Table of Contents

Core Material

Appendices

Introduction

> **The Short Version**
>
> *A Brief Introduction to Teaching* shows readers how to build and deliver high-quality learning experiences to adults. It is based on Software Carpentry's instructor training course, and all material is available under the Creative Commons - Attribution license.

Thousands of grassroots "learn to code" groups have sprung up in the past few years. They exist so that people don't have to figure out how to program on their own, but ironically, that's exactly what most of their founders are doing when it comes to teaching.

As many have discovered, there's more to teaching than talking. Good teachers break subjects up into digestible pieces, design lessons with verifiable goals in mind, check their students' progress at short intervals, and encourage collaboration and improvisation. Like good programming practices, these don't have to be reinvented by every teacher: they can and should be taught and learned. And while these practices won't magically make someone a *great* teacher, they do make most people *better* teachers.

This book is a brief introduction to modern evidence-based teaching practices and how to use them to teach programming to free-range learners. It covers:

- how people's thinking changes as they go from being novices to competent practitioners and then to being experts;

- how to tell if your learners are keeping up with you, and what to do or say when they're not;
- how to design and improve lessons efficiently and collaboratively;
- how and why live coding (i.e., writing programs step by step in front of learners) is a better way to teach programming than lectures or self-directed practice; and
- how insights and techniques borrowed from the performing arts can make you a better teacher.

How to Use This Material

This material has been taught as a multi-week online class, as a two-day in-person class, and as a two-day class in which the learners are in co-located groups and the instructor participates remotely.

> **Terminology**
>
> When we talk about workshops, we will try to be clear about whether we're discussing ones whose subject is programming, and which are aimed at general learners, and those whose subject is now to teach (and are using this material). We usually refer to the latter as "instructor training workshops".

In-Person

In our experience, this is the most effective way to deliver an instructor training workshop.

- Participants are physically together for one or two days. When they need to work in small groups (e.g., for practice teaching), some or all of them go to nearby breakout spaces. Participants bring their own tablets or laptops to view and edit online material during the class, and use pen and paper and/or whiteboards for some exercises.
- Participants use Etherpad or Google Doc for in-person training, both for shared note-taking and for posting exercise solutions and feedback on recorded lessons. Questions and discussion are done aloud.
- Several times during the training, participants are put in groups of three to teach for 2-3 minutes. The mechanics are described later, and while participants are initially intimidated at first, they routinely rank it as the most useful part of the class.

Two-Day Online With Groups

In this format, learners are in groups of 4-12, but those groups are geographically distributed.

- Each class uses an Etherpad or Google Doc for shared note-taking, and more importantly for asking and answering questions: having several dozen people try to talk on a call works poorly, so in most sessions, the instructor does the talking and learners respond through the note-taking tool's chat.

- Each group of learners is together in a room using one camera and microphone, rather than each being on the call separately. We have found that having good audio matters more than having good video, and that the better the audio, the more learners can communicate with the instructor and other rooms by voice rather than by using the Etherpad chat.
- We do the video lecture exercise as in the two-day in-person training.

Multi-Week Online

This was the first format we used, and we no longer recommend it.

- We met every week or every second week for an hour via web conferencing. Each meeting was held twice (or even three times) to accommodate learners' time zones and because video conferencing systems can't handle 60+ people at once.
- We used web conferencing and shared note-taking as described above for online group classes.
- Learners posted homework online between classes, and commented on each other's work. (In practice, comments were relatively rare: people seemed to prefer to discuss material in the web chats.)
- We used a WordPress blog for the first ten rounds of training, then a GitHub-backed blog, and finally Piazza. WordPress worked best: setting up accounts was tedious, but everything after that ran smoothly. Using a GitHub blog worked so poorly that we didn't try it again: a third of the participants found it extremely frustrating, and post-publication commentary was

> awkward. Piazza was better than GitHub, but still not as easy for participants to pick up as WordPress. In particular, it was hard to find things once there were more than a dozen homework categories.

History

I started teaching people how to program in the late 1980s. At first, I went too fast, used too much jargon, and had little idea of how much my learners actually understood. I got better over time–at least, I thought I did–but still had no idea how effective I was compared to other teachers.

In 2010, I rebooted a project called Software Carpentry. Its aim was (and is) to teach basic computing skills to researchers from a wide range of disciplines so that they could get more done in less time and with less pain. In the two years that followed, I discovered resources like Mark Guzdial's blog [Guzdial2017] and the book *How Learning Works* [Ambrose2010]. These led me to other sources like [Lemov2014], [Huston2009], and [Green2014] that showed me what I could do to make my teaching better, and why I should believe it would work.

We started using these ideas in Software Carpentry in 2012, and the results were everything we'd hoped for. Designing lessons with different stage of cognitive development in mind, using live coding instead of slides, and adopting lightweight real-time assessment mechanisms[1] all made our workshops more effective.

What made the biggest difference, though, was offering a short course to introduce people to these techniques and the ideas behind them. This course was originally delivered online over multiple weeks, but by 2014 we were teaching it in two intensive days (just like our regular software skills workshops). Over the last three years, I have run more than forty versions of that course for people who want to teach programming to children, recent immigrants, women re-entering the workforce, and a wide variety of other groups. Those experiences are the basis of this book.

Acknowledgments

I am grateful to everyone who helped create Software Carpentry's instructor training course, including Erin Becker, Karen Cranston, Neal Davis, Rayna Harris, Kate Hertweck, Christina Koch, Sue McClatchy, Lex Nederbragt, Elizabeth Patitsas, Aleksandra Pawlik, Ariel Rokem, Tracy Teal, Fiona Tweedie, Allegra Via, Anelda van der Walt, Belinda Weaver, Jason Williams, and the hundreds of people who have gone through it over the years. I am also grateful to Neil Brown, Warren Code, and Mark Degani for their feedback on this version. If you find it useful, I hope you will pass on whatever you learn to someone else.

Who You Are

Learner Profiles will explain how to define who a class is for. Here, we present profiles of two typical participants in a workshop based on this book.

Samira

1. An undergraduate student in mechanical engineering who first encountered the subject in an after-school club for girls and would now like to pass on her love for it.

2. Has done one programming class and one robotics class, and was a lab assistant for a couple of weekend introductions to engineering for high school students at her university.

3. Would like to learn techniques for explaining ideas and handling unexpected questions or situations.

4. This workshop will introduce her to some basic classroom practices and give her a chance to try them out in front of a supportive audience.

5. Feels insecure about standing up and teaching a subject that she isn't an expert in ("I'm not a professor!").

Moshe

1. A professional programmer with two young children. Their school doesn't offer a programming class, so he has volunteered to put one together. After reading a dozen different "programming for kids" books, he feels more confused than ever.

2. Has been programming in Visual Basic and C# for almost twenty years, during which time he has frequently given presentations to colleagues and management.

3. Wants to learn how to build lessons that both he and other people can use and maintain.

4. This class will show him how to design and deliver lessons tailored for his students, how to tell how well those lessons are working, and how to keep those lessons up to date.

5. Moshe is partially deaf, and most of his students have hearing disabilities as well.

Teaching Practices

We suggest that instructor training workshops use these three teaching practices right from the start:

- Have a code of conduct.
- Take notes together.
- Pre-assess learners' motivation and prior knowledge.

Dedication

This book is dedicated to my mother, Doris Wilson, who taught a lot of children how to read and write, and to believe in themselves.

Challenges

Favorite Class

In the online notes, write down your name, the best class you ever took, and what made it so great.

10 minutes

Footnotes

> 1. Most involving sticky notes. ↩

Helping Novices Build Mental Models

> **Objectives**
>
> - Learners can explain the cognitive differences between novices and competent practitioners in terms of mental models, and explain the implications of these differences for teaching.
> - Learners can define and differentiate formative and summative assessment.
> - Learners can construct multiple-choice questions with plausible distractors that have diagnostic power.

The first task in teaching is to figure out who your learners are and how best to help them. Our approach is based on the [Dreyfus model of skill acquisition][wikipedia-dreyfus], and more specifically on the work of researchers like Patricia Benner, who studied how nurses progress from being novices to being experts [Benner2000]. Benner identified five stages of cognitive development that most people go through in a fairly consistent way. (We say "most" and "fairly" because human beings are variable, and there will always be outliers. However, that shouldn't prevent us from making strong statements about what's true for the majority.)

For our purposes, we simplify the five stages to three:

1. A *novice* is someone who doesn't know what they don't know, i.e., they don't yet know what the key ideas in the domain are or how they relate. They reason by analogy and guesswork, borrowing bits and pieces of their mental models of other domains which seem superficially similar[1].

2. A *competent practitioner* is someone who has a mental model that's good enough for everyday purposes: they can do normal tasks with normal effort under normal circumstances. This model does not have to be completely accurate in order to be useful: for example, the average driver's mental model of how a car works probably doesn't include most of the complexities that a mechanical engineer would be concerned with.

3. An *expert* is someone who can easily handle situations that are out of the ordinary, diagnose the causes of problems, and so on. We will discuss expertise in more detail in Memory.

One example of a mental model is the ball-and-spring model of molecules that most of us encountered in high school chemistry. Atoms aren't actually balls, and their bonds aren't actually springs, but the model does a good job of helping people reason about chemical compounds and their reactions. Another model of an atom has a small central ball (the nucleus) surrounded by orbiting electrons. Again, this model is wrong, but useful for many purposes.

Novices, competent practitioners, and experts need to be taught differently. In particular, presenting novices with a pile of facts early on is counter-productive, because they don't yet have a model to fit those facts into. In fact, presenting too many facts too soon can actually reinforce the incorrect mental model they've cobbled together. As Derek Muller wrote about this in the context of video instruction for science students:

> Students have existing ideas about scientific phenomena before viewing a video. If the video presents scientific concepts in a clear, well illustrated way, students believe they are learning but they do not engage with the media on a deep enough level to realize that what was is presented differs from their prior knowledge.
>
> There is hope, however. Presenting students' common misconceptions in a video alongside the scientific concepts has been shown to increase learning by increasing the amount of mental effort students expend while watching it.

The goal with novices is therefore *to help them construct a working mental model* so that they have somewhere to put facts. As an example of what this means in practice, Software Carpentry's lesson on the Unix shell introduces fifteen commands in three hours. Twelve minutes per command may seem glacially slow, but the lesson's real purpose isn't to teach those fifteen commands: it's to teach learners about paths, history, tab completion, wildcards, pipes and filters, command-line arguments, redirection, and all the other big ideas that the shell depends on. Once they understand those concepts, people can quickly learn a repertoire of commands. What's more, later lessons on how to build functions in a programming language can refer back to pipes and filters, which helps solidify both ideas.

> **Different Kinds of Lessons**

> The cognitive differences between novices and competent practitioners underpin the differences between two kinds of teaching materials. A tutorial's purpose is to help newcomers to a field build a mental model; a manual's role, on the other hand, is to help competent practitioners fill in the gaps in their knowledge. Tutorials frustrate competent practitioners because they move too slowly and say things that are obvious (though of course they are anything but to newcomers). Equally, manuals frustrate novices because they use jargon and *don't* explain things. One of the reasons Unix and C became popular is that Kernighan et al's trilogy [Kernighan1982], [Kernighan1984], [Kernighan1988] somehow managed to be good tutorials *and* good manuals at the same time. Ray and Ray's book on Unix [Ray2014] and Fehily's introduction to SQL [Fehily2008] are among the very few other books in computing that have accomplished this.

One of the challenges in building a mental model is to clear away things that *don't* belong. As Mark Twain said, "It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so."

Broadly speaking, learners' misconceptions fall into three categories:

1. Simple *factual errors*, such as believing that Vancouver is the capital of British Columbia[2]. These are simple to correct, but getting the facts right is not enough on its own.

2. *Broken models*, such as believing that motion and acceleration must be in the same direction. We can address these by having them reason through examples to see contradictions.

3. *Fundamental beliefs*, such as "the world is only a few thousand years old" or "some kinds of people are just naturally better at programming than others" [Patitsas2016]. These are often deeply connected to the learner's social identity, and so are resistant to evidence and cannot be reasoned away in class.

Formative Assessment

Teaching is most effective when instructors have a way to identify and clear up learners' misconceptions *while they are teaching*. The technical term for this is *formative assessment*, which is assessment that takes place during the lesson in order to form or shape it. Learners don't pass or fail formative assessments; instead, its main purpose is to tell both the instructor and the learner how the learner is doing, and what to focus on next. For example, a music teacher might ask a student to play a scale very slowly in order to see whether she is breathing correctly, and if she is not, what she should change.

The counterpoint to formative assessment is *summative assessment*, which is used at the end of the lesson to tell whether the desired learning took place and whether the learner is ready to move on[3]. Learners either pass or fail a summative assessment. One example is a driving exam, which reassures the rest of society that someone can safely be allowed on the road.

> **Connecting Formative and Summative Assessment**
>
> One rule to use when designing lessons is that formative assessments should prepare people for summative assessments: no one should ever encounter a question on an exam for which the teaching did not prepare them. This doesn't mean that novel problems should not appear, but that if they do, learners should have have practice with and feedback on tackling novel problems beforehand.

In order to be useful during teaching, a formative assessment has to be quick to administer and give an unambiguous result. The most widely used kind of formative assessment is probably the multiple choice question (MCQ). When designed well, these can do much more than just tell whether someone knows something or not. For example, suppose we are teaching children multi-digit addition. A well-designed MCQ would be:

> Q: what is 27 + 15 ?
>
> 1. 42
> 2. 32
> 3. 312
> 4. 33

The correct answer is 42, but each of the other answers provides valuable insight:

- If the child answers 32, she is throwing away the carry completely.

- If she answers 312, she knows that she can't just discard the carried 1, but doesn't understand that it's actually a ten and needs to be added into the next column. In other words, she is treating each column of numbers as unconnected to its neighbors.
- If she answers 33 then she knows she has to carry the 1, but is carrying it back into the same column it came from.

Each of these incorrect answers is a *plausible distractor* with *diagnostic power*. "Plausible" means that it looks like it could be right: instructors will often put supposedly-silly answers like "a fish!" on MCQs, but they don't provide any insight and learners actually don't find them funny. "Diagnostic power" means that each of the distractors helps the instructor figure out what to explain to that particular learner next[4].

Instructors should use MCQs or some other kind of formative assessment at least every 10-15 minutes in order to make sure that the class is actually learning. Since the average attention span is usually only this long, formative assessments also help break up instructional time and re-focus attention. Formative assessments can also be used preemptively: if you start a class with an MCQ and everyone can answer it correctly, then you can safely skip the part of the lecture in which you were going to explain something that your learners already know. Doing this also helps show learners that the instructor cares about how much they are learning, and respects their time enough not to waste it.

> **When to Proceed?**

> As the instructor, what should you do if most of the class votes for one of the wrong answers? What if the votes are evenly spread between options? The answer is, "It depends." If the majority of the class votes for a single wrong answer, you should go back and work on correcting that particular misconception. If answers are pretty evenly split between options, learners are probably guessing randomly and it's a good idea to go back to a point where everyone was on the same page.
>
> If most of the class votes for the right answer, but a few vote for wrong ones, you have to decide whether you should spend time getting the minority caught up, or whether it's more important to keep the majority engaged. This is just one example of one of the most important rules of teaching: no matter how hard you work, or what teaching practices you use, you won't always be able to give everyone the help they need.

> **Notes on MCQ Design**
>
> 1. A good MCQ tests for conceptual misunderstanding rather than simple factual knowledge. If you are having a hard time coming up with diagnostic distractors, then either you need to think more about your learners' mental models, or your question simply isn't a good starting point for an MCQ.
>
> 2. When you are trying to come up with distractors, think about questions that learners asked or problems they had the last time you taught this subject. If you haven't taught it before, think about your own misconceptions

> or ask colleagues about their experiences. You can also ask open-ended questions in one class to collect misconceptions about material to be covered in a later class.

> **Concept Inventories**
>
> The Force Concept Inventory is a set of MCQs designed to gauge understanding of basic Newtonian mechanics. By interviewing a large number of respondents, correlating their misconceptions with patterns of right and wrong answers to questions, and then improving the questions, it's possible to construct a very precise diagnostic tool. However, it's very costly to do this, and students' ability to search for answers on the internet is an ever-increasing threat to its validity.

> **We Know Less Than We Think**
>
> [Brown2014] compared teachers' opinions about common programming errors with data from over 100,000 students, and finds only weak consensus amongst teachers and between teachers and data.

Designing an MCQ with plausible distractors is useful even if it is never used in class because it forces the instructor to think about the learners' mental models and how they might be broken–in short, to put themselves into the learners' heads and see the topic from their point of view.

> **Why Not MOOCs**

> *If you use robots to teach, you teach people to be robots.*
> – variously attributed
>
> Massive open online courses (MOOCs) in which students watch videos instead of attending lectures, and then do assignments that are (usually) robo-graded, were a hot topic a few years ago. Now that the hype has worn off, though, it's clear that they aren't as effective as their more enthusiastic proponents claimed they would be [Ubell2017].
>
> Recorded content is ineffective for most novices learners because it cannot intervene to clear up specific learners' misconceptions. Some people happen to already have the right conceptual categories for a subject, or happen to form them correctly early on; these are the ones who stick with most massive online courses, but many discussions of the effectiveness of such courses ignore this survivor bias.

Teaching Practices

If you haven't done so already, you should start using these three teaching practices in your instructor training workshop:

- Use sticky notes as status flags
- Use sticky notes to distribute attention
- Use sticky notes as minute cards

Challenges

Your Mental Models

What is one mental model you use to frame and understand your work? Write a few sentences describing it in the shared notes, and give feedback on other learners' contributions.

5 minutes

Symptoms of Being a Novice

What are the symptoms of being a novice? I.e., what does someone do or say that leads you to classify them as a novice in some domain?

5 minutes

Modelling Novice Mental Models

Create a multiple choice question related to a topic you intend to teach and explain the diagnostic power of each its distractors (i.e., what misconception each distractor is meant to identify).

When you are done, give your MCQ to a partner, and have a look at theirs. Is the question ambiguous? Are the misconceptions plausible? Do the distractors actually test for them? Are any likely misconceptions *not* tested for?

20 minutes

Other Kinds of Formative Assessment

A good formative assessment requires people to think through a problem. For example, consider this question from [Epstein2002]. Imagine that you have placed a cake of ice in a bathtub and then filled the tub to the rim with water. When the ice melts, does the water level go up (so that the tub overflows), go down, or stay the same?

