Teaching Tech Together

How to create and deliver lessons that work
nd build a teaching community around them

Compiled by Greg Wilson

ISBN 978-0-9881137-0-1

For my mother, Doris Wilson,
who taught hundreds of children to read and to believe in themselves.

And for my brother Jeff, who did not live to see it finished.
"Remember, you still have a lot of good times in front of you."

The Rules

1. Be kind: all else is details.
2. Remember that you are not your learners...
3. ...that most people would rather fail than change...
4. ...and that ninety percent of magic consists of knowing one extra thing.
5. Never teach alone.
6. Never hesitate to sacrifice truth for clarity.
7. Make every mistake a lesson.
8. Remember that no lesson survives first contact with learners...
9. ...that every lesson is too short from the teacher's point of view and too long from the learner's...
10. ...and that nobody will be more excited about the lesson than you are.

Contents

List of Figures

Teaching Tech Together

Hundreds of grassroots groups have sprung up around the world to teach programming, web design, robotics, and other skills to free-range learners outside traditional classrooms. These groups exist so that people don't have to learn these things on their own, but ironically, their founders and instructors are often teaching themselves how to teach.

There's a better way. Just as knowing a few basic facts about germs and nutrition can help you stay healthy, knowing a few things about psychology, instructional design, inclusivity, and community organization can help you be a more effective teacher. This book presents evidence-based practices you can use right now, explains why we believe they are true, and points you at other resources that will help you go further. Its four sections cover:

- how people learn;
- how to design lessons that work;
- how to deliver those lessons; and
- how to grow a community of practice around teaching.

This Book Belongs to Everyone

This book is a community resource. Parts of it were originally created for the Software Carpentry instructor training program, which has been run over several hundred times over the past six years, and all of it can be freely distributed and re-used under the Creative Commons - Attribution 4.0 license (s:license). Please see http://teachtogether.tech/ to download a digital version or purchase a printed copy at cost.

Contributions of all kinds are welcome, from errata and minor improvements to entirely new sections and chapters. All proposed contributions will be managed in the same way as edits to Wikipedia or patches to open source software, and all contributors will be credited for their work each time a new version is released. Please see s:joining for details and our code of conduct.

The Rules

1. Be kind: all else is details.
2. Remember that you are not your learners...
3. ...that most people would rather fail than change...
4. ...and that ninety percent of magic consists of knowing one extra thing.
5. Never teach alone.
6. Never hesitate to sacrifice truth for clarity.
7. Make every mistake a lesson.
8. Remember that no lesson survives first contact with learners...
9. ...that every lesson is too short from the teacher's point of view and too long from the learner's...
10. ...and that nobody will be more excited about the lesson than you are.

1 Introduction

Hundreds of grassroots groups have sprung up around the world to teach programming, web design, robotics, and other skills to free-range learners outside traditional classrooms. These groups exist so that people don't have to learn these things on their own, but ironically, their founders and instructors are often teaching themselves how to teach.

There's a better way. Just as knowing a few basic facts about germs and nutrition can help you stay healthy, knowing a few things about psychology, instructional design, inclusivity, and community organization can help you be a more effective teacher. This book presents evidence-based practices you can use right now, explains why we believe they are true, and points you at other resources that will help you go further. Its four sections cover:

- how people learn;
- how to design lessons that work;
- how to deliver those lessons; and
- how to grow a community of practice around teaching.

Throughout, we try to follow our own advice: for example, we start with ideas that are short, engaging, and actionable in order to motivate you to read further (Chapter 10), include lots of exercises that can be used to reinforce learning (Chapter 2), and include the original design for this book in s:v3 so that you can see what a lesson design looks like.

each time a new version is released. Please see s:joining for details and Section C.1 for our code of conduct.

1.1 Who You Are

Section 6.1 explains how to figure out who your learners are. The four I had in mind when writing this book are all end-user teachers: teaching isn't their primary occupation, they have little or no background in pedagogy, and they may work outside institutional classrooms.

Emily trained as a librarian, and now works as a web designer and project manager in a small consulting company. In her spare time, she helps run web design classes for women entering tech as a second career. She is now recruiting colleagues to run more classes in her area using the lessons that she has created, and wants to know how to grow a volunteer teaching organization.

Moshe is a professional programmer with two teenage children whose school doesn't offer programming classes. He has volunteered to run a monthly after-school programming club, and while he frequently gives presentations to colleagues, he has no experience designing lessons. He wants to learn how to build effective lessons in collaboration with others, and is interested in turning his lessons into a self-paced online course.

Samira is an undergraduate in robotics who is thinking about becoming a full-time teacher after she graduates. She wants to help teach weekend workshops for undergraduate women, but has never taught an entire class before, and feels uncomfortable teaching things that she's not an expert in. She wants to learn more about education in general in order to decide if it's for her.

Gene is a professor of computer science whose research area is operating systems. They have been teaching undergraduate classes for six years, and increasingly believe that there has to be a better way. The only training available through their university's teaching and learning center relates to posting assignments and grades in the learning management system, so they want to find out what else they ought to be asking for.

These people have *a variety of technical backgrounds* and *some previous teaching experience*, but *no formal training in teaching, lesson design, or community organization*. Most work with *free-range learners* and are *focused on teenagers and adults* rather than children; all *have limited time and resources*.

Section C.2 describes different ways people have used this material. (That discussion is delayed to an appendix because it refers to some of the ideas introduced later in this book.) We expect our made-up learners to use this material as follows:

Emily will take part in a weekly online reading group with her volunteers.

Moshe will cover part of this book in a two-day weekend workshop and study the rest on his own.

Samira will use this book in a one-semester undergraduate course with assignments, a project, and a final exam.

Gene will read the book on their own in their office or while commuting, wishing all the while that universities did more to support high-quality teaching.

1.2 What to Read Instead

If you are in a hurry, or want a taste of what this book will cover, [Brow2018] presents ten evidence-based tips for teaching computing. You can download the paper, or read it online, on the PLoS website.

I also recommend:

- The Carpentries instructor training, for which most of the first half of this book was originally developed.
- [Lang2016] and [Hust2012], which are short, approachable, and connect things you can do right now to the research that backs them.
- [Majo2015], [Broo2016], [Berg2012], and [Rice2018]. The first catalogs a hundred different kinds of exercises you can do with students; the second describes fifty different ways that groups can discuss things productively, while the third is a collection of patterns for teaching, and the fourth explains why to give learners breaks in class and ways to use them productively. These books can be used on their own, but I think they make more sense once Huston or Lang have given you a framework for understanding them.
- [DeBr2015], which conveys a lot of what *is* true about educational by explaining what *isn't*, and [Dida2016], which grounds learning theory in cognitive psychology.
- [Pape1993], which remains an inspiring vision of how computers could change education.
- [Gree2014], [McMi2017] and [Watt2014]. These three short books explain why so many attempts at educational reform have failed over the past forty years, how for-profit colleges are exploiting and exacerbating the growing inequality in our society, and how technology has repeatedly failed to revolutionize education.
- [Guzd2015a], [Hazz2014], and [Sent2018], which are academically-oriented books I've found useful about teaching computing.
- [Brow2007] and [Mann2015], because you can't teach computing well without changing the system in which we teach, and you can't do that on your own.

Of these, [Pape1993] is the one that shaped my ideas about teaching the most. Papert's central argument is that people don't absorb knowledge; instead, they (re-)construct it for themselves, and computers are a new and powerful tool for helping them do that. Andy Ko's excellent description does a better job of summarizing Papert's ideas than I possibly could, and [Craw2010] is a thought-provoking companion to both.

1.3 History

A lot of my stories aren't true, but this is a true story...

When I started teaching people how to program in the late 1980s, I went too fast, used too much jargon, and had no idea how much my learners actually understood. I got better over time, but still felt like I was stumbling around in a darkened room.

In 2010, I rebooted a project called Software Carpentry that teaches basic computing skills to researchers. (The name "carpentry" was chosen to distinguish what we taught from software engineering: we were trying to show people the digital equivalent of painting a bathroom, not building the Channel Tunnel.) In the years that followed, I discovered resources like Mark Guzdial's blog and the book *How Learning Works* [Ambr2010]. These in turn led me to books like [Hust2012, Lemo2014, Lang2016] that showed me how to build and deliver better lessons in less time and with less effort.

I started using these ideas in Software Carpentry in 2012. The results were everything I'd hoped for, so I began running training sessions to pass on what I'd learned. Those sessions became a training program that dozens of trainers have now taught to over a thousand people on six continents. Since then, I have run the course for people who teach programming to children, librarians, and women re-entering the workforce or changing careers, and all of those experiences have gone into this book.

1.4 Why Learn to Program?

Politicians, business leaders, and educators often say that people should learn to program because the jobs of the future will require it; for example, [Scaf2017] found that people who aren't software developers but who still program make higher wages than comparable workers who do not.

However, as Benjamin Doxtdator has pointed out, many of those claims are built on shaky ground. Even if they were true, education shouldn't prepare people for the jobs of the future: it should give them the power to decide what kinds of jobs there are, and to ensure that those jobs are worth doing. And as Mark Guzdial points out, there are actually many reasons to learn how to program:

1. To understand our world.

2. To study and understand processes.
3. To be able to ask questions about the influences on their lives.
4. To use an important new form of literacy.
5. To have a new way to learn art, music, science, and mathematics.
6. As a job skill.
7. To use computers better.
8. As a medium in which to learn problem-solving.

Part of what motivates me to teach is the hope that if enough people understand how to make technology work for them, we will be able to build a society in which *all* of the reasons above are valued and rewarded (Chapter 16).

1.5 Have a Code of Conduct

The most important thing I've learned about teaching in the last thirty years is how important it is for everyone to treat everyone else with respect, both in and out of class. If you use this material in any way, please adopt a Code of Conduct like the one in s:conduct and require everyone who takes part in your classes to abide by it.

A Code of Conduct can't 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. More importantly, having one tells people that there are rules, and that they can expect a friendly learning experience.

If someone challenges you about having a Code of Conduct, remind them that it *isn't* an infringement of free speech. People have a right to say what they think, but that doesn't mean they have a right to say it wherever and whenever they want. If they want to make someone feel unwelcome, they can go and find their own space in which to do it.

1.6 Acknowledgments

This book would not exist without the hard work and feedback of Erin Becker, Azalee Bostroem, Hugo Bowne-Anderson, Neil Brown, Gerard Capes, Francis Castro, Dav Clark, Warren Code, Ben Cotton, Richie Cotton, Karen Cranston, Katie Cunningham, Natasha Danas, Matt Davis, Neal Davis, Mark Degani, Tim Dennis, Michael Deutsch, Brian Dillingham, Kathi Fisler, Auriel Fournier, Bob Freeman, Nathan Garrett, Mark Guzdial, Rayna Harris, Ahmed Hasan, Ian Hawke, Felienne Hermans, Kate Hertweck, Toby Hodges, Dan Katz, Christina Koch, Shriram Krishnamurthi, Katrin Leinweber, Colleen Lewis, Lenny Markus, Sue McClatchy, Jessica McKellar, Ian Milligan, Lex Nederbragt, Aleksandra Nenadic, Jeramia Ory, Joel Ostblom, Elizabeth Patitsas, Aleksandra Pawlik, Sorawee Porncharoenwase, Emily Porta, Alex

Pounds, Thomas Price, Danielle Quinn, Ian Ragsdale, Erin Robinson, Rosario Robinson, Ariel Rokem, Pat Schloss, Malvika Sharan, Florian Shkurti, Juha Sorva, Tracy Teal, Tiffany Timbers, Richard Tomsett, Preston Tunnell Wilson, Matt Turk, Fiona Tweedie, Anelda van der Walt, Stéfan van der Walt, Allegra Via, Petr Viktorin, Belinda Weaver, Hadley Wickham, Jason Williams, Simon Willison, John Wrenn, and Andromeda Yelton. I am grateful to them, to Lukas Blakk for the cover image, and to everyone who has used this material over the years; any mistakes that remain are mine.

Breaking the Law
Much of the research reported in this book was publicly funded, but despite that, a lot of it is locked away behind paywalls. At a guess, I broke the law roughly 250 times to download papers from sites like Sci-Hub. I hope the day is coming when no one will need to do that; if you are a researcher, please hasten that day by publishing your research in open access venues, or by posting copies on open preprint servers.

1.7 Exercises

Each chapter ends with a variety of exercises that include a suggested format and an indication of how long they usually take in an in-person setting. Most can be used in other formats—in particular, if you are going through this book on your own, you can still do many of the exercises that are described as being for groups—and you can always spend more time on them than what's suggested.

The exercises in this chapter can be used as preassessment questions (Section 9.4) rather than as in-class exercises. If you have learners answer them a few days before a class or workshop starts, they will give you a much clearer idea of who they are and how best you can help them.

Highs and Lows (whole class/5)

Write brief answers to the following questions and share with your peers. (If you are taking notes together online as described in Section 9.7, put your answers there.)

1. What is the best class or workshop you ever took? What made it so good?
2. What was the worst one? What made it so bad?

Know Thyself (whole class/5)

Write brief answers to the following questions and share them as described above. Keep your answers somewhere so that you can refer to them as you go through the rest of this book.

1. What do you most want to teach?
2. Who do you most want to teach?
3. Why do you want to teach?
4. How will you know if you're teaching well?

Starting Points (individual/5)

Write brief answers to the following questions and share them as described above. Keep your answers somewhere so that you can refer to them as you go through the rest of this book.

1. What do you most want to learn about teaching and learning?
2. What is one specific thing you believe is true about teaching and learning?

Why Learn to Program? (individual/20)

Re-read Guzdial's list of reasons to learn to program in Section 1.4, then draw a 3x3 grid whose axes are labelled "low", "medium", and "high" and place each point in one sector according to how important it is to you (the X axis) and to the people you plan to teach (the Y axis).

1. Which points are closely aligned in importance (i.e., on the diagonal in your grid)?
2. Which points are misaligned (i.e., in the off-diagonal corners)?
3. How does this change what you teach?

2 Building Mental Models

The first task in teaching is to figure out who your learners are and what they already know. Our approach is based on the work of researchers like Patricia Benner, who studied how nurses progress from being novices to being experts [Benn2000]. 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 highly variable; obsessing over how a few geniuses taught or learned isn't generally useful.)

For our purposes, we can simplify Benner's progression to three stages:

- Novices don't know what they don't know, i.e., they don't yet have a usable mental model of the problem domain. As a result, they reason by analogy and guesswork, borrowing bits and pieces of mental models from other domains that seem superficially similar.
- Competent practitioners can do normal tasks with normal effort under normal circumstances because they have a mental model that's good enough for everyday purposes. That model doesn't have to be complete or accurate, just useful.
- Experts have mental models that include the complexities and special cases that competent practitioners' do not. This allows experts to handle situations that are out of the ordinary, diagnose the causes of problems, and so on. Like competent practitioners, experts know what they don't know and how to learn it; we will discuss expertise in more detail in Chapter 3.

So what *is* a mental model? As you may have gathered from the way we used the term above, it is a simplified representation of the most important parts of some problem domain that is good enough to enable problem solving. One example is the ball-and-spring models of molecules used 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. A more sophisticated model of an atom has a small central ball (the nucleus) surrounded by orbiting electrons. Again, it's wrong, but useful.

One sign that someone is a novice is that the things they say are not 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 Chapter 10 explains, it is very important not to make novices uncomfortable for doing this: until they have a better mental model, reasoning by (inappropriate) borrowing from their knowledge of other subjects is the best they can do.

Presenting novices with a pile of facts 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 [Mull2007a] observed in a study of video instruction for science students:

> *Students have existing ideas about... phenomena before viewing a video. If the video presents... 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 is presented differs from their prior knowledge... There is hope, however. Presenting students' common misconceptions in a video alongside the... concepts has been shown to increase learning by increasing the amount of mental effort students expend while watching it.*

Your goal when teaching novices should therefore be *to help them construct a mental model* so that they have somewhere to put facts. For example, Software Carpentry's lesson on the Unix shell introduces fifteen commands in three hours. That's one command every twelve minutes, which seems glacially slow until you realize that the lesson's real purpose isn't to teach those fifteen commands: it's to teach paths, history, tab completion, wildcards, pipes, command-line arguments, and redirection. Until novices understand those concepts, the commands don't make sense; once they do understand those concepts, they can quickly assemble a repertoire of commands.

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 they are anything *but* obvious to novices). Equally, manuals frustrate novices because they use jargon and *don't* explain things. This phenomenon is called the expertise reversal effect [Kaly2003], and is another reason you have to decide early on who your lessons are meant for.

A Handful of Exceptions

One of the reasons Unix and C became popular is that Kernighan et al's trilogy [Kern1978, Kern1983, Kern1988] 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 [Fehi2008] are among the very few other books in computing that have accomplished this; even after re-reading them several times, I don't know how they manage to do it.*

2.1 Are People Learning?

One of the exercises 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, novices' misconceptions fall into three categories:

Factual errors like believing that Vancouver is the capital of British Columbia (it's Victoria). These are simple to correct, but getting the facts right is not enough on its own.

Broken models like believing that motion and acceleration must be in the same direction. We can address these by having novices reason through examples that draw attention to contradictions.

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" [Guzd2015b, Pati2016]. These are also broken models, but often deeply connected to the learner's social identity, so they resist evidence and reason.

Teaching is most effective when teachers identify and clear up learners' misconceptions *during the lesson*. This is called formative assessment; the word "formative" means it is used to form or shape the teaching. Learners don't pass or fail formative assessment; instead, it tells the teacher and the learner how they are both doing and what they should focus on next. For example, a music teacher might ask a learner to play a scale very slowly in order to see if she is breathing correctly, while someone teaching web design could ask a learner to resize the images in a page to check if his explanation of CSS made sense.

The counterpoint to formative assessment is summative assessment, which you do at the end of the lesson to determine if your teaching was successful, i.e., whether the learner has understood what you have taught and is ready to move on. One way of thinking about the difference is that a chef tasting food as she cooks it is formative assessments, but the guests tasting it once it's served is summative.

In order to be useful during teaching, a formative assessment has to be quick to administer (so that it doesn't break the flow of the lesson) and give a clear result (so that it can be used with groups as well as individuals). The most widely used kind of formative assessment is probably the multiple choice question (MCQ). A lot of teachers have a low opinion of them, but

when they are designed well, they can reveal much more than just whether someone knows specific facts. For example, suppose you are teaching children how to do multi-digit addition [Ojos2015], and you give them this MCQ:

What is 37 + 15?

- *52*
- *42*
- *412*
- *43*

The correct answer is 52, but the other answers provide valuable insights:

- If the child chooses 42, she is throwing away the carry completely.
- If she chooses 412, she is treating each column of numbers as a separate problem unconnected to its neighbors.
- If she chooses 43 then she knows she has to carry the 1, but is carrying it back into the column it came from.

Each of these incorrect answers is a plausible distractor with diagnostic power. A distractor is a wrong or less-than-best answer; "plausible" means that it looks like it could be right, while "diagnostic power" means that each of the distractors helps us figure out what to explain next to that particular learner.

In order to come up with plausible distractors, think about the questions your 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, ask colleagues about their experiences, or look at the history of your field—if everyone misunderstood your subject in some way fifty years ago, the odds are that a lot of your learners will still misunderstand it that way today. You can also ask open-ended questions in class to collect misconceptions about material to be covered in a later class, or check question and answer sites like Quora or Stack Overflow to see what people learning the subject elsewhere are confused by.

MCQs aren't the only kind of formative assessment you can use: Parsons Problems (Chapter 4) and matching problems (Section 12.3) are also quick and unambiguous. Short-answer questions are another option: if answers are 2–5 words long, there are few enough plausible answers to make scalable assessment possible [Mill2016a].

Developing formative assessment is useful even if you don't use them in class because it forces you to think about your learners' mental models and how they might be broken—in short, to put yourself into your learners' heads and see the topic from their point of view. Whatever you pick, you should use something that takes a minute or two every 10–15 minutes to make sure that your learners are actually learning. That way, if a significant

number of people have fallen behind, only a short portion of the lesson will have to be repeated. This rhythm isn't based on an intrinsic attentional limit: [Wils2007] found little support for the often-repeated claim that students can only pay attention for 10–15 minutes. If you are teaching online (Chapter 11), you should check in much more often to keep learners engaged.

Formative assessments can also be used preemptively: if you start a class with an MCQ and everyone answers it correctly, you can skip the part of the lecture that was going to explain something your learners already know. Doing this also shows learners that you respect your learners' time enough not to waste it, which helps with motivation (Chapter 10).

If the majority of the class chooses the same wrong answer, you should go back and work on correcting the misconception that distractor points to. If their answers are pretty evenly split between several options they are probably just guessing, so you should back up and re-explain the idea in a different way.

What if most of the class votes for the right answer, but a few vote for wrong ones? In that case, 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. No matter how hard you work or what teaching practices you use, you won't always be able to give everyone what they need; it's your responsibility as a teacher to make the call.

Concept Inventories

Given enough data, MCQs can be made surprisingly precise. The best-known example is the Force Concept Inventory [Hest1992], which assesses understanding of basic Newtonian mechanics. By interviewing a large number of respondents, correlating their misconceptions with patterns of right and wrong answers, and then improving the questions, its creators constructed a diagnostic tool that can pinpoint specific misconceptions. Researchers can then use that tool to measure how effective changes in teaching methods are [Hake1998].

Tew and others developed and validated a language-independent assessment for introductory programming [Tew2011]; [Park2016] has replicated it, and [Hamo2017] is developing a concept inventory for recursion. However, it's very costly to build tools like this, and students' ability to search for answers online is an ever-increasing threat to their validity.

Working formative assessments into class requires only a little bit of preparation and practice. Giving students colored or numbered cards so that they can all answer an MCQ at once (rather than holding up their hands in turn), having one of the options be, "I have no idea", and encouraging them to talk to their neighbors for a few seconds before answering will all help ensure that your teaching flow isn't disrupted. Section 9.2 describes

a powerful, evidence-based teaching method that builds on these simple ideas.

Humor
Teachers sometimes put supposedly-silly answers like "my nose!" on MCQs, particularly ones intended for younger students. However, they don't provide any insight into learners' misconceptions, and most people don't actually find them funny (especially on re-reading).

A lesson's formative assessments should prepare learners for its summative assessment: no one should ever encounter a question on an exam that the teaching did not prepare them for. This doesn't mean you should never put new kinds of problems on an exam, but if you do, you should have given learners practice with (and feedback on) tackling novel problems beforehand.

2.2 Exercises

Your Mental Models (think-pair-share/15)

What is one mental model you use to understand your work? Write a few sentences describing it, and give feedback on a partner's. Once you have done that, have a few people share their models with the whole group. Does everyone agree on what a mental model is? Is it possible to give a precise definition, or is the concept useful precisely because it is a bit fuzzy?

Symptoms of Being a Novice (whole class/5)

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?

Modelling Novice Mental Models (pairs/20)

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

When you are done, trade MCQs with a partner. Is their question ambiguous? Are the misconceptions plausible? Do the distractors actually test for them? Are any likely misconceptions *not* tested for?

Other Kinds of Formative Assessment (whole class/20)

A good formative assessment requires people to think through a problem. For example, imagine that you have placed a block 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 (Figure 1)?

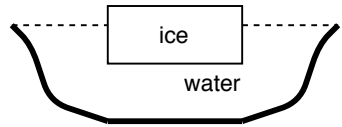

Figure 1: Ice in a Bathtub

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 out why helps people build a model of the relationship between weight, volume, and density [Epst2002].

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.

A Different Progression (individual/15)

The model of skill development described at the start of this chapter is sometimes called the Dreyfus model. Another commonly-used progression is the four stages of competence:

Unconscious incompetence: the person doesn't know what they don't know.

Conscious incompetence: the person realizes that they don't know something.

Conscious competence: the person has learned how to do something, but can only do it while concentrating, and may still need to break things down into steps.

Unconscious competence: the skill has become second nature, and the person can do it reflexively.

Identify one subject where you are at each level. What level are most of your learners at? What level are you trying to get them to?

What Kind of Book Is This? (small groups/5)

What are the chapters in the main body of this book: a tutorial or a manual? What about the appendices? Why?

What Kind of Computing? (individual/10)

[Tedr2008] summarizes three traditions in computing:

Mathematical: Programs are the embodiment of algorithms; they are either correct or incorrect, as well as more or less efficient.

Scientific: Programs are more or less accurate models of information processes that can be studied using the scientific method.

Engineering: Programs are built objects like dams and airplanes, and are more or less effective and reliable.

Which of these best matches your mental model of computing? If none of them do, what model do you have?

3 Expertise and Memory

Memory is the residue of thought.
— Dan Willingham

The previous chapter explained what distinguishes novices from competent practitioners. This one looks at expertise: what it is, how people acquire it, and how it can be harmful as well as helpful. It then shows 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 answer is that they can solve problems much faster than people who are "merely competent", or that they can recognize and deal with cases where the normal rules don't apply. They also somehow make this look effortless: in many cases, they instantly know what the right answer is [Parn2017].

