Mission Bit Teaching Tips

Here are some guiding principles for how we like to think about teaching and learning. They're not meant to be definitive or comprehensive by any means, just useful little nuggets to orient you. Ultimately, every student has different needs, and you should take those needs into account.

THE GOAL IS NOT TO CREATE PROFESSIONAL PROGRAMMERS.

> Teaching with that assumption can alienate students who don't think of themselves as becoming developers someday (which is a lot of people!). As much as we can, we don't want to limit the type of people that code. Instead, we want to create a welcoming environment where students can start to get a feel for programming and choose to continue learning it if they like.
>
> If a student's goal *is* to be a developer someday, great! Encourage them!
>
> A useful way of thinking about it is to empower students, help them feel like *they* are in control of the technology that increasingly surrounds them, that they can use it for their own needs. If they like programming, and have the resources, motivation, and encouragement to do so, then maybe they'll continue learning it. If not, then hopefully they had fun, feel more comfortable with technology, and have something cool to show for their experience.

The most effective learning experience you can have is working hard on something you care about.

> I think we've probably all had the experience of neglecting sleep, friends, and responsibilities in order to finish a program. Not that we *want* this to happen, just that intrinsic motivation is the fundamental force behind learning. Foster it in your students!
>
> Try to tap into students' interests (but don't be lame about it). Programming can be used in a huge variety of contexts, so help students find interesting, approachable projects. If someone is personally motivated to do something they deem worthy, then learning will be so much more fun and effective because *they* chose to pursue it.
>
> Most students are accustomed to the typical, passive classroom experience of being told what to do. See if you can nudge them out of that mindset. As much as possible, we want the student to be in control of their learning experiences, with us as resources rather than authorities.

A club-like atmosphere feels a lot better than a lecture/classroom environment.

We've all been stuck in boring lectures where we have no idea what's going on. From what I've observed, students (whether children or adults) don't learn programming very well from lectures. A much better approach seems to be working on projects together with peers, mentors, and teachers. It's more fun and the learning is better. Facilitators should try to create this environment and limit lecture time.

Try to let people figure things out for themselves. Don't over-teach.

Pretty often you'll see someone doing something that isn't going to work. Resist the impulse to step in and correct them unless they explicitly ask for help. Mistakes are great first-hand experiences for guiding a learner to a better understanding of how programming works, and over-teaching can often be less helpful than no teaching. Note that this is very different from typical school experiences where mistakes are shameful and ridiculed.

If you feel it would be better to step in, be polite. e.g. "Can I show you something?"

To rescue someone, you have to get to where they are.

Students don't have the mental models for programming that you've developed over the years. As much as possible, try to understand their thinking process to help guide them to a better understanding.

Remember, people usually make pretty reasonable guesses about how things work, even if they're wrong. Don't dismiss what they've learned, but recognize and mold it.

Peers can be really good teachers!

They're going through the same learning process and can often understand how someone is thinking well enough to get them over hurdles.

Students that "get it" quicker may be able to help others.

Letting the students work in groups seems helpful. They can ask and answer their own questions and look at each other's screens when they're stuck.

Be careful with jargon.

We take a lot of our jargon for granted. *Function*, *variable*, *library*, *for loop*, *class*, *attribute*, *method*, *tag*, *bracket, curly brace*. We use these and many other terms in strange ways for beginners. Especially at the start of a semester, it might be worth trying to speak in plain language about these concepts. After gently introducing terms and using them for a while, students will start making the connection between concepts and

vocabulary.

Show *and* tell. Depict *and* describe.

Try to not only tell students about how a concept works but show them, in excruciating detail if needed. The less a learner has to imagine, the better. People can understand what they can see!

For example, when creating an animation it is not at all obvious that the smooth motion of objects on the screen is the result of successively rendered static frames. Slowing the frame rate down to a crawl so students can see each frame rendered and slowly speeding it up illustrates the idea better than words ever could. Even better would be to let the students control the frame rate themselves.

Try not to be too prescriptive.

While you may have learned the “right” way to do something from hard-won experience, the students haven’t yet! They need to see for themselves (with your guidance) why certain methods are better than others. You might suggest alternative ways of doing something, but don’t get so stuck on the one “right” way of doing something, especially at the beginning when a learner has a lot to keep in their head.