The correct answer is that the level stays the same: the ice displaces its own weight in water, so it exactly fills the "hole" it has made when it melts. Figuring this out why helps people build a model of the relationship between weight, volume, and density.

Describe another kind of formative assessment you have seen or used and explain how it helps both the instructor and the learner figure out where they are and what they need to do next.

20 minutes

Footnotes

> [1]. One sign that someone is a novice is that the things they say aren't even wrong, e.g., they think there's a difference between programs they type in character by character and identical ones that they have copied and pasted. As we will discuss later, it is very important not to shame novices for this. ↩

> [2]. It's Victoria. ↩

> [3]. "When the cook tastes the soup, that's formative. when the guests taste the soup, that’s summative." (Michael Scriven, as quoted by Debra Dirksen.) ↩

4

> [4]. Most jokes are less funny when written down, and become even less funny with each re-reading. Being spontaneously funny while teaching usually works better, but can easily wrong: what's a joke to your circle of friends may turn out to be a serious political issue to your audience. If you do make jokes when teaching, don't make them at the expense of any group, or of anyone except possibly yourself. ↩

Teaching as a Performance Art

> **Objectives**
>
> - Learners can define *jugyokenkyu* and lateral knowledge transfer and explain their relationship to each other.
> - Learners can describe and enact at least three techniques for giving and receiving feedback on teaching performance.
> - Learners can explain at least two ways in which using a rubric makes feedback more effective.

Many people assume that teachers are born, not made. From politicians to researchers and teachers themselves, reformers have designed systems to find and promote those who can teach and eliminate those who can't. But as Elizabeth Green explains in [Green2014], that assumption is wrong, which is why educational reforms based on it have repeatedly failed.

The book is written as a history of the people who have put that puzzle together in the US. Its core begins with a discussion of what James Stigler discovered during a visit to Japan in the early 1990s:

> Some American teachers called their pattern "I, We, You": After checking homework, teachers announced the day's topic, demonstrating a new procedure (I)… Then they led the class in trying out a sample problem together (We)…

> Finally, they let students work through similar problems on their own, usually by silently making their way through a worksheet (You)…
>
> The Japanese teachers, meanwhile, turned "I, We, You" inside out. You might call their version "You, Y'all, We." They began not with an introduction, but a single problem that students spent ten or twenty minutes working through alone (You)… While the students worked, the teacher wove through the students' desks, studying what they came up with and taking notes to remember who had which idea. Sometimes the teacher then deployed the students to discuss the problem in small groups (Y'all). Next, the teacher brought them back to the whole group, asking students to present their different ideas for how to solve the problem on the chalkboard… Finally, the teacher led a discussion, guiding students to a shared conclusion (We).

It's tempting but wrong to think that this particular teaching technique is some kind of secret sauce. The actual key is a practice called *jugyokenkyu*, which means "lesson study":

> *Jugyokenkyu* is a bucket of practices that Japanese teachers use to hone their craft, from observing each other at work to discussing the lesson afterward to studying curriculum materials with colleagues. The practice is so pervasive in Japanese schools that it is…effectively invisible.
>
> In order to graduate, [Japanese] education majors not only had to watch their assigned master teacher work, they had to effectively replace him, installing themselves in his

> classroom first as observers and then, by the third week, as a wobbly…approximation of the teacher himself. It worked like a kind of teaching relay. Each trainee took a subject, planning five days' worth of lessons… [and then] each took a day. To pass the baton, you had to teach a day's lesson in every single subject: the one you planned and the four you did not… and you had to do it right under your master teacher's nose. Afterward, everyone–the teacher, the college students, and sometimes even another outside observer–would sit around a formal table to talk about what they saw.

Putting work under a microscope in order to improve it is commonplace in sports and music. A professional musician, for example, will dissect half a dozen different recordings of "Body and Soul" or "Smells Like Teen Spirit" before performing it. They would also expect to get feedback from fellow musicians during practice and after performances. Many other disciplines work this way too: the Japanese drew inspiration from Deming's ideas on continuous improvement in manufacturing, while the adoption of code review over the last 15 years has done more to improve everyday programming than any number of books or websites.

But this kind of feedback isn't part of teaching culture in the US, the UK, Canada, or Australia. There, what happens in the classroom stays in the classroom: teachers don't watch each other's lessons on a regular basis, so they can't borrow each other's good ideas. The result is that *every teacher has to invent teaching on their own*. They may get lesson plans and assignments from colleagues, the school board, a textbook publisher, or the Internet, but each teacher has to figure out on their own how to combine that with the theory they've learned in education school to deliver an actual lesson in an actual classroom for actual students.

Demonstration lessons, in which one teacher is in front of a room full of students while other teachers observe, seem like a way to solve this. However, Fincher and her colleagues studied how teaching practices are actually transferred using both a detailed case study [Fincher2007] and analysis of change stories [Fincher2012]. The abstract of the latter paper sums up their findings:

> Innovative tools and teaching practices often fail to be adopted by educators in the field, despite evidence of their effectiveness. Naïve models of educational change assume this lack of adoption arises from failure to properly disseminate promising work, but evidence suggests that dissemination via publication is simply not effective… We asked educators to describe changes they had made to their teaching practice… Of the 99 change stories analyzed, only three demonstrate an active search for new practices or materials on the part of teachers, and published materials were consulted in just eight… Most of the changes occurred locally, without input from outside sources, or involved only personal interaction with other educators.

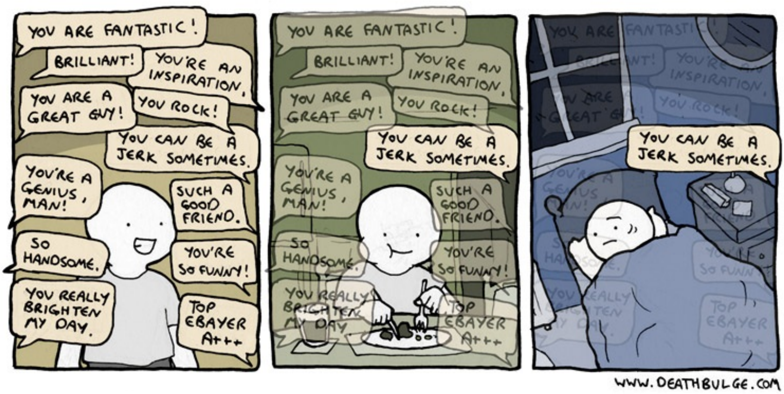
Barker et al found something similar [Barker2015]:

> Adoption is not a "rational action," however, but an iterative series of decisions made in a social context, relying on normative traditions, social cueing, and emotional or

> intuitive processes… Faculty are not likely to use educational research findings as the basis for adoption decisions… Positive student feedback is taken as strong evidence by faculty that they should continue a practice.

This phenomenon is sometimes called *lateral knowledge transfer*: someone sets out to teach X, but while watching them, their audience actually learns Y as well (or instead). For example, an instructor might set out to show people how to do a particular statistical analysis in R, but what her learners might take away is some new keyboard shortcuts in R Studio. Live coding makes this much more likely because it allows learners to see the "how" as well as the "what", and *jugyokenkyu* works because it creates more opportunities for this to happen.

Feedback



As the cartoon above suggests, sometimes it can be hard to receive feedback, especially negative feedback. The process is easier and more productive when the people involved share ground rules and expectations. This is especially important when they have different backgrounds or cultural expectations about what's appropriate to say and what isn't.

You can get better feedback on your work from other people using techniques like these:

1. *Initiate feedback.* It's better to ask for feedback than to receive it unwillingly.

2. *Choose your own questions*, i.e., ask for specific feedback. It's a lot harder for someone to answer, "What do you think?" than to answer either, "What is one thing I could have done as an instructor to make this lesson more effective?" or "If you could pick one thing from the lesson to go over again, what would it be?"

   Directing feedback like this is also more helpful to you. It's always better to try to fix one thing at once than to change everything and hope it's for the better. Directing feedback at something you have chosen to work on helps you stay focused, which in turn increases the odds that you'll see progress.

3. *Use a feedback translator.* Have a fellow instructor (or other trusted person in the room) read over all the feedback and give an executive summary. It can be easier to hear "It sounds like most people are following, so you could speed up" than to read several notes all saying, "this is too slow" or "this is boring".

4. Most importantly, *be kind to yourself*. Many of us are very critical of ourselves, so it's always helpful to jot down what we thought of ourselves *before* getting feedback from others. That allows us to compare what we think of our performance with what others think, which in turn allows us to scale the former more accurately. For example, it's very common for people to think that they're saying "um" and "err" all the time, when their audience doesn't notice it. Getting that feedback once allows instructors to adjust their assessment of themselves the next time they feel that way.

You can give feedback to others more effectively as well:

1. *Balance positive and negative feedback.* One method is a "compliment sandwiches" made up of one positive, one negative, and a second positive observation.

2. *Organize your feedback using a rubric.* Most people are more comfortable giving and receiving feedback when they they feel that understand the social rules governing what they are allowed to say and how they are allowed to say it. A facilitator can then transcribe items into a shared document (or onto a whiteboard) during discussion.

> **Two by Two**
>
> The rubric we find most useful for feedback on teaching is a 2x2 grid whose vertical axis is labelled "positive" and "negative", and whose horizontal axis is labelled "content" (what was said) and "presentation (how it was said). Observers write each of their comments in one of the grid's four squares as they are watching the demonstration.

Whatever methods are used, the most important thing to remember is feedback on teaching is meant to be formative: its goal is to help people figure out what they are doing well and what they still need to work on.

> **Studio Classes**
>
> Architecture schools often include studio classes, in which students solve small design problems and get feedback from their peers right then and there. These classes are most effective when the instructor critiques both the designs and the peer critiques, so that participants are learning not only how to make buildings, but how to give and get feedback [Schon1984]. Master classes in music serve a similar purpose, and a few people have experimented with using live coding at conferences or online in similar ways.

> **Tells**
>
> Everyone has nervous habits. For example, many of us become "Mickey Mouse" versions of ourselves when we're nervous, i.e., we talk more rapidly than usual, in a higher-pitched voice, and wave our arms around more than we usually would.
>
> Gamblers call nervous habits like this "tells". While these are often not as noticeable as you would think, it's good to know whether you pace, fiddle with your hair, look at your shoes, or rattle the change in your pocket when you don't know the answer to a question.

> You can't get rid of tells completely, and trying to do so can make you obsess about them. A better strategy is to try to displace them, e.g., to train yourself to scrunch your toes inside your shoes instead of cracking your knuckles.

If you are interested in knowing more about giving and getting feedback, you may want to read [Gormally2014] and discuss ways you could make peer-to-peer feedback a routine part of your teaching. You may also enjoy [Gawande2011], which looks at the value of having a coach.

How to Practice Teaching

One of the key elements of instructor training is recording trainees and having them, and their peers, critique those recordings. We were introduced to this practice by UBC's Warren Code, who learned it from the Instructional Skills Workshop, and it has evolved to the following:

1. Split into groups of three.
2. Each person rotates through the roles of instructor, audience, and videographer. As the instructor, they have two minutes to explain one key idea from their research (or other work) as if they were talking to a class of interested high school students. The person pretending to be the audience is there to be attentive, while the videographer records the session using a cellphone or similar device.

3. After everyone in the group of three has finished teaching, watch the videos as a group. Everyone gives feedback on all three videos, i.e., people give feedback on themselves as well as on others.

4. After everyone has given feedback on all of the videos, return to the main group and put all of the feedback into the notes. Again, try to divide positive from negative and content from presentation. Try also to identify each person's tells: what do they do that betrays nervousness, and how noticeable is it?

It's important to record all three videos and then watch all three: if the cycle is teach-review-teach-review, the last person to teach runs out of time. Doing all the reviewing after all the teaching also helps put a bit of distance between the teaching and the reviewing, which makes the exercise slightly less excruciating.

In order for this exercise to work well:

- Groups must be physically separated to reduce audio cross-talk between their recordings. In practice, this means 2-3 groups in a normal-sized classroom, with the rest using nearby breakout spaces, coffee lounges, offices, or (on one occasion) a janitor's storage closet.

- Do all three recordings before reviewing any of them, because otherwise the person to go last is short-changed on time.

- People must give feedback on themselves, as well as giving feedback on each other, so that they can calibrate their impressions of their own teaching according to the impressions of other people. (We find that most people are harder on themselves than others are, and it's important for them to realize this.)

- At the end of day 1, ask trainees to review the lesson episode you will use for the live coding demonstration at the start of day 2.
- Try to make at least one mistake during the demonstration of live coding so that trainees can see you talk through diagnosis and recovery, and draw attention afterward to the fact that you did this.

The announcement of this exercise is often greeted with groans and apprehension, since few people enjoy seeing or hearing themselves. However, it is consistently rated as one of the most valuable parts of the class, and also serves as an ice breaker: we want pairs of instructors at actual workshops to give one another feedback, and that's much easier to do once they've had some practice and have a rubric to follow.

> **Setting Up Your Teaching Environment**
>
> If the room setup allows it, try to set up your environment to mimic what you would use in an actual classroom: have a glass of water handy, stand instead of sitting, and so on.

Challenges

Give Feedback

1. Watch this video as a group and then give feedback on it. Organize feedback along two axes: positive vs. negative and content vs. presentation.

2. Have each person in the class add one point to a 2x2 grid on a whiteboard (or in the shared notes) without duplicating any points that are already up there.

What did other people see that you missed? What did they think that you strongly agree or disagree with?

20 minutes

Practice Giving Feedback

Use the process described above to practice teaching in groups of three. When your group is done, the instructor will add one point of feedback from each participant to a 2x2 grid on the whiteboard or in the shared notes, without accepting duplicates. Participants should not say whether the point they offer was made by them, about them, or neither: the goal at this stage is primarily for people to become comfortable with giving and receiving feedback, and to establish a consensus about what sorts of things to look for.

45 minutes

Expertise and Memory

> **Objectives**
>
> - Learners can define expertise and explain its operation using a graph metaphor for cognition.
> - Learners can explain the difference between repetition and deliberate practice.
> - Learners can define and construct concept maps, and explain the benefits of externalizing cognition.
> - Learners can differentiate long-term and short-term memory, describe the capacity limits of the latter, and explain the the impact of these limits on teaching.

The previous chapter looked at what distinguishes novices from competent practitioners. Here, we will look at expertise: what it is, how people acquire it, and how it can be harmful as well as helpful. We will then see how concept maps can be used to figure out how to turn knowledge into lessons.

To start, what do we mean when we say someone is an expert? The usual response is that they can solve problems much faster than people who are "merely competent", or that they can recognize and deal with the cases where the normal rules don't apply. They also somehow make this look effortless: in most cases, they just know what the right answer is.

What makes someone an expert? The answer isn't just that they know more facts: competent practitioners can memorize a lot of trivia without any noticeable improvement to their performance.

Instead, imagine for a moment that we store knowledge as a graph in which facts are nodes and relationships are arcs. (This is emphatically *not* how our brains work, but it's a useful metaphor.) The key difference between experts and people who are "merely competent" is that experts have many more connections, i.e., their mental models are much more densely connected.

This metaphor helps explain many observed aspects of expert behavior:

- Experts can jump directly from a problem to its solution because there actually is a direct link between the two in their mind. Where a competent practitioner would have to reason "A, B, C, D, E", the expert can go from A to E in a single step. We call this *intuition*, and it isn't always a good thing: when asked to explain their reasoning, experts often can't, because they didn't actually reason their way to the solution–they just recognized it.

- Experts are frequently so familiar with their subject that they can no longer imagine what it's like to *not* see the world that way. As a result, they are often less good at teaching the subject than people with less expertise who still remember what it's like to have to learn the things. This phenomenon is called *expert blind spot*, and while it can be overcome with training, it's part of why there is no correlation between how good someone is at doing research in an area and how good they are at teaching it [Marsh2002].

- Densely-connected knowledge graphs are also the basis for experts' *fluid representations*, i.e., their ability to switch back and forth between different views of a problem [Petre2016].

For example, when trying to solve a problem in mathematics, we might switch between tackling it geometrically and representing it as a set of equations to be solved.

- Finally, this metaphor also explains why experts are better at diagnosis than competent practitioners: more linkages between facts makes it easier to reason backward from symptoms to causes. (And this in turn is why asking programmers to debug during job interviews gives a more accurate impression of their ability than asking them to program.)

> **The J Word**
>
> Experts often betray their blind spot by using the word "just" in explanations, as in, "Oh, it's easy, you just fire up a new virtual machine and then you just install these four patches to Ubuntu and then you just re-write your entire program in a pure functional language." As we discuss later in Motivation, the J word (also sometimes called the passive dismissive adjective) should be banned from classrooms, primarily because using it gives learners the very clear signal that the instructor thinks their problem is trivial and that they therefore must be stupid.

The graph model of knowledge explains why helping learners make connections is as important as introducing them to facts. To use another analogy, the more people you know in a group, the more likely you are to remain part of that group. Similarly, the more connections a fact has to other facts, the more likely the fact is to be remembered.

Repetition vs. Deliberate Practice

The idea that ten thousand hours of practice will make someone an expert in some field is widely quoted, but reality is more complex. Doing exactly the same thing over and over again is much more likely to solidify bad habits than perfect performance. What actually works is *deliberate practice* (also sometimes called *reflective practice*), which is doing similar but subtly different things, paying attention to what works and what doesn't, and then changing behavior in response to that feedback to get cumulatively better.

A common progression is for people to go through three stages:

1. They *learn how to do something given feedback from others*. For example, they might write an essay about what they did on their summer holiday, and get feedback from a teacher telling them how to improve it.

2. They *learn how to give feedback*. For example, they might write an essay about character development in *The Catcher in the Rye*, and get feedback on their critique from a teacher.

3. They *apply what they've learned about feedback to themselves*. At some point, they start critiquing their own work in real time (or nearly so) using the critical skills they've built up in steps 1 and

    i. Doing this is so much faster than waiting for feedback from others that proficiency suddenly starts to take off.

A meta-study conducted in 2014 [Macnamara2014] found that "… deliberate practice explained 26% of the variance in performance for games, 21% for music, 18% for sports, 4% for education, and less than 1% for professions." One explanation for this variation is that deliberate practice works best when the rules for evaluating success are very stable, but is less effective when there are more factors at play (i.e., when it's harder to connect cause to effect).