Expertise is more than just knowing more facts: competent practitioners can memorize a lot of trivia without any noticeable improvement in their performance. Instead, imagine for a moment that we store knowledge as a network or graph in which facts are nodes and relationships are arcs. (This is definitely *not* how our brains work, but it's a useful metaphor.) The key difference between experts and competent practitioners is that experts' mental models are much more densely connected, i.e., they are much more likely to know of a connection between any two randomly-selected pieces of information.

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.
- Densely-connected graphs are also the basis for experts' fluid representations, i.e., their ability to switch back and forth between different views of a problem [Petr2016]. For example, when trying to solve a problem in

mathematics, an expert might switch between tackling it geometrically and representing it as a set of equations to be solved.

- 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.)
- Finally, experts are often 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 learning it themselves.

The last of these points is important enough to have a name of its own: expert blind spot. As originally defined in [Nath2003], it is the tendency of experts to organize explanation according to the subject's deep principles, rather than being guided by what their learners already know. 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 [Mars2002].

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 in Chapter 10, doing this signals that the speaker thinks the problem is trivial and that the person struggling with it must therefore be stupid. Don't do this.

3.1 Concept Maps

The graph metaphor explains why helping learners make connections is as important as introducing them to facts: without those connections, it's hard for people to recall things that they know. To use another analogy, the more people you know at a party, the less likely you are to leave early.

Our tool of choice for representing someone's mental model as a graph is a concept map, in which 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, different people can have different concept maps for the same topic, but one of the benefits of concept mapping is that it makes those differences explicit.

As an example, Figure 2 reproduces a concept map taken from the IHMC CMap site showing why the Earth has seasons, and Figure 12 uses a concept map to explain how to create a good screencast.

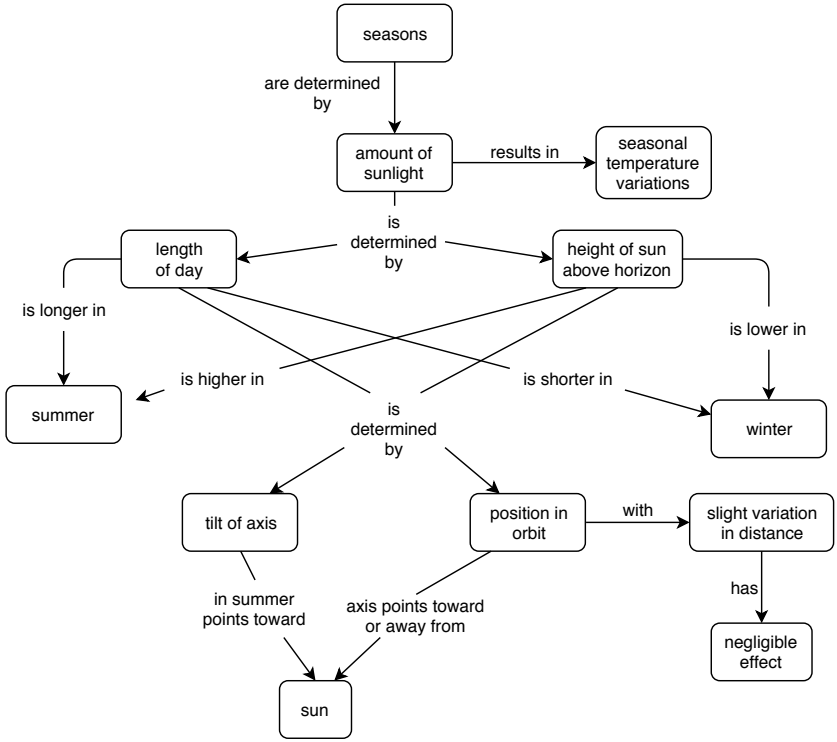

Figure 2: Concept Map for Seasons (from <https://cmap.ihmc.us/>)

To show how concept maps can be using in teaching programming, consider this `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 top of Figure 3, but they are only half the story. The expanded version in the bottom shows the *relationships* between those things, which are as important for understanding as the concepts themselves.

Concept maps can be used in many ways:

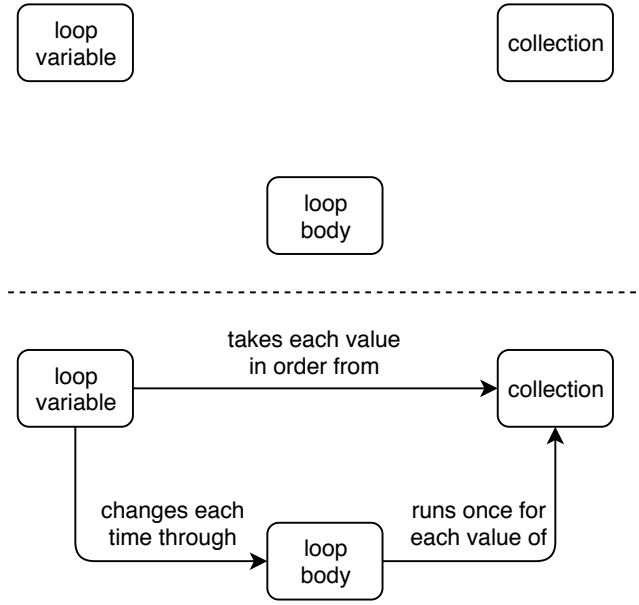

Figure 3: Concept Map for a For Loop

Helping teachers 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. (In technical terms, they reduce the teacher's cognitive load—we will discuss this again in Chapter 4.)

Aiding communication between lesson designers. Teachers 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 helps.

Aiding 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 teacher said. (We will return to this idea in Section 4.1.)

For assessment. Having learners draw pictures of what they think they just heard shows the teacher what they missed and what was miscommunicated. Reviewing learners' concept maps is too time-consuming to do as in-class formative assessment, 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.

[Kepp2008] looked at the use of concept mapping in computing education. One of their findings was that, "... concept mapping is troublesome for many students because it tests personal understanding rather than knowledge that was merely learned by rote." As someone who values understanding over rote knowledge, I count that as a benefit.

Some teachers are also skeptical of whether novices can effectively map their understanding, since introspection and explanation of understanding are generally more advanced skills than understanding itself. Like any other new tool or technique, concept maps have to be taught and practiced if they are to be effective.

Start Anywhere

When asked to draw their first concept map, many people will stare at the blank page in front of them, not knowing where to start. When this happens, write down two words associated with the topic you're trying to map, then draw a line between them and add a label explaining how those two ideas are related. You can then ask what other things are related in the same way, what parts those things have, or what happens before or after the concepts already on the page in order to discover more nodes and arcs. After that, the hard part is often stopping.

Concept maps are just one way to represent our understanding of a subject; others include mind maps (which are usually radial and hierarchical), conceptual diagrams (which use predefined categories and relationships), and visual metaphors (which are striking images overlaid with text) [Eppl2006]. Maps, flowcharts, and blueprints can also be useful in some contexts, as can decision trees like [Abel2009] that shows how to choose the right kind of chart for different kinds of questions and data.

What each does is externalize cognition, i.e., make thought processes and mental models visible so that they can be compared, contrasted, and combined. [Cher2007] suggests that externalizing cognition may be the main reason developers draw diagrams when they are discussing things. They found that most developers can't identify the parts of their own diagrams shortly after having created them—instead of archiving information for posterity, diagrams are actually a cache for short-term memory that lets a participant in the discussion point at a wiggly bubble and say "that" to trigger recall of several minutes of debate.

Rough Work and Honesty

Many user interface designers believe that it's better to show people rough sketches of their ideas rather than polished mock-ups because people are more likely to give honest feedback on something that they think only took a few minutes to create—if it looks as though what they're critiquing took

hours to create, most will pull their punches. When drawing concept maps to motivate discussion, you should therefore use pencils and scrap paper (or pens and a whiteboard) rather than fancy computer drawing tools.

3.2 Seven Plus or Minus Two

While the graph model of knowledge is wrong but useful, another simple model has a sounder physiological basis. As a rough approximation, human memory can be divided into two distinct layers. The first, called long-term or persistent memory, is where we store things like our friends' names, 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: [Mill1956] estimated that the average adult's working memory could only hold 7 ± 2 items at a time. 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 several times. As Section 3.3 discusses, short-term memory may actually be as small as 4 ± 1 items; our innate tendency to remember things together gives the illusion of it being larger.

Participation
The size of working memory is sometimes used to explain why sports teams tend to have about half a dozen members or be broken down into subgroups like the forwards and backs in rugby. It is also used to explain why meetings are only productive up to a certain number of participants: if twenty people try to discuss something, either three meetings are going on at once or half a dozen people are talking while everyone else listens. The argument is that people's ability to keep track of their peers is constrained by the size of working memory, but so far as I know, the link has never been proven.

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, they need to keep a bunch of arbitrary facts straight in their head: what does this variable represent, what value does it currently hold, etc. If the number of facts grows too large, their mental model of the program comes crashing down (something we have all experienced).

7 ± 2 is also the most important number in teaching. A teacher cannot push information directly into a learner's long-term memory. Instead, whatever they present is first stored in the learner's short-term memory, and is only transferred to long-term memory after it has been held there and

rehearsed (Section 5.1). If the teacher presents too much information too quickly, the new will displace the old before it has a chance to consolidate in long-term memory.

This is one of the reasons to create a concept map for a lesson when designing it: doing so helps the teacher identify how many pieces of separate information the learner will need to store in memory as the lesson unfolds. In practice, I often draw a concept map, realize there's far too much in it to teach in a single pass, and then carve out tightly-connected subsections to break the lesson into digestible pieces, each of which leads to a formative assessment.

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.

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. Once everyone realizes how different their mental models of the project are, a lot of interesting discussion will ensue. . . .

The simple model of memory presented here has largely been replaced by a more sophisticated one in which short-term memory is broken down into several modal stores (e.g., for visual vs. linguistic memory), each of which does some involuntary preprocessing [Mill2016a]. Our presentation is therefore an example of a mental model that aids learning and everyday work, but is eventually superseded by something more complicated.

Research also now indicates that the limiting factor for long-term memory is not retention, but rather the ability to recall memories that are present. Studying in short, spaced periods in a variety of contexts improves recall; the reason may be that doing so creates more cues than cramming (Section 5.1).

3.3 Pattern Recognition

The preceding section said that short-term memory can only store 7 ± 2 items at a time, and recent research have suggested that its actual size might be as low as 4 ± 1 items [Dida2016]. In order to handle larger information sets, our minds create chunks. For example, most of us remember words as single items, rather than as sequences 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.

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. However, chunking can also mislead us if we mis-identify things: newcomers really can sometimes see things that experts have looked at and missed.

Given how important chunking is to thinking, it is tempting to try to teach patterns directly. One way to do this is to identify design patterns, which are reusable solutions to common problems. Patterns help competent practitioners think and talk to each other in many domains (including teaching [Berg2012]), but pattern catalogs are too dry and too abstract for novices to make sense of on their own. That said, giving names to a small number of patterns does seem to help with teaching, primarily by giving the learners a richer vocabulary to think and communicate with [Kuit2004, Byck2005, Saja2006]. We will return to this in Section 7.1.

3.4 Becoming an Expert

So how does someone become an expert? The idea that ten thousand hours of practice will do it is widely quoted but probably not true: doing the same thing over and over again is much more likely to solidify bad habits than improve 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:

Act on feedback from others. For example, a student might write an essay about what they did on their summer holiday and get feedback from a teacher telling them how to improve it.

Give feedback to others. For example, they might critique character development in *The Catcher in the Rye*. For this to be effective, it's essential that they get feedback on their feedback, i.e., that the teacher critiques their analysis.

Give feedback to themselves. At some point, they start critiquing their own work in real time (or nearly so) using the skills they have now built up. Doing this is so much faster than waiting for feedback from others that proficiency suddenly starts to take off.

What Counts as Deliberate Practice?

[Macn2014] 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." However, [Eric2016] critiqued this finding by saying, "Summing up every hour of any type of practice during an individual's career implies that the impact of all types

of practice activity on performance is equal—an assumption that... is inconsistent with the evidence." To be effective, deliberate practice requires both a clear performance goal and immediate informative feedback.

3.5 Exercises

Concept Mapping (pairs/30)

Draw a concept map for something you would teach in five minutes. 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?

Concept Mapping (Again) (small groups/20)

Working in groups of 3–4, have each person independently draw a concept map showing their mental model of what goes on in a classroom. When everyone is done, compare the concept maps. Which concepts and relationships are common? Which are different? Where do your mental models agree and disagree?

A Concept Map for This Material (individual/30)

After you have finished going through this material (not just this chapter), pick one small topic, draw a concept map for it, and send it to us (s:joining). If we decide to add it to this book, we will add you to the credits in the introduction.

Noticing Your Blind Spot (small groups/10)

Consider all the things you have to know to understand this one line of Python source code:

```
answers = ['tuatara', 'tuataras', 'bus', "lick"]
```

As Elizabeth Wickes points out:

- The square brackets surrounding the content mean we're working with a list (as opposed to square brackets immediately to the right of something, which is a data extraction notation).
- The elements are separated by commas, which are outside/between the quotes (rather than inside, as they would be for quoted speech).
- Each element is a character string, and we know that because of the quotes. We could have number or other data types in here if we wanted; we need quotes because we're working with strings.

- We're mixing our use of single and double quotes, and Python doesn't care (so long as they balance around the individual strings).
- Each comma is followed by a space, which is not required by Python, but we prefer it for readability.

Each of these details might be overlooked by an expert. Working in groups of 3–4, select something equally short from a lesson you have recently taught or taken and break it down to this level of detail.

4 Cognitive Load

In [Kirs2006], Kirschner, Sweller and Clark wrote:

> *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.*

Their paper set off a minor academic storm, because beneath the jargon the authors were claiming that allowing learners to ask their own questions, set their own goals, and find their own path through a subject, as they would when solving problems in real life, isn't effective. This approach—called inquiry-based learning—is intuitively appealing, but the authors argued that it overloads learners by requiring them to master a domain's factual content and its problem-solving strategies at the same time.

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

Intrinsic load is what people have to keep in mind in order to absorb new material. 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. Intrinsic load can't be reduced except by reducing the amount of content being taught.

Germane load is the (desirable) mental effort required to link new information to old, which is one of the things that distinguishes learning from memorization. An example might be remembering that a loop variable is assigned a new value each time the loop executes.

Extraneous load is everything else in the instructional material that distracts from learning, such as matching the highlight colors in the instruc-

tor's examples to the different color scheme used by the learner's own editor.

Cognitive load theory holds that people have to split a fixed amount of working memory between these three things. Our goal as instructors is to maximize the memory available to handle germane load, which means reducing the intrinsic load at each step and eliminating as much of the extraneous load as possible.

For example, searching for a solution strategy is an extra burden on top of actually applying that strategy. We can therefore accelerate learning by giving learners worked examples that break a solution procedure down into steps, each of which can be mastered on its own before being combined with other steps (which is a step in its own right).

One way to do this is to give learners a series of faded examples. The first example 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 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.

Faded examples can be used in almost every kind of teaching, from sport and music to contract law. Someone teaching programming might use them by first explaining how to calculate the total length of a list of words:

```
# total_length(["red", "green", "blue"]) => 12
define total_length(list_of_words):
    total = 0
    for each word in list_of_words:
        total = total + word.length()
    return total
```

and then asking learners to fill in the blanks in this (which focuses their attention on control structures):

```
# word_lengths(["red", "green", "blue"]) => [3, 5, 4]
define word_lengths(list_of_words):
    list_of_lengths = []
    for each ____ in ____:
        list_of_lengths.append(____)
    return list_of_lengths
```

The next problem might be this (which focuses their attention on updating the final result):

```
# join_all(["red", "green", "blue"]) => "redgreenblue"
define join_all(list_of_words):
    joined_words = ____
```

```
    for each ____ in ____:
        ____
    return joined_words
```

Learners would finally be asked to write an entire function on their own:

```
# make_acronym(["red", "green", "blue"]) => "RGB"
define make_acronym(list_of_words):
    ____
```

Faded examples work because they introduce the problem-solving strategy piece by piece: at each step, learners have one new problem to tackle, which is less intimidating than a blank screen or a blank sheet of paper (Section 9.11). It also encourages learners to think about the similarities and differences between various approaches, which helps create the linkages in their mental models that help retrieval.

Efficiency vs. Extent

Seeing worked examples accelerates learning more than having students write lots of code themselves [Skud2014]. As we will see in Chapter 7, deconstructing code by tracing it or debugging it also increases the efficiency of learning [Grif2016]. However, this isn't the same as saying that people learn more unless they see additional problems.

The key to constructing a good faded example is to think about the problem-solving strategy 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.

Critics have sometimes argued that any result can be justified after the fact by labelling things that hurt performance as extraneous load and things that don't as intrinsic or germane. However, instruction based on cognitive load theory is undeniably effective. For example, [Maso2016] redesigned a database course to remove split attention and redundancy effects, and provide worked examples and sub-goals. The new course reduced exam failure rate by 34% on an identical final exam and increased student satisfaction.

A decade after the publication of [Kirs2006], a growing number of people believe that cognitive load theory and inquiry-based approaches are compatible if viewed in the right way. [Kaly2015] argues that cognitive load theory is basically micro-management of learning within a broader context that considers things like motivation, while [Kirs2018] extends cognitive load theory to include collaborative aspects of learning. As with [Mark2018] (discussed in Section 5.1), researchers' perspectives may differ, but the practical implementation of their theories often wind up being the same.

Cognitive Apprenticeship

An alternative model of learning and instruction that also uses scaffolding and fading is cognitive apprenticeship, which emphasizes the way in which a master passes on skills and insights to an apprentice. The master provides models of performance and outcomes, then coaches novices as they take their first steps by explaining what they're doing and why [Coll1991, Casp2007]. The apprentice reflects on their own problem solving, e.g., by thinking aloud or critiquing their own work, and eventually explores problems of their own choosing.

This model tells us that instructors should present several examples when presenting a new idea so that learners can see what to generalize, and that we should vary the form of the problem to make it clear what are and aren't superficial features. (For a long time, I believed that the variable holding the value a function was going to return had *to be called* `result` *because my instructor always used that name in examples.) Problems should be presented in real-world contexts, and we should encourage self-explanation, since it helps learners organize and make sense of what they have just been taught. This is discussed in more detail in Section 5.1.*

Parsons Problems

Another kind of exercise that can be explained in terms of cognitive load theory is called a Parsons Problem (named after one of their creators [Pars2006]). 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. Multiple studies have shown that Parsons Problems take less time for learners to do, but produce equivalent educational outcomes [Eric2017].

Labelled Subgoals

Subgoal labelling means giving names to the steps in a step-by-step description of a problem-solving process. [Marg2016, Morr2016] all found that students with labelled subgoals solved Parsons Problems better than students without, and the same benefit is seen in other problem domains [Marg2012]. Returning to the Python example used earlier, the subgoals in finding the total length of a list of words or constructing an acronym are:

1. Create an empty value of the type to be returned.
2. Get the value to be added to the result from the loop variable.
3. Update the result with that value.

Labelling subgoals works because grouping related steps in a chunk (Section 3.3) and giving each chunk a name helps learners distinguish between generic information and information that is specific to the problem at hand, which reduces cognitive load. It also helps them build a mental model of that kind of problem, so that they can solve other problems of that kind, and gives them a natural opportunity for self-explanation (Section 5.1).

4.1 Split Attention

Research by Mayer and colleagues on the split-attention effect is closely related to cognitive load theory [Maye2003]. 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 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 (a topic we'll return to when discussing dual coding in Section 5.1).

Learning is therefore more effective when information is presented simultaneously in two different channels, but when that information is complementary rather than redundant. For example, people generally 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, because some of their attention has to be devoted to checking that the narration and the captions agree with each other. Two notable exceptions to this are people who do not yet speak the language well and people with hearing exercises or other special needs, both of whom may find that the extra effort is a net benefit.

This explains why it's more effective to draw a diagram piece by piece while teaching rather than to present 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. Pointing at part of the diagram later is then more likely to trigger recall of what was being said when that part was being drawn.

The split-attention effect does *not* mean that learners shouldn't try to reconcile multiple incoming streams of information—after all, this is something they have to do in the real world [Atki2000]. Instead, it means that instruction shouldn't require it while people are mastering unit skills; instead, using multiple sources of information simultaneously should be treated as a separate learning task.

Not All Graphics Are Created Equal

[Sung2012] presents an elegant study that distinguishes seductive *graphics (which are highly interesting but not directly relevant to the instructional goal),* decorative *graphics (which are neutral but not directly relevant to the instructional goal), and* instructive *graphics (directly relevant to the instructional goal). Students who received any kind of graphic gave significantly higher satisfaction ratings to material than those who didn't get graphics, but only students who got instructive graphics actually performed better.*
Similarly, [Stam2013, Stam2014] found that having more information can actually lower performance. They showed children pictures, pictures and numbers, or just numbers for two tasks: fraction equivalence and fraction addition. For equivalence, having pictures or pictures and numbers outperformed having numbers only. For addition, however, having pictures outperformed pictures and numbers, which outperformed just having numbers.

4.2 Minimal Manuals

The most extreme use of cognitive load theory may be the "minimal manual" method introduced in [Carr1987]. Its starting point is a quote from a user: "I want to do something, not learn how to do everything." Carroll and colleagues therefore redesigned training to present every idea as a single-page self-contained task: a title describing what the page was about, step-by-step instructions of how to do something really simple (like how to delete a blank line in a text editor), and then several notes how to recognize and debug common problems.

Carroll and colleagues found that rewriting training materials this way made them shorter overall, and that people using them learned faster. Later studies like [Lazo1993] confirmed that this approach outperformed the traditional approach regardless of prior experience with computers.

Looking back, [Carr2014] summarized this work by saying:

Our "minimalist" designs sought to leverage user initiative and prior knowledge, instead of controlling it through warnings and ordered steps. It emphasized that users typically bring much expertise and insight to this learning, for example, knowledge about the task domain, and that such knowledge could be a resource to instructional designers. Minimalism leveraged episodes of error recognition, diagnosis, and recovery, instead of attempting to merely forestall error. It framed troubleshooting and recovery as learning opportunities instead of as aberrations.

He goes on to say that at the time, instruction decomposed skills into subskills hierarchically and then drilled people on the sub-skills. However, this meant context was lost: the goals weren't apparent until people had learned

the pieces. Since people want to dive in and do real tasks, well-designed instruction should help them do that.

4.3 Exercises

Create a Faded Example (pairs/30)

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, try to place them in different groups, and have them play the part of learners for those groups. Alternatively, choose a different problem domain and develop a faded example for it.

Classifying Load (small groups/15)

Working in groups of 3–4, choose a short lesson that one of you has taught or taken recently, make a point-form list of the ideas, instructions, and explanations it contains, and then classify each as intrinsic, germane, or extraneous. (The exercise "Noticing Your Blind Spot" in Section 3.5 will give you an idea of how detailed your point-form list should be.)

Create a Parsons Problem (pairs/20)

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. Again, if your group includes people who aren't programmers, try using a different problem domain, such as making guacamole.

Minimal Manuals (individual/20)

Write a one-page guide to doing something simple that your learners might encounter in one of your classes, such as centering text horizontally or printing a number with a certain number of digits after the decimal points. Try to list at least three or four incorrect behaviors or outcomes the learner might see, and include a one- or two-line explanation of why each happens and how to correct it (i.e., go from symptoms to cause to fix).

Cognitive Apprenticeship (pairs/15)

Pick a small coding problem (something you can do in two or three minutes) and think aloud as you work through it while your partner asks questions about what you're doing and why. As you work, do not just comment on what you're doing, but also on why you're doing it, how you know it's the right thing to do, and what alternatives you've considered but discarded. When you are done, swap roles with your partner and repeat the exercise.

Critiquing Graphics (individual/30)

[Maye2009] presents six principles for designing good diagrams for teaching. As summarized in [Mill2016a], they are:

Signalling: visually highlight the most important points that you want students to retain so that they stand out from less-critical material.

Spatial contiguity: if using captions or other text to accompany graphics, place them as close to the graphics as practical to offset the cost of shifting between the two. If using diagrams or animations, place captions right next to relative components instead of putting them in one big block of text.

Temporal contiguity: present spoken narration and graphics as close in time as practical—presenting both at once is better than presenting them one after another.

Segmenting: when presenting a long sequence of material or when students are inexperienced with the subject, break up the presentation into shorter segments and let students control how quickly they advance from one part to the next.

Pretraining: if students don't know the major concepts and terminology used in your presentation, set up a module just to teach those concepts and terms and make sure they complete that module beforehand.

Modality: students learn better from pictures plus audio narration than from pictures plus text, unless there are technical words or symbols, or the students are non-native speakers.

Choose a video of a lesson or talk online that uses slides or other static presentations, and rate its graphics as “poor”, “average”, or “good” according to these six criteria.

5 Individual Learning

The previous three chapters have looked at what instructors can do to help their learners. This chapter looks at what learners can do for themselves by changing their study strategies and getting enough rest.