Not: “These lines of codes should be made into a function.” Better: “Making these lines of code into a function might make it easier for you to understand your program.”

Be flexible.

If the students are taking you in a certain direction, don’t be afraid to go there. Don’t stick blindly to curricula if your students aren’t into it or have something better to do.

Lower the “time to first awesome” as much as possible.

The faster you can get something cool happening, the better. No one likes jumping through a million hoops they don’t understand in order to get something done, especially when they’re just starting out.

Harvard’s popular CS50 “Intro to Computer Science” class found a much lower dropout rate when introducing programming using Scratch, a programming environment for children that takes almost no work to make something interesting happening.

Have infinite patience.

You are a Zen master. Learning can be a slooow process and everyone has a different

pace.

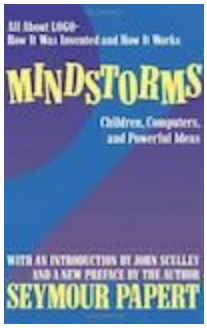
Students will have many different backgrounds and learning styles. Respect them all!

If you or a student is bored or discouraged or frustrated, recognize that and try to change the situation. You're both human and it happens.

Be a real person.

When teaching or mentoring, it's easy to disappear into that role—a Teacher or a Mentor. You only expose small parts of yourself to the students in order to fulfill that role. But more likely than not, programming is a significant part of your life, filled with success and failures, laughter and stress, a continued drive to create and improve, and more. This is the real act of programming and you should strive to share that experience genuinely with your students.

Resources if you want to learn more:



Mindstorms: Children, Computers, and Powerful Ideas by Seymour Papert (1980)
http://en.wikipedia.org/wiki/Mindstorms:_Children,_Computers,_and_Powerful_Ideas

This is the Bible of learning to program and programming to learn. It describes how people learn and how programming can facilitate the process.

"For me, getting to know a domain of knowledge is much like coming into a new community of people. Sometimes one is initially overwhelmed by a bewildering array of undifferentiated faces. Only gradually do the individuals begin to stand out. [...] Similarly, when one enters a new domain of knowledge, one initially encounters a crowd of new ideas. Good learners are able to pick out those who are powerful and congenial. Others who are less skillful need help from teachers and friends. But we must not forget that while good teachers play the role of mutual friends who can provide introductions, the

actual job of getting to know an idea or a person cannot be done by a third party."

How Children Fail by John Holt (1964, 1982)
http://en.wikipedia.org/wiki/How_Children_Fail

An invaluable book for understanding how children think in schools and why they think that way. Crucial for empathizing with your students.

"Kids have trouble with arithmetic, not only because they have to memorize a host of facts that seem to have no pattern, meaning, or interest, but also because they are given a host of rules for manipulating these facts, which they have to take on faith."

"A child who has really learned something can use it, and does use it. It is connected with reality in his mind, there he can make other connections between it and reality when the chance comes. A piece of unreal learning has no hooks on it; it can't be attached to anything, it is of no use to the learner."

Learnable Programming by Bret Victor (2012)
http://worrydream.com/LearnableProgramming/

How to design a programming system for understanding programs. If nothing else, this essay does a great job showing some of the biggest stumbling blocks in programming today.

"Think about this. We expect programmers to write code that manipulates variables, without ever seeing the values of those variables. We expect readers to understand code that manipulates variables, without ever seeing the values of the variables. The entire purpose of code is to manipulate data, and we never see the data. We write with blindfolds, and we read by playing pretend with data-phantoms in our imaginations."

Designing for Tinkerability (Scratch and MakeyMakey) by Eric Rosenbaum and Mitch Resnick (2012)
http://web.media.mit.edu/~mres/papers/designing-for-tinkerability.pdf

Recognizing the value of playful exploration and tinkering in learning and how learning environments can be designed to encourage that style. Shows how tinkering is an often-used, but rarely encouraged technique when learning.

"There are many different approaches to making things, and some lead to richer learning experiences than others. In this chapter, we focus on a particular approach to making that we describe as tinkering*. The tinkering approach is characterized by a playful, experimental, iterative style of engagement, in which makers are continually reassessing their goals, exploring new paths, and imagining new possibilities. Tinkering is undervalued (and even discouraged) in many educational settings today, but it is well aligned with the goals and spirit of the progressive-constructionist tradition—and, in our view, it is exactly what is needed to help young people prepare for life in today's society."*