1 between programming skills, computational thinking, and free-choice work. Around half of the time was spent working on programming challenges (such as the regular polygon one mentioned above) that were chosen to highlight different aspects of programming: logic, loops, input-output, functional composition, and graphics. While varying the content from day to day, we also varied the context: on one day teachers wrote scripts for spreadsheets in Google Apps Scripts, and on another they created interactive web applications on codepen.io, a great site for hosting work-in-progress webapps. In this way, teachers were exposed to both a number of programming skills and a lot of different places where programs could be run to accomplish something.

We also spent about a quarter of each day on computational thinking work. The material at CS Unplugged is a great resource to start exploring computational thinking. We worked through some of activities from CS Unplugged and elsewhere, and we talked about how computational thinking could be incorporated into content classes.

Finally, we spent the last quarter of each day in a free-choice session. Some (most) participants returned to the challenges from earlier in the day, digging more deeply and turning their work into something that is useful to them either in their classroom or in their everyday life (or just following through on an

intellectual challenge they felt). Others worked on larger-scale programming projects (a popular one is a tool that will generate randomly-assigned small groups given a class list and constraints about which students cannot be paired with each other), and some worked on expanding their ideas about adding computational thinking to their teaching.

Overall, these general guidelines worked quite well to create a coherent and cohesive Code Camp. We had small groups of teachers working on projects (programming and computational thinking) at varying levels of difficulty.

The outcomes from Code Camp, especially on how well our teachers took up notions of computational thinking and using programming to teach content, will be discussed in a forthcoming post.

References

White, B., & Frederiksen, J. (1998). “Inquiry, modeling, and metacognition: Making science accessible to all students.” *Cognition and Instruction*, *16*(1), 3-118.