The key to getting more out of learning is metacognition, or thinking about one's own thinking processes. Just as good musicians listen to their own playing, and good teachers reflect on their teaching (Chapter 8), learners will learn better and faster if they make plans, set goals, and monitor their progress. It's difficult for learners to master these skills in the abstract—for example, just telling them to make plans doesn't have any effect—but lessons can be designed to encourage certain study practices, and drawing attention to these practices in class helps them realize that learning is a skill that can be improved like any other [McGu2015, Miya2018].

The big prize is transfer of learning, which occurs when one thing we have learned helps us learn something else more quickly. Researchers distinguish between near transfer, which occurs between similar or related areas like fractions and decimals, or loops in different programming languages, and far transfer, which occurs between dissimilar domains—the idea that learning to play chess will help mathematical reasoning or vice versa.

Near transfer undoubtedly occurs—no kind of learning beyond simple memorization could occur if it didn't—and instructors leverage it all the time by giving learners exercises that are close in form or content to what has just been presented in a lesson. However, [Sala2017] recently analyzed many studies of far transfer and concluded that while we might *want* to believe in it:

> *. . . the results show small to moderate effects. However, the effect sizes are inversely related to the quality of the experimental design. . . We conclude that far transfer of learning rarely occurs.*

When far transfer *does* occur, it seems to happen only once a subject has been mastered [Gick1987]. In practice, this means that learning to program won't help you play chess and vice versa.

5.1 Six Strategies

Psychologists study learning in a wide variety of ways, but have reached similar conclusions about what actually works [Mark2018]. The Learning Scientists have catalogued six of these strategies and summarized them in a set of downloadable posters. Teaching these strategies to students, and mentioning them by name when you use them in class, can help them learn how to learn faster and better [Wein2018].

Spaced Practice

Ten hours of study spread out over five days is more effective than two five-hour days, and far better than one ten-hour day. You should therefore create a study schedule that spreads study activities over time: block off at least half an hour to study each topic each day rather than trying to cram everything in the night before an exam [Kang2016].

You should also review material after each class (but not immediately after—take at least a half-hour break). When reviewing, be sure to include at least a little bit of older material: for example, spend 20 minutes looking over notes from that day's class, and then 5 minutes each looking over material from the previous day and from a week before. (Doing this also helps you catch any gaps or mistakes in previous sets of notes while there's still time to correct them or ask questions: it's painful to realize the night before the exam that you have no idea why you underlined "Demodulate!!" three times.)

When reviewing, make notes about things that you had forgotten: for example, make a flash card for each fact that you couldn't remember, or that you remembered incorrectly. This will help you focus the next round of study on things that most need attention.

The Value of Lectures

According to [Mill2016a], "The lectures that predominate in face-to-face courses are relatively ineffective ways to teach, but they probably contribute to spacing material over time, because they unfold in a set schedule over time. In contrast, depending on how the courses are set up, online students can sometimes avoid exposure to material altogether until an assignment is nigh."

Retrieval Practice

Researchers now believe that the limiting factor for long-term memory is not retention (what is stored), but recall (what can be accessed). Recall of specific information improves with practice, so outcomes in real situations can be improved by taking practice tests or summarizing the details of a topic from memory and then checking what was and wasn't remembered.

For example, [Karp2008] found that repeated testing improved recall of word lists from 35% to 80%.

Research also shows that recall is better when practice uses activities similar to those used in testing; for example, writing personal journal entries helps with multiple-choice quizzes, but less than doing multiple-choice quizzes [Mill2016a]. This is called transfer-appropriate processing.

One way to exercise retrieval skills is to solve problems twice. The first time, do it entirely from memory without notes or discussion with peers. After grading your own work against a rubric supplied by the instructor, solve the problem again using whatever resources you want. The difference between the two shows you how well you were able to retrieve and apply knowledge.

Another method (mentioned above) is to create flash cards. In physical form, a question or other prompt is written on one side, and the answer is written on the other; in digital form, these are ideal for deployment on mobile devices like phones. If you are studying as part of a group, you can exchange flash cards with a partner; this also helps you discover important ideas that you may have missed or misunderstood.

A quicker version of this is read-cover-retrieve: as you read something, cover up key terms or sections with small sticky notes. When you are done, go through it a second time and see how well you can guess what's under each of those stickies.

Whatever method you use, don't just practice recalling facts and definitions: make sure you also check your understanding of big ideas and the connections between them. Sketching a concept map and then comparing it to your notes or to a previously-drawn concept map is a quick way to do this.

Hypercorrection

One powerful finding in learning research is the hypercorrection effect [Metc2016]. Most people don't like to be told they're wrong, so it's reasonable to assume that the more confident someone is that the answer they've given in a test is correct, the harder it is to change their mind if they were actually wrong. However, it turns out that the opposite is true: the more confident someone is that they were right, the more likely they are not to repeat the error if they are corrected.

Interleaving

One way you can space your practice is to interleave study of different topics: instead of mastering one subject, then the next, then a third, shuffle study sessions. Even better, switch up the order: A-B-C-B-A-C is better than A-B-C-A-B-C, which in turn is better than A-A-B-B-C-C [Rohrer2015]. This is effective because interleaving fosters creation of more links between different topics, which in turn increases retention and recall.

How long you should spend on each item depends on the subject and how well you know it, but somewhere between 10 and 30 minutes is long enough for you to get into a state of flow (Section 5.2) but not for your mind to wander. Interleaving study will initially feel harder than focusing on one topic at a time, but that's a sign that it's working. If you are making flash cards for yourself, or doing practice tests, you should see improvement after only a couple of days.

Elaboration

Explaining things to yourself as you go through them helps you understand and remember them. One way to do this is to follow up each answer on a practice quiz with an explanation of why that answer is correct, or conversely with an explanation of why some other plausible answer isn't. Another is to tell yourself how a new idea is similar to or different from one that you have seen previously.

Talking to yourself may seem like an odd way to study, but [Biel1995] explicitly trained people in self-explanation, and yes, they outperformed those who hadn't been trained. An exercise that builds on this is to go through code line by line with a group, having a different person to explain each line in turn and say why it is there and what it accomplishes.

[Chi1989] found that some learners simply halt when they hit an unexplained step (or a step whose explanation they don't understand) when doing mechanics problems in a physics class. Others who pause their "execution" of the example to generate an explanation of what's going on learn faster. Instructors should therefore demonstrate the latter strategy to learners.

Explaining things to others even works on exams, though the extent of the benefits are still being studied. [Cao2017a, Cao2017b] looked at two-stage exams, i.e., a normal (individual) exam which is then immediately followed by a second exam in which students work in small groups to solve a set of problems. They found significant short-term gains for students doing exams collaboratively, but not long-term gains, i.e., the benefits visible a couple of weeks after the mid-term had faded by the final. They also found that students in the middle of the class benefited strongly, and that homogeneous-ability groups benefited, while heterogeneous groups did not.

Concrete Examples

One specific form of elaboration is so useful that it deserves its own heading, and that is the use of concrete examples. Whenever you have a statement of a general principle, try to provide one or more examples of its use, or conversely take each particular problem and list the general principles it

embodies. [Raws2014] found that interleaving examples and definitions made it more likely that learners would remember the latter correctly.

One structured way to do this is the ADEPT method: give an **A**nalogy, draw a **D**iagram, present an **E**xample, describe the idea in **P**lain language, and then give the **T**echnical details. Again, if you are studying with a partner or in a group, you can swap and check work: see if you agree that other people's examples actually embody the principle being discussed, or which principles are used in an example that they haven't listed.

Another useful technique is to teach by contrast, i.e., to show learners what a solution is *not*, or what kind of problem a technique *won't* solve. For example, when showing children how to simplify fractions, it's important to give them a few like 5/7 that can't be simplified so that they don't become frustrated looking for answers that don't exist.

Dual Coding

The last of the six core strategies that the Learning Scientists describe is to present words and images together. As discussed in Section 4.1, different subsystems in our brains handle and store linguistic and visual information, and if complementary information is presented through both channels, then they can reinforce one another. However, learning is more effective when the same information is *not* presented simultaneously in two different channels [Maye2003], because then the brain has to expend effort to check the channels against each other.

One way to take advantage of dual coding is to draw or label timelines, maps, family trees, or whatever else seems appropriate to the material. (I am personally fond of pictures showing which functions call which other functions in a program.) Drawing a diagram *without* labels, then coming back later to label it, is excellent retrieval practice.

5.2 Time Management

I used to brag about the hours I was working. Not in so many words, of course—I had *some* social skills—but I would show up for class around noon, unshaven and yawning, and casually mention how I'd been up working 'til 6:00 a.m.

Looking back, I can't remember who I was trying to impress. Instead, what I remember is how much of the work I did in those all-nighters I threw away once I'd had some sleep, and how much damage the stuff I didn't throw away did to my grades.

My mistake was to confuse "working" with "being productive". You can't produce software (or anything else) without doing some work, but you can easily do lots of work without producing anything of value. Convincing

people of this can be hard, especially when they're in their teens or twenties, but pays tremendous dividends.

Scientific study of overwork and sleep deprivation goes back to at least the 1890s (see [Robi2005] for a short, readable summary). The most important results for learners are:

1. Working more than eight hours a day for an extended period of time lowers your total productivity, not just your hourly productivity—i.e., you get less done in total (not just per hour) when you're in crunch mode than you do when you work regular hours.
2. Working over 21 hours in a stretch increases the odds of you making a catastrophic error just as much as being legally drunk.
3. Productivity varies over the course of the workday, with the greatest productivity occurring in the first four to six hours. After enough hours, productivity approaches zero; eventually it becomes negative.

These facts have been reproduced and verified for over a century, and the data behind them is as solid as the data linking smoking to lung cancer. The catch is that *people usually don't notice their abilities declining*. Just like drunks who think they're still able to drive, people who are deprived of sleep don't realize that they're not finishing their sentences (or thoughts). Five eight-hour days per week has been proven to maximize long-term total output in every industry that has ever been studied; studying or programming are no different.

But what about short bursts now and then, like pulling an all-nighter to meet a deadline? That has been studied too, and the results aren't pleasant. Your ability to think drops by 25% for each 24 hours you're awake. Put it another way, the average person's IQ is only 75 after one all-nighter, which puts them in the bottom 5% of the population. Two all nighters in a row, and their effective IQ is 50, which is the level at which people are usually judged incapable of independent living.

When You Just Can't Say No

Research has shown that our ability to exert willpower runs out, just like our ability to use muscles: if we have to resist eating the last donut on the tray when we're hungry, we are less likely to fold laundry and vice versa. This is called ego depletion [Mill2016a], and an effective counter is to build up habits so that doing the right thing is automatic.

"But—but—we have so many assignments to do!", your learners say. "And they're all due at once! We *have* to work extra hours to get them all done!" No: in order to be productive, they have to prioritize and focus, and in order to do that, they have to be taught how. One widely-used technique is to make a list of things that need to be done, sort them by priority, and then switch off email and other interruptions for 30-60 minutes

and complete one of those tasks. If any task on a to-do list is more than an hour long, break it down into smaller pieces and prioritize those separately.

The most important part of this is switching off interruptions. Despite what many people want to believe, people are not good at multi-tasking. What we can become good at is automaticity, which is the ability to do something routine in the background while doing something else [Mill2016a]. Most of us can talk while chopping onions, or drink coffee while reading; with practice, we can also take notes while listening, but we can't study effectively, program, or do other mentally challenging tasks while paying attention to something else.

The point of all this organization and preparation is to get into the most productive mental state possible. Psychologists call it flow [Csik2008]; athletes call it "being in the zone", while musicians talk about losing themselves in what they're playing. Whatever name you use, people produce much more per unit of time in this state than normal.

That's the good news. The bad news is that it takes roughly ten minutes to get back into a state of flow after an interruption, no matter how short the interruption was. This means that if you are interrupted half a dozen times per hour, you are *never* at your productive peak.

5.3 Peer Assessment

Asking people on a team to rate their peers is a common practice in industry. [Sond2012] surveyed the literature on student peer assessment, distinguishing between grading and reviewing. The benefits they found included increasing the amount, diversity, and timeliness of feedback, helping students exercise higher-level thinking, encouraging reflective practice, and supporting development of social skills. The concerns were predictable: validity and reliability, motivation and procrastination, trolls, collusion, and plagiarism. However, while these concerns are legitimate, the evidence shows that they aren't significant in class. For example, [Kauf2000] compared confidential peer ratings and grades on several axes for two undergraduate engineering courses and found that self-rating and peer ratings statistically agreed, that collusion (i.e., everyone giving their peers the same grades) wasn't significant, that students didn't inflate their self-ratings, and crucially, that ratings were not biased by gender or race.

One important variation on peer assessment and review is contributing student pedagogy, in which students produce artifacts to contribute to other students' learning. This can be developing a short lesson and sharing it with the class, adding to a question bank, or writing up notes from a particular lecture for in-class publication. For example, [Fran2018] found that students who made short videos to teach concepts to their peers had

a significant increase in their own learning compared to those who only studied the material or viewed the videos.

Another is calibrated peer review, in which a student reviews one or more examples using a rubric and compares their evaluation against the instructor's review of the same work [Kulk2013]. Only once student's evaluations are close enough to the instructor's are they allowed to start evaluating peers' actual work.

As long as evaluation is based on observables, rather than personality traits, peer assessment can actually be as accurate as assessment by TAs and other outsiders. "Observables" means that instead of asking, "Is the person outgoing," or "Does the person have a positive attitude," assessments should ask, "Does the person listen attentively during meetings," or, "Does the person attempt to solve problems before asking for help." The evaluation form in s:peereval shows a sample to get you started. To use it, rank yourself and each of your teammates, then calculate and compare scores.

5.4 Exercises

Learning Strategies (individual/20)

1. Which of the six learning strategies do you regularly use? Which ones do you not?
2. Write down three general concepts that you want your learners to master, and then give two specific examples of each. (This uses the "concrete examples" practice).
3. For each of those concepts, work backward from one of your examples to explain how the concept explains it. (This uses the "elaboration" practice).

Connecting Ideas (pairs/5)

This exercise is an example of using elaboration to improve retention. Pick a partner, and have each person independently choose an idea, then announce your ideas and try to find a four-link chain that leads from one to the other. For example, if the two ideas are "Saskatchewan" and "statistics", the links might be:

- Saskatchewan is a province of Canada;
- Canada is a country;
- countries have governments;
- governments are elected; and
- people try to predict election results using statistics

Convergent Evolution (pairs/15)

One practice that wasn't covered above is guided notes, which are instructor-prepared notes that cue students to respond to key information in a lecture or discussion. The cues can be blank spaces where students add information, asterisks next to terms students should define, etc.

Create 2–4 guided note cards for a lesson you have recently taught or are going to teach. Swap cards with your partner: how easy is it to understand what is being asked for? How long would it take to fill in the prompts?

Changing Minds (pairs/10)

[Kirs2013] argues that myths about digital natives, learning styles, and self-educators are all reflections of the mistaken belief that learners know what is best for them, and cautions that we may be in a downward spiral in which every attempt by education researchers to rebut these myths confirms their opponents' belief that learning science is pseudo-science. Pick one thing you have learned about learning so far in this book that surprised you or contradicted something you previously believed, and practice explaining it to a partner in 1–2 minutes. How convincing are you?

Flash Cards (individual/15)

Use sticky notes or anything else you have at hand to make up a dozen flash cards for a topic you have recently taught or learned, trade with a partner, and see how long it takes each of you to achieve 100% perfect recall. When you are done, set the cards aside, then come back after an hour and see what your recall rate is.

Using ADEPT (whole class/15)

Pick something you have recently taught or been taught and outline a short lesson that uses the five-step ADEPT method to introduce it.

The Cost of Multi-Tasking (pairs/10)

The Learning Scientists blog describes a simple experiment you can do without preparation or equipment other than a stopwatch to demonstrate the mental cost of multi-tasking. Working in pairs, measure how long it takes each person to do each of these three tasks:

- Count from 1 to 26.
- Recite the alphabet from A to Z.
- Interleave the numbers and letters, i.e., say, "1, A, 2, B, . . . " and so on.

Have each pair report their numbers: you will probably find that the third (in which you are multi-tasking) takes significantly longer than either of the component tasks.

Myths in Computing Education (whole class/20)

[Guzd2015b] presents a list of the top 10 mistaken beliefs about computing education. His list of things that many people believe, but which aren't true, includes the following:

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. Student evaluations are the best way to evaluate teaching.
4. Good teachers personalize education for students' learning styles.
5. A good CS teacher should model good software development practice, because their job is to produce excellent software engineers.
6. Some people are just naturally better programmers than others.

Have everyone vote +1 (agree), -1 (disagree), or 0 (not sure) for each point, then read the full explanations in the original article) and vote again. Which ones did people change their minds on? Which ones do they still believe are true, and why?

Calibrated Peer Review (pairs/20)

1. Create a 5–10 point rubric for grading programs of the kind you would like your learners to write that has entries like "good variable names", "no redundant code", and "properly-nested control flow".
2. Choose or create a small program that contains 3–4 violations of these entries.
3. Grade the program according to your rubric.
4. Have your partner grade the same program with the same rubric. What do they accept that you did not? What do they critique that you did not?

6 A Lesson Design Process

Most people design lessons like this:

1. Someone asks you to teach something you haven't thought about in 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 and promise yourself that you'll be more organized next time.

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 write tests to check that the software is doing the right thing. 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 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 backward design; developed independently in [Wigg2005, Bigg2011, Fink2013], it is summarized in [McTi2013], and in simplified form, its steps are:

1. Brainstorm to get a rough idea of what you want to cover, how you're going to do it, what problems or misconceptions you expect to encounter, what's *not* going to be included, and so on. You may also want to draw some concept maps at this stage.

2. Create or recycle learner personas (discussed in the next section) to figure out who you are trying to teach and what will appeal to them. (This step can also be done first, before the brainstorming.)
3. Create formative assessments that will give the learners a chance to practice the things they're trying to learn and tell you and them whether they're making progress and where they need to focus their work.
4. Put the formative assessments in order based on their complexity and dependencies to create a course outline.
5. Write just enough to get learners from one formative assessment to the next. Each hour in the classroom will then consist of three or four such episodes.

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. It is *not* the same thing as "teaching to the test". When using backward design, 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. The outcomes of those summative assessments directly affect the teachers' pay and promotion, which means teachers have an incentive to focus on having learners pass test rather than on helping them learn.

Measure... And Then?

[Gree2014] argues that this focus on measurement is appealing to those with the power to set the tests, but unlikely to improve outcomes unless it is coupled with support for teachers to make improvements based on test outcomes. The latter is often missing because large organizations usually value uniformity over productivity [Scot1998]; we will return to this topic in Chapter 8.

It's important to note that while lesson design is *described* as a sequence, it's almost never *done* that way: we may, for example, change our mind about what we want to teach based on something that occurs to us while we're writing an MCQ, or re-assess who we're trying to help once we have a lesson outline. However, it's important that the notes we leave behind present things in the order described above, because that's the easier way for whoever has to use or maintain the lesson to retrace our thinking. The same rewriting of history is useful for the same reasons in software design and many other fields [Parn1986].

Appendix M presents the design notes for this version of this book. A few things have been added, dropped, or rearranged, but what you are reading now matches the plan pretty closely.

6.1 Learner Personas

A key step in the lesson design process described above is figuring out who your audience is. One way to do this is to write two or three learner personas. This technique is borrowed from user interface designers, who create short profiles of typical users to help them think about their audience.

Learner personas have five parts:

1. the person's general background,
2. what they already know,
3. what *they* think they want to do (as opposed to what someone who already understands the subject thinks),
4. how the course will help them, and
5. any special needs they might have.

The personas in Section 1.1 have the five points listed above, rearranged to flow more readably; a learner persona for a weekend workshop aimed at 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 well, but still struggles sometimes to keep up with spoken conversation (especially if it involves a lot of new jargon).

A Gentle Reminder

When designing lessons, you must always remember that you are not your learners. *You may be younger (if you're teaching seniors) or wealthier (and therefore able to afford to download videos without foregoing a meal to pay for the bandwidth), but you are almost certainly more knowledgeable about technology. Don't assume that you know what they need or will understand: ask them, and pay attention to their answer. After all, it's only fair that learning should go both ways.*

Rather than writing new personas for every lesson or course, it's common for teachers to create and share a handful that cover everyone they are likely to teach, then pick a few from that set to describe who particular material is

intended for. When personas are used this way, they become a convenient shorthand for design issues: when speaking with each other, teachers can say, "Would Jorge understand why we're doing this?" or, "What installation problems would Jorge face?"

Brainstorming the broad outlines of what you're going to teach and then deciding who you're trying to help is one approach; it's equally valid to pick an audience and then brainstorm their needs. Either way, [Guzd2016] offers the following guidance:

1. Connect to what learners know.
2. Keep cognitive load low.
3. Use authentic tasks (see Section 10.1).
4. Be generative and productive.
5. Test your ideas rather than trusting your instincts.

Of course, one size won't fit all. [Alha2018] reported improvement in learning outcomes and student satisfaction in a course for students from a variety of academic backgrounds which allowed them to choose between different domain-related assignments. It's extra work to set up and grade, but that's manageable if the projects are open-ended (so that they can be used repeatedly) and if the load is shared with other teachers (Section 6.3). Other work has shown that building courses for science students around topics as diverse as music, data science, and cell biology will also improve outcomes [Pete2017, Dahl2018, Ritz2018].

6.2 Learning Objectives

Formative and summative assessments help teachers figure out what they're going to teach, but in order to communicate that to learners and other teachers, a course description should also have learning objectives. These help 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 which 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 learning outcomes with learning objectives.

A learning objective is a single sentence describing how a learner will demonstrate what they have learned once they have successfully completed a lesson. More specifically, it 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 you, your fellow teachers, 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 way to understand what makes for a good learning objective is to see how a poor one can be improved:

- "The learner will be given opportunities to learn good programming practices." This describes the lesson's content, not the attributes of successful students.
- "The learner will have a better appreciation for good programming practices." This 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.
- "The learner will understand how to program in R." While this starts with an active verb, it doesn't define the level of learning, and the subject of learning is still too vague for assessment.
- "The learner will write one-page data analysis scripts to read, filter, summarize, and print results for tabular data using R and RStudio." This starts with an active verb, defines the level of learning, and provides context to ensure that outcomes can be assessed.

When it comes to choosing verbs, many teachers use Bloom's taxonomy. First published in 1956, it was updated at the turn of the century [Ande2001], and is the most widely used framework for discussing levels of understanding. Its most recent form has six categories; the list below defines each, and gives a few of the verbs typically used in learning objectives written for each:

Remembering: Exhibit memory of previously learned material by recalling facts, terms, basic concepts, and answers. *(recognize, list, describe, name, find)*

Understanding: Demonstrate understanding of facts and ideas by organizing, comparing, translating, interpreting, giving descriptions, and stating main ideas. *(interpret, summarize, paraphrase, classify, explain)*

Applying: Solve new problems by applying acquired knowledge, facts, techniques and rules in a different way. *(build, identify, use, plan, select)*

Analyzing: Examine and break information into parts by identifying motives or causes. Make inferences and find evidence to support generalizations. *(compare, contrast, simplify)*

Evaluating: Present and defend opinions by making judgments about information, validity of ideas, or quality of work based on a set of criteria. *(check, choose, critique, prove, rate)*

Creating: Compile information together in a different way by combining elements in a new pattern or proposing alternative solutions. *(design, construct, improve, adapt, maximize, solve)*

[Masa2018] found that even experienced educators have trouble agreeing on how to classify a question or idea according to Bloom's Taxonomy, but the material in most introductory programming courses fits into the first four of these levels; only once that material has been mastered can learners start to think about evaluating and creating. (As Daniel Willingham has said, people can't think without something to think about [Will2010].)

Another way to think about learning objectives comes from [Fink2013], which defines learning in terms of the change it is meant to produce in the learner. Fink's Taxonomy also has six categories, but unlike Bloom's, they are complementary rather than hierarchical:

Foundational Knowledge: understanding and remembering information and ideas. *(remember, understand, identify)*

Application: skills, critical thinking, managing projects. *(use, solve, calculate, create)*

Integration: connecting ideas, learning experiences, and real life. *(connect, relate, compare)*

Human Dimension: learning about oneself and others. *(come to see themselves as, understand others in terms of, decide to become)*

Caring: developing new feelings, interests, and values. *(get excited about, be ready to, value)*

Learning How to Learn: becoming a better student. *(identify source of information for, frame useful questions about)*

A set of learning objectives based on this taxonomy for an introductory course on HTML and CSS might be:

By the end of this course, learners will be able to:

- *Explain the difference between markup and presentation, what CSS properties are, and how CSS selectors work.*
- *Write and style a web page using common tags and CSS properties.*
- *Compare and contrast authoring with HTML and CSS to authoring with desktop publishing tools.*
- *Identify issues in sample web pages that would make them difficult for the visually impaired to interact with and provide appropriate corrections.*
- *Explain the role that JavaScript plays in styling web pages and want to learn more about how to use it.*

6.3 Maintainability

It takes a lot of effort to create a good lesson, but once it has been built, someone needs to maintain it, and doing that is a lot easier if it has been built in a maintainable way. But what exactly does "maintainable" mean? The short answer is that a lesson is maintainable if it's cheaper to update it than to replace it. This equation depends on three factors. The first is *how well documented the course's design is.* If the person doing maintenance doesn't know (or doesn't remember) what the lesson is supposed to accomplish or why topics are introduced in a particular order, it will take her more time to update it. One of the reasons to use the design process described earlier in this chapter is to capture decisions about why each course is the way it is.