Concept Maps

Our tool of choice to represent a knowledge graph (expert or otherwise) is a *concept map(gloss.md#concept-map)*. A concept map is simply a picture of someone's mental model of a domain: facts are bubbles, and connections are labelled arcs. It is important that they are labelled: saying "X and Y are related" is only helpful if we explain what the relationship *is*. And yes, one person's fact may be another person's connection, but one of the benefits of concept mapping is that it makes those differences explicit.

> **Externalizing Cognition**
>
> Concept maps are just one way to represent our understanding of a subject. For example, Andrew Abela's decision tree presents a mental model of how to choose the right kind of chart for different kinds of questions and data. Maps, flowcharts, and blueprints can also be useful in some contexts. What each does is *externalize cognition*, i.e., make thought processes and mental models visible so that they can be compared, contrasted, and combined.

To show what concept maps look like, consider this simple `for loop in Python:

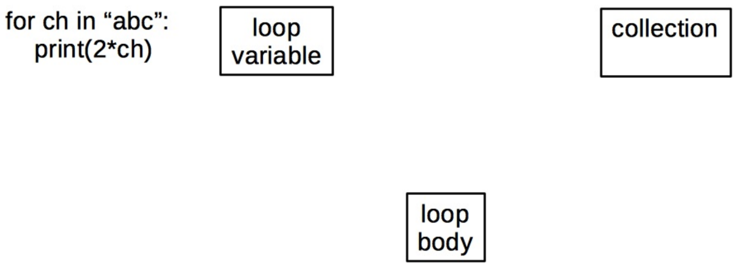
```
for letter in "abc":
```

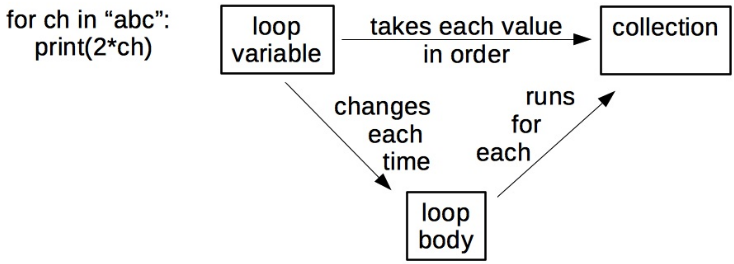
```
        print('*' + letter)
```

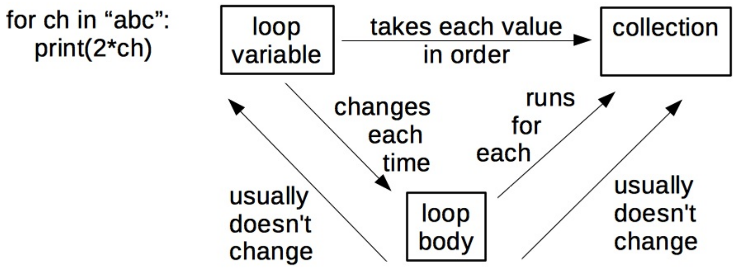
whose output is:

```
*a
*b
*c
```

The three key "things" in this loop are shown in the first part of the figure below, but they are only half the story–and arguably, the less important half. The second part shows the *relationships* between those things. We can go further and add two more relationships that are usually (but not always) true as shown in the third part.







Concept maps can be used in many ways:

1. Concept maps aid design of a lesson by helping authors figure out what they're trying to teach. Crucially, a concept map separates content from order: in our experience, people rarely

wind up teaching things in the order in which they first drew them.

2. They also aid communication between lesson designers. Instructors with very different ideas of what they're trying to teach are likely to pull their learners in different directions. Drawing and sharing concept maps isn't guaranteed to prevent this, but it certainly helps.

3. Concept maps also aid communication with learners. While it's possible to give learners a pre-drawn map at the start of a lesson for them to annotate, it's better to draw it piece by piece while teaching to reinforce the ties between what's in the map and what the instructor said. (We will return to this idea when we discuss Mayer's work on multimedia learning in Cognitive Load.

4. Concept maps are also a useful for assessment: having learners draw concept maps of what they think they just heard shows the instructor what was missed and what was mis-understood. However, reviewing learners' concept maps is too time-consuming for use in class, but very useful in weekly lectures *once learners are familiar with the technique*. The qualification is necessary because any new way of doing things initially slows people down–if a student is trying to make sense of basic programming, asking them to figure out how to draw their thoughts at the same time is an unfair load. Finally, some instructors are skeptical of whether novices can effectively map their understanding, since introspection and explanation of understanding are generally more advanced skills than understanding itself.

> **Meetings, Meetings, Meetings**
>
> The next time you have a team meeting, give everyone a sheet of paper and have them spend a few minutes drawing a concept map of the project you're all working on–separately. On the count of three, have everyone reveal their concept maps simultaneously. The discussion that follows everyone's realization of how different their mental models of the project's aims and organization are is always interesting…

Seven Plus or Minus Two

The graph model of knowledge is wrong but useful, but another simple model has a sound physical basis. As a rough approximation, human memory can be divided into two distinct layers. The first is called *long-term* or *persistent memory*. It is where we store things like our password, our home address, and what the clown did at our eighth birthday party that scared us so much. It is essentially unbounded: barring injury or disease, we will die before it fills up. However, it is also slow to access–too slow to help us handle hungry lions and disgruntled family members.

Evolution has therefore given us a second system called *short-term* or *working memory*. It is much faster, but also much smaller: in 1956, Miller estimated that the average adult's working memory could hold 7±2 items for a few seconds before things started to drop out[1]. This is why phone numbers are typically 7 or 8 digits long: back when phones had dials instead of keypads, that was the longest string of numbers most adults could remember accurately for as long as it took the dial to go around and around. It's also why sports teams tend to have about half a dozen members, or be broken down into smaller groups (such as the forwards and backs in rugby).

> **Chunking**
>
> Our minds can store larger numbers of facts in short-term memory by creating *chunks*. For example, most of us will remember a word we read as a single item, rather than as a sequence of letters. Similarly, the pattern made by five spots on cards or dice is remembered as a whole rather than as five separate pieces of information. Chunks allow us to manage larger problems, but can also mislead us if we mis-identify something, i.e., see it as something it isn't. We will discuss this in more detail later.

7±2 is probably the most important number in programming. When someone is trying to write the next line of a program, or understand what's already there, she needs to keep a bunch of arbitrary facts straight in her head: what does this variable represent, what value does it currently hold, etc. If the number of facts grows too large, her mental model of the program comes crashing down (something we have all experienced).

7±2 is also the most important number in teaching. An instructor cannot push information directly into a learner's long-term memory. Instead, whatever she presents is first represented in the learner's short-term memory, and is only transferred to long-term memory after it has been held there and rehearsed. If we present too much information too quickly, the new will displace the old before it has a chance to consolidate in long-term memory.

This is why it's very important to use a technique like concept mapping a lesson before teaching it - an instructor needs to identify just how many pieces of separate information will need to be "stored" in memory as part of the lesson.

> **Building Concept Maps Together**
>
> Concept maps can be used as a classroom discussion exercise. Put learners in small groups (2-4 people each), give each group some sticky notes on which a few key concepts are written, and have them build a concept map on a whiteboard by placing those sticky notes, connecting them with labelled arcs, and adding any other concepts they think they need.

> **What Are We Doing Again?**
>
> Concept maps can also be used to help build a shared understanding of what a project is trying to accomplish. Everyone independently draws a concept map to show what they think the project's goals and constraints are. Those concept maps are then revealed simultaneously. The ensuing discussion can be…vigorous.

Challenges

Concept Mapping

Create a hand drawn concept map for something you would teach in five minutes. (If possible, do it for the same subject that created a multiple choice question for earlier.) Trade with a partner, and critique each other's maps. Do they present concepts or surface detail? Which of the relationships in your partner's map do you consider concepts and vice versa?

30 minutes

Footnotes

> 1. More recent estimates put the number closer to 4±1, which means that effective chunking is even more important than first thought. ↩

Cognitive Load

> **Objectives**
>
> - Learners can define cognitive load and explain how consideration of it can be used to shape instruction.
> - Learners can explain what faded examples are and construct faded examples for use in programming workshops.
> - Learners can explain what Parson's Problems are and construct Parson's Problems for use in programming workshops.
> - Learners can describe ways in which they differ from their own learners and what effect those differences have on instruction.

In 2006, Kirschner, Sweller, and Clark published a paper titled "Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching" [Kirschner2006]. Its abstract says:

> Although unguided or minimally guided instructional approaches are very popular and intuitively appealing… these approaches ignore both the structures that constitute human cognitive architecture and evidence from empirical studies over the past half-century that consistently indicate that minimally guided instruction is less effective and less efficient than instructional approaches that place a strong

> emphasis on guidance of the student learning process. The advantage of guidance begins to recede only when learners have sufficiently high prior knowledge to provide "internal" guidance.

The paper set off a minor academic firestorm, because beneath the jargon the authors were claiming that *inquiry-based learning* doesn't actually work very well. Inquiry-based learning is the practice of allowing learners to ask their own questions, set their own goals, and find their own path through a subject, just as they would when solving problems in real life. It is intuitively appealing, but Kirschner argued that it overloads learners, since it requires them to simultaneously master both a domain's factual content and its problem-solving strategies.

More specifically, *cognitive load theory* posits that people have to deal with three things when they're learning:

1. *Intrinsic* load is what people have to keep in mind in order to carry out a learning task. In a programming class, this might be understanding what a variable is, or understanding how assignment in a programming language is different from creating a reference to a cell in a spreadsheet.

2. *Germane* load is the (desirable) mental effort required to create linkages between new information and old (which is one of the things that distinguishes learning from memorization). An example might be learning how to loop through a collection in Python.

3. *Extraneous* load is everything else that distracts or gets in the way, such as knowing that tabs look like multiple characters but only count as one when indenting Python code.

According to this theory, searching for a solution strategy is an extra burden on top of applying that strategy. We can therefore accelerate learning by giving learners worked examples that show them a problem and a detailed step-by-step solution, followed by a series of *faded examples*. The first of these presents a nearly-complete use of the same problem-solving strategy just demonstrated, but with a small number of blanks for the learner to fill in. The next problem is also of the same type, but has more blanks, and so on until the learner is asked to solve the entire problem. (The material that *isn't* blank is often referred to as *scaffolding*, since it serves the same purpose as the scaffolding set up temporarily at a building site.)

For example, someone teaching Python might start by explaining this:

```
# total_length(["red", "green", "blue"]) => 12
def total_length(words):
      total = 0
      for word in words:
          total += len(word)
      return total
```

then ask learners to fill in the blanks in:

```
# word_lengths(["red", "green", "blue"]) => [3, 5, 4]
def word_lengths(words):
      lengths = ____
      for word in words:
```

```
        lengths ____
    return lengths
```

The next problem might be:

```
# concatenate_all(["red", "green", "blue"]) => "redgreenb
lue"
def concatenate_all(words):
    result = ____
    for ____ in ____:
        ____
    return result
```

and learners would finally be asked to tackle:

```
# acronymize(["red", "green", "blue"]) => "RGB"
def acronymize(words):
    ____
```

Faded examples work because they introduce the problem-solving strategy piece by piece. At each step, learners have one new problem to tackle. As discussed later, this is less intimidating than a blank screen or a blank sheet of paper. It also encourages learners to think about the similarities and differences between various approaches, which helps create the linkages in the mental model that instructors want them to form.

The key to constructing a good faded example is to think about the problem-solving strategy or solution pattern that it is meant to teach. For example, the series of problems are all examples of the *accumulator pattern*, in which the results of processing items from a collection are repeatedly added to a single variable in some way to create the final result.

Cognitive load theory has been criticized as being unfalsifiable: since there's no way to tell in advance of an experiment whether something is germane or not, any result can be justified after the fact by labelling things that hurt performance as "extraneous" and things that don't "germane". However, there is no doubt that faded examples are effective.

> **Split Attention**
>
> Research by Mayer and colleagues on the split-attention effect is closely related to cognitive load theory [Mayer2003]. Linguistic and visual input are processed by different parts of the human brain, and linguistic and visual memories are stored separately as well. This means that correlating linguistic, auditory, and visual streams of information takes cognitive effort: when someone reads something while hearing it spoken aloud, their brain can't help but check that it's getting the same information on both channels.
>
> Learning is therefore more effective when redundant information is *not* being presented simultaneously in two different channels. For example, people find it harder to learn from a video that has both narration and on-screen captions than from one that has either the narration or the captions but not both.

> This is also why it's more effective to draw a diagram piece by piece while teaching rather than presenting the whole thing at once. If parts of the diagram appear at the same time as things are being said, the two will be correlated in the learner's memory, so that pointing at part of the diagram will trigger recall of what was being said.

Another way to use cognitive load theory to construct exercises is called a *Parson's Problem*. If you are teaching someone to speak a new language, you could ask them a question, and then give them the words they need to answer the question, but in jumbled order. Their task is to put the words in the right order to answer the question grammatically, which frees them from having to think simultaneously about what to say *and* how to say it.

Similarly, when teaching people to program, you can give them the lines of code they need to solve a problem, and ask them to put them in the right order. This allows them to concentrate on control flow and data dependencies, i.e., on what has to happen before what, without being distracted by variable naming or trying to remember what functions to call.

Pattern Recognition

An earlier section described how people chunk related or correlated information together so that they can fit more into short-term memory. One key finding in cognition research is that experts have more and larger chunks than non-experts, i.e., experts "see" larger patterns, and have more patterns to match things against. This allows them to reason at a higher level, and to search for information more quickly and more accurately.

It is therefore tempting to try to teach patterns directly–in fact, supporting this is one of the reasons programmers have been so enthusiastic about design patterns. In practice, though, pattern catalogs are too large to flick through and too dry to memorize directly. Giving names to a small number of patterns, though, does seem to help with teaching, primarily by giving the learners a richer vocabulary to think and communicate with [Kuittinen2004].

You Are Not Your Learners

People learn best when they care about the topic and believe they can master it. Neither fact is particularly surprising, but their practical implications have a lot of impact on what we teach, and the order in which we teach it.

First, as noted in Motivation, most people don't actually want to program: they want to build a website or check on zoning regulations, and programming is just a tax they have to pay along the way. They don't care how hash tables work, or even that hash tables exist; they just want to know how to process data faster. We therefore have to make sure that everything we teach is useful right away, and conversely that we don't teach anything just because it's "fundamental".

Second, believing that something will be hard to learn is a self-fulfilling prophecy. This is why it's important not to say that something is easy: if someone who has been told that tries it, and it doesn't work, they are more likely to become discouraged.

It's also why installing and configuring software is a much bigger problem for us than experienced programmers like to acknowledge. It isn't just the time we lose at the start of boot camps as we try to get a Unix shell working on Windows, or set up a version control client on some idiosyncratic Linux distribution.

It isn't even the unfairness of asking students to debug things that depend on precisely the knowledge they have come to learn, but which they don't yet have. The real problem is that every such failure reinforces the belief that computing is hard, and that they'd have a better chance of making next Thursday's deadline at work if they kept doing things the way they always have. For these reasons, we have adopted a "teach most immediately useful first" approach described in Motivation.

Challenges

Create a Faded Example

It's very common for programs to count how many things fall into different categories: for example, how many times different colors appear in an image, or how many times different words appear in a paragraph of text.

1. Create a short example (no more than 10 lines of code) that shows people how to do this, and then create a second example that solves a similar problem in a similar way, but has

a couple of blanks for learners to fill in. How did you decide what to fade out? What would the next example in the series be?

2. Define the audience for your examples. For example, are these beginners who only know some basics programming concepts? Or are these learners with some experience in programming but not in Python?

3. Show your example to a partner, but do *not* tell them what level it is intended for. Once they have filled in the blanks, ask them what level they think it is for.

If there are people among the trainees who don't program at all, make sure that they are in separate groups and ask to the groups to work with that person as a learner to help identify different loads.

30 minutes

Create a Parson's Problem

Write five or six lines of code that does something useful, jumble them, and ask your partner to put them in order. If you are using an indentation-based language like Python, do not indent any of the lines; if you are using a curly-brace language like Java, do not include any of the curly braces.

20 minutes

Designing Lessons

> **Objectives**
>
> - Learners can describe the steps in reverse instructional design and explain why it generally produces better lessons than the usual "forward" lesson development process.
> - Learners can define "teaching to the test" and explain why reverse instructional design is *not* the same thing.
> - Learners can construct and critique five-part learner profiles.
> - Learners can construct good learning objectives and critique learning objectives with reference to Bloom's Taxonomy.

Most people design lessons as follows:

1. Someone tells you that you have to teach something you haven't thought about in ten years.

2. You start writing slides to explain what you know about the subject.

3. After two or three weeks, you make up an assignment based more or less on what you've taught so far.

4. You repeat step 3 several times.

5. You stay awake into the wee hours of the morning to create a final exam.

There's a better way, but to explain it, we first need to explain how *test-driven development* (TDD) is used in software development. Programmers who are using TDD don't write software and then (possibly) write tests. Instead, they write the tests first, then write just enough new software to make those tests pass, and then clean up a bit.

TDD works because writing tests forces programmers to specify exactly what they're trying to accomplish and what "done" looks like. It's easy to be vague when using a human language like English or Korean; it's much harder to be vague in Python or R.

TDD also reduces the risk of endless polishing, and also the risk of confirmation bias: someone who hasn't written a program is much more likely to be objective when testing it than its original author, and someone who hasn't written a program *yet* is more likely to test it objectively than someone who has just put in several hours of hard work and really, really wants to be done.

A similar "backward" method works very well for lesson design. This method is something called *reverse instructional design* or *understanding by design* after a book by that name [Wiggins2005]; a similar method is described in [Fink2013][1]. In brief, lessons should be designed as follows:

1. Create learner profiles (discussed in the next section) to figure out who you are trying to teach and what will appeal to them.

2. Draw concept maps to describe the mental model you want them to construct.

3. Create a summative assessment, such as a final exam or performance, that will show you whether learning has actually taken place.
4. Create formative assessments that will give the learners a chance to practice the things they'll be asked to demonstrate in the summative assessment, and tell you and them whether they're making progress and where they need to focus their work.
5. Put the formative assessments in order based on their complexity and dependencies.
6. Write just enough to get learners from one formative assessment to the next. An actual classroom lesson will typically then consist of three or four such episodes, each building toward a short check that learners are keeping up.

This method helps to keep teaching focused on its objectives. It also ensures that learners don't face anything on the final exam that the course hasn't prepared them for.