The second factor is *how easy it is for collaborators to collaborate technically.* Teachers usually share material by mailing PowerPoint files to each other or putting them in a shared drive. Collaborative writing tools like Google Docs and wikis are a big improvement, as they allow many people to update the same document and comment on other people's updates. The version control systems used by programmers, such as GitHub, are another big advance, since they let any number of people work independently and then merge their changes back together in a controlled, reviewable way. Unfortunately, version control systems have a long, steep learning curve, and (still) don't handle common office document formats.

The third factor, which is the most important in practice, is *how willing people are to collaborate.* The tools needed to build a "Wikipedia for lessons" have been around for twenty years, but most teachers still don't write and share lessons the way that they write and share encyclopedia entries, even though commons-based lesson development and maintenance actually works very well (Section 13.4 and Section C.3).

[Leak2017] interviewed 17 computer science teachers to find out why they don't use resource sharing sites. They found that most of the reasons were operational. For example, respondents said that sites need good landing pages that ask "what is your current role?" and "what course and grade level are you interested in?", and should display all their resources in context, since visitors may be new teachers who are struggling to connect the dots themselves. They also said that sites should allow anonymous posts on discussion forums to reduce fear of looking foolish in front of peers.

One interesting observation is that while teachers don't collaborate at scale, they *do* remix by finding other people's materials online or in textbooks and reworking them. That suggests that the root problem may be a flawed analogy: rather than lesson development being like writing a Wikipedia article or some open source software, perhaps it's more like sampling in music.

If this is true, then lessons may be the wrong granularity for sharing, and collaboration might be more likely to take hold if the thing being

collaborated on was smaller. This fits well with Caulfield's theory of choral explanations. He argues that sites like Stack Overflow succeed because they provide a chorus of answers for every question, each of which is most suitable for a slightly different questioner. If Caulfield is right, the lessons of tomorrow may include guided tours of community-curated Q&A repositories designed to accommodate learners at widely different levels.

6.4 Exercises

Create Learner Personas (small groups/30)

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

Classify Learning Objectives (pairs/10)

Look at the example learning objectives given for an introductory course on HTML and CSS in Section 6.2 and classify each according to Bloom's Taxonomy. Compare your answers with those of your partner: where did you agree and disagree, and why?

Write Learning Objectives (pairs/20)

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

Write More Learning Objectives (pairs/20)

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

Building Lessons by Subtracting Complexity (individual/20)

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 code that you give them. This typically includes things like importing libraries and loading data.

Take a program or web page that you want your learners to be able to create on their own at the end of a lesson and work backward to break it

into digestible parts. How many are there? What key idea is introduced by each one?

Inessential Weirdness (individual/15)

Betsy Leondar-Wright coined the phrase "inessential weirdness" to describe things groups do that aren't really necessary, but which alienate people who aren't members of that group. Sumana Harihareswara later used this notion as the basis for a talk on inessential weirdnesses in open source software, which includes things like making disparaging comments about Microsoft Windows, command-line tools with cryptic names, and the command line itself. Take a few minutes to read these articles, then make a list of inessential weirdnesses you think your learners might encounter when you first teach them. How many of these can you avoid with a little effort?

PRIMM (individual/15)

One approach to introducing new ideas in computing is PRIMM: **P**redict a program's behavior or output, **R**un it to see what it actually does, **I**nvestigate why it does that (e.g., by stepping through it in a debugger or drawing the flow of control), **M**odify it (or its inputs), and then **M**ake something similar from scratch. Pick something you have recently taught or been taught and outline a short lesson that follows these five steps.

Evaluating Lessons (pairs/20)

[Mart2017] specifies eight dimensions along which lessons can be evaluated:

Closed vs. open: is there a well-defined path and endpoint, or are learners exploring?
Cultural relevance: how well is the task connected to things they do outside class?
Recognition: how easily can the learner share the product of their work?
Space to play: seems to overlap closed vs. open
Driver shift: how often are learners in control of the learning experience (tight cycles of "see then do" score highly)
Risk reward: to what extent is taking risks rewarded or recognized?
Grouping: is learning individual, in pairs, or in larger groups?
Session shape: theater-style classroom, dinner seating, free space, public space, etc.

Working with a partner, go through a set of lessons you have recently taught, or have recently been taught, and rate them as "low", "medium", "high", or "not applicable" on each of these criteria. Which two criteria are most important to you personally as a teacher? As a learner?

Concrete-Representational-Abstract (pairs/15)

Concrete-Representational-Abstract (CRA) is another approach to introducing new ideas that is used primarily with younger learners. The first step is the concrete stage, and involves physically manipulating objects to solve a problem (e.g., piling blocks to do addition). In the representational stage, images are used to represent those objects, and in the final abstract stage, the learner uses numbers or symbols.

1. Write each of the numbers 2, 7, 5, 10, 6 on a sticky note.
2. Simulate a loop that finds the largest value by looking at each in turn (concrete).
3. Sketch a diagram of the process you used, labelling each step (representational).
4. Write instructions that someone else could follow to go through the same steps you used (abstract).
5. Compare your representational and abstract materials with those of your partner.

7 Actionable Approximations of the Truth

Every instructor needs three things:

- content knowledge, such as how to program;
- general pedagogical knowledge, such as an understanding of the psychology of learning; and
- pedagogical content knowledge (PCK), which is the domain-specific knowledge of how to teach a particular concept to a particular audience.

In computing, PCK includes things like what examples to use when teaching how parameters are passed to a function, or what misconceptions about nesting HTML tags are most common. This chapter summarizes some results from research into teaching and learning programming that will add to your store of PCK.

Computing education research is still a young discipline: while the American Society for Engineering Education was founded in 1893, and the National Council of Teachers of Mathematics in 1920, the Computer Science Teachers Association didn't exist until 2005. The simple truth is that we don't know as much about how people learn to program as we do about how they learn to read, play a sport, or do basic arithmetic. However, conferences like SIGCSE, ITiCSE and ICER are delivering an ever-increasing stream of rigorous, insightful studies with practical application. (For those interested in methods these studies rely on, [Ihan2016] summarizes the ones used most often.)

As with all research, though, some caution is required to interpret these results. Most studies look at school children and university undergraduates, both because those are the populations that researchers have easiest access to [Henr2010] and because those are the ages at which people most often learn to program; less is known about adults learning to program in free-range settings.

Theories may change as more and better data becomes available, so if this was an academic treatise, it would preface most claims with statements like, "Research may seem to indicate that. . . " However, since actual teachers

in actual classrooms have to make decisions regardless of whether research has clear answers yet or not, this chapter presents actionable approximations of the truth rather than nuanced perhapses.

Jargon
Like any specialty, computing education research has its jargon. The term CS1 refers to an introductory semester-long programming course in which learners meet variables, loops, and functions for the first time, while CS2 refers to a second course that covers basic data structures like stacks and queues. The term CS0 is also now being used to refer to an introductory course for people without any prior experience who aren't intending to continue with computing (at least not right away).
A CS1 course is often useful for undergraduates in other disciplines; a CS2 course designed for computer science learners is usually less relevant for artists, ecologists, and other end-user programmers, but is sometimes the only next step available at their institution. Full definitions for these terms and others can be found in the ACM Curriculum Guidelines.

7.1 How Do Novices Program?

[Solo1984, Solo1986] pioneered the exploration of novice and expert programming strategies. The key finding is that experts know both "what" and "how", i.e., they understand what to put into programs *and* they have a set of patterns or plans to guide their construction. Novices lack both, but most teachers focus solely on the former, even though bugs are often caused by not having a strategy for solving the problem rather than to lack of knowledge about the language.

The most important recommendation in this chapter is therefore to *show learners how to program*. This is consistent with the work on cognitive load theory presented in Chapter 4, and [Mull2007b] is just one of many studies proving its benefits; live coding (Section 8.4) is effective in part because it puts "how" front and center.

When demonstrating the act of programming, teachers should emphasize the importance of small steps with frequent feedback [Blik2014]. They should also emphasize the importance of picking a plan and sticking to it rather than making more-or-less random changes to the program and hoping that they'll work—as [Spoh1985] found, merging plans and/or goals can yield bugs because of goals being dropped or fragmented.

Another aspect of "how" that teachers should present and discuss is the order in which code is written. [Ihan2011] describes a tool for solving Parsons Problems. They found that experienced programmers often drag the method signature to the beginning, then add the majority of the control flow (i.e., loops and conditionals), and only then add details like variable initialization and handling of corner cases. This out-of-order authoring is

foreign to novices, who read and write code in the order it's presented on the page; again, one of the benefits of live coding (Section 8.4) is that it gives them a chance to see the sequence that more advanced programmers actually use.

Roles of Variables

One body of work that I have found very useful in teaching programming plans to novices is the collection of single-variable design patterns in [Kuit2004, Byck2005, Saja2006]. Labelling the parts of novices' programs gives them a vocabulary to think with and a set of programming plans for constructing code of their own. The patterns are listed on the Roles of Variables website, which also includes examples of each:

Fixed value: A data item that does not get a new proper value after its initialization.
Stepper: A data item stepping through a systematic, predictable succession of values.
Walker: A data item traversing in a data structure.
Most-recent holder: A data item holding the latest value encountered in going through a succession of unpredictable values, or simply the latest value obtained as input.
Most-wanted holder: A data item holding the best or otherwise most appropriate value encountered so far.
Gatherer: A data item accumulating the effect of individual values.
Follower: A data item that gets its new value always from the old value of some other data item.
One-way flag: A two-valued data item that cannot get its initial value once the value has been changed.
Temporary: A data item holding some value for a very short time only.
Organizer: A data structure storing elements that can be rearranged.
Container: A data structure storing elements that can be added and removed.

7.2 How Do Novices Debug and Test?

A decade ago, [McCa2008] wrote, "It is surprising how little page space is devoted to bugs and debugging in most introductory programming textbooks." Little has changed since: there are hundreds of books on compilers and operating systems, but only a handful about debugging, and I have never seen an undergraduate course on the subject. One reason is that debugging is a "how" rather than a "what"; again, one of the benefits of live coding is that it gives teachers a chance to demonstrate the process in a way that textbooks cannot (Section 8.4).

[List2004, List2009] found that many novices struggled to predict the output of short pieces of code and to select the correct completion of the code from a set of possibilities when told what it was supposed to do. More recently, [Harr2018] found that the gap between being able to trace code and being able to write it has largely closed by CS2, but that novices who still have a gap (in either direction) are likely to do poorly in the course.

Our second recommendation is therefore to *teach novices how to debug*. [Fitz2008, Murp2008] found that good debuggers were good programmers, but not all good programmers were good at debugging. Those who were used a symbolic debugger to step through their programs, traced execution by hand, wrote tests, and re-read the spec frequently, which are all teachable habits. However, tracing execution step by step was sometimes used ineffectively: for example, a novice might put the same `print` statement in both parts of an `if-else`. Novices would also comment out lines that were actually correct as they tried to isolate a problem; teachers can make both of these mistakes deliberately, point them out, and correct them to help novices get past them.

Teaching novices how to debug can also help make classes easier to manage. [Alqa2017] found that learners with more experience solved debugging problems significantly faster, but times varied widely: 4–10 minutes was a typical range for individual exercises, which means that some learners need 2–3 times longer than others to get through the same exercises. Teaching the slower learners what the faster ones are doing will help make the group's overall progress more uniform.

Debugging depends on being able to read code, which multiple studies have shown is the single most effective way to find bugs [Basi1987, Keme2009, Bacc2013]. The code quality rubric developed in [Steg2014, Steg2016a], which is online at [Steg2016b], is a good checklist of things to look for, though it is best presented in chunks rather than all at once.

Having learners read code and summarize its behavior is a good exercise (Section 5.1), but often takes too long to be practical in class. Having them predict a program's output just before it is run, on the other hand, helps reinforce learning (Section 9.11) and also gives them a natural moment to ask "what if" questions. Instructors or learners can also trace changes to variables as they go along (Figure 4), which [Cunn2017] found was effective.

When it comes to testing, novices seem just as reluctant to do it as professional programmers. There's no doubt it's valuable—[Cart2017] found that high-performing novices spent a lot of time testing, while low performers spent much more time working on code with errors—and many instructors require learners to write tests for assignments. The question is, how well do they do this?

One answer comes from [Bria2015], which scored learners' programs by how many teacher-provided test cases those programs passed, and

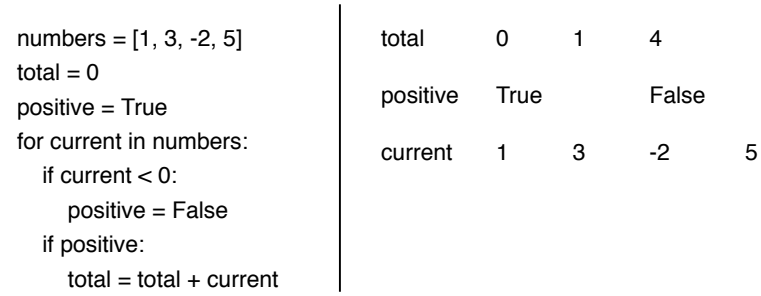

Figure 4: Tracing the Values of Variables

conversely scores test cases written by learners according to how many deliberately-seeded bugs they caught. They found that novices' tests often have low coverage (i.e., they don't test most of the code) and that they often test many things at once, which makes it hard to pinpoint the causes of errors.

Another answer comes from [Edwa2014b], which looked at all of the bugs in all novices' code submissions combined and identified those detected by the novices' test suite. They found that novices' tests only detected an average of 13.6% of the faults present in the entire program population. What's more, 90% of the novices' tests were very similar, which indicates that novices mostly write tests to confirm that code is doing what it's supposed to rather than to find cases where it isn't.

One approach to teaching better testing practices is to define a programming problem by providing a set of tests to be passed rather than through a written description (Section 12.1). Before doing this, though, take a moment to look at how many tests you've written for your own code recently, and then decide whether you're teaching what you believe people should do, or what they (and you) actually do.

7.3 What Misconceptions Do Novices Have?

Chapter 2 explained why clearing up novices' misconceptions is just as important as teaching them strategies for solving problems. The biggest misconception novices have—sometimes called the "superbug" in coding—is the belief that they can communicate with a computer in the same way that they would with a human being, i.e., that the computer understands intention the way that a human being would [Pea1986]. Our third recommendation is therefore to *teach novices that computers don't understand*

programs, i.e., that calling a variable "cost" doesn't guarantee that its value is actually a cost.

[Sorv2018] presents over 40 other misconceptions that instructors can also try to clear up, many of which are also discussed in [Qian2017]'s survey. One is the belief that variables in programs work the same way they do in spreadsheets, i.e., that after executing:

```
grade = 65
total = grade + 10
grade = 80
print(total)
```

the value of `total` will be 90 rather than 75 [Kohn2017]. (This is an example of the way in which novices construct a plausible-but-wrong mental model by making analogies.) Other misconceptions include:

- A variable holds the history of the values it has been assigned, i.e., it remembers what its value used to be.
- Two objects with the same value for a `name` or `id` attribute are guaranteed to be the same object.
- Functions are executed as they are defined, or are executed in the order in which they are defined.
- A `while` loop's condition is constantly evaluated, and the loop stops as soon as it becomes false. Conversely, the conditions in `if` statements are also constantly evaluated, and their statements are executed as soon as the condition becomes true, no matter where the flow of control is at the time.
- Assignment moves values, i.e., after `a = b`, the variable `b` is empty.

Instead of looking directly at misconceptions, [Muhl2016] analyzed 350 concept maps and compared those who had done a CS course and those who had not. Unsurprisingly, they found that the maps drawn by those with previous experience looked more like the maps experts would draw, but the details highlighted what exactly learners were taking away from their lessons: "program" was a central concept in both sets of concept maps, but the next most central for those with prior exposure were "class" (in the object-oriented sense) and "data structure", while for those without, they were "processor" and "data".

7.4 What Mistakes Do Novices Make?

The mistakes novices make can tell us what to prioritize in our teaching, but it turns out that most teachers don't know how common different kinds of mistakes actually are. The largest study of this is [Brow2017]'s study of novice Java programs, which found that mismatched quotes and parentheses are the most common type of error, but also the easiest to fix, while some

mistakes (like putting the condition of an `if` in `{}` instead of `()`) are most often made only once. Unsurprisingly, mistakes that produce compiler errors are fixed much faster than ones that don't.

Some mistakes, however, are made many times, like invoking methods with the wrong arguments (e.g., passing a string instead of an integer). One caution when reading this research is how important it is to distinguish mistakes from work in progress: for example, an empty `if` statement or a method that's defined but not yet used may be a sign of incomplete code rather than an error.

[Brow2017] also compared the mistakes novices actually make with what their teachers thought they made. They found that, "... educators formed only a weak consensus about which mistakes are most frequent, that their rankings bore only a moderate correspondence to the students in the... data, and that educators' experience had no effect on this level of agreement." For example, mistaking = (assignment) and == (equality) in loop condition tests wasn't nearly as common as most teachers believed.

Not Just for Code

[Park2015] collected data from an online HTML editor during an introductory web development course. Nearly all learners made syntax errors that remained unresolved weeks into the course. 20% of these errors related to the relatively complex rules that dictate when *it is valid for HTML elements to be nested in one another, but 35% related to the simpler tag syntax determining* how *HTML elements are nested. (The tendency of many instructors to say, "But the rules are simple," is a good example of expert blind spot discussed in Chapter 3...)*

7.5 What Are We Teaching Them Now?

Very little is known about what coding bootcamps and other free-range initiatives teach, in part because many are reluctant to share their curriculum. We do know more about what is taught in schools: [Luxt2017] surveyed the topics included in introductory programming courses, categorized their findings under a dozen headings, and ranked them by frequency:

Topic	Number of Courses	(%)
Programming Process	90	(87%)
Abstract Programming Thinking	65	(63%)
Data Structures	41	(40%)
Object-Oriented Concepts	37	(36%)
Control Structures	34	(33%)
Operations & Functions	27	(26%)
Data Types	24	(23%)
Input/Output	18	(17%)

Libraries	15	(15%)
Variables & Assignment	14	(14%)
Recursion	10	(10%)
Pointers & Memory Management	5	(5%)

This paper also showed how concepts are connected. For example, it's impossible to explain how operator precedence works without first explaining a few operators, and difficult to explain those in a meaningful way without first introducing variables (because otherwise you're comparing constants in expressions like `5<3`, which is confusing).

Similarly, [Rich2017] reviewed a hundred articles to find learning trajectories for computing classes in elementary and middle schools, and presented results for sequencing, repetition, and conditionals. These are essentially collective concept maps, as they combine and rationalize the implicit and explicit thinking of many different educators. Figure 5 shows the learning trajectories for conditionals.

But there can be a world of difference between what instructors teach and what learners learn, and study after study has shown that teaching evaluations don't correlate with actual learning outcomes [Star2014, Uttl2017]. To find out how much novices are actually learning, we therefore have to use other measures or do direct studies. Taking the former approach, roughly two-thirds of post-secondary students pass their first computing course, with some variations depending on class size and so on, but with no significant differences over time or based on language [Benn2007a, Wats2014].

How does prior experience affect these results? To find out, [Wilc2018] compared the performance and confidence of novices with and without prior programming experience in CS1 and CS2. They found that novices with prior experience outscored novices without by 10% in CS1, but those differences disappeared by the end of CS2. They also found that women with prior exposure outperformed their male peers in all areas, but were consistently less confident in their abilities; we will return to this issue in Section 10.4.

As for direct studies of how much novices learn, [McCr2001] presented a multi-site international study, which was later replicated by [Utti2013]. According to the first study, "... the disappointing results suggest that many students do not know how to program at the conclusion of their introductory courses." More specifically, "For a combined sample of 216 students from four universities, the average score was 22.89 out of 110 points on the general evaluation criteria developed for this study." This result may say as much about teachers' expectations as it does about student ability, but either way, our fourth recommendation is to *measure and track results* in ways that can be compared over time, so that you can tell if your lessons are becoming more or less effective.

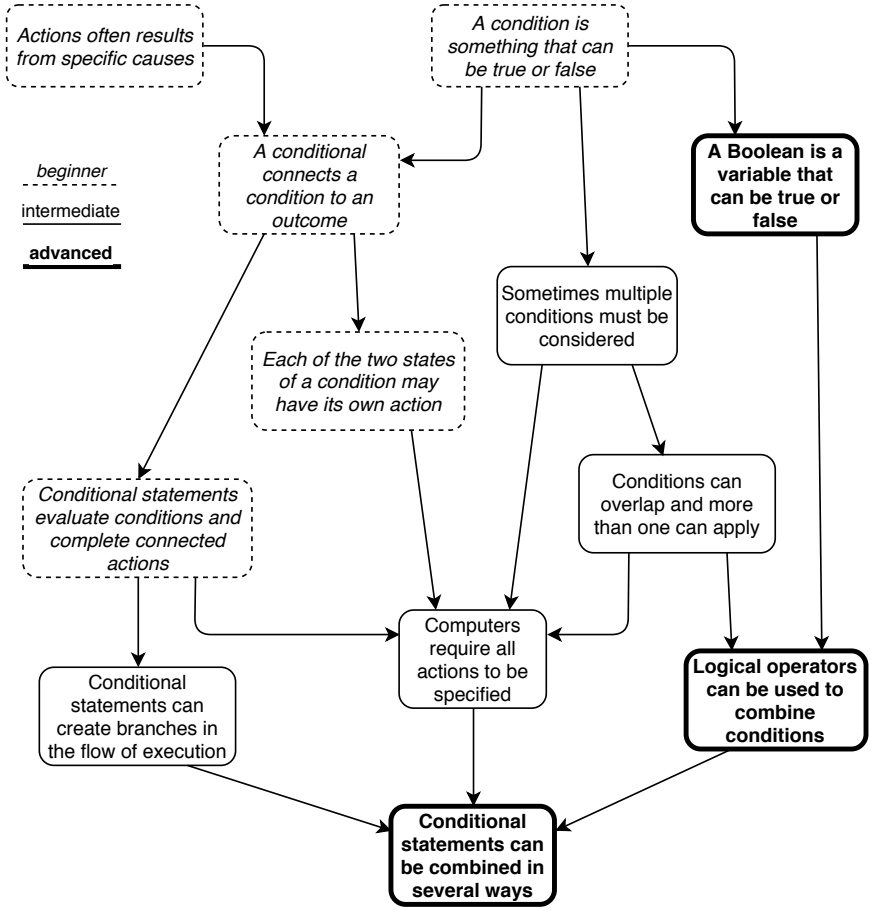

Figure 5: Learning Trajectory for Conditions (from [Rich2017])

7.6 Do Languages Matter?

The short answer is “yes”: novices learn to program faster and also learn more using blocks-based tools like Scratch (Figure 6) that make syntax errors impossible [Wein2017b]. And block interfaces encourage exploration in a way that text does not; like all good tools, Scratch can be learned accidentally [Malo2010].

Our fifth recommendation is therefore to *start children and teens with blocks-based interfaces* before moving to text-based systems. The age qualification is there because Scratch (deliberately) looks like it’s meant for younger users; while imitators like Blockly look more grown-up, it can still be hard to convince adults to take them seriously.

Figure 6: Scratch (from https://opensource.com/article/18/4/designing-game-scratch-open-jam)

Scratch has probably been studied more than any other programming tool, and we know a great deal about how it is used. As just one example, [Aiva2016] analyzed over 250,000 Scratch projects and found (among other things) that about 28% of projects have some blocks that are never called or triggered. The authors hypothesize that users may be using them as a scratchpad to keep bits of code they don’t (yet) want to throw away.

[Grov2017, Mlad2017] studied novices learning about loops in Scratch, Logo, and Python, and found that misconceptions about loops are minimized when using a block-based language rather than a text-based language. What’s more, as tasks become more complex (such as using nested loops) the differences become larger.

[Wein2017a] studied people using a tool that allowed them to switch between blocks and text for programming. They found that learners tend to migrate from blocks to text over time, but when learners shifted from text to blocks, their next action was to add a new type of command. This may be because browsing available commands is easier with blocks, or because blocks make syntax errors with unfamiliar new commands impossible. The authors say, "While it is often claimed that blocks-based programming environments offer the advantage of reducing syntax errors, our findings suggest that blocks also offer information about what is possible in the space and provide a low-stakes means of exploring unfamiliar code." New tools like Stride are trying to smooth the transition between blocks and text even further; when combined with programming notebooks like Jupyter and Stencila, they may eventually eliminate the distinction altogether.

Harder Than Necessary
[Stef2013] has shown that the creators of programming language make those languages harder to learn by not doing basic usability testing. For example, "... the three most common words for looping in computer science, `for`, `while`, and `foreach`, were rated as the three most unintuitive choices by non-programmers." More fundamentally, their work shows that C-style syntax (as used in Java and Perl) is just as hard for novices to learn as a randomly-designed syntax, but that the syntax of languages such as Python and Ruby is significantly easier to learn, and the syntax of their own language, Quorum, is easier still, because they are testing each new feature before adding it to the language. ([Stef2017] is a useful brief summary of what we actually know about designing programming languages and why we believe it's true.)

Object-Oriented and Functional Programming

Objects and classes are power tools for experienced programmers, and many educators advocate an "objects first" approach to teaching programming (though they sometimes disagree on exactly what that means [Benn2007b]). [Sorv2014] describes and motivates this approach, and [Koll2015] describes three generations of tools designed to support novice programming in object-oriented environments.

Introducing objects early has a few special challenges. [Mill2016b] found that most novices using Python struggled to understand `self` (which refers to "this object"): they omitted it in method definitions, failed to use it when referencing object attributes, or both. Object reference errors were also more common than other errors; the authors speculate that this is partly due to the difference in syntax between `obj.method(param)` and `def method(self, param)`. [Rago2017] found something similar in high school students, and that high school teachers often weren't clear on the concept either.

Another approach is exemplified by the Bootstrap project, which is based on the functional programming paradigm. This work draws on a rich tradition going back to languages like Scheme and Lisp, and to classic textbooks like [Fell2001, Frie1995, Abel1996]. If functional programming continues to gain ground among professional programmers, this approach may grow more popular for teaching.

On balance, we recommend that instructors *use procedural languages* to start with, i.e., that defining classes and using higher-order functions not be taught until learners understand basic control structures and data types. How quickly these topics should be introduced depends on the audience: if learners want to build web applications in JavaScript, for example, they're going to have to master callbacks much earlier than if they want to generate reports using C#.

Type Declarations

Programmers have argued for decades about whether variables' data types should have to be declared or not. One recent empirical finding is [Gao2017], which found that about 15% of bugs in JavaScript programs could be caught by requiring type declarations, which is either high or low depending on what answer you wanted in the first place.

However, programming and learning to program are different activities, and results from the former don't necessarily apply to the latter. [Endr2014] found that requiring novices to declare variable types does add some complexity to programs, but it pays off fairly quickly by acting as documentation for a method's use—in particular, by forestalling questions about what's available and how to use it.

We don't know enough yet to recommend typed or untyped languages for novices. Now that Python allows optional typing, though, it may be feasible for researchers to explore whether it can or should be introduced gradually.

Does Variable Naming Style Matter?

[Kern1999] says, "Programmers are often encouraged to use long variable names regardless of context. This is a mistake: clarity is often achieved through brevity." Lots of programmers believe this, but [Hofm2017] found that using full words in variable names led to an average of 19% faster comprehension compared to letters and abbreviations.

In contrast, [Beni2017] found that using single-letter variable names didn't affect novices' ability to modify code. This may be because their programs are shorter than professionals', or because some single-letter variable names have implicit types and meanings: most programmers assume `i`, `j`,

and `n` are integers, and `s` is a string, while `x`, `y`, and `z` are either floating-point numbers or integers more or less equally.

How important is this? [Bink2012] reported a series of studies that found that reading and understanding code is fundamentally different from reading prose: "... the more formal structure and syntax of source code allows programmers to assimilate and comprehend parts of the code quite rapidly independent of style. In particular... beacons and program plans play a large role in comprehension." It also found that experienced developers are relatively unaffected by identifier style, so our recommendation is just to use consistent style in all examples.

Since most languages have style guides (e.g., PEP 8 for Python) and tools to check that code follows these guidelines, our full recommendation is to *use tools to ensure that all code examples adhere to a consistent style*.

7.7 Does Better Feedback Help?

Incomprehensible error messages are a major source of frustration for novices (and sometimes for experienced programmers as well). Several researchers have therefore explored whether better error messages would help alleviate this. For example, [Beck2016] rewrote some of the Java compiler's messages so that instead of:

```
C:\stj\Hello.java:2: error: cannot find symbol
        public static void main(string[ ] args){
^
1 error
Process terminated ... there were problems.
```

learners would see:

```
Looks like a problem on line number 2.
If "string" refers to a datatype, capitalize the 's'!
```

Sure enough, novices given these messages made fewer repeated errors and fewer errors overall.

[Bari2017] went further and used eye tracking to show that despite the grumblings of compiler writers, people really do read error messages—in fact, they spend 13–25% of their time doing this. However, reading error messages turns out to be as difficult as reading source code, and how difficult it is to read the error messages strongly predicts task performance. Instructors should therefore *give learners practice in reading and interpreting error messages*. [Marc2011] has a rubric for responses to error messages that can be useful in grading such exercises.

Does Visualization Help?

The idea of visualizing programs is perennially popular, and tools like [Guo2013] (a web-based tool for visualizing the execution of Python programs) and Loupe (which shows how JavaScript's event loop works) are both useful teaching aids. However, people learn more from constructing visualizations than they do from viewing visualizations constructed by others [Stas1998, Ceti2016], so does visualization actually help learning?

To answer this, [Cunn2017] replicated an earlier study of the kinds of sketching students do when tracing code execution. They found that not sketching at all correlates with lower success, while tracing changes to variables' values by writing new values near their names as they change was the most effective strategy (Figure 4).

One possible confounding effect they checked was time: since sketchers take significantly more time to solve problems, do they do better just because they think for longer? The answer is no: there was no correlation between the time taken and the score achieved. Our recommendation is therefore to *teach students to trace variables' values when debugging*.

Flowcharts
One often-overlooked finding about visualization is that students understand flowcharts better than pseudocode if both are equally well structured *[Scan1989]. Earlier work showing that pseudocode outperformed flowcharts used structured pseudocode and tangled flowcharts; when the playing field was levelled, novices did better with the graphical representation.*

7.8 What Else Can We Do to Help?

[Viha2014] examined the average improvement in pass rates of various kinds of intervention in programming classes. As they themselves point out, there are many reasons to take their findings with a grain of salt: the pre-change teaching practices are rarely stated clearly, the quality of change is not judged, and only 8.3% of studies reported negative findings, so either there is positive reporting bias or the way we're teaching right now is almost the worst way possible and anything would be an improvement. And like many other studies discussed in this chapter, they were only looking at university classes, so their findings may not generalize to other groups.

With all those caveats in mind, they found ten things instructors can do to improve outcomes (Figure 7):

Collaboration: Activities that encourage student collaboration either in classrooms or labs.
Content Change: Parts of the teaching material were changed or updated.

Contextualization: Course content and activities were aligned towards a specific context such as games or media.

CS0: Creation of a preliminary course to be taken before the introductory programming course; could be organized only for some (e.g., at-risk) students.

Game Theme: A game-themed component was introduced to the course.

Grading Scheme: A change in the grading scheme; the most common change was to increase the amount of points rewarded from programming activities, while reducing the weight of the course exam.

Group Work: Activities with increased group work commitment such as team-based learning and cooperative learning.

Media Computation: Activities explicitly declaring the use of media computation (Chapter 10).

Peer Support: Support by peers in form of pairs, groups, hired peer mentors or tutors.

Other Support: An umbrella term for all support activities, e.g. increased teacher hours, additional support channels, etc.

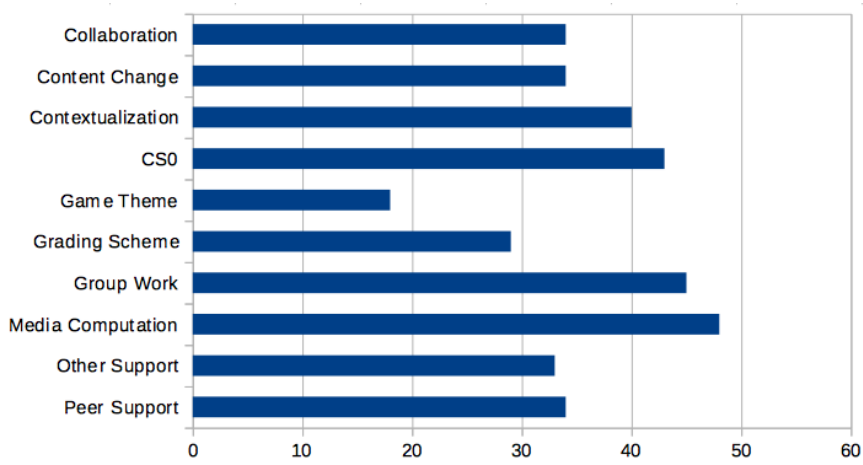

Figure 7: Effectiveness of Interventions

This list highlights the importance of cooperative learning. [Beck2013] looked at this specifically over three academic years in courses taught by two different instructors, and found significant benefits overall and for many subgroups: they not only had higher grades, they left fewer questions blank on the final exam, which indicates greater self-efficacy and willingness to try to debug things.

As noted earlier, writing code isn't the only way to teach people how to program. [Shel2017] reports that having novices work on computational creativity exercises improves grades at several levels. A typical exercise is to identify an everyday object (such as nail clipper, a paper clip, Scotch tape) and describe the object in terms of its inputs, outputs and functions. This kind of teaching is sometimes called "unplugged"; the CS Unplugged site has a collection of lessons and exercises for doing this.

7.9 Exercises

Checking for Common Errors (individual/20)

This list of common errors is taken from [Sirk2012]. Pick three, and write an exercise for each to check that learners *aren't* making that mistake.

Inverted assignment: The student assigns the value of the left-hand variable to the right-hand side variable, rather than the other way around.

Wrong branch: Even though the conditional evaluates to `False`, the student jumps to the `then` clause.

Wrong `False`: As soon as the conditional evaluates to `False` , the student returns `False` from the function.

Executing function instead of defining it: The student believes that a function is executed as it is defined.

Unevaluated parameters: The student believes the function starts running before the parameters have been evaluated.

Parameter evaluated in the wrong frame: The student creates parameter variables in the caller's frame, not in the callee's.

Failing to store return value: The student does not assign the return value in the caller.

Assignment copies object: The student creates a new object rather than copying a reference.

Method call without subject: The student tries to call a method from a class without first creating an instance of the class.

Mangled Code (pairs/15)

[Chen2017] describes exercises in which students reconstruct code that has been mangled by removing comments, deleting or replacing lines of code, moving lines, inserting extra unneeded lines, and so on. Student performance on these correlates strongly with performance on assessments in which students write code (i.e., whatever traditional assignments are measuring, these are measuring as well), but these questions require less (in-person) work to mark. Take the solution to a programming exercise you've created in the past, mangle it in two different ways, and swap with a partner.

The Rainfall Problem (pairs/10)

[Solo1986] introduced the Rainfall Problem: write a program that repeatedly reads in positive integers until it reads the integer 99999. After seeing 99999, the program should print out the average of the numbers seen. This problem has been used in many subsequent studies of programming [Fisl2014, Simo2013, Sepp2015].

Solve the Rainfall Problem in the programming language of your choice. Compare your solutions with those of your partner.

Roles of Variables (pairs/15)

Take a short program you have written (5–15 lines) and classify each of its variables using the categories defined in Section 7.1. Compare your classifications with those of a partner: where did you agree? When you disagreed, did you understand each other's view?

Choose Your Own Adventures (individual/10)

Which of the three approaches described in [Sorv2014] (Section 7.5) do you use when teaching? Or is your approach best described in some other way?

What Are You Teaching? (individual/10)

Compare the topics you teach to the list developed in [Luxt2017] (Section 7.5). Which topics do you cover? What extra topics do you cover that aren't in their list?

Beneficial Activities (individual/10)

Look at the list of interventions developed by [Viha2014] (Section 7.8). Which of these things do you already do in your classes? Which ones could you easily add? Which ones are irrelevant?

Visualizations (individual/10)

What visualization do you most like to use when teaching? Is it a static image or an animation? Do you show it to your learners, do they discover it on their own, or something in between?

Misconceptions and Challenges (small groups/15)

The Professional Development for CS Principles Teaching site includes a detailed list of student misconceptions and exercises. Working in small

groups, choose one section (such as data structures or functions) and go through their list. Which of these misconceptions do you remember having when you were a learner? Which do you still have? Which have you seen in your learners?

8 Teaching as a Performance Art

As Chapter 7 explained, every teacher needs content knowledge, general pedagogical knowledge, and pedagogical content knowledge in order to be effective. We can elaborate this framework by adding technology to the mix [Koeh2013], but that doesn't change the key point: it isn't enough to know the subject, or how to teach—you have to know how to teach that particular subject [Maye2004].

This chapter therefore focuses on one key aspect of teaching: giving a lecture or a live demonstration in front of a class. It isn't the only way to teach, but it is probably the most common, and the techniques that will make you better at doing it can be applied elsewhere as well.

Teaching Tips

The CS Teaching Tips site is collecting PCK for teaching programming, and I hope that one day we will have catalogs like [Ojos2015], teacher training materials like [Hazz2014, Guzd2015a, Sent2018], or more personal collections like [Gelm2002] to help us all do it better.

8.1 Lesson Study

From politicians to researchers and teachers themselves, educational reformers have designed systems to find and promote people who can teach well and eliminate those who cannot. But the assumption that some people are born teachers is wrong; instead, like any other performance art, the keys to better teaching are practice and collaboration. As [Gree2014] explains, the Japanese approach to this is 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. She would also expect to get feedback from fellow musicians during practice and after performances. Many other professions work this way as well: for example, the Japanese drew inspiration from Deming's ideas on continuous improvement in manufacturing.

But continuous feedback isn't part of teaching culture in most English-speaking countries. 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 or a textbook publisher, or go through a few MOOCs on the Internet, but each teacher has to figure out for herself how to combine that content with the theory she learned in education school to deliver an actual lesson in an actual classroom for actual students.

Writing up new techniques and giving demonstration lessons, in which one person teaches actual students while other teachers observe, seem like a way to solve this. However, [Finc2007, Finc2012] found that they are usually ineffective: of the 99 change stories analyzed, teachers only searched actively for new practices or materials in three cases, and only consulted published material in eight cases. Most changes occurred locally, without input from outside sources, or involved only personal interaction with other educators.

[Bark2015] found something similar:

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, a teacher might intend to show learners how to search for email addresses in a text file, but what her audience might take away is some new keyboard shortcuts in the editor. What *jugyokenkyu* does is maximize the opportunity for this to happen between teachers.

8.2 Giving and Getting Feedback on Teaching

Observing someone helps you; giving them feedback helps them. But as the cartoon in Figure 8 suggests, it can be hard to receive feedback, especially when it's negative.

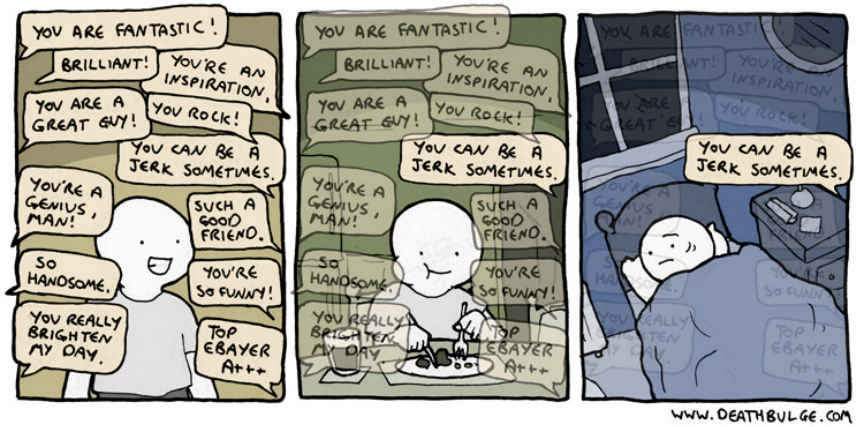

Figure 8: Feedback Feelings (copyright © Deathbulge 2013)

Feedback is easier to give and receive when both parties 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 by using these techniques:

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

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 a teacher 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.

Use a feedback translator. Have someone else read over all the feedback and give you a 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".

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 teachers to adjust their assessment of themselves the next time they feel that way.

You can give feedback to others more effectively as well:

Balance positive and negative feedback. A common method is a "compliment sandwich" made up of one positive, one negative, and a second positive observation (though this can get tiresome after a while).

Organize your feedback using a rubric. Most people are more comfortable giving and receiving feedback when they feel that they 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.

The simplest rubric for feedback on teaching is a 2x2 grid whose vertical axis is labelled "what went well" and "what can be improved", and whose horizontal axis is labelled "content" (what was said) and "presentation" (how it was said). Observers write their comments on sticky notes as they watch the demonstration, then post those in the quadrants of a grid drawn on a whiteboard (Figure 9).

A more sophisticated rubric developed for assessing 5–10 minute videos of programming instruction is given in Appendix J. A rubric this detailed is best presented as a checklist with items more or less in the order that they'll be used (e.g., questions about the introduction come before questions about the conclusion).

Question Budgets

Rubrics like the one in Appendix J have a tendency to grow over time as people think of things they'd like to add. A good way to keep them manageable is to insist that the total length stay constant, i.e., that if someone wants to add a question, they have to identify one that's less important and can be removed.

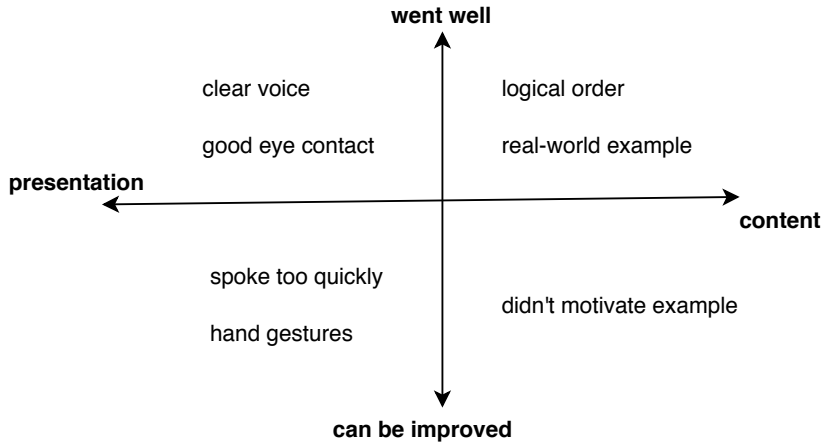

Figure 9: Teaching Rubric

If you are interested in giving and getting feedback, [Gorm2014] has good advice that you can use to make peer-to-peer feedback a routine part of your teaching, while [Gawa2011] looks at the value of having a coach. However feedback is collected, remember that it 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. Please also remember that these guidelines are for peer-to-peer feedback on your lesson delivery; gathering feedback from your learners should be as close to continuous as you can make it, and you should prompt for questions and reflections as well as positives and negatives.

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 teacher critiques both the designs and the peer critiques, so that participants learn not only how to make buildings, but how to give and get feedback [Scho1984]. Master classes in music serve a similar purpose.

8.3 How to Practice Performance

The best way to improve your in-person lesson delivery is to watch yourself do it. This method is borrowed from Warren Code at the University of British Columbia.

1. Work in groups of three.
2. Each person rotates through the roles of teacher, audience, and videographer. The teacher has two minutes to explain one key idea from their teaching or other work as if they were talking to a class of 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 other handheld device.
3. After everyone has finished teaching, the whole group watches the videos together. Everyone gives feedback on all three videos, i.e., people give feedback on themselves as well as on others.
4. After the videos have been discussed, they are deleted. (Many people are increasingly uncomfortable with the prospect of images of themselves appearing online.)
5. Finally, return to the main group and add the feedback to a shared 2x2 grid that separates positive from negative and content from presentation.

In order for this exercise to work well:

- 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.
- Let people know at the start of the class that they will be asked to teach something so that they have time to choose a topic. (Telling them this in advance can be counter-productive, since some people will fret over how much they should prepare.)
- 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.
- 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. (Most people are harder on themselves than others are, and it's important for them to realize this.)

The announcement of this exercise is often greeted with groans and apprehension, since few people enjoy seeing or hearing themselves. However, those same people consistently rate it as one of the most valuable parts of workshops based on these notes. It's also good preparation for co-teaching (Section 9.3): teachers find it a lot easier to give each other informal feedback if they have had some practice doing so and have a shared rubric to set expectations.

Tells

Everyone has nervous habits. For example, many of us talk more rapidly and in a higher-pitched voice than usual, while others play with their hair or crack their knuckles. 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.

8.4 Live Coding

Teaching is theater, not cinema.
— Neal Davis

One technique that completely changed the way I teach programming is live coding. Instead of using slides, teachers actually write code in front of their class as their learners follow along. It's more effective than slides for many reasons:

- Watching a program being written is more compelling than watching someone page through slides that present bits and pieces of the same code.
- It enables teachers to be more responsive to "what if?" questions. Where a slide deck is like a railway track, live coding allows teachers to go off road and follow their learners' interests.
- It facilitates lateral knowledge transfer: people learn more than we realized we were teaching by watching *how* teachers do things.
- It slows the teacher 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.
- It helps to keep the load on short-term memory down because it makes the teacher more aware of how much they are throwing at their learners. (This isn't true of slides or of copy-and-paste.)
- Learners get to see teachers' 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.
- Watching teachers make mistakes shows learners that it's all right to make mistakes of their own. Most people model the behavior of their teachers: if the teacher isn't embarrassed about making and talking about mistakes, learners will be more comfortable doing so too.

Teachers need a bit of practice to get comfortable with thinking aloud as they code in front of an audience, but most report that it is then no more difficult than talking around a deck of slides, and research seems to back up its effectiveness [Rubi2013, Haar2017]. The sections below offer tips on how to make your live coding better.

Embrace Your Mistakes

The typos are the pedagogy.
— Emily Jane McTavish

The most important rule of live coding is to embrace your mistakes. No matter how well you prepare, you will make some; when you do, think through them with your audience. While data is hard to come by, professional programmers spend anywhere from 25% to 60% of their time debugging; novices spend much more (Section 7.2), but most textbooks and tutorials spend little time diagnosing and correct problems. If you talk aloud while you figure out what you mistyped or where you took the wrong path, and explain how you've corrected yourself, you will give your learners a toolbox they can use when they make their own mistakes.

This is at odds with advice like that in [Kran2015], which says, "... you should have your material *absolutely mastered* before you enter the classroom. If... you have a proof or example that is not quite right... and stand in front of the group trying to fix it, then you will lose all but the diehards quickly." In contrast, the feedback we've had in Software Carpentry workshops and other settings is that watching the teacher make mistakes actually motivates most students, since it gives them permission to be less than perfect as well.

Deliberate Fumbles
If you've given a lesson several times, you're unlikely to make anything other than basic typing mistakes (which can still be informative). You can try to remember past mistakes and make them deliberately, but that often feels forced (unless the mistake and how to correct it is the primary purpose of the lesson). A better approach is sometimes called twitch coding: ask learners one by one to tell you what to type next. This is pretty much guaranteed to get you into the weeds.

Ask For Predictions

One way to keep students engaged while you are live coding is to ask them to make predictions, e.g., to say, "What is going to happen when I run this code?" You can then either show them, or write down the first few suggestions they make, have the whole class vote on which they think is most likely, and then run the code. As well as keeping their attention on

task, this gives them practice at reasoning about code's behavior, which is a useful skill in its own right.

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. And if you use tab completion, say it out loud the first few times so that your learners understand what you're doing: "Let's use turtle dot 'r' 'i' and tab to get 'right'."

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. If that's not practical, execute the same command a second time, or copy and paste the last command(s) into the workshop's shared notes.

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.

Mirror Your Learner's Environment

You may have customized your environment with a fancy Unix shell prompt, a custom color scheme for your development environment, or a plethora of keyboard shortcuts. Your learners won't have any of this, so try to create an environment that mirrors what they *do* have. Some teachers create a

separate bare-bones user (login) account on their laptop, or a separate teaching-only account if they're using an online service like Scratch or GitHub.

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. 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.

To cope with this, 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.

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.

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. Raise the bottom of your window(s) to compensate, but be aware that this gives you even less space for your typing.

If you can get a second projector and screen, use it: the extra 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.

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).

Accessibility Aids Help Everyone

Tools like Mouseposé (for Mac) and PointerFocus (for Windows) will highlight the position of your mouse cursor on the screen, and screen recording software tools like Camtasia will echo invisible keys like tab and Control-J as you type them. These take a bit of practice to get used to, but are extremely helpful as you start teaching more advanced tools.

Double Devices

Some people now use two devices when teaching: a laptop plugged into the projector for learners to see, and a tablet beside it so that they can view

their own notes and the shared notes that the learners are taking together (Section 9.7). This is more reliable than displaying one virtual desktop while flipping back and forth to another. Of course, printouts of the lesson material are still the most reliable backup technology...