> **Building Lessons by Subtracting Complexity**
>
> One way to build a programming lesson is to write the program you want learners to finish with, then remove the most complex part that you want them to write and make it the last exercise. You can then remove the next most complex part you want them to write and make it the penultimate exercise, and so on. Anything that's left–i.e., anything you don't want them to write as an exercise–

> becomes the starter file(s) that you give them. This typically includes things like importing libraries or helper functions to access data.

> **How and Why to Fake It**
>
> One of the most influential papers in the history of software engineering was Parnas and Clements' "A Rational Design Process: How and Why to Fake It". In it, the authors pointed out that in real life we move back and forth between gathering requirements, interface design, programming, and testing, but when we write up our work it's important to describe it as if we did these steps one after another so that other people can retrace our steps. The same is true of lesson design: while we may change our mind about what we want to teach based on something that occurs to us while we're writing an MCQ, we want the notes we leave behind to present things in the order described above.

> **Teaching to the Test**
>
> Reverse instructional design is *not* the same thing as "teaching to the test". When using RID, teachers set goals to aid in lesson design, and may never actually give the final exam that they wrote. In many school systems, on the other hand, an external authority defines assessment criteria for all learners, regardless of their individual situations, and the outcomes of those summative assessments directly affect the teachers' pay and promotion. Green's *Building a Better Teacher* [Green2014] argues that this focus on measurement is appealing to

> those with the power to set the tests, but is unlikely to improve outcomes unless it is coupled with support for teachers to make improvements based on test outcomes. This is often missing, because as Scott pointed out in [Scott1999], large organizations usually value uniformity over productivity.

Learner Profiles

The first piece of the process above is figuring out who your audience is. One way to do this is to write two or three *learner profiles*. This technique is borrowed from user interface design, where short profiles of typical users are created to help designers think about their audience's needs, and to give them a shorthand for talking about specific cases.

Learner profiles have five parts: the person's general background, what they already know, what *they* think they want to do, how the course will help them, and any special needs they might have. A learner profile for a weekend workshop aimed at new college students might be:

1. Jorge has just moved from Costa Rica to Canada to study agricultural engineering. He has joined the college soccer team, and is looking forward to learning how to play ice hockey.

2. Other than using Excel, Word, and the Internet, Jorge's most significant previous experience with computers is helping his sister build a WordPress site for the family business back home in Costa Rica.

3. Jorge needs to measure properties of soil from nearby farms using a handheld device that sends logs in a text format to his computer. Right now, Jorge has to open each file in Excel, crop the first and last points, and calculate an average.

4. This workshop will show Jorge how to write a little Python program to read the data, select the right values from each file, and calculate the required statistics.

5. Jorge can read English proficiently, but still struggles sometimes to keep up with spoken conversation (especially if it involves a lot of new jargon).

A single learner profile is sometimes enough, but two or three that cover the whole range of potential learners is better. One of the ways they help is by serving as a shorthand for design issues: when speaking with each other, lesson authors can say, "Would Jorge understand why we're doing this?" or, "What installation problems would Jorge face?"

Learning Objectives

Summative and formative assessments help instructors figure out what they're going to teach, but in order to communicate that to learners and other instructors, a course description should also have *learning objectives* (sometimes also called a *learning goal*). A learning objective is a single sentence describing what a learner will be able to do once they have sat through the lesson in order to demonstrate what they have learned.

Learning objectives are meant to ensure that everyone has the same understanding of what a lesson is supposed to accomplish. For example, a statement like "understand Git" could mean any of the following, each of this would be backed by a very different lesson:

- Learners can describe three scenarios in which version control systems like Git are better than file-sharing tools like Dropbox, and two in which they are worse.
- Learners can commit a changed file to a Git repository using a desktop GUI tool.
- Learners can explain what a detached HEAD is and recover from it using command-line operations.

> **Objectives vs. Outcomes**
>
> A learning objective is what a lesson strives to achieve. A *learning outcome* is what it actually achieves, i.e., what learners actually take away. The role of summative assessment is therefore to compare outcomes with objectives.

More specifically, a good learning objective has a *measurable or verifiable verb* that states what the learner will do, and specifies the *criteria for acceptable performance*. Writing these kinds of learning objectives may initially seem restrictive or limiting, but will make both you, your fellow instructors, and your learners happier in the long run. You will end up with clear guidelines for both your teaching and assessment, and your learners will appreciate the clear expectations.

One tool that can help when writing learning objectives is Bloom's taxonomy, which was first published in 1956. It attempts to define levels of understanding in a way that is hierarchical, measurable, stable, and cross-cultural. The list below defines the levels in Bloom's Taxonomy and shows some of the verbs typically used in learning objectives written for each level.

- Knowledge: recalling learned information (name, define, recall).
- Comprehension: explaining the meaning of information (restate, locate, explain, recognize).
- Application: applying what one knows to novel, concrete situations (apply, demonstrate, use).
- Analysis: breaking down a whole into its component parts and explaining how each part contributes to the whole (differentiate, criticize, compare).

Synthesis: assembling components to form a new and integrated whole (design, construct, organize).

Evaluation: using evidence to make judgments about the relative merits of ideas and materials (choose, rate, select).

Another way to understand what makes for a good learning objective is to see how a poor one can be improved:

- Learner will be given opportunities to learn good programming practices.
  Describes the lesson's content, not the attributes of successful students.

- Learner will have a better appreciation for good programming practices.
  Doesn't start with an active verb or define the level of learning, and the subject of learning has no context and is not specific.

- Learner will understand how to program in R.
  Starts with an active verb, but doesn't define the level of learning, and the subject of learning is still too vague for assessment.

- Learner will write one-page read-filter-summarize-print data analysis scripts for tabular data using R and R Studio.
  Starts with an active verb, defines the level of learning, and provides context to ensure that outcomes can be assessed.

Baume's guide to writing and using good learning outcomes [Baume2009] is a good longer discussion of these issues. This list of learning goals from the University of British Columbia may also provide some useful guidance.

Challenges

Learner Profiles

Working in pairs or small groups, create a five-point profile that describes one of your typical learners.

30 minutes

Write Learning Objectives

Write one more learning objectives for something you currently teach or plan to teach. Working with a partner, critique and improve the objectives.

20 minutes

Footnotes

> [1]. A summary of [Fink2013] is freely available online. ↩

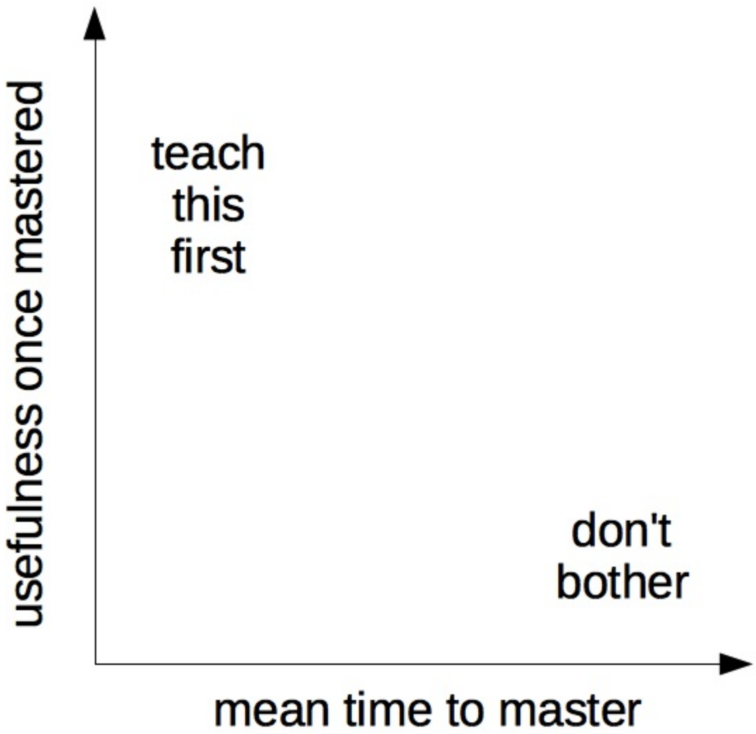
Motivation and Demotivation

> **Objectives**
>
> - Learners can name and describe the three principal ways in which they can demotivate their own learners.
> - Learners can define impostor syndrome and stereotype threat, and describe ways in which to combat each.
> - Learners can describe the difference between fixed and growth mindset and explain the importance of encouraging the latter.
> - Learners can describe and enact at least three things they can do to make their programming workshops more accessible.
> - Learners can describe and enact at least three things they can do to make their programming workshops more inclusive.

Learners need encouragement to step out into unfamiliar terrain, so this chapter discusses ways instructors can motivate them. More importantly, it discusses ways that we can accidentally *demotivate* them, and how we can avoid doing that.

People learn best when they care about the topic and believe they can master it. This presents us with a problem because most people don't actually want to program: they want to make music or compare changes to zoning laws with family incomes, and rightly regard programming as a tax they have to pay in order to do so. In addition, their early experiences with programming are often demoralizing, and believing that something will be hard to learn is a self-fulfilling prophecy.

Imagine a grid whose axes are labelled "mean time to master" and "usefulness once mastered". Everything that's quick to master, and immediately useful should be taught first; things in the opposite corner that are hard to learn and have little near-term application don't belong in this course.



> **Actual Time**

> Any useful estimate of how long something takes to master must take into account how frequent failures are and how much time is lost to them. For example, editing a text file seems like a simple task, but most graphical editors save things to the user's desktop or home directory. If people need to run shell commands on the files they've edited, a substantial fraction won't be able to navigate to the right directory without help. If this seems like a small problem to you, please revisit the discussion of expert blind spot in Memory.

Many of the foundational concepts of computer science, such as computability, inhabit the "useful but hard to learn" corner of the grid described above. This doesn't mean that they aren't worth learning, but if our aim is to convince people that they *can* learn this stuff, and that doing so will help them do more science faster, they are less compelling than things like automating repetitive tasks.

We therefore recommend a "teach most immediately useful first" approach. Have learners do something that *they* think is useful in their daily work within a few minutes of starting each lesson. This not only motivates them, it also helps build their confidence in us, so that if it takes longer to get to the payoff of a later topic, they'll stick with us.

The best-studied use of this idea is the media computation approach developed by Guzdial and Ericson at Georgia Tech [Guzdial2013]. Instead of printing "hello world" or summing the first ten integers, their students' first program opens an image, resizes it to create a thumbnail, and saves the result. This is an *authentic* *task*, i.e., something that learners believe they would actually do in real life. It is also has a *tangible artifact*: if the image comes out the wrong size, learners have a concrete starting point for debugging.

> **Strategies for Motivating Learners**
>
> [Ambrose2010] contains a list of evidence-based methods to motivate learners. None of them are surprising–it's hard to imagine someone saying that we *shouldn't* identify and reward what we value–but it's useful to check lessons against these points to make sure they're doing at least a few of these things.
>
> What's missing from this list is strategies to motivate the *instructor*. Learners respond to an instructor's enthusiasm, and instructors need to care about a topic in order to keep teaching it, particularly when they are volunteers.

Demotivation

> *Women aren't leaving computing because they don't know what it's like; they're leaving because they *do* know.
> – variously attributed

If you are teaching free-range learners, they are probably already motivated–if they weren't, they wouldn't be in your classroom. The challenge is therefore not to demotivate them. Unfortunately, we can do this by accident much more easily than you might think.

The three most powerful demotivators are *unpredictability*, *indifference*, and *unfairness*. Unpredictability demotivates people because if there's no reliable connection between what they do and what outcome they achieve, there's no reason for them to try to do anything[1]. If learners believe that the instructor or the educational system doesn't care about them or the lesson, they won't care either. And if people believe the class is unfair, they will also be demotivated, even if it is unfair in their favor (because consciously or unconsciously they will worry that they will some day find themselves in the group on the losing end [Wilkinson2011]).

Here are some quick ways to demotivate your learners:

- A "holier-than-thou" or contemptuous attitude from an instructor.
- Tell learners they are rubbish because they use Excel and/or Word, don't modularize their code, etc.
- Repeatedly make digs about Windows and praise Linux, e.g., say that the former is for amateurs.
- Criticize GUI applications (and by implication their users) and describe command-line tools as the One True Way.
- Dive into complex or detailed technical discussion with the one or two people in the audience who clearly don't actually need to be there.
- Pretend to know more than you do. People will actually trust you more if you are frank about the limitations of your knowledge, and will be more likely to ask questions and seek help.
- Use the J word ("just"). As discussed in Memory, this signals to the learner that the instructor thinks their problem is trivial and by extension that they therefore must be stupid for not

being able to figure it out.

- Feign surprise. Saying things like "I can't believe you don't know X" or "you've never heard of Y?" signals to the learner that they do not have some required pre-knowledge of the material you are teaching, that they are in the wrong place, and it may prevent them from asking questions in the future. (This idea comes from the Recurse Center's Social Rules).

> **Code of Conduct Revisited**
>
> As noted in Welcome, we believe very strongly that classes should have a Code of Conduct. Its details are important, but the most important thing about it is that it exists: knowing that we have rules tells people a great deal about our values and about what kind of learning experience they can expect.

> **Never Learn Alone**
>
> One way to support learners who have been subject to systematic exclusion or discrimination (overt or otherwise) is to have people sign up for workshops in small teams rather than as individuals. If an entire lab group comes, or if attendees are drawn from the same (or closely-related) disciplines, everyone in the room will know in advance that they will be with at least a few people they trust, which increases the chances of them actually coming. It also helps after the workshop: if people come with their friends or colleagues, they can work together to implement what they've learned.

Impostor Syndrome

Impostor syndrome is the belief that one is not good enough for a job or position, that one's achievements are lucky flukes, and an accompanying fear of being "found out". Impostor syndrome seems to be particularly common among high achievers who undertake publicly visible work.

Academic work is frequently undertaken alone or in small groups but the results are shared and criticized publicly. In addition, we rarely see the struggles of others, only their finished work, which can feed the belief that everyone else finds it easy. Women and minority groups who already feel additional pressure to prove themselves in some settings may be particularly affected.

Two ways of dealing with your own impostor syndrome are:

1. Ask for feedback from someone you respect and trust. Ask them for their honest thoughts on your strengths and achievements, and commit to believing them.

2. Look for role models. Who do you know who presents as confident and capable? Think about how they conduct themselves. What lessons can you learn from them? What habits can you borrow? (Remember, they quite possibly also feel as if they are making it up as they go.)

As an instructor, you can help people with their impostor syndrome by sharing stories of mistakes that you have made or things you struggled to learn. This reassures the class that it's OK to find topics hard. Being open with the group makes it easier to build trust and make students confident to ask questions. (Live coding is great for this: typos let the class see you're not superhuman.)

You can also emphasize that you want questions: you are not succeeding as a teacher if no one can follow your class, so you're asking students for their help to help you learn and improve. Remember, it's much more important to *be* smart than to *look* smart.

The Ada Initiative has some excellent resources for teaching about and dealing with imposter syndrome.

Stereotype Threat

Reminding people of negative stereotypes, even in subtle ways, makes them anxious about the risk of confirming those stereotypes, which in turn reduces their performance. This is called *stereotype threat*, and the clearest examples in computing are gender-related. Depending on whose numbers you trust, only 12-18% of programmers are women, and those figures have actually been getting worse over the last 20 years. There are many reasons for this (see [Margolis2003] and [Margolis2010]), and [Steele2011] summarizes what we know about stereotype threat in general and presents some strategies for mitigating it in the classroom.

However, while there's lots of evidence that unwelcoming climates demotivate members of under-represented groups, it's not clear that stereotype threat is the underlying mechanism. Part of the problem is that the term has been used in many ways; another is [questions about the replicability of key studies][stereotype-threadreplicability]. What *is* clear is that we need to avoid thinking in terms of a deficit model (i.e., we need to change the members of under-represented groups because they have some deficit, such as lack of prior experience) and instead use a systems approach (i.e., we need to change the system because it produces these disparities).

A great example of how stereotypes work in general was presented in Patitsas et al's "Evidence That Computer Science Grades Are Not Bimodal" [Patitsas2016]. This thought-provoking paper showed that people see evidence for a "geek gene" where none exists. As the paper's abstract says:

> Although it has never been rigorously demonstrated, there is a common belief that CS grades are bimodal. We statistically analyzed 778 distributions of final course grades from a large research university, and found only 5.8% of the distributions passed tests of multimodality. We then devised a psychology experiment to understand why CS educators believe their grades to be bimodal. We showed 53 CS professors a series of histograms displaying ambiguous distributions and asked them to categorize the distributions. A random half of participants were primed to think about the fact that CS grades are commonly thought to be bimodal; these participants were more likely to label ambiguous distributions as "bimodal". Participants were also more likely to label distributions as bimodal if they believed that some students are innately predisposed to do better at CS. These results suggest that bimodal grades are instructional folklore in CS, caused by confirmation bias and instructor beliefs about their students.

It's easy to use language that suggests that some people are natural programmers and others aren't, but Mark Guzdial has called this belief the biggest myth about teaching computer science.

Mindset

Learners can be demotivated in subtler ways as well. For example, Dweck and others have studied the differences of fixed mindset and growth mindset. If people believe that competence in some area is intrinsic (i.e., that you either "have the gene" for it or you don't), *everyone* does worse, including the supposedly advantaged. The reason is that if they don't get it at first, they figure they just don't have that aptitude, which biases future performance. On the other hand, if people believe that a skill is learned and can be improved, they do better on average.

A person's mindset can be shaped by subtle cues. For example, if a child is told, "You did a good job, you must be very smart," they are likely to develop a fixed mindset. If on the other hand they are told, "You did a good job, you must have worked very hard," they are likely to develop a growth mindset, and subsequently achieve more. Studies have also shown that the simple action of telling learners about the different mindsets before a course can improve learning outcomes for the whole group.

As with stereotype threat, there are concerns that research on grown mindset has been oversold, or will be much more difficult to put into practice than its more enthusiastic advocates have implied. While some people interpret this back and forth of claim and counter-claim as evidence than education research isn't reliable, what it really shows is that anything involving human subjects is both subtle and difficult.

Accessibility

Not providing equal access to lessons and exercises is about as demotivating as it gets. If you look at the old Software Carpentry lessons, for example, the text beside the slides includes all of the narration–but none of the Python source code. Someone using a screen reader would therefore be able to hear what was being said about the program, but wouldn't know what the program actually was.

While it may not be possible to accommodate everyone's needs, it *is* possible to get a good working structure in place without any specific knowledge of what specific disabilities people might have. Having at least some accommodations prepared in advance also makes it clear that hosts and instructors care enough to have thought about problems in advance, and that any additional concerns are likely to be addressed.