Use Diagrams

Diagrams are almost always a good idea. Creating them in advance to bring up on screen is a common practice—I often have a slide deck full of diagrams in the background when I'm doing live coding—but don't underestimate the value of sketching on the whiteboard as you go through your lesson. This allows you to build diagrams step by step, which helps with retention (Section 4.1) and allows you to improvise.

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 it can be awkward when a message pops up you'd rather not have others see. If you are teaching frequently, you might want to create a second account on your computer that doesn't have email or other tools set up at all.

Improvise After You Know the Material

The first time you teach a new lesson, stick fairly closely to the lesson plan you've drawn up or borrowed. 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. Resist: 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 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, run through it beforehand as you plan to in class *using the same computer that you'll be teaching on*. Installing several hundred megabytes of software updates over high school WiFi in front of increasingly bored 16-year-olds isn't something you want to do twice.

Direct Instruction

Direct Instruction is a teaching method centered around meticulous curriculum design delivered through a prescribed script—i.e., it's more like

an actor reciting lines than it is like the improvisatory approach we recommend. [Stoc2018] surveys studies and finds statistically significant positive effect, even though DI sometimes gets knocked for being mechanical. We still prefer improvisation because DI requires a far greater up-front investment than most free-range learning groups can afford.

Face the Screen—Occasionally

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. 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.

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.

Drawbacks

Live coding does have some drawbacks, but with practice, these can be avoided or worked around. A common one is going too slowly, either because you are not a good typist or because you are spending too much time looking at notes trying to figure out what to type next. The fix for the first is a bit of typing practice; the fix for the second is to break the lesson into very short pieces, so that you only ever have to remember one small step to take next.

A deeper exercise is that typing in library import statements, class headers, and other boilerplate code increases the extraneous cognitive load on your learners (Chapter 4). If you spend a lot of time doing this, it may be all that learners take away, so give yourself and your learners skeleton code to start with (Section 9.9).

8.5 Exercises

Give Feedback on Bad Teaching (whole class/20)

1. Watch this video of bad teaching as a group and 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?

Practice Giving Feedback (small groups/45)

Use the process described above to practice teaching in groups of three. When your group is done, the teacher 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.

The Bad and the Good (whole class/20)

Watch the videos of live coding done poorly and live coding done well and summarize your feedback on both using the usual 2x2 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.

See Then Do (pairs/30)

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?

Tells (small groups/15)

1. Read the description of tells at the end of Section 8.2, then make a note of what you think your tells are, but do not share them with other people.
2. Teach a short (3–5 minute) lesson.
3. Ask your audience how they think you betray nervousness. Is their list the same as yours?

Teaching Tips (small groups/15)

The CS Teaching Tips site has a large number of practical tips on teaching computing, as well as a collection of downloadable tip sheets. In small groups, go through the tip sheets on their home page and classify each tip as "use all the time", "use occasionally", "never use". Where do your practice and your peers' practice differ? Are there any tips you strongly disagree with, or think would be ineffective?

9 In the Classroom

The previous chapter described how to practice in-class teaching and described one method—live coding—that allows teachers to adapt to their learners' pace and interests. This chapter describes other practices that have proven helpful in programming classes.

Before describing these practices, it's worth pausing for a moment to set expectations. The best teaching method we know is individual tutoring: [Bloo1984] found that students taught one-to-one using mastery learning techniques performed two standard deviations better than those who learned through conventional lecture, i.e., that individually-tutored students did better than 98% of students who were lectured to. However, hiring one teacher for every student is impossibly expensive (and despite the hype, artificial intelligence isn't going to take the place of human instructors any time soon). Every method is essentially an attempt to get as much of the value of individual attention as possible, but at scale.

9.1 Enforce the Code of Conduct

Chapter 1 said that every workshop should have and enforce a Code of Conduct like the one in s:conduct. If you are a teacher, and believe that someone has violated it, 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.

If you expel someone, say so to the rest of the class and explain why. This helps prevent exaggerated rumors from taking hold, and also signals very clearly to everyone that you're serious about making your class safe and respectful for them.

Contact the host of your class as soon as you can and describe what happened.

A Code of Conduct is meaningless without procedures for reporting violations and enforcing its rules. However much you don't enjoy doing the latter, remember that the former can be a much greater burden for people who have been targets.

9.2 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 [Mazu1996], it has been studied extensively in a wide variety of contexts, including programming [Crou2001, Port2013], and [Port2016] found that students value peer instruction even at first contact.

Peer instruction is essentially a way to provide one-to-one mentorship in a scalable way. It interleaves formative assessment with student discussion as follows:

1. Give a brief introduction to the topic.
2. Give students a multiple choice question that probes for misconceptions (rather than simple factual knowledge).
3. Have all the students vote on their answers to the MCQ.

    - If the students all have the right answer, move on.
    - If they all have the same wrong answer, address that specific misconception.
    - 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 from Avanti's learning center in Kanpur 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 teacher 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.

But could this be a false positive? Are results improving because of increased understanding during discussion, or simply from a follow-the-leader effect ("vote like Jane, she's always right")? [Smit2009] tested this by following the first question with a second one that students answer individually. Sure enough, peer discussion actually does enhance understanding, even when none of the students in a discussion group originally knew the correct answer.

Taking a Stand

It is important to have learners vote publicly 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 hypercorrection of having their answer be wrong and having to think through the reasons why (Section 5.1).

9.3 Teach Together

Co-teaching describes any situation in which two teachers work together in the same classroom. [Frie2016] describes several ways to do this:

Team teaching: Both teachers deliver a single stream of content in tandem, taking turns the way that musicians taking solos would.

Teach and assist: Teacher A teaches while Teacher B moves around the classroom to help struggling students.

Alternative teaching: Teacher A provides a small set of students with more intensive or specialized instruction while Teacher B delivers a general lesson to the main group.

Teach and observe: Teacher A teaches while Teacher B observes the students, collecting data on their understanding to help plan future lessons.

Parallel teaching: The class is divided into two equal groups and the teachers present the same material simultaneously to each.

Station teaching: The students are divided into several small groups that rotate from one station or activity to the next while both teachers supervise where needed.

All of these models create more opportunities for lateral knowledge transfer than teaching alone. Team teaching is particularly beneficial in day-long workshops: not only does it give each teacher's voice a chance to rest, it reduces the risk that they will be so tired by the end of the day that they will start snapping at their students or fumbling at their keyboard.

Helping

Many people who aren't comfortable teaching are still willing and able to provide in-class technical support. They can help learners with setup and installation, answer technical questions during exercises, monitor the room to spot people who may need help, or keep an eye on the shared notes (Section 9.7) and either answer questions there or remind the instructor to do so during breaks.

Helpers are sometimes people training to become teachers (i.e., they're Teacher B in the teach and assist model), but they can also be members of the host institution's technical support staff, alumni, or advanced learners who already know the material well. Using the latter as helpers is doubly effective: not only are they more likely to understand the problems their peers are having, it also stops them from getting bored.

If you and a partner are co-teaching, try to follow these rules:

- Take 2–3 minutes before the start of each class to confirm who's teaching what with your partner. (If you have time to do some advance preparation, try drawing a concept map together.)
- Use that time to work out a couple of hand signals as well. "You're going too fast", "speak up", "that learner needs help", and, "It's time for a bathroom break" are all useful.
- Each person should teach for at least 10–15 minutes at a stretch, since students may be distracted by more frequent interleaving.
- The person who *isn't* teaching shouldn't interrupt, offer corrections, elaborations, or amusing personal anecdotes, or do anything else to distract from what the person teaching at the time is doing or saying. The one exception is that it's sometimes helpful to ask leading questions, particularly if the learners seem unsure of themselves.
- Each person should take a couple of minutes before they start teaching to see what their partner is going to teach after they're done, and then *not* present any of that material.
- The person who isn't teaching should stay engaged with the class, not catch up on their email. Monitor the shared notes (Section 9.7), keep an eye on the students to see who's struggling, jot down some feedback to give your teaching partner at the next break—anything that contributes to the lesson is better than anything that doesn't.

Most importantly, take a few minutes when the class is over to either congratulate or commiserate with each other. In teaching as in life, shared misery is lessened and shared joy increased: no one will understand how pleased you are that you helped someone understand loops better than the person you just taught with.

9.4 Assess Prior Knowledge

The more you know about your learners before you start teaching, the more you will be able to help them. If you're working inside a formal school system, you can probably infer their incoming knowledge by looking at what's (actually) covered in the prerequisites to your course. If you're in a free-range setting, though, your learners may be much more diverse, so you may want to give them a short survey or questionnaire in advance of your class to find out what they do and don't already know.

But doing this is risky. School trains people to treat all assessment as summative, i.e., to believe that anything that looks like an exam is something they have to pass, rather than a chance to shape instruction. If they answer "I don't know" to even a handful of questions on your preassessment, they might conclude that your class is too advanced for them. In short, you might scare off many of the people you most want to help.

And self-assessment is unreliable because of the Dunning-Kruger effect [Krug1999]: the less people know about a subject, the less accurate their estimate of their knowledge is. Conversely, people who are competent may underrate their skills because they regard their level of competence as normal.

Rather than asking people to rate their knowledge from 1 to 5, you should therefore try to ask them how easily they could complete some specific tasks, but that still runs the risk of scaring them away. s:preassess presents a short preassessment questionnaire that most potential learners are unlikely to find intimidating; if you use it or anything like it, please be sure to follow up with people who *don't* respond to find out why not.

9.5 Plan for Mixed Abilities

If your learners have widely varying levels of prior knowledge, then you can easily wind up in a situation where a third of your class is lost and a third is bored. That's unsatisfying for everyone, but there are some strategies you can use to manage the situation:

- 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 they will be asked to complete.
- Provide extra self-paced exercises 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).
- 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 single lesson can possibly meet everyone's individual needs. If you slow down to accommodate two people who are struggling, the other 38 are not being well served. Equally, if you spend a few minutes talking about an advanced topic to a learner who is bored, the rest of the class will feel left out.

False Beginners

A false beginner is someone who has studied a language before but is learning it again. False beginners may be indistinguishable from absolute beginners on preassessment tests, but are able to move much more quickly through the material once they start—in mathematical terms, their intercept is the same, but their slope is very different. False beginners are common in free-range programming classes: for example, a child may have taken a Scratch class a couple of years ago and built a mental model of loops and conditionals, but do poorly on a pre-test because the material

isn't fresh in their mind. All of the strategies described above can be used in classes with false beginners.
Being a false beginner is an example of preparatory privilege [Marg2010]. In many cases, it's a result of coming from a home that's secure enough and affluent enough to have several computers and parents who are familiar with how to use them. Whether or not this is fair depends on what you choose to include in your assessment.

9.6 Pair Programming

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

Pair programming is an effective practice in professional work [Hann2009], and is also a good way to teach: benefits include increased success rate in introductory courses, better software, and higher student confidence in their solutions; there is also evidence that students from underrepresented groups benefit even more than others [McDo2006, Hank2011, Port2013, Cele2018]. 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. I have found it particularly helpful with mixed-ability classes, since pairs are likely to be more homogeneous than individuals.

When you use pairing, put *everyone* in pairs, not just 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) on a regular basis, and 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.

To facilitate pairing, use a flat (dinner-style) seating rather than banked (theater-style) seating; this also makes it easier for helpers to reach learners who need assistance. And take a few minutes to demonstrate what it actually looks like so that they understand the person who doesn't have their hands on the keyboard isn't supposed to just sit and watch. Finally, tell them about [Lewi2015], who studied pair programming in a Grade 6 classroom, and found that pairs that focused on trying to complete the task as quickly as possible were less fair in their sharing.

Switching Partners
Teachers 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, moving computers and power adapters to new desks several times a day is disruptive, and pairing can be uncomfortable for introverts. That said, [Hann2010] found weak correlation between the "Big Five" personality traits and performance in pair programming, although an earlier study [Wall2009] found that pairs whose members had differing levels of personality traits communicated more often.

9.7 Take Notes. . . Together?

Many studies have shown that taking notes while learning improves retention [Aike1975, Boha2011]. Taking notes is essentially a form of real-time elaboration (Section 5.1): it 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* can is also effective, [Ornd2015, Yang2015], even though taking notes on a computer is generally less effective than taking notes using pen and paper [Muel2014].

The first time students encounter the practice, they sometimes report that they find it distracting, as it's one more thing they have to keep an eye on. Some of the arguments in favor of doing it are:

- 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 can 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 teacher would prepare in advance, since the learners are more likely to write down what they actually found new, rather than what the teacher predicted would be new.
- Glancing at the notes as they're being taken helps the teacher discover that the class didn't hear something important, or misunderstood it.

We usually use Etherpad or Google Docs for taking shared notes. 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 teachers their work (by copying and pasting it in).

If you are going to have a group take notes together, make a list of everyone's name and paste it into the document each time you want every person to answer a question or contribute an exercise solution. This prevents the situation in which everyone is trying to edit the same couple of lines at the same time.

In my experience, the benefits of shared note-taking outweigh the costs. If you are only working with a particular group once, though, please heed the advice in Section 9.12 and stick to whatever they are used to.

9.8 Sticky Notes

Sticky notes are one of my favorite teaching tools, and judging from [Ward2015], I'm not alone in loving their versatility, portability, stickability, foldability, and subtle yet alluring aroma.

As Status Flags

Give each learner two sticky notes of different colors, e.g., orange 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, they put up the orange 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 teacher can quickly see from the front of the room what state the class is in.

To Distribute Attention

Sticky notes can also be used to ensure the teacher's attention is fairly distributed. Have each learner write their name on a sticky note and put it on their laptop. Each time the teacher 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 teacher 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.

As Minute Cards

You can 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 hasn't yet been answered or something that they're still confused about. While they are enjoying their coffee or lunch, the teachers 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.

9.9 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, novices should never start doing exercises with a blank page (or screen), since they often find this intimidating or bewildering. If they have been following along as you do live coding, you can ask them either to add a few more lines or to modify the example you have built up. Alternatively, if there is a shared note-taking space, you can paste in a few lines of starter code for them to extend or modify.

Modifying existing code instead of writing new code from scratch doesn't just give learners structure: it is also closer to what they will do in real life. 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. Java's `public static void main()` or a handful of `import` statements at the top of a Python program may make sense to you, but is extraneous load to them (Chapter 4).

9.10 Setting Up Your Learners

Adult learners tell us that it is important to them to leave programming classes with their own computers set up to do real work. We therefore strongly recommend that teachers 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 do this, put detailed setup instructions for all three platforms on your class 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 teachers 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.*

You can try using tools like Docker to put virtual machines on learners' computers to reduce installation problems, but those introduce problems of their own. Older or smaller machines simply aren't fast enough to run them, learners often struggle to switch back and forth between two different sets of keyboard shortcuts for things like copying and pasting, and even competent practitioners will become confused about what exactly is happening where.

All of this is so complicated that many teachers now use browser-based tools instead. This solves the installation issues, but makes the class dependent on institutional WiFi (which can be of highly variable quality). It also doesn't satisfy adult learners' desire to leave with their own machines ready for real-world use, but as cloud-native development tools like Glitch enter widespread use, that is less and less important.

9.11 Other Teaching Practices

None of the smaller practices described below are essential, but all will improve lesson delivery. As with chess and marriage, success in teaching is often a matter of slow, steady progress.

Start With Introductions

To begin your class, the teachers 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 shared notes, but it's also good for everyone at a given site to know who all is in the group. (This can be done while setting up before the start of the class.)

Set 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, 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 in s:events.

Avoid Homework in All-Day Formats

Learners who have spent an entire day programming will be tired. If you give them homework to do after hours, they'll start the next day tired as well, so don't do this.

Don't 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.

Repeat the Question

Whenever someone asks a question in class, repeat it back to them before answering it to check that you've understood it, and to give people who might not have heard it a chance to do so. This is particularly important when presentations are being recorded or broadcast, since your microphone will usually not pick up what other people are saying. Repeating questions back also gives you a chance to redirect the question to something you're more comfortable answering if need be...

One Up, One Down

We frequently ask for summary feedback at the end of each day. The teachers 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.

Have Learners Make Predictions

Research has shown that people learn more from demonstrations if they are asked to predict what's going to happen [Mill2013]. 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.

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 flat (dinner-style) seating rather than banked (theater-style) seating, so that you can reach learners who need help more easily, and so that learners can pair with one another (Section 9.5). In-floor power outlets so that you don't have to run power cords across the floor make life easier as well as safer, but are still the exception.

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.

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. Good ones will also mask the onset of coffee breath, for which your learners will probably be grateful.

Think-Pair-Share

Think-pair-share is a lightweight technique that helps people 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.

Morning, Noon, and Night

[Smar2018] found that if students' classes and other work is scheduled at times that don't line up with their natural body clocks, they do less well—i.e., that if a morning person takes night classes or vice versa, their grades suffer. It's usually not possible to accommodate this in small groups, but larger ones should try to stagger start times. This can also help people with childcare responsibilities and other constraints on their time.

Humor

Humor should be used sparingly when teaching: 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 go 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.

9.12 Limit Innovation

Each of the techniques presented in this chapter will make your classes better, but you shouldn't try to adopt them all at once. In fact, it may be best for your students if you don't use *any* of them, particularly in situations where you and the students are only together for brief periods. The reason is that every new practice increases the student's cognitive load: as well as absorbing what you're trying to teach them about programming, they're also having to learn a new way to learn. If you are working with them repeatedly, you can introduce one new technique every few lessons; if you only have them for a one-day workshop, it's probably best to be conservative in your approach.

9.13 Exercises

Create a Questionnaire (individual/20)

Using the questionnaire in s:preassess 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?

One of Your Own (whole class/15)

Think of one teaching practice that hasn't been described so far. Present your idea to a partner, listen to theirs, and select one to present to the group as a whole. (This exercise is an example of think-pair-share.)

May I Drive? (pairs/10)

Swap computers with a partner (preferably one who uses a different operating system than you) and work through a simple programming exercise. How frustrating is it? How much insight does it give you into what novices have to go through all the time?

Pairing (pairs/15)

Watch this video of pair programming, then practice doing it with a partner. Remember to switch roles between driver and navigator every few minutes. How long does it take you to fall into a working rhythm?

Compare Notes (small groups/15)

From groups of 3–4 people and compare the notes that each person has taken while reading this material or following along with it in class. What did you think was noteworthy that your peers missed and vice versa? What did you understand differently?

Credibility (individual/15)

[Fink2013] describes three things that make teachers credible in their learners' eyes:

Competence: knowledge of the subject as shown by the ability to explain complex ideas or reference the work of others.

Trustworthiness: having the student's best interests in mind. This can be shown by giving individualized feedback, offering a rational explanation for grading decisions, and treating all students the same.

Dynamism: excitement about the subject (Chapter 8).

Describe one thing you do when teaching that fits into each category, and then describe one thing you *don't* do but should for each category as well.

Measuring Effectiveness (individual/15)

[Kirk1994] defines four levels at which to evaluate training:

Reaction: how did the learners feel about the training?
Learning: how much did they actually learn?
Behavior: how much have they changed their behavior as a result?
Results: how have those changes in behavior affected their output or the output of their group?

What are you doing at each level to evaluate what and how you teach? What could you do that you're not doing?

Objections and Counter-Objections (think-pair-share/15)

You have decided not to ask your learners if your class was useful, because you know that there is no correlation between their answers and how much they actually learn (Section 7.5). Instead, you have put forward four proposals, each of which your colleagues have shot down:

See if they recommend the class to friends. Why would this be any more meaningful than asking them how they feel about the class?
Give them an exam at the end. But how much learners know at the end of the day is a poor predictor of how much they will remember two or three months later, and any kind of final exam will change the feel of the class, because school has conditioned learners to believe that exams are always high-stakes affairs.
Give them an exam two or three months later. But that's practically impossible with free-range learners, and 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 skewed.
See if they keep using what they learned. Again, since installing spyware on learners' computers is frowned upon, how will this be implemented?

Working on your own, come up with answers to these objections, then swap responses with a partner and discuss the approaches you have come up with. When you are done, share your best counter-argument with the entire class.

10 Motivation and Demotivation

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

Our starting point is the difference between extrinsic motivation, which we feel when we do something to avoid punishment or earn a reward, and intrinsic motivation, which is what we feel when we find something personally rewarding. Both affect most situations—for example, people teach because they enjoy it and because they get paid—but we learn best when we are intrinsically motivated [Wlod2017]. According to self-determination theory, the three drivers of intrinsic motivation are:

Competence: the feeling that you know what you're doing.
Autonomy: the feeling of being in control of your own destiny.
Relatedness: the feeling of being connected to others.

A well-designed lesson encourages all three. For example, a programming exercise would give students practice with all the tools they need to use to solve a larger problem (competence), let them tackle the parts of that problem in whatever order they want (autonomy), and allow them to talk to their peers (relatedness).

The Problem of Grades

I've never had an audience in my life. My audience is a rubric.
– quoted by Matt Tierney

Grades and the way they distort learning are often used as an example in discussion of extrinsic motivation, but as [Mill2016a] observes, they aren't going to go away any time soon, so it's pointless to try to build a system that ignores them. Instead, [Lang2013] explores how courses that emphasize grades can incentivize students to cheat, and offers some tips on how to diminish this effect, while [Covi2017] looks at the larger problem of balancing intrinsic and extrinsic motivation in institutional education, and the constructive alignment approach advocated in [Bigg2011] seeks to bring learning activities and learning outcomes into line with each other.

[Ambr2010] 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. One strategy I particularly like is to have students who struggled but succeeded come in and tell their stories to the rest of the class. Learners are far more likely to believe stories from people like themselves [Mill2016a], and people who have been through your course will always have advice that you would never have thought of.

Not Just for Students

Discussions of motivation in education often overlook the need to motivate the teacher. *Learners respond to a teacher's enthusiasm, and teachers need to care about a topic in order to keep teaching it, particularly when they are volunteers. This is another powerful reason to co-teach (Section 9.3): just as having a running partner makes it more likely that you'll keep running, having a teaching partner helps get you up and going on those days when you have a cold and the projector bulb has burned out and nobody knows where to find a replacement and why are they doing construction work today of all days. . .*

Teachers can do other positive things as well. [Bark2014] found three things that drove retention for all students: meaningful assignments, faculty interaction with students, and student collaboration on assignments. Pace and workload (relative to expectations) were also significant drivers, but primarily for male students. Things that *didn't* drive retention were interactions with teaching assistants and interactions with peers in extracurricular activities. These results may seem obvious, but the reverse would seem obvious too: if the study had found that extracurricular activities drove retention, we would also say "of course". Noticeably, two of the four retention drivers (faculty interaction and student collaboration) take extra effort to replicate online (Chapter 11).

10.1 Authentic Tasks

As Dylan Wiliam points out in [Hend2017], motivation doesn't always lead to achievement, but achievement almost always leads to motivation: helping students succeed motivates them far more than telling them how wonderful they are. We can use this idea in teaching by creating a grid whose axes are "mean time to master" and "usefulness once mastered" (Figure 10).

Things that are quick to master and immediately useful should be taught first, even if they aren't considered fundamental by people who are already competent practitioners, because a few early wins will build learners' confidence in their own ability and their teacher's judgment. Conversely, things that are hard to learn and have little near-term application should be skipped

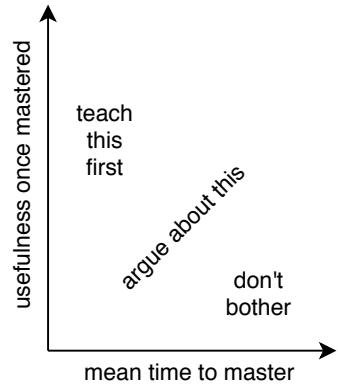

Figure 10: What to Teach

entirely, while topics along the diagonal need to be weighed against each other.

Many of the foundational concepts of computer science, such as recursion and computability, inhabit the "useful but hard to learn" corner of this grid. This doesn't mean that they aren't worth learning, but if our aim is to convince people that they *can* learn to program, and that doing so will help them do things that they care about, these big ideas can and should be deferred. Remember, people often don't want to program for its own sake: they want to make music or explore changes to family incomes over time, and (rightly) regard programming as a tax they have to pay in order to do so.

A well-studied instance of prioritizing what's useful without sacrificing what's fundamental is the media computation approach developed at Georgia Tech [Guzd2013]. Instead of printing "hello world" or summing the first ten integers, a student's first program might open an image, resize it to create a thumbnail, and save the result. This is an authentic task, i.e., something that learners believe they would actually do in real life. It also has a tangible artifact: if the image comes out the wrong size, learners have a concrete starting point for debugging. [Lee2013] describes an adaption of this approach from Python to MATLAB, while others are building similar courses around data science, image processing, and biology [Dahl2018, Meys2018, Ritz2018].

There will always be tension between giving learners authentic problems and exercising the individual skills that they will need to solve those problems. People don't answer multiple choice questions or do Parsons Problems outside of a classroom, any more than most musicians play scales over and over again in front of an audience. Finding the balance is hard, but one easy

first step is to make sure that exercises don't include anything arbitrary or meaningless. For example, programming examples shouldn't use variables called `foo` and `bar`, and if you're going to have learners sort lines of text, give them album titles or people's names or something relatable.

10.2 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 in a free-range setting, your learners are probably volunteers, and probably want to be in your classroom. The exercise therefore isn't how to motivate them, but how to not demotivate them. Unfortunately, you can do this by accident much more easily than you might think. For example, [Cher2009] reported four studies showing that subtle environmental clues have a measurable difference on the interest that people of different genders have in computing: changing objects in a CS classroom from those considered stereotypical of computer science (e.g., Star Trek posters and video games) to objects not considered stereotypical (e.g., nature poster, phone books) boosted female undergraduates' interest in CS to the level of their male peers. Similarly, [Gauc2011] reports a trio of studies showing that gendered wording commonly employed in job recruitment materials can maintain gender inequality in traditionally male-dominated occupations.

The three most powerful demotivators for adult learners 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. Indifference demotivates because learners who believe that the teacher or educational system doesn't care about them or the material won't care about it either. And people are also demotivated if they believe something is unfair, even if it is unfair in their favor, because they will worry (consciously or unconsciously) that they will some day find themselves in the group on the losing end [Wilk2011]. In extreme situations, learners may develop learned helplessness: when repeatedly subjected to negative feedback in a situation that they can't change, they may learn not to even try to change the things they could.

Here are a few specific things that will demotivate your learners:

A holier-than-thou or contemptuous attitude from a teacher or a fellow learner.

Telling them that their existing skills are rubbish. Unix users sneer at Windows, programmers of all kinds make jokes about Excel, and

no matter what web application framework you already know, some programmer will tell you that it's out of date. Learners have often invested a lot of time and effort into acquiring the skills they have; disparaging them is a good way to guarantee that they won't listen to anything else you have to say.

Diving into complex or detailed technical discussion with the most advanced learners in the class.

Pretending that you know more than you do. Learners will trust you more if you are frank about the limitations of your knowledge, and will be more likely to ask questions and seek help.

Using the J word ("just") or feigning surprise (i.e., saying things like "I can't believe you don't know X" or "you've never heard of Y?"). As discussed in Chapter 3, this signals to the learner that the teacher thinks their problem is trivial and by extension that they must be stupid for not being able to figure it out.

Software installation headaches. People's first contact with new programming tools, or programming in general, is often demoralizing, and believing that something is hard to learn is a self-fulfilling prophecy. It isn't just the time it takes to get set up, or the feeling that it's unfair to have to debug something that depends on precisely the knowledge they don't yet have; the real problem is that every such failure reinforces their belief that they'd have a better chance of making next Thursday's deadline if they kept doing things the way they always have.

It is even easier to demotivate people online than in person, but there are now evidence-based strategies for dealing with this. [Ford2016] found that five barriers to contribution on Stack Overflow are seen as significantly more problematic by women than men: lack of awareness of site features, feeling unqualified to answer questions, intimidating community size, discomfort interacting with or relying on strangers, and the perception that they shouldn't be slacking (i.e., the feeling that searching for things online wasn't "real work"). Fear of negative feedback didn't quite make this list, but would have been the next one added if the authors weren't quite so strict about their statistical cutoffs. All of these factors can and should be addressed in both in-person and online settings using methods like those in Section 10.4, and doing so improves outcomes for everyone [Sved2016].

Productive Failure and Privilege

Some recent work has explored the notion of productive failure, where learners are deliberately given problems that can't be solved with the knowledge they have, and have to go out and acquire new information in order to make progress [Kapu2016]. Ensuring that learners are blocked but not frustrated depends more on classroom culture and expectations than it does on the details of particular exercises.

Productive failure is superficially reminiscent of tech's "fail fast, fail often" mantra, but the latter is more a sign of privilege than of understanding. People can only afford to celebrate failure if they're sure they'll get a chance to try again; many of your learners, and many people from marginalized or underprivileged groups, can't be sure of that, and talking otherwise is a great way to turn them off.

Impostor Syndrome

Impostor syndrome is the belief that you aren't really good enough for a job or position—that your achievements are lucky flukes—and an accompanying fear of someone finding out. Impostor syndrome is common among high achievers who undertake publicly visible work, but most people suffer from it occasionally to some extent. It disproportionately affects members of under-represented groups: as discussed in Section 7.5, [Wilc2018] found that female students with prior exposure to computing outperformed their male peers in all areas in introductory programming courses, but were consistently less confident in their abilities, in part because society keeps signalling in subtle and not-so-subtle ways that they don't really belong.

Traditional classrooms can fuel impostor syndrome. Schoolwork is frequently undertaken alone or in small groups, but the results are shared and criticized publicly; as a result, we rarely see the struggles of others, only their finished work, which can feed the belief that everyone else finds it easy. Members of underrepresented groups who already feel additional pressure to prove themselves may be particularly affected.

The Ada Initiative has created some guidelines for fighting your own impostor syndrome, which include:

Talk about the issue with people you trust. When you hear from others that impostor syndrome is a common problem, it becomes harder to believe your feelings of being a fraud are real.

Go to an in-person impostor syndrome session. There's nothing like being in a room full of people you respect and discovering that 90% of them have impostor syndrome.

Watch your words, because they influence how you think. Saying things like, "I'm not an expert in this, but..." takes away from the knowledge you actually possess.

Teach others about your field. You will gain confidence in your own knowledge and skill, and you will help others avoid some impostor syndrome shoals.

Ask questions. Asking questions can be intimidating if you think you should know the answer, but getting answers eliminates the extended agony of uncertainty and fear of failure.

Build alliances. Reassure and build up your friends, who will reassure and build you up in return. (And if they don't, find new friends.)

Own your accomplishments. Keep actively recording and reviewing what you have done, what you have built, and what successes you've had.

As a teacher, 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: as noted in Section 8.4, your typos show your class that you're human.) 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.

Stereotype Threat

Reminding people of negative stereotypes, even in subtle ways, can make them anxious about the risk of confirming those stereotypes, which in turn reduces their performance. This is called stereotype threat; [Stee2011] summarizes what we know about stereotype threat in general and presents some strategies for mitigating it in the classroom.

Unwelcoming climates demotivate everyone, particularly members of under-represented groups, but it's less clear that stereotype threat is the primary cause. Part of the problem is that the term has been used in many ways [Shap2007]; another is questions about the replicability of key studies. What *is* clear is that both instructors and learners must avoid using language that suggests that some people are natural programmers and others aren't. Guzdial has called this the biggest myth about teaching computer science, and [Pati2016] backed this up by showing that people see evidence for a "geek gene" where none exists:

> *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 instructors' beliefs about their students.

Belief that some people get it and some don't is particularly damaging because of feedback effects. Consciously or unconsciously, teachers tend to focus their attention on learners who seem to be doing well. That extra attention increases the odds that they will, while the corresponding neglect of other learners leaves them further and further behind [Alvi1999, Brop1983, Juss2005].

Mindset

Carol 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.

As with stereotype threat, there are concerns that growth mindset has been oversold, or that research is much more difficult to put into practice than its more enthusiastic advocates have implied. [Sisk2018] reported two meta-analyses, one looking at the strength of the relationship between mindset and academic achievement, the other at the effectiveness of mindset interventions on academic achievement. The overall effects for both were weak, but some results supported specific tenets of the theory, namely, that students with low socioeconomic status or who are academically at risk might benefit from mindset interventions.

10.3 Accessibility

Not providing equal access to lessons and exercises is about as demotivating as it gets. This is often inadvertent: for example, my old online programming lessons presented the full script of the narration beside the slides—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.

It isn't always possible to accommodate everyone's needs, but 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 for us without us") predates accessibility rights, but is always the right place to start. A few specific recommendations are:

Find out what you need to do. Each of these posters offers do's and don'ts for people on the autistic spectrum, users of screen readers, and people with low vision, physical or motor disabilities, hearing exercises, and dyslexia.

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.

[Coom2012, Burg2015] are good guides to visual design for accessibility. Their recommendations include:

Format documents with actual headings and other landmarks, rather than just changing font sizes and styles.

Avoid using color alone to convey meaning in text or graphics: use color plus cross-hatching or colors that are noticeably different in grayscale.

Remove all unnecessary elements rather than just making them invisible, because screen readers will still often say them aloud.

Allow self-pacing and repetition for people with reading or hearing issues.

Include narration of on-screen action in videos.

Conduct Revisited

We said in Section 1.5 that classes should enforce a Code of Conduct like the one in s:conduct. This is a form of accessibility: while closed captions make video accessible to people with hearing disabilities, a Code of Conduct makes lessons accessible to people who would otherwise be marginalized.

As discussed in Section 9.1, the details of the Code of Conduct are important, but the most important thing about it is that it exists and is enforced. Knowing that there are rules tells people a great deal about your values and about what kind of learning experience they can expect.

Group Signup

One way to support learners from marginalized groups is to have people sign up for workshops in groups rather than individually. That way, 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 use what they've learned.

10.4 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, under-represented racial or ethnic groups, people with various sexual orientations, the elderly, the physically exercised, the formerly incarcerated, the economically disadvantaged, and everyone else who doesn't fit Silicon Valley's white/Asian male demographic. [Lee2017] is a brief, practical guide to doing that with references to the research literature. The practices it describes 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 your notes to make sure they are free from gendered pronouns, 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, but demonstrate it first so that learners understand the roles of driver and navigator.

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

At a higher level, committing to inclusive teaching may mean fundamentally rethinking content. This is a lot of work, but the rewards can be significant. For example, [DiSa2014a] found that 65% of male African-American participants in a game testing program went on to study computing, in part because the gaming aspect of the program was something their peers respected.

Work like this has to be done carefully. [Lach2018] explored two strategies:

Community representation highlights students' social identities, histories, and community networks using after-school mentors or role models from students' neighborhoods, or activities that use community narratives and histories as a foundation for a computing project.

Computational integration incorporates ideas from the learner's community, e.g., reverse engineering indigenous graphic designs in a visual programming environment.

The major risks of these approaches are shallowness (for community representation), e.g., using computers to build slideshows rather than do any real computing, and cultural appropriation (for computational integration), e.g., using practices without acknowledging origins. When in doubt, ask your learners and members of their community what they think you ought to do and give them control over content and direction. We return to this in Chapter 13.

Spoons

In 2003, Christine Miserandino started using spoons as a way to explain what it's like to live with chronic illness. Healthy people start each day with an unlimited supply of spoons, but people with lupus or other debilitating conditions only have a few, and everything they do costs them one. Getting out of bed? That's a spoon. Making a meal? That's another spoon, and pretty soon, you've run out.

> *You cannot simply just throw clothes on when you are sick... If my hands hurt that day buttons are out of the question. If I have bruises that day, I need to wear long sleeves, and if I have a fever I need a sweater to stay warm and so on. If my hair is falling out I need to spend more time to look presentable, and then you need to factor in another 5 minutes for feeling badly that it took you 2 hours to do all this.*

Spoons are often invisible, but as Elizabeth Patitsas has argued, people who have a lot can accumulate more, but people whose supply of spoons is limited may struggle to get ahead of the game. When you are designing classes and exercises, try to take into account the fact that some of your learners may have physical or mental obstacles that aren't obvious.

Again, when in doubt, ask your learners: they almost certainly have more experience with what works and what doesn't than anyone else.

Moving Past the Deficit Model

Depending on whose numbers you trust, only 12–18% of people getting computer science degrees are women, which is less than half the percentage seen in the mid-1980s (Figure 11). And western countries are the odd ones for having such low percentage of women in computing: women are still often 30–40% of computer science students elsewhere [Galp2002, Varm2015].

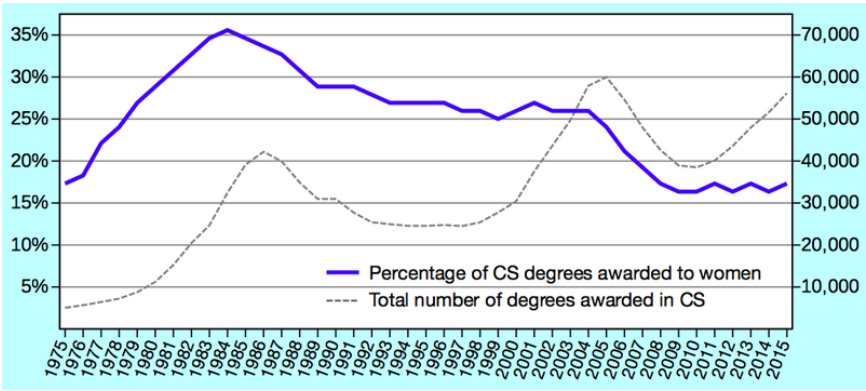

Figure 11: Degrees Awarded and Female Enrollment ([Robe2017](#BIB))

Since it's unlikely that women have changed drastically in the last thirty years, we have to look for structural causes to understand what's gone wrong and how to fix it. One reason is the way that home computers were marketed as "boys' toys" starting in the 1980s [Marg2003]; another is the way that computer science departments responded to explosive growth in enrollment in the 1980s and again in the 2000s by changing admission requirements [Robe2017], and as noted at the start of this section, these factors have excluded many other people as well. None of these factors may seem dramatic to people who aren't affected by them, but they act like the steady drip of water on a stone: over time, they erode motivation, and with it, participation.

The first and most important step toward fixing this is to stop thinking in terms of a "leaky pipeline" [Mill2015]. More generally, we need to move past a deficit model i.e., to stop thinking that the members of under-represented groups lack something and are therefore responsible for not getting ahead. Believing that puts the burden on people who already have to work harder

because of the inequities they face, and (not coincidentally) gives those who benefit from the current arrangements an excuse not to look at themselves too closely.

Rewriting History
[Abba2012] describes the careers and accomplishments of the women who shaped the early history of computing, but have all too often been written out of that history; [Ensm2003, Ensm2012] describes how programming was turned from a female into a male profession in the 1960s, while [Hick2018] looks at how Britain lost its early dominance in computing by systematically discriminating against its most qualified workers: women. [Milt2018] is a good review of all three books. Discussing this can make some men in computing very uncomfortable; in my opinion, that's a good reason to do it more often.

Misogyny in video games, the use of "cultural fit" in hiring to excuse conscious or unconscious bias, a culture of silence around harassment, and the growing inequality in society that produces preparatory privilege (Section 9.5) may not be any one person's fault, but they are everyone's responsibility. This workshop has excellent practical advice on how to be a good ally in tech; we will return to this topic in Chapter 13.

10.5 Exercises

Authentic Tasks (pairs/15)

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.

1. Pair up with your neighbor and decide where this exercise fits on a 2x2 grid of "short/long time to master" and "low/high usefulness".
2. Write the task and where it fits on the grid.
3. Discuss how these relate back to the "teach most immediately useful first" approach.

Core Needs (whole class/10)

Paloma Medina identifies six core needs for people at work: belonging, improvement (i.e., making progress), choice, equality, predictability, and significance. After reading her description of these, order them from most to least significant for you personally, then compare rankings with your class using 6 points for most important, 5 for next, and so on down to 1 for least important. How do you think your rankings compare with those of your learners?

Implement One Strategy for Inclusivity (individual/5)

Pick one activity or change in practice from [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.

Brainstorming Motivational Strategies (think-pair-share/20)

1. Think back to a programming course (or any other) that you took in the past, and identify one thing the instructor did that demotivated 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.

Demotivational Experiences (think-pair-share/15)

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.

Walk the Route (whole class/15)

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 exercises? And did you notice that the first sentence in this exercise assumed you could actually walk?

Who Decides? (whole class/15)

In [Litt2004], 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 by Cameron Cottrill, and then describe an example from your own experience of “objective” assessments that reinforced the status quo.

Common Stereotypes (pairs/10)

You will (still) sometimes hear people say, "It's so simple that even your grandmother could use it." In pairs, list two or three other phrases that reinforce stereotypes about computing.

Not Being a Jerk (individual/15)

This short article by Gary Bernhardt rewrites an unnecessarily hostile message to be less rude. Using it as a model, find something unpleasant on Stack Overflow or some other public discussion forum and rewrite it to be less repellant.

Saving Face (individual/10)

Are there any aspects of what you want to teach that members of your hoped-for audience might be embarrassed to admit to not knowing already? Are there any that they would rather their peers didn't know they were learning? If so, what can you do to help them save face?

After the Fact (whole class/15)

[Cutt2017] surveyed adult computer users about their childhood activities and found that the strongest correlation between confidence and computer use were based on reading on one's own and playing with construction toys with no moving parts (like Lego). Spend a few minutes searching online for ideas programmers have about how to tell if someone is going to be a good coder, or what non-coding activities correlate with programming ability, and see if these two ever come up.

How Accessible Are Your Lessons? (pairs/30)

In pairs, choose a lesson whose materials are online and independently rank it according to the do's and don'ts in these posters. Where did you and your partner agree? Where did you disagree? How well did the lesson do for each of the six categories of user?

Tracing the Cycle (small groups/15)

[Coco2018] traces a depressingly common pattern in which good intentions are undermined by an organization's leadership being unwilling to actually change. Working in groups of 4–6, write brief emails that you imagine each of the parties involved would send to the other at each stage in this cycle.

11 Teaching Online

If you use robots to teach, you teach people to be robots.
— variously attributed

Technology has changed teaching and learning many times. Before blackboards were introduced into schools in the early 1800s, for example, there was no way for teachers to share an improvised example, diagram, or exercise with an entire class at once. Combining low cost, low maintenance, reliability, ease of use, and flexibility, blackboards enabled teachers to do things quickly and at scale that they had only been able to do slowly and piecemeal before. Similarly, the hand-held video camera revolutionized athletics training, just as the tape recorder revolutionized music instruction a decade earlier.

Many of the people pushing the Internet into classrooms don't know this history, and don't realize that it is just the latest in a long series of attempts to use machines to teach [Watt2014]. From the printing press through radio and television to desktop computers and mobile devices, every new way to share knowledge has produced a wave of aggressive optimists who believe that education is broken and that technology can fix it. However, ed tech's strongest advocates have often known less about "ed" than they do about "tech", and have often been driven more by the prospect of profit than by the desire to improve learning.

Today's debate is often muddied by the fact that "online" and "automated" don't have to be the same thing. Live online teaching can be a lot like leading a small-group discussion. Conversely, the only way to teach several hundred people at a time is to standardize and automate assessment; the learner's experience is largely the same from whether the automation uses software or a squad of teaching assistants working to a tightly-defined rubric.

This chapter therefore looks at how the Internet can and should be used to deliver automated instruction, i.e., to teach with recorded videos and assess via automatically-graded exercises. The next chapter will then explore ways of combining automated instruction with live teaching delivered either online or in person.

11.1 MOOCs

The highest-profile effort to reinvent education using the Internet is the Massive Open Online Course, or MOOC. The term was invented by David Cormier in 2008 to describe a course organized by George Siemens and Stephen Downes. That course was based on a connectivist view of learning, which holds that knowledge is distributed and learning is the process of finding, creating, and pruning connections.

The term "MOOC" was quickly co-opted by creators of courses that kept more closely to the hub-and-spoke model of a traditional classroom, with the instructor at the center defining goals and the learners seen as recipients or replicators of knowledge. Classes that use the original connectivist model are now sometimes referred to as "cMOOCs", while classes that centralize control are called "xMOOCs". (The latter kind of course is also sometimes called a "MESS", for Massively Enhanced Sage on the Stage.)

Two strengths of the MOOC model are that learners can work when it's convenient for them, and that they have access to a wider range of courses, both because the Internet brings them all next door and because online courses typically have lower direct and indirect costs than in-person courses. Five years ago, you couldn't cross the street on a major university campus without hearing some talking about how MOOCs would revolutionize education, destroy it, or possibly both.

But MOOCs haven't been nearly as effective as their more enthusiastic proponents claimed they would be [Ubel2017]. One reason is that recorded content is ineffective for many novices because it cannot clear up their individual misconceptions (Chapter 2): if they don't understand an explanation the first time around, there usually isn't a different one on offer. Another is that the automated assessment necessary in order to put the "massive" in MOOC only works well at the lower levels of Bloom's Taxonomy. It's also now clear that learners have to shoulder much more of the burden of staying focused in a MOOC, and that the impersonality of working online can demotivate people and encourage uncivil behavior.

[Marg2015] examined 76 MOOCs on various subjects, and found that the quality of lesson design was poor, though organization and presentation of material was good. Closer to home, [Kim2017] studied 30 popular online coding tutorials, and found that they largely teach the same content the same way: bottom-up, starting with low-level programming concepts and building up to high-level goals. Most require learners to write programs, and provide some form of immediate feedback, but this feedback is typically very shallow. Few explain when and why concepts are useful (i.e., they don't show how to transfer knowledge) or provide guidance for common errors, and other than rudimentary age-based differentiation, none personalize lessons based on prior coding experience or learner goals.

Personalized Learning

Few terms have been used and abused in as many ways as personalized learning. To most ed tech proponents, it means dynamically adjusting the pace or focus of lessons based on learner performance, which in practice means that if someone answers several questions in a row correctly, the computer will skip some of the subsequent questions.

Doing this can produce modest improvements in outcomes, but better is possible. For example, if many learners find a particular topic difficult, the teacher can prepare multiple alternative explanations of that point—essentially, multiple paths forward through the lesson rather than accelerating a single path—so that if one explanation doesn't resonate, others are available. However, this requires a lot more design work on the teacher's part, which may be why it's a less popular approach with the tech crowd.

And even if it does work, the effects are likely to be much less than some of its advocates believe. A good teacher makes a difference of 0.1–0.15 standard deviations in end-of-year performance in grade school [Chet2014] (see this article for a brief summary). It's simply unrealistic to believe that any kind of automation can outdo this any time soon.

So how should the web be used in teaching and learning tech skills? From an educational point of view, its pros and cons are:

Learners can access more lessons, more quickly, than ever before. Provided, of course, that a search engine considers those lessons worth indexing, that their internet service provider and government don't block it, that the truth isn't drowned in a sea of attention-sapping disinformation.

Learners can access *better* lessons than ever before, unless they are being steered toward second-rate material in order to redistribute wealth from the have-nots to the haves [McMi2017]. (It's worth remembering that scarcity increases perceived value, so as online education becomes cheaper, it will be seen as being worth less.)

Learners can access far more people than ever before as well. But only if those learners actually have access to the required technology, can afford to use it, and aren't driven offline by harassment or marginalized because they don't conform to the social norms of whichever group is talking loudest. In practice, most MOOC users come from secure, affluent backgrounds [Hansen2015].

Teachers can get far more detailed insight into how learners work. So long as learners are doing things that are amenable to large-scale automated analysis and either don't object to the use of ubiquitous surveillance in the classroom, or aren't powerful enough for their objections to matter.

[Marg2015, Mill2016a, Nils2017] describe ways to accentuate the positives in the list above while avoiding the negatives:

Make deadlines frequent and well-publicized, and enforce them, so that learners will get into a work rhythm.

Keep synchronous all-class activities like live lectures to a minimum so that people don't miss things because of scheduling conflicts.

Have learners contribute to collective knowledge, e.g., take notes together (Section 9.7), serve as classroom scribes, or contribute problems to shared problem sets (Section 5.3).

Encourage or require learners to do some of their work in small groups that *do* have synchronous online activities such as a weekly online discussion to help learners stay engaged and motivated without creating too many scheduling headaches. (See s:meetings for some tips on how to make these discussions fair and productive.)

Create, publicize, and enforce a code of conduct so that everyone can actually (as opposed to theoretically) take part in online discussions (Section 1.5).

Use lots of short lesson episodes rather than a handful of lecture-length chunks in order to minimize cognitive load and provide lots of opportunities for formative assessment. This also helps with maintenance: if all of your videos are short, you can simply re-record any that need maintenance, which is often cheaper than trying to patch longer ones.

Use video to engage rather than instruct, since, disabilities aside, learners can read faster than you can talk. The exception to this rule is that video is actually the best way to teach people verbs (actions): short screencasts that show people how to use an editor, step through code in a debugger, and so on are more effective than screenshots with text.

Identify and clear up misconceptions early (Chapter 2). If data shows that learners are struggling with some parts of a lesson, create alternative explanations of those points and extra exercises for them to practice on.

All of this has to be implemented somehow, which means that you need some kind of teaching platform. You can either use an all-in-one learning management system like Moodle or Sakai, or assemble something yourself using Slack or Zulip for chat, Google Hangouts or appear.in for video conversations, and WordPress, Google Docs, or any number of wiki systems for collaborative authoring. If you are just starting out, then use whatever requires the least installation and administration on your side, and the least extra learning effort on your learners' side. (I once ran a half-day class using group text messages because that was the only tool everyone was already familiar with.)