> **It Helps Everyone**
>
> Curb cuts (the small sloped ramps joining a sidewalk to the street) were originally created to make it easier for the physically disabled to move around, but proved to be equally helpful to people with strollers and grocery carts. Similarly, steps taken to make lessons more accessible to people with various disabilities also help everyone else.

> Proper captioning of images, for example, doesn't just give screen readers something to say: it also makes the images more findable by exposing their content to search engines.

The first and most important step in making lessons accessible is to *involve people with disabilities in decision-making*: the slogan *nihil de nobis, sine nobis* (literally, "nothing about us, without us") predates accessibility rights, but is always the right place to start. A few other recommendations are:

- *Find out what you need to do.* The W3C Accessibility Initiative's checklist for presentations is a good starting point focused primarily on assisting the visually impaired, while Liz Henry's blog post about [accessibility at conferences][henry-accessibility] has a good checklist for people with mobility issues, and this interview with Chad Taylor is a good introduction to issues faced by the hearing impaired.

- *Know how well you're doing.* For example, sites like WebAIM allow you to check how accessible your online materials are to visually impaired users.

- *Don't do everything at once.* We don't ask learners in our workshops to adopt all our best practices or tools in one go, but instead to work things in gradually at whatever rate they can manage. Similarly, try to build in accessibility habits when preparing for workshops by adding something new each time.

- *Do the easy things first.* There are plenty of ways to make workshops more accessible that are both easy and don't create extra cognitive load for anyone: font choices, general text size, checking in advance that your room is accessible via an elevator or ramp, etc.

Inclusivity

Inclusivity is a policy of including people who might otherwise be excluded or marginalized. In computing, it means making a positive effort to be more welcoming to women, people of color, people with various sexual orientations, the elderly, the physically challenged, the formerly incarcerated, the economically disadvantaged, and everyone else who doesn't fit Silicon Valley's white/Asian male demographic. Lee's paper "What can I do today to create a more inclusive community in CS?" [Lee2017] is a brief, practical guide to doing that with references to the research literature. These help learners who belong to one or more marginalized or excluded groups, but help motivate everyone else as well; while they are phrased in terms of term-long courses, many can be applied in our workshops:

- Ask learners to email you before the workshop to explain how they believe the training could help them achieve their goals.
- Review notes to make sure they are free from gendered pronouns, that they include culturally diverse names, etc.
- Emphasize that what matters is the rate at which they are learning, not the advantages or disadvantages they had when they started.
- Encourage pair programming.

- Actively mitigate behavior that some learners may find intimidating, e.g., use of jargon or "questions" that are actually asked to display knowledge.

Challenges

Several of these challenges use "think-pair-share". If you have not already covered this, please look at the description now.

Authentic Tasks

Think about something you did this week that uses one or more of the skills you teach, (e.g., wrote a function, bulk downloaded data, did some stats in R, forked a repo) and explain how you would use it (or a simplified version of it) as an exercise or example in class.

Pair up with your neighbor and decide where this exercise fits on a 2x2 grid of "short/longtime to master" and "low/high usefulness"? In the shared notes, write the task and where it fits on the grid. As a group, discuss how these relate back to the "teach most immediately useful first" approach.

15 minutes

Implement One Strategy for Inclusivity

Pick one activity or change in practice from Lee's paper [Lee2017] that you would like to work on. Put a reminder in your calendar three months in the future to self-check whether you have done something about it.

5 minutes

Brainstorming Motivational Strategies

1. Think back to a programming course (or any other) that you took in the past, and identify one thing the instructor did that motivated you, and describe what could have been done afterward to correct the situation.
2. Pair up with your neighbor and discuss your stories, then add your comments to the shared notes.
3. Review the comments in the shared notes as a group. Rather than read them all out loud, highlight and discuss a few of the things that could have been done differently. This will give everyone some confidence in how to handle these situations in the future.

20 minutes

Demotivational Experiences

Think back to a time when you demotivated a student (or when you were demotivated as a student). Pair up with your neighbor and discuss what you could have done differently in the situation, and then share the story and what could have been done in the group notes.

15 minutes

Walk the Route

Find the nearest public transportation drop-off point to your building and walk from there to your office and then to the nearest washroom, making notes about things you think would be difficult for someone with mobility issues. Now borrow a wheelchair and repeat the journey. How complete was your list of challenges? And did you notice that the first sentence in this challenge assumed you could actually walk?

15 minutes

Who Decides?

In [Littky2004], Kenneth Wesson wrote, "If poor inner-city children consistently outscored children from wealthy suburban homes on standardized tests, is anyone naive enough to believe that we would still insist on using these tests as indicators of success?" Read this article, and then describe an example from your own experience of "objective" assessments that reinforced the status quo.

15 minutes

Footnotes

> 1. In extreme situations, learners may developed [learned helplessness]: when repeatedly subjected to negative feedback that they have no way to escape, they may learn not to even try to escape when they could. ↩

Teaching Practices

> **Objectives**
>
> - Learners can name, describe, and enact four teaching practices that are appropriate to use in programming workshops for adults, and give a pedagogical justification for each.
> - Learners can explain why instructors should *not* introduce new pedagogical practices in a short workshop.

Just as domain expertise is often a matter of pattern matching, teaching expertise often comes down to using good practices consistently. None of the practices described below are essential (except having a code of conduct), but each will improve lesson delivery.

Have a Code of Conduct

An important part of making a class productive is to treat everyone with respect. We therefore strongly recommend that every group offering classes based on this material adopt a Code of Conduct like the one in Conduct, and require people taking part in the class to abide by it.

We believe equally strongly that your actual programming classes should also have and enforce a Code of Conduct. Programming is a scary topic for many novices, and workshops are meant to be a judgment free space to learn and experiment. The behavior of the instructor and other participants may make more of an impression on a novice learner than any "technical" topic you teach.

If you do this, hosts should point people at it during registration, and instructors should remind attendees of it at the start of the workshop. The Code of Conduct doesn't just tell everyone what the rules are: it tells them that there *are* rules, and that they can therefore expect a safe and welcoming learning experience.

If you are an instructor, and believe that someone in a workshop has violated the Code of Conduct, you may warn them, ask them to apologize, and/or expel them, depending on the severity of the violation and whether or not you believe it was intentional. Whatever you do:

- Do it in front of witnesses. Most people will tone down their language and hostility in front of an audience, and having someone else present ensures that later discussion doesn't degenerate into conflicting claims about who said what.
- Contact the organizer or host of your class as soon as you can and describe what happened. Remember, a Code of Conduct is meaningless without a procedure for enforcing it.

A Code of Conduct cannot stop people from being offensive, any more than laws against theft stop people from stealing. What it *can* do is make expectations and consequences clear. In our experience, people rarely violate the Code of Conduct in person, though some are more likely to online, where they feel less inhibited. And remember, a Code of Conduct is *not* an infringement on free speech. People have a right to say what they think, but that doesn't mean they have a right to speak wherever and whenever they want. If someone wishes to say something disparaging about someone else, they can go and find a space of their own in which to say it.

Starting Out

To begin your class, the instructors should give a brief introduction that will convey their capacity to teach the material, accessibility and approachability, desire for student success, and enthusiasm. Tailor your introduction to the students' skill level so that you convey competence (without seeming too advanced) and demonstrate that you can relate to the students. Throughout the workshop, continually demonstrate that you are interested in student progress and that you are enthusiastic about the topics.

Students should also introduce themselves (preferably verbally). At the very least, everyone should add their name to the Etherpad, but its also good for everyone at a given site to know who all is in the group. Note: this can be done while setting up before the start of the class.

Overnight Homework

In a two-day class, have learners read the operations checklists as overnight homework and do their demotivational story just before lunch on day 2: it means day 2 starts with *their* questions (which wakes them up), and the demotivational story is a good lead-in to lunchtime discussion.

Never a Blank Page

Programming workshops (and other kinds of classes) can be built around a set of independent exercises, develop a single extended example in stages, or use a mixed strategy. The main advantages of independent exercises are that people who fall behind can easily re-synchronize, and that lesson developers can add, remove, and rearrange material at will. A single extended example, on the other hand, will show learners how the bits and pieces they're learning fit together: in educational parlance, it provides more opportunity for them to integrate their knowledge.

Whichever approach you take, learners should never start with a blank page (or screen), since they often find this intimidating or bewildering. Modifying existing code instead of writing new code from scratch doesn't just give them structure: it is also more realistic. Keep in mind, however, that starter code may increase cognitive load, since learners can be distracted by trying to understand it all before they start their own work.

Take Notes Together

Many studies have shown that taking notes while learning improves retention [Aiken1975], [Bohay2011]. As discussed in Memory, this happens because taking notes forces you to organize and reflect on material as it's coming in, which in turn increases the likelihood that you will transfer it to long-term memory in a usable way.

Our experience, and some recent research findings, lead us to believe that taking notes *collaboratively* helps learning even more [Orndorff2015], even though taking notes on a computer is generally less effective than taking notes using pen and paper [Mueller2014]. Taking notes collaboratively:

- It allows people to compare what they think they're hearing with what other people are hearing, which helps them fill in gaps and correct misconceptions right away.
- It gives the more advanced learners in the class something useful to do. Rather than getting bored and checking Twitter during class, they often take the lead in recording what's being said, which keeps them engaged, and allows less advanced learners to focus more of their attention on new material. Keeping the more advanced learners busy also helps the whole class stay engaged because boredom is infectious: if a handful of people start updating their Facebook profiles, the people around them will start checking out too.
- The notes the learners take are usually more helpful *to them* than those the instructor would prepare in advance, since the learners are more likely to write down what they actually found new, rather than what the instructor predicted would be new.
- Glancing at the notes as they're being taken helps the instructor discover that the class didn't hear something important, or misunderstood it.

We usually use Etherpad or Google Docs for collaborative note-taking. The former makes it easy to see who's written what, while the latter scales better and allows people to add images to the notes. Whichever is chosen, classes also use it to share snippets of code and small datasets, and as a way for learners to show instructors their work (by copying and pasting it in).

Shared note-taking is almost always mentioned positively in post-workshop feedback. However, it's also common for participants to report that they find it distracting, as it's one more thing they have to keep an eye on. We believe the positives outweigh the negatives, but think that some careful controlled studies would tell us whether we're right, and how to use it better.

Assess Learners' Motivation and Prior Knowledge

Most formal educational systems train people to treat all assessment as summative, i.e., to think of every interaction with a teacher as an evaluation, rather than as a chance to shape instruction. For example, we use a short pre-assessment questionnaire to profile learners before workshops to help instructors tune the pace and level of material. We send this questionnaire out after people have registered rather than making it part of the sign-up process because when we did the latter, many people concluded that since they couldn't answer all the questions, they shouldn't enrol. We were therefore scaring off many of the people we most wanted to help.

Instead of asking people how easily they could complete specific tasks, we could just ask them to rate their knowledge of various subjects on a scale from 1 to 5. However, self-assessments of this kind are usually inaccurate because of the Dunning-Kruger effect: the less people know about a subject, the less accurate their estimate of their knowledge is.

That said, there *are* things we can do:

- Before running a workshop, communicate its level clearly to everyone who's thinking of signing up by listing the topics that will be covered and showing a few examples of exercises that people will be asked to complete.
- Provide multiple exercises for each teaching episode so that more advanced learners don't finish early and get bored.
- Ask more advanced learners to help people next to them. They'll learn from answering their peers' questions (since it will force them to think about things in new ways).
- The helpers and the instructor who isn't teaching the particular episode should keep an eye out for learners who are falling behind and intervene early so that they don't become frustrated and give up.

The most important thing is to accept that no class can possibly meet everyone's individual needs. If the instructor slows down to accommodate two people who are struggling, the other 38 are not being well served. Equally, if she spends a few minutes talking about an advanced topic because two learners are bored, the 38 who don't understand it will feel left out. All we can do is tell our learners what we're doing and why, and hope that they'll understand.

It's important to design lessons with a particular audience in mind. It's equally important to find out who's in each specific audience, since this will influence how you introduce yourself, motivate topics, and pace the lessons. Before the start of a Software Carpentry instructor training class, we ask people to fill in a short questionnaire like the one below. It doesn't tell us everything we might want to know, but it does give trainers a pretty clear idea of who they're speaking to.

1. Have you ever participated in a Software Carpentry workshop? (Check all that apply.)

    - Yes, as a learner.
    - Yes, as a helper.
    - Yes, as an organizer.
    - Yes, as an instructor.
    - No, but I am familiar with what is taught at a workshop.
    - No, and I am not familiar with what is taught at a workshop.
2. Which of these describes your teaching experience? (Check all that apply.)

    - I have none.
    - I have taught a seminar, workshop, or other short or informal course.
    - I have been a graduate or undergraduate teaching assistant for a college- or university-level course.
    - I have been the instructor-of-record for a college- or university-level course.
    - I have taught at the K-12 level.

3. Which of these describes your previous formal training in teaching? (Please choose only one.)
   - None
   - A few hours
   - A workshop
   - A certification or short course
   - A full degree
4. How frequently do you work with the tools that Software Carpentry teaches, such as R, Python, MATLAB, Perl, SQL, Git, and the Unix Shell?
   - Every day
   - A few times a week
   - A few times a month
   - A few times a year
   - Never or almost never
5. How often would you expect to teach a Software Carpentry workshop after this training?
   - Not at all
   - Once a year
   - Several times a year
6. Why do you want to take this training course?

Use Sticky Notes as Status Flags

Give each learner two sticky notes of different colours, e.g., red and green. These can be held up for voting, but their real use is as status flags. If someone has completed an exercise and wants it checked, they put the green sticky note on their laptop; if they run into a problem and need help, the put up the red one. This is better than having people raise their hands because:

- it's more discreet (which means they're more likely to actually do it),
- they can keep working while their flag is raised, and
- the instructor can quickly see from the front of the room what state the class is in.

Sometimes a red sticky involves a technical problem that takes a bit more time to solve. To prevent this issue from slowing down the whole class too much, you could use the occasion to take the small break you had planned to take a bit later, giving the helper(s) time to fix the problem.

Use Sticky Notes to Distribute Attention

Sticky notes can also be used to ensure that the instructor's attention is fairly distributed. Have each learner write their name on a sticky note and put it on their laptop. Each time the instructor calls on them or answers one of their questions, their sticky note comes down. Once all the sticky notes are down, everyone puts theirs up again.

This technique makes it easy for the instructor to see who they haven't spoken with recently, which in turn helps them avoid the unconscious trap of only interacting with the most extroverted of their learners. It also shows learners that attention is being distributed fairly, so that when they *are* called on, they won't feel like they're being picked on.

Never Touch the Learner's Keyboard

It's often tempting to fix things for learners, but when you do, it can easily seem like magic (even if you narrate every step). Instead, talk your learners through whatever they need to do. It will take longer, but it's more likely to stick.

Minute Cards

We frequently use sticky notes as *minute cards*: before each break, learners take a minute to write one positive thing on the green sticky note (e.g., one thing they've learned that they think will be useful), and one thing they found too fast, too slow, confusing, or irrelevant on the red one. They can use the red sticky note for questions that haven't yet been answered. While they are enjoying their coffee or lunch, the instructors review and cluster these to find patterns. It only takes a few minutes to see what learners are enjoying, what they still find confusing, what problems they're having, and what questions are still unanswered.

One Up, One Down

We frequently ask for summary feedback at the end of each day. The instructors ask the learners to alternately give one positive and one negative point about the day, without repeating anything that has already been said. This requirement forces people to say things they otherwise might not: once all the "safe" feedback has been given, participants will start saying what they really think.

Minute cards are anonymous; the alternating up-and-down feedback is not. Each mode has its strengths and weaknesses, and by providing both, we hope to get the best of both worlds.

Pair Programming

Pair programming is a software development practice in which two programmers share one computer. One person (called the driver) does the typing, while the other (called the navigator) offers comments and suggestions. The two switch roles several times per hour.

Pair programming is a good practice in real life, and also a good way to teach [Hannay2009] [Porter2013]. Partners can not only help each other out during the practical, but can also clarify each other's misconceptions when the solution is presented, and discuss common research interests during breaks. To facilitate this, we strongly prefer flat (dinner-style) seating to banked (theater-style) seating; this also makes it easier for helpers to reach learners who need assistance.

When pair programming is used it's important to put *everyone* in pairs, not just the learners who are struggling, so that no one feels singled out. It's also useful to have people sit in new places (and hence pair with different partners) after each coffee or meal break. It's also important to have people switch roles within each pair three or four times per hour, so that the stronger personality in each pair doesn't dominate the session[1].

> **Switching Partners**
>
> Instructors have mixed opinions on whether people should be required to change partners at regular intervals. On the one hand, it gives everyone a chance to gain new insights and make new friends. On the other, it is uncomfortable for introverts, and moving computers and power adapters to new desks several times a day is disruptive.

Have Learners Make Predictions

Research has shown that people learn more from demonstrations if they are asked to predict what's going to happen [Miller2013]. Doing this fits naturally into live coding: after adding or changing a few lines of a program, ask someone what is going to happen when it's run.

Collaborative Debugging

If you are live coding and your program doesn't work, explain the symptoms to your learners. The underlying cause often then becomes clear; if it doesn't, have them take turns suggesting things to try next. Be careful not to let one or two people dominate the discussion.

Peer Instruction

No matter how good a teacher is, she can only say one thing at a time. How then can she clear up many different misconceptions in a reasonable time?

The best solution developed so far is a technique called *peer instruction*. Originally created by Eric Mazur at Harvard, it has been studied extensively in a wide variety of contexts, including programming [Porter2013]. Peer instruction combines formative assessment with student discussion and looks something like this:

1. Give a brief introduction to the topic.

2. Give students an MCQ that probes for misconceptions (rather than simple factual knowledge).

3. Have all the students vote on their answers to the MCQ.

    i. If the students all have the right answer, move on.
    ii. If they all have the same wrong answer, address that specific misconception.
    iii. If they have a mix of right and wrong answers, give them several minutes to discuss those answers with one another in small groups (typically 2-4 students) and then reconvene and vote again.

As [this video][video-peer-instruction] shows, group discussion significantly improves students' understanding because it forces them to clarify their thinking, which can be enough to call out gaps in reasoning. Re-polling the class then lets the instructor know if they can move on, or if further explanation is necessary. A final round of additional explanation and discussion after the correct answer is presented gives students one more chance to solidify their understanding.

Peer instruction is essentially a way to provide one-to-one mentorship in a scalable way. Despite this, we usually do not use it in either programming workshops or instructor training workshops because it takes people time to adapt to a new way of learning–time that we typically don't have in our compressed two-day format.

> **Taking a Stand**
>
> Note that it is important to have learners record their votes so that they can't change their minds afterward and rationalize it by making excuses to themselves like "I just misread the question". Much of the value of peer instruction comes from the jarring experience of having their answer be wrong and having to think through the reasons why.

Setting Up Your Learners

Adult learners tell us that it is important to them to leave programming workshops with their own machine set up to do real work. We therefore strongly recommend that instructors be prepared to teach on all three major platforms (Linux, Mac OS, and Windows), even though it would be simpler to require learners to use just one.

To aid in this, put detailed setup instructions for all three platforms on the workshop's website, and email learners a couple of days before the workshop starts to remind them to do the setup. Even with this, a few people will always show up without the right software, either because their other commitments didn't allow them to go through the setup or because they ran into problems. To detect this, have everyone run some simple command as soon as they arrive and show the instructors the result, and then have helpers and other learners assist people who have run into trouble.

> **Common Denominators**
>
> If you have participants using several different operating systems, avoid using features which are OS-specific, and point out any that you *do* use. For example, some shell commands take different options on Mac OS than on Linux, while the "minimize window" controls and behavior on Windows are different from those on other platforms.

> **Virtual Machines**
>
> We have experimented with virtual machines (VMs) on learners' computers to reduce installation problems, but those introduce problems of their own: older or smaller machines simply aren't fast enough, and learners often struggle to switch back and forth between two different sets of keyboard shortcuts for things like copying and pasting.
>
> Some instructors use VPS over SSH or web browser pages instead. This solve the installation issues, but makes us dependent on host institutions' WiFi (which can be of highly variable quality), and has the issues mentioned above with things like keyboard shortcuts.

Setting Up Tables

You may not have any control over the layout of the desks or tables in the room in which your programming workshop takes place, but if you do, we find it's best to have:

- all tables on the same level (rather than banked seating),
- four people per table (so that each table can have two pairs if you choose to use pair programming), and
- in-floor power outlets (so that you don't have to run power cords across the floor that people can trip over).

Whatever layout you have, try to make sure the seats have good back support, since people are going to be in them for an extended period, and check that every seat has an unobstructed view of the screen.

Setting Up Your Own Environment

Setting up your environment is just as important as setting up your learners', but more involved. As well as having all the software that they need, and network access to the tool they're using to take notes, you should also have a glass of water, or a cup of tea or coffee. This helps keep your throat lubricated (as discussed in the next section), but its real purpose is to give you an excuse to pause for a couple of seconds and think when someone asks a hard question or you lose track of what you were going to say next.

You will probably also want some whiteboard pens and a few of the other things described in the travel kit checklist.

Cough Drops

If you talk all day to a room full of people, your throat gets raw because you are irritating the epithelial cells in your larynx and pharynx. This doesn't just make you hoarse–it also makes you more vulnerable to infection (which is part of the reason people often come down with colds after teaching).

The best way to protect yourself against this is to keep your throat lined, and the best way to do that is to use cough drops early and often. The right ones will also help delay the onset of coffee breath, for which your learners will probably be grateful.

Teaching Online

Many learners find it difficult to get to a workshop, either because there isn't one locally or because it's difficult to schedule time around other commitments, so why don't we create video recordings of the lessons and offer the workshop as a MOOC (Massive Open Online Course)?

The first answer is that we did in 2010-11, but found the maintenance costs unsustainable. Making a small change to this webpage only takes a few minutes. but making *any* change to a video takes an hour or more. In addition, most people are much less comfortable recording themselves than contributing written material.

The second answer is that doing outperforms watching. Specifically, a recent paper by Koedinger et al [Koedinger2015] estimated "…the learning benefit from extra doing (1 SD increase) to be more than six times that of extra watching or reading." "Doing", in this case, refers to completing an interactive or mimetic task with feedback, while "benefit" refers to both "completion rates" and "overall performance".

And while we do not (yet) have empirical data, we believe very strongly that many novices would give up in despair if required to debug setup and installation lessons on their own, but are more likely to get past these obstacles if someone is present to help them.

An intermediate approach that has proven quite successful is real-time remote instruction, in which the learners are co-located at one (or a few) sites, with helpers present, while the instructor(s) teaching via online video. This model has worked well for this instructor training course, and for a handful of regular workshops, but more work is needed to figure out its pros and cons.

Think-Pair-Share

Think-pair-share is a lightweight technique that helps refine their ideas and compare them with others'. Each person starts by thinking individually about a question or problem and jotting down a few notes. Participants are then paired to explain their ideas to each another, and possibly to merge them or select the more interesting ones. Finally, a few pairs present their ideas to the whole group.

Think-pair-share works because, to paraphrase Oscar Wilde's Lady Windermere, people often can't know what they're thinking until they've heard themselves say it. Pairing gives people new insight into their own thinking, and forces them to think through and resolve any gaps or contradictions *before* exposing their ideas to a larger group.

Challenges

Create a Questionnaire

Using the questionnaire shown earlier as a template, create a short questionnaire you could give learners before teaching a class of your own. What do you most want to know about their background?

20 minutes

Footnotes

> [1]. The Dunning-Kruger Effect can easily come into play in pair programming: whoever *thinks* they know the most can dominate the session regardless of how much they *actually* know. ↩

Live Coding

> **Objectives**
>
> - Learners can describe live coding and explain its advantages as a teaching practice for programming workshops.
> - Learners can enact and critique live coding.

Teaching is a performance art, just like drama, music, and athletics. And as in those fields, we have a collection of small tips and tricks to make teaching work better.

The first of our recommended teaching practices is so central that it deserves a chapter of its own: *live coding*. When they are live coding, instructors don't use slides. Instead, they go through the lesson material, typing in the code or instructions, with their learners following along. Its advantages are:

1. Watching a program being written is more compelling than watching someone page through slides that present bits and pieces of the same code.

2. It enables instructors to be more responsive to "what if?" questions. Where a slide deck is like a railway track, live coding allows instructors to go off road and follow their learners' interests.

3. It facilitates lateral knowledge transfer: people learn more than we realized we were teaching by watching *how* instructors do things.

4. It slows the instructor down: if she has to type in the program as she goes along, she can only go twice as fast as her learners, rather than ten-fold faster as she could with slides.

5. Learners get to see instructors' mistakes *and how to diagnose and correct them*. Novices are going to spend most of their time doing this, but it's left out of most textbooks.

6. Watching instructors make mistakes shows learners that it's all right to make mistakes of their. Most people model the behavior of their teachers: if the instructor isn't embarrassed about making and talking about mistakes, learners will be more comfortable doing so too.

Live coding does have some drawbacks, but with practice, these can be avoided or worked around:

1. Instructors can go too slowly, either because they are not good typists or by spending too much time looking at notes to try to remember what they meant to type.

2. Typing in boilerplate code that is needed by the lesson, but not directly relevant to it (such as library import statements) increases the extraneous cognitive load on your learners. Willingham says "Memory is the residue of thought" [Willingham2010], so if you spend your time typing boilerplate, that may be what learners will take away.

> *Teaching is theater not cinema.*
> – Neal Davis

Live coding is an example of the "I/We/You" approach to teaching discussed in Performance. It takes a bit of practice for instructors to get used to thinking aloud while coding in front of an audience, but most report that it is then no more difficult to do than talking off a deck of slides.

> **Double Devices**
>
> Many instructors now use two devices when teaching: a laptop plugged into the projector for learners to see, and a tablet beside it on which they can view their notes and the shared notes that the learners are taking. This seems to be more reliable than displaying one virtual desktop while flipping back and forth to another.

Here are some tips to make your live coding better:

1. *Be seen and heard.* If you are physically able to stand up for a couple of hours, do it while you are teaching. When you sit down, you are hiding yourself behind others for those sitting in the back rows. Make sure to notify the workshop organizers of your wish to stand up and ask them to arrange a high table, standing desk, or lectern.

   Regardless of whether you are standing or sitting, make sure to move around as much as reasonable. You can for example go to the screen to point something out, or draw something on the white/blackboard (see below). Moving around makes the teaching more lively, less monotonous. It draws the learners' attention away from their screens, to you, which helps get the point you are making across.

Even though you may have a good voice and know how to use it well, it may be a good idea to use a microphone, especially if the workshop room is equipped with one. Your voice will be less tired, and you increase the chance of people with hearing difficulties being able to follow the workshop.

2. *Take it slow.* For every command you type, every word of code you write, every menu item or website button you click, say out loud what you are doing while you do it, then point to the command and its output on the screen and go through it a second time. This not only slows you down, it allows learners who are following along to copy what you do, or to catch up, even when they are looking at their screen while doing it. Whatever you do, *don't* copy and paste code: doing this practically guarantees that you'll race ahead of your learners.

   If the output of your command or code makes what you just typed disappear from view, scroll back up so learners can see it again - this is especially needed for the Unix shell lesson. Other options are to execute the same command a second time, or to copy and paste the last command(s) into the workshop's shared notes.

3. *Mirror your learner's environment as much as possible.* You may have set up your environment to your liking, with a very simple or rather fancy Unix prompt, colour schemes for your development environment, keyboard shortcuts etc. Your learners usually won't have all of this. Try to create an environment that mirrors what your learners have, and avoid using keyboard shortcuts. Some instructors create a separate

`bare-bone' user (login) account on their laptop, or a separate` teaching-only' account on the service being taught (e.g., Github).

4. *Use the screen wisely.* You will need to enlarge your font considerably in order for people to read it from the back of the room, which means you can put much less on the screen than you're used to[1]. Maximize your window, and then ask everyone to give you a thumbs-up or thumbs-down on its readability. Use a black font on a lightly-tinted background rather than a light font on a dark background–the light tint will glare less than a pure white background.

   When the bottom of the projector screen is at the same height, or below, the heads of the learners, people in the back won't be able to see the lower parts. Draw up the bottom of your window(s) to compensate.

   Pay attention to the room lighting as well: it should not be fully dark, and there should be no lights directly on or above the presenter's screen. If needed, reposition the tables so all learners can see the screen.

   If you can get a second screen, use it: the extra screen real estate will allow you to display your code on one side and its output or behavior on the other. The second screen may require its own PC or laptop, so you may need to ask a helper to control it.

   > **Multiple Personalities**

> If you teach using a console window, such as a Unix shell, it's important to tell people when you run an in-console text editor and when you return to the console prompt. Most novices have never seen a window take on multiple personalities in this way, and can quickly become confused (particularly if the window is hosting an interactive interpreter prompt for Python or some other language as well as running shell commands and hosting an editor).

5. *Use illustrations.* Most lesson material comes with illustrations, and these may help learners to understand the stages of the lesson and to organize the material. What can work really well is when you as instructor generate the illustrations on the white/blackboard as you progress through the material. This allows you to build up diagrams, making them increasingly complex in parallel with the material you are teaching. It helps learners understand the material, makes for a more lively workshop (you'll have to move between your laptop and the blackboard) and gathers the learners' attention to you as well.

6. *Avoid distractions.* Turn off any notifications you may use on your laptop, such as those from social media, email, etc. Seeing notifications flash by on the screen distracts you as well as the learners - and may even result in awkward situations when a message pops up you'd rather not have others see.

7. *Improvise after you know the material.* The first time you teach a new lesson, you should stick fairly closely to the topics it lays out and the order they're in. It may be tempting to deviate from

the material because you would like to show a neat trick, or demonstrate some alternative way of doing something. Don't do this, since there is a fair chance you'll run into something unexpected that you then have to explain.

Once you are more familiar with the material, though, you can and should start improvising based on the backgrounds of your learners, their questions in class, and what you find most interesting about the lesson. This is like a musician playing a new song: the first few times, you stick to the sheet music, but after you're comfortable with it, you can start to put your own stamp on it.

If you really want to use something outside of the material, try it out thoroughly before the workshop: run through the lesson as you would during the actual teaching and test the effect of your modification.

Some instructors use printouts of the lesson material during teaching. Others use a second device (tablet or laptop) when teaching, on which they can view their own notes and the shared notes the learners are taking. This seems to be more reliable than displaying one virtual desktop while flipping back and forth to another.

8. *Embrace mistakes.*

   No matter how well prepared you are, you will be making mistakes. Typo's are hard to avoid, you may overlook something from the lesson instructions, etc. This is OK! It allows learners to see instructors' mistakes and how to diagnose and correct them. Some mistakes are actually an opportunity to point something out, or reflect back on something covered earlier. Novices are going to spend most of their time making the same and other mistakes, but how to deal with them is left out of most textbooks.

> *The typos are the pedagogy.*
> – Emily Jane McTavish

Note: if you've given a lesson several times, you're unlikely to make anything other than basic typing mistakes (which usually aren't informative). It's worth remembering "real" mistakes and making them deliberately, but that often feels forced. A better approach is to get learners to tell you what to do next in the hope that this will get you into the weeds.

9. It's OK to *face the screen occasionally*, particularly when you are walking through a section of code statement by statement or drawing a diagram, but you shouldn't do this for more than a few seconds at a time[2]. A good rule of thumb is to treat the screen as one of your learners: if it would be uncomfortable to stare at someone for as long as you are spending looking at the screen, it's time to turn around and face your audience.

10. *Have fun.* Teaching is performance art and can be rather serious business. On the one hand, don't let this scare you - it is much easier than performing Hamlet. You have an excellent script at your disposal, after all! On the other hand, it is OK to add an element of play, i.e. use humor and improvisation to liven up the workshop. How much you are able and willing to do this is really a matter of personality and taste - as well as experience. It becomes easier when you are more familiar with the material, allowing you to relax more. Choose your words

and actions wisely, though. Remember that we want the learners to have a welcoming experience and a positive learning environment - a misplaced joke can ruin this in an instant. Start small, even just saying `that was fun' after something worked well is a good start. Ask your co-instructors and helpers for feedback when you are unsure of the effect your behaviour has on the workshop.

Challenges

The Bad and the Good

Watch this video of live coding done poorly and then this video of live coding done well as a group and then summarize your feedback on both using the usual 2×2 grid. These videos assume learners know what a shell variable is, know how to use the `head` command, and are familiar with the contents of the data files being filtered.

20 minutes

See Then Do

Teach 3-4 minutes of a lesson using live coding to a fellow trainee, then swap and watch while that person live codes for you. Don't bother trying to record the live coding sessions–we have found that it's difficult to capture both the person and the screen with a handheld device–but give feedback the same way you have previously (positive and negative, content and presentation).

Explain in advance to your fellow trainee what you will be teaching and what the learners you teach it to are expected to be familiar with.

- What felt different about live coding (versus standing up and lecturing)? What was harder/easier?
- Did you make any mistakes? If so, how did you handle them?
- Did you talk and type at the same time, or alternate?
- How often did you point at the screen? How often did you highlight with the mouse?
- What will you try to do differently next time?

30 minutes

Footnotes

> [1]. You will often be reduced to 60-70 columns and 20-30 rows, which basically means that you're using a 21st Century supercomputer to emulate an early-1980s VT100 terminal. ↩

> [2]. Looking at the screen for a few seconds can help lower your anxiety levels, since it gives you a brief break from being looked at by all… those… eyes… ↩

Code of Conduct

> To make clear what is expected, everyone participating in this class is required to conform to the following Code of Conduct. This code of conduct applies to all spaces managed by our group including, but not limited to, workshops, mailing lists, and online forums (including source code repositories). Workshop hosts are expected to assist with enforcement of the Code of Conduct.
>
> If you believe someone is violating the Code of Conduct we ask that you report it to the course organizer and/or the course's hosts. All reports will be kept confidential.

We are dedicated to providing a welcoming and supportive environment for all people, regardless of background or identity. However, we recognize that some groups in our community are subject to historical and ongoing discrimination, and may be vulnerable or disadvantaged. Membership in such a specific group can be on the basis of characteristics such as such as gender, sexual orientation, disability, physical appearance, body size, race, nationality, sex, colour, ethnic or social origin, pregnancy, citizenship, familial status, veteran status, genetic information, religion or belief, political or any other opinion, membership of a national minority, property, birth, age, or choice of text editor. We do not tolerate harassment of participants on the basis of these categories, or for any other reason.

Harassment is any form of behaviour intended to exclude, intimidate, or cause discomfort. Because we are a diverse community, we may have different ways of communicating and of understanding the intent behind actions. Therefore we have chosen to prohibit certain forms of behaviour in our community, regardless of intent. Prohibited harassing behaviour includes but is not limited to:

- written or verbal comments which have the effect of excluding people on the basis of membership of a specific group listed above;
- causing someone to fear for their safety, such as through stalking, following, or intimidation;
- the display of sexual or violent images;
- unwelcome sexual attention;
- nonconsensual or unwelcome physical contact;
- sustained disruption of talks, events or communications;
- incitement to violence, suicide, or self-harm;
- continuing to initiate interaction (including photography or recording) with someone after being asked to stop; and
- publication of private communication without consent.

Behaviour not explicitly mentioned above may still constitute harassment. The list above should not be taken as exhaustive but rather as a guide to make it easier to enrich all of us and the communities in which we participate. All interactions should be professional regardless of location: harassment is prohibited whether it occurs on or offline, and the same standards apply to both.

Enforcement of the Code of Conduct will be respectful and not include any harassing behaviors.

Thank you for helping make this a welcoming, friendly community for all.

This code of conduct is a modified version of that used by PyCon, which in turn is forked from a template written by the Ada Initiative and hosted on the Geek Feminism Wiki.

License

The Rules

1. Be kind: all else is details.
2. Never teach alone.
3. No lesson survives first contact with learners.
4. Nobody will be more excited about the lesson than you are.
5. Every lesson is too short from the teacher's point of view and too long from the learner's.
6. Never hesitate to sacrifice truth for clarity.
7. Every mistake is a lesson.
8. "I learned this a long time ago" is not the same as "this is easy".
9. Ninety percent of magic consists of knowing one extra thing.
10. You can't help everyone, but you can always help someone.

Bibliography

What to Read Next

[Ambrose2010] Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, and Marie K. Norman: *How Learning Works: Seven Research-Based Principles for Smart Teaching* Jossey-Bass, 2010, 978-0470484104.
An excellent overview of what we know about education and why we believe it's true, covering everything from cognitive psychology to social factors.

[Brookfield2016] Stephen D. Brookfield and Stephen Preskill: *The Discussion Book: 50 Great Ways to Get People Talking* Jossey-Bass, 2016, 978-1119049715.
Describes fifty different ways to get groups talking productively.

[Green2014] E. Green: *Building a Better Teacher: How Teaching Works (and How to Teach It to Everyone)* W. W. Norton, 2014, 978-0393244151.
A well-written look at why educational reforms in the past 50 years have mostly missed the mark, and what we should be doing instead.

[Guzdial2015b] Mark Guzdial: *Learner-Centered Design of Computing Education: Research on Computing for Everyone*
Morgan & Claypool, 2015, 978-1627053518.

An evidence-based argument that we must design computing education for everyone, not just people who think they are going to become full-time professional programmers.

[Huston2009] Therese Huston: *Teaching What You Don't Know* Harvard University Press, 2009, 978-0674035805.
A pointed, funny, and very useful book that explores exactly what the title suggests.

[Lang2016] James M. Lang: *Small Teaching: Everyday Lessons from the Science of Learning* Jossey-Bass, 2016, 978-1118944493.
Presents a selection of accessible evidence-based practices that teachers can adopt when they little time and few resources.

[Lemon2014] Doug Lemov: *Teach Like a Champion 2.0: 62 Techniques that Put Students on the Path to College* (2nd edition). Jossey-Bass, 2014, 978-1118901854.
Presents 62 classroom techniques drawn from intensive study of thousands of hours of video of good teachers in action.

[Margolis2003] J. Margolis and A. Fisher: *Unlocking the Clubhouse: Women in Computing* MIT Press, 2003, 978-0262632690.
A groundbreaking report on the gender imbalance in computing, and the steps Carnegie-Mellon took to address the problem.

Other References

[Aiken1975] Edwin G. Aiken, Gary S. Thomas, and William A. Shennum: "Memory for a Lecture: Effects of Notes, Lecture Rate, and Informational Density." *Journal of Educational Psychology*, 67(3), June 1975, 10.1037/h0076613.
A landmark study showing that taking notes improves retention when learning.

[Aveling2013] Emma-Louise Aveling, Peter McCulloch, and Mary Dixon-Woods: "A Qualitative Study Comparing Experiences of the Surgical Safety Checklist in Hospitals in High-Income and Low-Income Countries." *BMJ Open*, 3(8), 2013, 10.1136/bmjopen-2013-003039.
Reports on surgical checklist implementations and effects in the UK and Africa.

[Barker2015] Lecia Barker, Christopher Lynnly Hovey, and Jane Gruning: "What Influences CS Faculty to Adopt Teaching Practices?", *Proc. 46th ACM Technical Symposium on Computer Science Education*, 2015, 10.1145/2676723.2677282.
Describes findings from a two-part study of how computer science educators adopt new teaching practices.

[Baume2009] D. Baume: "Writing and Using Good Learning Outcomes", Leeds Metropolitan University Press, 2009, 978-0-9560099-5-1.
A useful detailed guide to constructing useful learning outcomes.

[Benner2000] Patricia Benner: *From Novice to Expert: Excellence and Power in Clinical Nursing Practice* Pearson, 2000, 978-0130325228.
A classic study of clinical judgment and how expertise develops.

[Bohay2011] Mark Bohay, Daniel P. Blakely, Andrea K. Tamplin, and Gabriel A. Radvansky: "Note Taking, Review, Memory, and Comprehension." *American Journal of Psychology*, 124(1), 2011, 10.5406/amerjpsyc.124.1.0063.
Presents a study showing that note-taking improves retention most at deeper levels of understanding.

[Brown2014] Neil C. C. Brown and Amjad Altadmri: "Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data." *Proc Tenth Annual Conference on International Computing Education Research*, 2014, 10.1145/2632320.2632343.
Uses data from over 100,000 students to show that educators know less than they think about what mistakes novice programmers actually make.

[DeBruyckere2015] Pedro De Bruyckere, Paul A. Kirschner, and Casper D. Hulshof: *Urban Myths about Learning and Education* Academic Press, 2015, 978-0128015377.
Describes and debunks some widely-held myths about how people learn.

[Epstein2002] L.C. Epstein: *Thinking Physics is Gedanken Physics* Insight Press, 2002: 978-0935218084.
An entertaining problem-based introduction to thinking like a physicist.

[Fehily2008] Chris Fehily: *SQL: Visual QuickStart Guide* (3rd edition). Peachpit Press, 2008 978-0321553577.
An introduction to SQL that is both a good tutorial and a good reference guide.

[Fincher2012] Sally Fincher, Brad Richards, Janet Finlay, Helen Sharp, and Isobel Falconer: "Stories of Change: How Educators Change Their Practice." *Proc. Frontiers in Education Conference*, 2012, 10.1109/FIE.2012.6462317.
A detailed look at how educators actually adopt new teaching practices.

[Fincher2007] Sally Fincher and Josh Tenenberg: "Warren's Question." *Proc. Third International Workshop on Computing Education Research*, 2007, 10.1145/1288580.1288588.
A detailed look at a particular instance of transferring a teaching practice.

[Fink2013] L. Dee Fink: *Creating Significant Learning Experiences: An Integrated Approach to Designing College Courses* (2nd edition). Jossey-Bass, 2013, 978-1118124253.
A step-by-step guide to a systematic lesson design process.

[Gawande2011] Atul Gawande: "Personal Best." *The New Yorker*, October 3, 2011.
Describes how having a coach can improve practice in a wide variety of fields.

[Gelman2002] Andrew Gelman and Deborah Nolan: *Teaching Statistics: A Bag of Tricks* Oxford University Press, 2002, 978-0198572244.
A collection of useful motivating examples for teaching statistics.

[Gormally2014] Cara Gormally, Mara Evans, and Peggy Brickman: "Feedback about Teaching in Higher Ed: Neglected Opportunities to Promote Change." *CBE Life Sciences Education*, 13(2), 2014,

10.1187/cbe.13-12-0235.
Summarizes the best practices for providing instructional feedback, and recommends specific strategies for providing feedback.

[Guzdial2017] Mark Guzdial: "Computing Education Blog". https://computinged.wordpress.com/, viewed April 2017.
An informative, frequently-updated blog about computing education.

[Guzdial2013] Mark Guzdial: "Exploring Hypotheses About Media Computation." *Proc. Ninth Annual International ACM Conference on International Computing Education Research*, 2013, 10.1145/2493394.2493397.
A look back on 10 years of media computation research.

[Guzdial2015a] Mark Guzdial: "Top 10 Myths About Teaching Computer Science". http://cacm.acm.org/blogs/blog-cacm/189498-top-10-myths-about-teaching-computer-science/fulltext, 2015.
Ten things many people believe that aren't true.

[Hannay2009] Jo E. Hannay, Tore Dybå, Erik Arisholm, and Dag I.K. Sjøberg: "The Effectiveness of Pair Programming: A Meta-Analysis." *Information and Software Technology*, 51(7), 2009, 10.1016/j.infsof.2009.02.001.
A summary of research on the effectiveness of pair programming.

[Henderson2011] Charles Henderson, Andrea Beach, and Noah Finkelstein: "Facilitating Change in Undergraduate STEM Instructional Practices: An Analytic Review of the Literature." *Journal of Research in Science Teaching*, 48(8), 2011, 10.1002/tea.20439.

Describes eight approaches to effecting change in STEM education that form a useful framework for thinking about how free-range workshops can go mainstream.

[Kernighan1982] Brian W. Kernighan and P.J. Plauger: *The Elements of Programming Style* (2nd edition). McGraw-Hill, 1982, 978-0070342071.
An early and influential description of the Unix programming philosophy.

[Kernighan1984] Brian W. Kernighan and Rob Pike: *The UNIX Programming Environment* Prentice Hall, 1984. 978-0139376818.
An influential early description of Unix.

[Kernighan1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language* (2nd edition). Prentice Hall, 1988. 978-0131103709.
The book that made C a popular programming language.

[Kirschner2006] Paul Kirschner, John Sweller, and Richard Clark: "Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching." *Educational Psychologist*, 41(2), 2006.
Argues that inquiry-based learning is less effective for novices than guided instruction.

[Koedinger2015] Kenneth R. Koedinger, Jihee Kim, Julianna Zhuxin Jia, Elizabeth A. McLaughlin, and Norman L. Bier: "Learning is Not a Spectator Sport: Doing is Better Than Watching

for Learning from a MOOC" *Proc. Second ACM Conference on Learning @ Scale*, 2015, 10.1145/2724660.2724681.
Measures the benefits of doing rather than watching.

[Marja Kuittinen and Jorma Sajaniemi:] Marja Kuittinen and Jorma Sajaniemi: "Teaching Roles of Variables in Elementary Programming Courses" *Proc. 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2004, 10.1145/1007996.1008014.
Presents a few patterns used in novice programming and looks at the pedagogical value of teaching them.

[Lee2017] Cynthia Lee: "What Can I Do Today to Create a More Inclusive Community in CS?" https://docs.google.com/document/d/1hLivou9-_wmsZuzKI2pCGQu0KHVIgYfJSaYhvTgO0Wo/, viewed April 2017.
A practical checklist of things instructors can do to make their computing classes more inclusive.

[Littky2004] D. Littky and S. Grabelle: *The Big Picture: Education is Everyone's Business* Association for Supervision and Curriculum Development, 2004, 978-0871209719.
A personal exploration of the purpose of education and how to make schools better.

[Macnamara2014] Brooke N. Macnamara, David Z. Hambrick, and Frederick L. Oswald: "Deliberate Practice and Performance in Music, Games, Sports, Education, and Professions." *Psychological Science*, 25(8), 2014, 10.1177/0956797614535810.
A meta-study of the effectiveness of deliberate practice.

[Margolis2010] J. Margolis, R. Estrella, J. Goode, J.J. Holme, and K. Nao: *Stuck in the Shallow End: Education, Race, and Computing* MIT Press, 2010, 978-0262260961.
A hard-hitting look at racial inequities in computing education.

[Marsh2002] Herbert W. Marsh and John Hattie: "The Relation Between Research Productivity and Teaching Effectiveness." *Journal of Higher Education*, 73(5), 2002.
One study of many showing there is zero correlation between research ability and teaching effectiveness.

[Mayer2003] Richard E. Mayer and Roxana Moreno: "Nine Ways to Reduce Cognitive Load in Multimedia Learning." *Educational Psychologist*, 38, 2003.
Shows how research into how we absorb and process information can be applied to the design of instructional materials.

[Miller2013] Kelly Miller, Nathaniel Lasry, Kelvin Chu, and Eric Mazur: "Role of Physics Lecture Demonstrations in Conceptual Learning." *Physical Review Physics Education Research*, 9(2), 2013, 10.1103/PhysRevSTPER.9.020113.
Reports a detailed study of what students learn during demonstrations and why.

[Mueller2014] Pam A. Mueller and Daniel M. Oppenheimer: "The Pen Is Mightier Than the Keyboard." *Psychological Science*, 25(6), 2014, 10.1177/0956797614524581.
Presents evidence that taking notes by hand is more effective than taking notes on a laptop.

[Orndorff2015] Harold N. Orndorff III: "Collaborative Note-Taking: The Impact of Cloud Computing on Classroom Performance." *International Journal of Teaching and Learning in Higher Education*, 27(3), 2015.
Presents a study showing that collaborative note-taking improves grades and learning outcomes.

[Patitsas2016] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook: "Evidence That Computer Science Grades Are Not Bimodal." *Proc. 2016 ACM Conference on International Computing Education Research*, 2016, 10.1145/2960310.2960312.
Presents a statistical analysis and an experiment which jointly show that despite popular myth, grades in computing classes are not bimodal–i.e., there is no geek gene.

[Petre2016] Marian Petre, André van der Hoek, and Yen Quach: *Software Design Decoded: 66 Ways Experts Think* MIT Press, 2016, 978-0262035187.
A short illustrated overview of how expert software developers think.

[Porter2013] Leo Porter, Mark Guzdial, Charlie McDowell, and Beth Simon: "Success in Introductory Programming: What Works?" *Communications of the ACM*, 56(8), August 2013, 10.1145/2492007.2492020.
Summarizes the evidence that three techniques–peer instruction, media computation, and pair programming–can significantly improve outcomes in introductory programming courses.

[Ray2014] Eric J. Ray and Deborah S. Ray: *Unix and Linux: Visual QuickStart Guide* (5th edition). Peachpit Press, 2014, 978-0321997548.

An introduction to Unix that is both a good tutorial and a good reference guide.

[Schon1984] Donald A. Schön: *The Reflective Practitioner: How Professionals Think In Action* Basic Books, 1984, 978-0465068784.
A groundbreaking look at how professionals in different fields actually solve problems.

[Scott1999] J. C. Scott: *Seeing Like a State: How Certain Schemes to Improve the Human Condition Have Failed: Yale University Press, 1999, 978-0300128789.*
Presents a compelling case that centrally-managed, top-down ventures will always be less effective than grassroots alternatives because large organizations consistently prefer uniformity over productivity.*

[Steele2011] C. M. Steele: *Whistling Vivaldi: And Other Clues to How Stereotypes Affect Us* W. W. Norton, 2011, 978-0393341485.
Explains and explores stereotype threat and strategies for addressing it.

[Ubell2017] Robert Ubell: "How the Pioneers of the MOOC Got It Wrong *http://spectrum.ieee.org/tech-talk/at-work/education/how-the-pioneers-of-the-mooc-got-it-wrong, January 16, 2017.*
A brief exploration of why MOOCs haven't lived up to claims made for them five years ago.*

[Urbach2014] David R. Urbach, Anand Govindarajan, Refik Saskin, Andrew S. Wilton, and Nancy N. Baxter: "Introduction of Surgical Safety Checklists in Ontario, Canada." *New England Journal of*

Medicine, 370(11), 2014, 10.1056/NEJMsa1308261.
Reports a study showing that the introduction of surgical checklists did not have a significant effect on operative outcomes.

[Uttl2016] Bob Uttl, Carmela A. White, and Daniela Wong Gonzalez: "Meta-Analysis of Faculty's Teaching Effectiveness: Student Evaluation of Teaching Ratings and Student Learning Are Not Related" *Studies in Educational Evaluation*, 2016, 10.1016/j.stueduc.2016.08.007
A summary of studies shown that how students rate a course and how much they actually learn are not related.

[Wiggins2005] G.P. Wiggins and J. McTighe: *Understanding by Design: Association for Supervision and Curriculum Development, 2005, 978-1416600350.*
A lengthy presentation of reverse instructional design.*

[Wilkinson2011] Richard Wilkinson and Kate Pickett: *The Spirit Level: Why Greater Equality Makes Societies Stronger* Bloomsbury Press, 2011, 978-1608193417.
Presents evidence that inequality harms everyone, both economically and otherwise.

[Willingham2010] Daniel T. Willingham: *Why Don't Students Like School?* Jossey-Bass, 2010, 978-0470591963.
A cognitive scientist looks at how the mind works and what it means in the classroom.

Glossary

Authentic Task: A task which contains important elements of things that learners would do in real (non-classroom situations). To be authentic, a task should require learners to construct their own answers rather than choose between provided answers, and to work with the same tools and data they would use in real life.

Behaviorism: A theory of learning whose central principle is stimulus and response, and whose goal is to explain behavior without recourse to internal mental states or other unobservables. See also cognitivism.

Chunking: The act of grouping related concepts together so that they can be stored and processed as a single unit.

Cognitive Load Theory: Cognitive load is the amount of mental effort required to solve a problem. Cognitive load theory divides this effort into *intrinsic*, *extraneous*, and *germane*, and holds that people learn faster and better when extraneous load is reduced.

Cognitivism: A theory of learning that holds that mental states and processes can and must be included in models of learning.
See also behaviorism.

Community of Practice: A self-perpetuating group of people who share and develop a craft or occupation, such as knitters, musicians, or programmers.
See also legitimate peripheral participation.

Competent Practitioner: Someone who can do normal tasks with normal effort under normal circumstances.
See also novice and expert.

Concept Map: A picture of a mental model in which concepts are nodes in a graph and relationships are (labelled) arcs.

Connectivism: A theory of learning which emphasizes its social aspects, particularly as enabled by the Internet and other technologies.

Constructivism: A theory of learning that views learners as actively constructing knowledge.

Content Knowledge: A person's understanding of a subject.
See also general pedagogical knowledge and pedagogical content knowledge.

Deliberate Practice: The act of observing performance of a task while doing it in order to improve ability.

Diagnostic Power: The degree to which a wrong answer to a question or exercise tells the instructor what misconceptions a particular learner has.

Educational Psychology: The study of how people learn.
See also instructional design.

Expert: Someone who can diagnose and handle unusual situations, knows when the usual rules do not apply, and tends to recognize solutions rather than reasoning to them.
See also competent practitioner and novice.

Expert Blind Spot: The inability of experts to empathize with novices who are encountering concepts or practices for the first time.

Externalized Cognition: The use of graphical, physical, or verbal aids to augment thinking.

Faded Example: A series of examples in which a steadily increasing number of key steps are blanked out.
See also scaffolding.

Fixed Mindset: The belief that an ability is innate, and that failure is due to a lack of some necessary attribute.
See also growth mindset.

Fluid Representation: The ability to move quickly between different models of a problem.

Formative Assessment: Assessment that takes place during a lesson in order to give both the learner and the instructor feedback on actual understanding.
See also summative assessment.

General Pedagogical Knowledge: A person's understanding of the general principles of teaching.
See also content knowledge and pedagogical content knowledge.

Growth Mindset: The belief that ability comes with practice.
See also fixed mindset.

Impostor Syndrome: A feeling of insecurity about one's accomplishments that manifests as a fear of being exposed as a fraud.

Inclusivity: Working actively to include people with diverse backgrounds and needs.

Inquiry-Based Learning: The practice of allowing learners to ask their own questions, set their own goals, and find their own path through a subject.

Instructional Design: The craft of creating and evaluating specific lessons for specific audiences.
See also educational psychology.

Jugyokenkyu: Literally "lesson study", a set of practices that includes having teachers routinely observe one another and discuss lessons to share knowledge and improve skills.

Lateral Knowledge Transfer: The "accidental" transfer of knowledge that occurs when an instructor is teaching one thing, and the learner picks up another.

Learner Profile: A brief description of a typical target learner for a lesson that includes their general background, what they already know, what they want to do, how the lesson will help them, and any special needs they might have.

Learning Objective: What a lesson is trying to achieve.

Learning Outcome: What a lesson actually achieves.

Legitimate Peripheral Participation: Newcomers' participation in simple, low-risk tasks that a community of practice recognizes as valid contributions.

Live Coding: The act of teaching programming by writing software in front of learners as the lesson progresses.

Long-Term Memory: The part of memory that stores information for long periods of time. Long-term memory is very large, but slow. See also short-term memory.

Minute Cards: A feedback technique in which learners spend a minute writing one positive thing about a lesson (e.g., one thing they've learned) and one negative thing (e.g., a question that still hasn't been answered).

Novice: Someone who has not yet built a usable mental model of a domain.
See also competent practitioner and expert.

Pair Programming: A software development practice in which two programmers share one computer. One programmer (the driver) does the typing, while the other (the navigator) offers comments and suggestions in real time. Pair programming is often used as a teaching practice in programming classes.

Parson's Problem: An assessment technique developed by Dale Parsons and others in which learners rearrange given material to construct a correct answer to a question.

Pedagogical Content Knowledge (PCK): The understanding of how to teach a particular subject, i.e., the best order in which to introduce topics and what examples to use.
See also content knowledge and general pedagogical knowledge.

Peer Instruction: A teaching method in which an instructor poses a question and then students commit to a first answer, discuss answers with their peers, and commit to a (revised) answer.

Persistent Memory: see long-term memory.

Plausible Distractor: A wrong answer to a multiple-choice question that looks like it could be right.
See also diagnostic power.

Reflective Practice: see deliberate practice.

Reverse Instructional Design: An instructional design method that works backwards from a summative assessment to formative assessments and thence to lesson content.

Scaffolding: Extra material provided to early-stage learners to help them solve problems.

Short-Term Memory: The part of memory that briefly stores information that can be directly accessed by consciousness.

Situated Learning: A model of learning that focuses on people's transition from being newcomers to be accepted members of a community of practice.

Stereotype Threat: A situation in which people feel that they are at risk of being held to stereotypes of their social group.

Summative Assessment: Assessment that takes place at the end of a lesson to tell whether the desired learning has taken place.

Tangible Artifact: Something a learner can work on whose state gives feedback about the learner's progress and helps the learner diagnose mistakes.

Test-Driven Development: A software development practice in which programmers write tests first in order to give themselves concrete goals and clarify their understanding of what "done" looks like.

Understanding by Design: see reverse instructional design.

Working Memory: see short-term memory.

Contributing

1. Read the GitBook documentation.
2. Create a branch in your desktop Git repository.
3. Edit the Markdown files in this directory and its subdirectories.
4. Commit your changes.
5. Push that branch to your repository on GitHub.
6. Submit a pull request from your branch in your repository to the master repository.

Checklists

Atul Gawande's 2007 article "The Checklist" popularized the idea that using checklists can save lives (and make many other things better too). The results of recent studies have been more nuanced [Aveling2013] [Urbach2014], but we still find them useful, particularly when bringing new instructors onto a team.

The checklists below are used before, during, and after instructor training events, and can easily be adapted for end-learner workshops as well. We recommend that every group build and maintain its own checklists customized for its instructors' and learners' needs.

Scheduling the Event

1. Decide if it will be in person, online for one site, or online for several sites.
2. Talk through expectations with the host(s) and make sure that everyone agrees on who is covering travel costs.
3. Determine who is allowed to take part: is the event open to all comers, restricted to members of one organization, or something in between?
4. Arrange trainers.
5. Arrange space. Make sure there are breakout rooms for video recording.

6. Choose dates. If it is in person, book travel.
7. Get names and email addresses of attendees from host(s).
8. Make sure people are added to whatever registration system is being used.

Setting Up

1. Set up a web page with details on the workshop, including date, location, and a list of what participants are expected to bring.
2. Check whether any attendees have special needs.
3. If the workshop is online, test the video conferencing link.
4. Make sure attendees will all have network access.
5. Create an Etherpad or Google Doc for shared notes.
6. Email attendees a welcome message that includes:
    - a link to the workshop home page
    - background readings
    - a description of any pre-requisite tasks

At the Start of the Event

1. Remind everyone of the code of conduct.
2. Collect attendance.
3. Distribute sticky notes.

4. Collect relevant online account IDs (e.g., GitHub IDs).

At the End of the Event

1. Update attendance records. Be sure to also record who participated as an instructor or helper.
2. Administer a post-workshop survey.
3. Update the course notes and/or checklists.

Travel Kit

Here are a few things instructors take with them when they travel to teach:

- sticky notes
- cough drops
- good shoes (because you'll be standing most of the day)
- a spare power adapter
- a spare shirt
- deodorant
- a variety of video adapters
- laptop stickers
- a toothbrush or some mouthwash
- a granola bar or some other emergency snack
- Eno or some other antacid (because road food)
- business cards
- a printed copy of notes for reference during teaching (or a tablet or other "extra" device)

- an insulated cup for tea/coffee
- spare glasses/contacts
- a notebook and pen
- a portable WiFi hub (in case networking in the room isn't working)
- extra whiteboard markers
- a laser pointer
- a packet of wet wipes (because spills happen)
- a few USB drives with installers for various operating systems (just in case)
- running shoes, a bathing suit, a yoga mat, or whatever else you exercise in or with
- a small notepad for writing down snippets of code for participants

Extra Material

This appendix includes material that doesn't naturally fit anywhere else.

Key Terms

Educational psychology is the study of how people learn. It touches on everything from the neuropsychology of perception and the mechanisms of memory to the sociology of school systems and the philosophical question of what we actually mean by "learning" (which turns out to be pretty complicated once you start looking beyond the standardized Western classroom). Within the broad scope of educational psychology, two specific perspectives have primarily influenced our teaching practices (and by extension, this instructor training).

The first perspective is *cognitivism*, which treats learning as a problem in neuropsychology. Cognitivists focus their attention on things like pattern recognition, memory formation, and recall. It is good at answering low-level questions, but generally ignores larger issues like, "What do we mean by 'learning'?" and, "Who gets to decide?"

The second perspective is *situated learning*, which focuses on how *legitimate peripheral participation* leads to people becoming members of a *[community of practice][gloss.md#community-of-practice]*. Unpacking those terms, the situated learning perspective focuses on the transition from being a newcomer to being accepted as a peer by those who already do the activity in question. Situated learning is directly relevant to our learners, many of whom ease into scientific computing by doing small tasks that experienced practitioners would regard as straightforward, but who learn how to take on bigger and more novel challenges both from what they do and from the feedback (and welcome) it elicits. It is equally relevant to our instructors (i.e., you), who are approaching evidence-based teaching in the same way.

For example, Software Carpentry aims to serve researchers who are exploring data management and programming on their own (legitimate peripheral practice) and make them aware of other people doing that work (simply by attending the workshop) and the best practices and ideas of that community of practice, thereby giving them a way to become members of that community. Situated learning thus describes why we teach, and recognizes that teaching and learning is necessarily rooted in a social context. We then depend on the cognitivist perspective to drive *how* we teach the specific content associated with the community of practice.

> **Other Perspectives**
>
> There are many other perspectives outside cognitivist theory—see this site for summaries. Besides cognitivism, those encountered most frequently include *behaviorism* (which treats education as stimulus/response conditioning), *constructivism* (which considers learning an active process during which learners construct knowledge for themselves),

> and *connectivism* (which emphasizes the social aspects of learning, particularly those made possible by the Internet). And yes, it would help if their names were less similar…

Educational psychology does not tell us how to teach on its own because it under-constrains the problem: in real life, several different teaching methods might be consistent with what we currently know about how learning works. We therefore have to try those methods in the class, with actual learners, in order to find out how well they balance the different forces in play.

Doing this is called *instructional design*. If educational psychology is the science, instructional design is the engineering. For example, there are good reasons to believe that children will learn how to read best by starting with the sounds of letters and working up to words. However, there are equally good reasons to believe that children will learn best if they are taught to recognize entire simple words like "open" and "stop", so that they can start using their knowledge sooner[1]. The only way to tell which approach works best for most children, most of the time, is to try them both out[2].

Unsurprisingly, effective teaching depends on what the teacher knows, which can be divided into:

- *content knowledge*, such as the "what" of programming;
- *general pedagogical knowledge*, i.e., an understanding of the psychology of learning; and
- the *pedagogical content knowledge* (PCK) that connects the two. PCK is things like what examples to use when teaching how parameters are passed to a function, or what misconceptions about wildcard expansion are most common.

> For example, an instructor could write variable names and values on paper plates and then stack and unstack them to show how the call stack works.

A great example of PCK is Gelman and Nolan's *[Teaching Statistics: A Bag of Tricks]* [Gelman2002], which is full of PCK for teaching introductory statistics. The CS Teaching Tips site is gathering similar ideas for computing.

Myths

One well-known scheme characterizes learners as visual, auditory, or kinesthetic according to whether they like to see things, hear things, or do things. This scheme is easy to understand, but as de Bruyckere and colleagues point out in *Urban Myths About Learning and Education* [DeBruyckere2015], it is almost certainly false. Unfortunately, that hasn't stopped a large number of companies from marketing products based on it to parents and school boards.

This is not the only myth to plague education. The learning pyramid that shows we remember 10% of what we read, 20% of what we hear, and so on? Myth. The idea that "brain games" can improve our intelligence, or at least slow its decline in old age? Also a myth, as are the claims that the Internet is making us dumber or that young people read less than they used to.

Computing education has its own myths. Mark Guzdial's "Top 10 Myths About Teaching Computer Science" [Guzdial2015a] are:

1. The lack of women in Computer Science is just like all the other STEM fields.

2. To get more women in CS, we need more female CS faculty.
3. A good CS teacher is a good lecturer.
4. Clickers and the like are an add-on for a good teacher.
5. Student evaluations are the best way to evaluate teaching.
6. Good teachers personalize education for students' learning styles.
7. High schools just can't teach CS well, so they shouldn't do it at all.
8. The real problem is to get more CS curriculum into the hands of teachers.
9. All I need to do to be a good CS teacher is model good software development practice, because my job is to produce excellent software engineers.
10. Some people are just born to program.

The last of these is the most pervasive and most damaging. As discussed in Motivation, Elizabeth Patitsas and others have shown that grades in computing classes are *not* bimodal [Patitsas2016], i.e., there isn't one group that gets it and another that doesn't. Many of the participants in our workshops have advanced degrees in intellectually demanding subjects, but have convinced themselves that they just don't have what it takes to be programmers. If all we do is dispel that belief, we will have done them a service.

Feedback on Live Coding Demo Videos

The two lists below summarize key feedback on the two videos used in the discussion of live coding.

Part 1: How Not to Do It

- Instructor ignores a red sticky clearly visible on a learner's laptop.
- Instructor is sitting, mostly looking at the laptop screen.
- Instructor is typing commands without saying them out loud.
- Instructor uses fancy shell prompt in the console window.
- Instructor uses small font in not full-screen console window with black background.
- The console window bottom is partially blocked by the learner's heads for those sitting in the back.
- Instructor receives a a pop-up notification in the middle of the session.
- Instructor makes a mistake (a typo) but simply fixes it without pointing it out, and redoes the command.

Part 2: How to Do It Right

- Instructor checks if the learner with the red sticky on her laptop still needs attention.

- Instructor is standing while instructing, making eye-contact with participants.
- Instructor is saying the commands out loud while typing them.
- Instructor moves to the screen to point out details of commands or results.
- Instructor simply uses `$` as shell prompt in the console window.
- Instructor uses big font in wide-screen console window with white background.
- The console window bottom is above the learner's heads for those sitting in the back.
- Instructor makes mistake (a typo) and uses the occasion to illustrate how to interpret error-messages.

Effecting Change

This guide is aimed primarily at people working outside mainstream educational institutions, but in order for us to reach and help as many people as possible, we must eventually find ways to work with schools as they are. Henderson et al's "Facilitating Change in Undergraduate STEM Instructional Practices" [Henderson2011] discusses ways to get educational institutions to actually change what they teach. The approaches they identify include:

- *Diffusion*: STEM undergraduate instruction will be changed by altering the behavior of a large number of individual instructors. The greatest influences for changing instructor behavior lie in optimizing characteristics of the innovation and exploiting the characteristics of individuals and their networks.
- *Implementation*: STEM undergraduate instruction will be changed by developing research-based instructional "best practices" and training instructors to use them. Instructors must use these practices with fidelity to the established standard.
- *Scholarly Teaching*: STEM undergraduate instruction will be changed when more individual faculty members treat their teaching as a scholarly activity.
- *Faculty Learning Communities*: STEM undergraduate instruction will be changed by groups of instructors who support and sustain each other's interest, learning, and reflection on their teaching.
- *Quality Assurance*: STEM undergraduate instruction will be changed by requiring institutions (colleges, schools, departments, and degree programs) to collect evidence demonstrating their success in undergraduate instruction. What gets measured is what gets improved.
- *Organizational Development*: STEM undergraduate instruction will be changed by administrators with strong vision who can develop structures and motivate faculty to adopt improved instructional practices.

- *Learning Organizations*: Innovation in higher education STEM instruction will occur through informal communities of practice within formal organizations in which individuals develop new organizational knowledge through sharing implicit knowledge about their teaching. Leaders cultivate conditions for both formal and informal communities to form and thrive.
- *Complexity Leadership*: STEM undergraduate instruction is governed by a complex system. Innovation will occur through the collective action of self-organizing groups within the system. This collective action can be stimulated, but not controlled.

Evaluating Impact

A key part of effecting change is to convince people that what you're doing is actually having a positive impact. That turns out to be surprisingly hard[3] for free-range programming workshops:

1. *Ask learners if the workshop was useful.* Study after study has shown that there is no correlation between how highly learners rate a course and how much they actually learn [Uttl2016], and most people working in education are now aware of that.
2. *Give them an exam at the end of the workshop.* Doing that dramatically changes the feel of the workshop, and how much they know at the end of the day is a poor predictor of how much they will remember two or three months later.

3. *Give them an exam two or three months later.* That's hard enough to do in a traditional battery-farmed learning environment; doing it with free-range learners is even harder. In addition:

    i. The people who didn't get anything out of the workshop are probably less likely to take part in follow-up, so feedback gathered this way will be subject to self-selection bias.

    ii. The fact that learners *remember* something doesn't necessarily mean it was useful (although they are more likely to remember things that are useful than things that aren't).

4. *See if they keep using what they learned.* This is a good way to evaluate employment-oriented skills, but equally useful for things people have learned for fun. The problem is how to do it: you probably shouldn't put spyware on their computers, and follow-up surveys suffer from the same low return rate and self-selection bias as exams.

5. *See if they recommend the workshop to friends.* This method often strikes the best balance between informative and doable: if people are recommending your workshop to other people, that's a pretty good sign.

There are many other options; the most important thing is to figure out early on how you're going to know whether you're teaching the right things the right way, and how you're going to convince potential backers that you're doing so.

Footnotes

> [1]. The first approach is called "phonics", and the second, "whole language". The whole language approach may seem nonsensical, but more than a billion people have learned to read and write Chinese and similar ideogrammatic languages in exactly this way. ↩

> [2]. And to control other variables, because in practice, the teacher's enthusiasm for the teaching method may have more of an impact than the method itself, since children will model their teacher's excitement (or lack thereof) for a subject. ↩

> [3]. Well, it surprised me… ↩

Why I Teach

When I first started volunteering at the University of Toronto, students occasionally asked me why. This was my answer:

> When I was your age, I thought universities existed to teach people how to learn. Later, in grad school, I thought universities were about doing research and creating new knowledge. Now that I'm in my forties, though, I've realized that what we're really teaching you is how to take over the world, because you're going to have to one day whether you like it or not.
>
> My parents are in their seventies. They don't run the world any more; it's people my age who pass laws, set interest rates, and make life-and-death decisions in hospitals. As scary as it may be, *we've* become the grownups.
>
> Twenty years from now, though, we'll be heading for retirement and *you* will be in charge, because there won't be anyone else to do it. That may sound like a long time when you're nineteen, but let me tell you, take three breaths and it's gone. That's why we give you problems whose answers can't be cribbed from last year's notes. That's why we put you in situations where you have to figure out what needs to be done right now, what can be left for later, and what you can simply ignore. It's because if you don't start learning how to do these things now, you won't be ready to do them when you have to.

It was stirring stuff, but it wasn't the whole story. I didn't just want my students to make the world a better place so that I could retire in comfort. I want them to make it a better place because it's the great adventure of our time. Just think: a hundred and fifty years ago, most societies still practiced slavery. A hundred years ago, when my grandmother was young, she wasn't legally a person in Canada. Fifty years ago, most of the world's people suffered under totalitarian rule; in the year I was born, judges could---and did---order electroshock therapy to "cure" homosexuals. Yes, there's still a lot wrong with the world, but look at how many more choices we have than our grandparents did. Look at how many more things we can know, and be, and enjoy. And most importantly, look at how many other people can too.

This didn't happen by chance. It happened because millions of us made millions of little decisions, the sum of which was a better world. We don't think of these day-to-day decisions as political, but every time we buy one brand of running shoe instead of another or shout an anatomical insult instead of a racial one at a cab driver, we're choosing one vision of the world instead of another. We don't think about this bigger picture most of the time, but the last century and a half shows that once enough people make "doing the right thing" a habit, the big picture more or less takes care of itself.

At least, that's what I believed in 2004. I am less optimistic today. One upside of nuclear apocalypse from a preventive point of view was that the people pushing the buttons would pay the price. The bill for my generation's cowardice, lethargy, and greed, on the other hand, won't come due 'til my daughter is grown, by which point there will be no easy solutions (and quite possibly no solutions at all). The only things that will save us then will be more science and more courage, both of which need to be taught and practiced. Insisting on evidence, and then *acting* on it when the cost is immediate and the reward delayed, requires as great a social change as emancipation or universal suffrage.

In his 1947 essay "Why I Write", George Orwell said:

> In a peaceful age I might have written ornate or merely descriptive books, and might have remained almost unaware of my political loyalties. As it is I have been forced into becoming a sort of pamphleteer… Every line of serious work that I have written since 1936 has been written, directly or indirectly, against totalitarianism… It seems to me nonsense, in a period like our own, to think that one can avoid writing of such subjects. Everyone writes of them in one guise or another. It is simply a question of which side one takes…

Replace "writing" with "teaching" and you'll have the reason I've done what I've done for the last thirteen years. The world doesn't get better on its own. It gets better because people *make* it better: penny by penny, vote by vote, and one lesson at a time.