The most important thing when choosing technology is to *ask your learners what they are already using*. Normal people don't use IRC, and find its arcane conventions and interface offputting. Similarly, while this book

lives in a GitHub repository, requiring non-experts to submit pull requests has been an unmitigated disaster, even with GitHub's online editing tools. As a teacher, you're asking people to learn a lot; the least you can do in return is learn how to use the tools they prefer.

Points for Improvement

One way to demonstrate to learners that they are learning with *you, not just* from *you, is to allow them to edit your course notes. In live courses, we recommend that you enable them to do this as you lecture (Section 9.7); in online courses, you can put your notes into a wiki, a Google Doc, or anything else that allows you to review and comment on changes. Giving people credit for fixing mistakes, clarifying explanations, adding new examples, and writing new exercises doesn't reduce your workload, but increases engagement and the lesson's lifetime (Section 6.3).*

A major concern with any online community, learning or otherwise, is how to actually make it a community. Hundreds of books and presentations discuss this, but most are based on their authors' personal experiences. [Krau2016] is a welcome exception: while it predates the accelerating descent of Twitter and Facebook into weaponized abuse and misinformation, most of what was true then is true now. [Foge2005] is also full of useful tips for the community of practice that learners may hope to join.

Freedom To and Freedom From

Isaiah Berlin's 1958 essay "Two Concepts of Liberty" made a distinction between positive liberty, which is the ability to actually do something, and negative liberty, which is the absence of rules saying that you can't do it. Unchecked, online discussions usually offer negative liberty (nobody's stopping you from saying what you think) but not positive liberty (many people can't actually be heard). One way to address this is to introduce some kind of throttling, such as only allowing each learner to contribute one message per discussion thread per day. Doing this gives those with something to say a chance to say it, while clearing space for others to say things as well.

One other concern people have about teaching online is cheating. Day-to-day dishonesty is no more common in online classes than in face-to-face settings [Beck2014], but the temptation to have someone else write the final exam, and the difficulty of checking whether this happened, is one of the reasons educational institutions have been reluctant to offer credit for pure online classes. Remote exam proctoring is possible, usually by using a webcam to watch the learner take the exam. Before investing in this, read [Lang2013], which explores why and how learners cheat, and how courses can be structured to avoid giving them a reason to do so.

11.2 Video

A core element of cMOOCs is their reliance on recorded video lectures. As mentioned in Chapter 8, a teaching technique called Direct Instruction that is based on precise delivery of a well-designed script has repeatedly been shown to be effective [Stoc2018], so recorded videos can in principle be effective. However, scripts for direct instruction have to be designed, tested, and refined very carefully, which is an investment that many MOOC authors have been unwilling or unable to make. Making a small change to a web page or a slide deck only takes a few minutes; making even a small change to a short video takes an hour or more, so the cost to the teacher of acting on feedback can be unsupportable. And even when they're well made, videos have to be combined with activities to be beneficial: [Koed2015] estimated, "... the learning benefit from extra doing... to be more than six times that of extra watching or reading."

If you are teaching programming, you may use screencasts instead of slides, since they offer some of the same advantages as live coding (Section 8.4). [Chen2009] offers useful tips for creating and critiquing screencasts and other videos; Figure 12 reproduces the patterns that paper presents and the relationships between them, and is also a good example of a concept map (Section 3.1).

[Guo2014] measured engagement by looking at how long learners watched MOOC videos. Some of its key findings were:

- Shorter videos are much more engaging—videos should be no more than six minutes long.
- A talking head superimposed on slides is more engaging than voice over slides alone.
- Videos that felt personal could be more engaging than high-quality studio recordings, so filming in informal settings could work better than professional studio work for lower cost.
- Drawing on a tablet is more engaging than PowerPoint slides or code screencasts, though it's not clear whether this is because of the motion and informality, or because it reduces the amount of text on the screen.
- It's OK for teachers to speak fairly fast as long as they are enthusiastic.

One thing [Guo2014] didn't address is the chicken-and-egg problem: do learners find a certain kind of video engaging because they're used to it, so producing more videos of that kind will increase engagement simply because of a feedback loop? Or do these recommendations reflect some deeper cognitive processes? Another thing this paper didn't look at is learning outcomes: we know that learner evaluations of courses don't correlate with learning [Star2014, Uttl2017], and while it's plausible that learners won't learn from things they don't watch, it remains to be proven that they *do* learn from things they *do* watch.

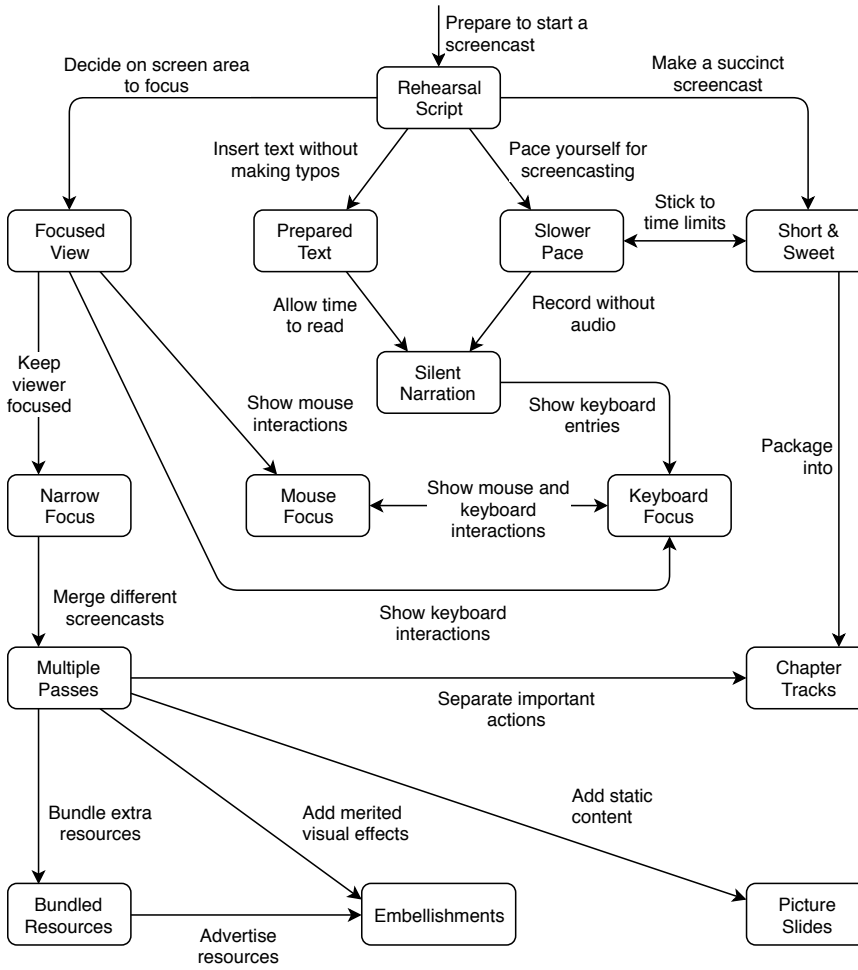

Figure 12: Patterns for Screencasting ([Chen2009](#BIB))

I'm a Little Uncomfortable

[Guo2014]'s research was approved by a university research ethics board, the learners whose viewing habits were monitored almost certainly clicked "agree" on a terms of service agreement at some point, and I'm glad to have these insights. On the other hand, I attended the conference at which this paper was published, and the word "privacy" didn't appear in the title or abstract of any *of the dozens of papers or posters presented. Given a choice, I'd rather not know how engaged learners are than see privacy become obsolete.*

There are many different ways to record video lessons; to find out which are most effective, [Mull2007a] assigned 364 first-year physics learners to online multimedia treatments of Newton's First and Second Laws in one of four styles:

Exposition: concise lecture-style presentation.
Extended Exposition: as above with additional interesting information.
Refutation: Exposition with common misconceptions explicitly stated and refuted.
Dialog: Learner-tutor discussion of the same material as in the Refutation.

Refutation and Dialogue produced the greatest learning gains compared to Exposition; learners with low prior knowledge benefited most, and those with high prior knowledge were not disadvantaged.

11.3 Flipped Classrooms

Fully automated teaching is one way to use the web in teaching; in practice, almost all learning in affluent societies has an online component: sometimes officially, and if not, through peer-to-peer back channels and surreptitious searches for answers to homework questions. Combining live and automated instruction allows instructors to use the strengths of both. In a classroom, the instructor can answer questions immediately, but it takes time for learners to get feedback on their coding exercises (sometimes days or weeks). Online, it can take a long time to get an answer, but learners can get immediate feedback on their coding (at least for those kinds of exercises we can auto-grade).

Similarly, online exercises have to be more detailed because they're anticipating questions. I find that in-person lessons start with the intersection of what everyone needs to know and expands on demand, while online lessons have to include the union of what everyone needs to know because you aren't there to do the expanding.

The most popular hybrid teaching strategy today is the flipped classroom, in which learners watch recorded lessons on their own, and class time is used for discussion and to work through problem sets. Originally

proposed in [King1993], the idea was popularized as part of peer instruction (Section 9.2), and has been studied intensively over the past decade. For example, [Camp2016] compared students who chose to take a CS1 class online with those who took it in person in a flipped classroom. Completion of (unmarked) practice exercises correlated with exam scores for both, but the completion rate of rehearsal exercises by online students was significantly lower than lecture attendance rates for in-person students. Looking at what did affect the grade, they found that the students' perception of the material's intrinsic value was only a factor for the flipped section (and only once results were controlled for prior programming experience). Conversely, test anxiety and self-efficacy were factors only for the online section; the authors recommend trying to improve self-efficacy by increasing instructor presence online.

But are lectures worth attending at all? Or should we just provide recordings? [Nord2017] examined the impact of recordings on both lecture attendance and students' performance at different levels. In most cases the study found no negative consequences of making recordings available; in particular, students don't skip lectures when recordings are available (at least, not any more than they usually do). The benefits of providing recordings are greatest for students early in their careers, but diminish as students become more mature.

11.4 Life Online

[Nuth2007] found that there are three overlapping worlds in every classroom: the public (what the teacher is saying and doing), the social (peer-to-peer interactions between learners), and the private (inside each learner's head). Of these, the most important is usually the social: learners pick up as much via cues from their peers as they do from formal instruction.

The key to making any form of online teaching effective is therefore to facilitate peer-to-peer interactions. To aid this, courses almost always have some kind of discussion forum. [Vell2017] analyzes discussion forum posts from 395 CS2 students at two universities by dividing them into four categories:

Active: request for help that does not display reasoning and doesn't display what the student has already tried or already knows.

Constructive: reflect students' reasoning or attempts to construct a solution to the problem.

Logistical: course policies, schedules, assignment submission, etc.

Content clarification: request for additional information that doesn't reveal the student's own thinking.

They found that constructive and logistical questions dominated, and that constructive questions correlated with grades. They also found that

students rarely ask more than one active question in a course, and that these *don't* correlate with grades. While this is disappointing, knowing it helps set instructors' expectations: while we might all want our courses to have lively online communities, most won't.

Learners use forums in very different ways, and with very different results. [Mill2016a] observed that, "... procrastinators are particularly unlikely to participate in online discussion forums, and this reduced participation, in turn, is correlated with worse grades. A possible explanation for this correlation is that procrastinators are especially hesitant to join in once the discussion is under way, perhaps because they worry about being perceived as newcomers in an established conversation. This aversion to jump in late causes them to miss out on the important learning and motivation benefits of peer-to-peer interaction."

Co-opetition

[Gull2004] describes an innovative online coding contest that combines collaboration and competition. The contest starts when a problem description is posted along with a correct, but inefficient, solution. When it ends, the winner is the person who has made the greatest overall contribution to improving the performance of the overall solution. All submissions are in the open, so that participants can see one another's work and borrow ideas from each other; as the paper shows, the final solution is almost always a hybrid borrowing ideas from many people.

[Batt2018] described a small-scale variation of this used in an introductory computing class. In stage one, each student submitted a programming project individually. In stage two, students were paired to create an improved solution to the same problem. The assessment indicates that two-stage projects tend to improve students' understanding, and that they enjoyed the process.

Discussion isn't the only way to get students to work together online. [Pare2008] and [Kulk2013] report experiments in which learners grade each other's work, and the grades they assign are then compared with grades given by graduate-level teaching assistants or other experts. Both found that student-assigned grades agreed with expert-assigned grades as often as the experts' grades agreed with each other, and that a few simple steps (such as filtering out obviously unconsidered responses or structuring rubrics) decreased disagreement even further. And as discussed in Section 5.3, collusion and bias are *not* significant factors in peer grading.

[Cumm2011] looked at the use of shareable feedback tags on homework; students could attach tags to specific locations in coding assignments (like code review) so that there's no navigational cost for the reader, and they controlled whether to share their work and feedback anonymously. Students found tag clouds of feedback on their own work useful, but that the tags were really only meaningful in context. This is unsurprising: the greater

the separation between action and feedback, the greater the cognitive load. What wasn't expected was that the best and worst students were more likely to share than middling students.

Trust, but Educate

The most common way to measure the validity of feedback is to compare students' grades to experts' grades, but calibrated peer review (Section 5.3) can be equally effective. Before asking learners to grade each others' work, they are asked to grade samples and compare their results with the grades assigned by the teacher. Once the two align, the learner is allowed to start giving grades to peers. Given that critical reading is an effective way to learn, this result may point to a future in which learners use technology to make judgments, rather than being judged by technology.

One technique we will definitely see more of in coming years is online streaming of live coding sessions [Haar2017]. This has most of the benefits discussed in Section 8.4, and when combined with collaborative note-taking (Section 9.7) it can come pretty close to approximating an in-class experience.

Looking even further ahead, [Ijss2000] identified four levels of online presence, from realism (we can't tell the difference) through immersion (we forget the difference) and involvement (we're engaged but aware of the difference) to suspension of disbelief (we are doing most of the work). Crucially, they distinguish physical presence, which is the sense of actually being somewhere, and social presence, which is the sense of being with others. In most learning situations, the latter is more important, and one way to foster it is to bring the technology learners use every day into the classroom. For example, [Deb2018] found that doing in-class exercises with realtime feedback using mobile devices improved concept retention and student engagement while reducing failure rates.

Hybrid Presence

Combining online and in-person instruction can be more effective than either on its own. I have delivered very successful classes using real-time remote instruction, in which the learners are co-located at 2–6 sites, with helpers present, while I taught via streaming video (Section C.2). This scales well, saves on travel costs, and is less disruptive for learners (particularly those with family responsibilities). What doesn't *work is having one group in person and one or more groups remotely: with the best will in the world, the local participants get far more attention.*

Online teaching is still in its infancy: [Luxt2009] surveyed peer assessment tools that could be useful in computing education, and [Broo2016] describes many other ways groups can discuss things, but only a handful of these ideas are widely known or used.

I think that our grandchildren will probably regard the distinction we make between what we call the real world and what they think of as simply the world as the quaintest and most incomprehensible thing about us.
— William Gibson

11.5 Exercises

Give Feedback on a Bad Screencast (whole class/20)

Watch this screencast as a group and give feedback on it. Organize feedback along two axes: positive vs. negative and content vs. presentation. When you are done, 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? (You can compare your answers with the checklist in s:teacheval.)

Two-Way Video (pairs/10)

Record a 2–3 minute video of yourself doing something, then swap machines with a partner so that each of you can watch the other's video at 4X speed. How easy is it to follow what's going on? What if anything did you miss?

Viewpoints (individual/10)

According to [Irib2009], different disciplines focus on different factors affecting the success or otherwise of online communities:

Business: customer loyalty, brand management, extrinsic motivation.
Psychology: sense of community, intrinsic motivation.
Sociology: group identity, physical community, social capital, collective action.
Computer Science: technological implementation.

Which of these perspectives most closely corresponds to your own? Which are you least aligned with?

Helping or Harming (small groups/30)

Susan Dynarski's article in the *New York Times* explains how and why schools are putting students who fail in-person courses into online courses, and how this sets them up for even further failure.

1. Working in small groups, read the article, come up with 2–3 things that schools could do to compensate for these negative effects, and create rough estimates of their per-student costs.
2. Compare your suggestions and costs with those of other groups. How many full-time teaching positions do you think would have to be cut in order to free up resources to implement the most popular ideas for 100 students?
3. As a class, do you think that would be a net benefit for the students or not?

Budgeting exercises like this are a good way to tell who's serious about educational change. Everyone can think of things they'd like to do; far fewer are willing to talk about the tradeoffs needed to make change happen.

12 Exercise Types

Every good carpenter has a set of screwdrivers, and every good teacher has different kinds of formative assessment exercises to check what learners are actually learning, help them practice their new skills, and keep them engaged. This chapter starts by describing several kinds of exercises you can use to check if your teaching has been effective. It then looks at the state of the art in automated grading, and closes by exploring discussion, projects, and other important kinds of work that require more human attention to assess. Our discussion draws in part on the Canterbury Question Bank [Sand2013], which has entries for various languages and topics in introductory computing.

12.1 The Classics

As Section 2.1 discussed, *multiple choice questions* (MCQs) are most effective when the wrong answers probe for specific misconceptions. In terms of Bloom's Taxonomy (Section 6.2), MCQs are usually designed to test recall and understanding ("What is the capital of Saskatchewan?"), but they can also require learners to exercise judgment.

A Multiple Choice Question
In what order do operations occur when the computer evaluates the expression `price = addTaxes(cost - discount)`*?*

1. *subtraction, function call, assignment*
2. *function call, subtraction, assignment*
3. *function call, then assignment and subtraction simultaneously*
4. *none of the above*

The second classic type of programming exercise is *code and run* (C&R), in which the learner writes code that produces a specified output. C&R exercises can be as simple or as complex as the teacher wants, but for in-class use, they should be brief and have only one or two plausible correct answers. For novices, it's often enough to ask them to call a specific function: experienced teachers often forget how hard it can be to figure out which

parameters go where. For more advanced learners, figuring out which function to call is more engaging and a better gauge of their understanding.

Code & Run
The variable `picture` contains a full-color image read from a file. Using one function, create a black and white version of the image and assign it to a new variable called `monochrome`.

Write and run exercises can be combined with MCQs. For example, this MCQ can only be answered by running the Unix `ls` command:

Combining MCQ with Code & Run
You are in the directory `/home/greg`. Which of the following files is not *in that directory?*

1. `autumn.csv`
2. `fall.csv`
3. `spring.csv`
4. `winter.csv`

C&Rs help learners practice the skills they most want to learn, but they can be hard to assess: learners can find lots of unexpected ways to get the right answer, and are demoralized if an automatic grading system rejects their code because it doesn't match the instructor's. One way to reduce how often this occurs is to assess only their output, but that doesn't give them feedback on how they are programming. Another is to give them a small test suite they can run their code against before they submit it (at which point it is run against a more comprehensive set of tests). Doing this helps them figure out if they have completely misunderstood the intent of the exercise before they do anything that they think might cost them grades.

Instead of writing code that satisfies some specification, learners can be asked to write tests to determine whether a piece of code conforms to a spec. This is a useful skill in its own right, and doing it may give students a bit more sympathy for how hard their teachers work.

Inverting Code & Run
The function `monotonic_sum` calculates the sum of each section of a list of numbers in which the values are strictly increasing. For example, given the input `[1, 3, 3, 4, 5, 1]`, the output is `[4, 12, 1]`. Write and run unit tests to determine which of the following bugs the function contains:

- *Considers every negative number the start of a new sub-sequence.*
- *Does not include the first value of each sub-sequence in the sub-sum.*
- *Does not include the last value of each sub-sequence in the sub-sum.*
- *Only re-starts the sum when values decrease rather than fail to increase.*

Fill in the blanks is a refinement of C&R in which the learner is given some starter code and has to complete it. (In practice, most C&R exercises are actually fill in the blanks because the teacher will provide comments to remind the learners of the steps they should take.) As discussed in Chapter 4, novices often find filling in the blanks less intimidating than writing all the code from scratch, and since the teacher has provided most of the answer's structure, submissions are much more predictable and therefore easier to check.

Fill in the Blanks
Fill in the blanks so that the code below prints the string `'hat'`.

```
text = 'all that it is'
slice = text[____:____]
print(slice)
```

As described in Chapter 4, Parsons Problems also avoid the "blank screen of terror" problem. The learner is given the lines of code needed to solve a problem, but has to put them in the right order. Research over the past few years has shown that Parsons Problems are effective because they allow learners to concentrate on control flow separately from vocabulary [Pars2006, Eric2015, Morr2016, Eric2017]. The same research shows that giving the learner more lines than she needs, or asking her to rearrange some lines and add a few more, makes this kind of problem significantly harder [Harm2016]. Tools for building and doing Parsons Problems online exist [Ihan2011], but they can be emulated (albeit somewhat clumsily) by asking learners to rearrange lines of code in an editor.

Parsons Problem
Rearrange and indent these lines to sum the positive values in a list. (You will need to add colons in appropriate places as well.)

```
total = 0
if v > 0
total += v
for v in values
```

12.2 Tracing

Tracing execution is the inverse of a Parsons Problem: given a few lines of code, the learner has to trace the order in which those lines are executed. This is an essential debugging skill, and is a good way to solidify learners' understanding of loops, conditionals, and the evaluation order of function and method calls. The easiest way to implement it is to have learners write out a sequence of labelled steps. Having them choose the correct sequence from a set (i.e., presenting this as an MCQ) adds cognitive load without

adding value, since they have to do all the work of figuring out the correct sequence, then search for it in the list of options.

Tracing Execution Order
In what order are the labelled lines in this block of code executed?

```
A)      vals = [-1, 0, 1]
B)      inverse_sum = 0
        try:
            for v in vals:
C)              inverse_sum += 1/v
        except:
D)          pass
```

Tracing values is similar to tracing execution, but instead of spelling out the order in which code is executed, the learner has to list the values that one or more variables take on as the program runs. It can also be implemented by having learners provide a list of values, but another approach is to give the learner a table whose columns are labelled with variable names and whose rows are labelled with line numbers, and asking them to fill in all of the values taken on by all of the variables.

Tracing Values
What values do `left` and `right` take on as this program executes?

```
left = 24
right = 6
while right:
    left, right = right, left % right
```

You can also require learners to trace code backwards, e.g., to figure out what the input must have been if the code produced a particular result [Armo2008]. These *reverse execution* problems require search and deductive reasoning, but they are particularly useful when the "output" is an error message, and help learners develop valuable debugging skills.

Reverse Execution
Fill in the missing number in `values` that caused this function to crash.

```
values = [ [1.0, -0.5], [3.0, 1.5], [2.5, ___] ]
runningTotal = 0.0
for (reading, scaling) in values:
    runningTotal += reading / scaling
```

Minimal fix exercises also help learners develop debugging skills. Given a few lines of code that contain a bug, the learner must either make or identify the smallest change that will produce the correct output. Making the change can be done using C&R, while identifying it can be done as a multiple choice question.

Minimal Fix
This function is supposed to test whether a number lies within a range. Make one small change so that it actually does so.

```
def inside(point, lower, higher):
    if (point <= lower):
        return false
    elif (point <= higher):
        return false
    else:
        return true
```

Theme and variation exercises are similar, but instead of making a change to fix a bug, the learner is asked to make a small alteration that changes the output in some specific way. Allowed changes can include replacing one function call with another, changing one variable's initial value, swapping an inner and outer loop, changing the order of tests in a chain of conditionals, or changing the nesting of function calls or the order in which methods are chained. Again, this kind of exercise gives learners a chance to practice a useful real-world skill: the fastest way to produce a working program is often to tweak one that already does something useful.

Theme and Variations
Change the inner loop in the function below so that it fills the upper left triangle of an image with a specified color.

```
function fillTriangle(picture, color) is
    for x := 1 to picture.width do
        for y := 1 to picture.height do
            picture[x, y] = color
        end
    end
end
```

Refactoring exercises are the complement of theme and variation exercises: given a working piece of code, the learner has to modify it in some way *without* changing its output. For example, the learner could be asked to replace loops with vectorized expressions, to simplify the condition in a while loop, etc. The exercise here is that there are often so many ways to refactor a piece of code that grading requires human intervention.

Refactoring
Write a single list comprehension that has the same effect as this loop.

```
result = []
for v in values:
    if len(v) > threshold:
        result.append(v)
```

12.3 Diagrams

Having students draw concept maps and other diagrams gives insight into how they're thinking (Section 3.1), but free-form diagrams take human time and judgment to assess. *Labelling diagrams*, on the other hand, is almost as useful from a pedagogical point of view but much easier to scale.

Rather than having learners create diagrams from scratch, provide them with a diagram and a set of labels and have them put the latter in the right places on the former. The diagram can be a complex data structure ("after this code is executed, which variables point to which parts of this structure?"), the graph that a program produces ("match each of these pieces of code with the part of the graph it generated"), the code itself ("match each term to an example of that program element"), or many other things; the key is that constraining the set of solutions makes this usable in class and at scale.

Labelling a Diagram
Figure 13 shows how a small fragment of HTML is represented in memory. Put the labels 1–10 on the elements of the tree to show the order in which they are reached in a depth-first traversal.

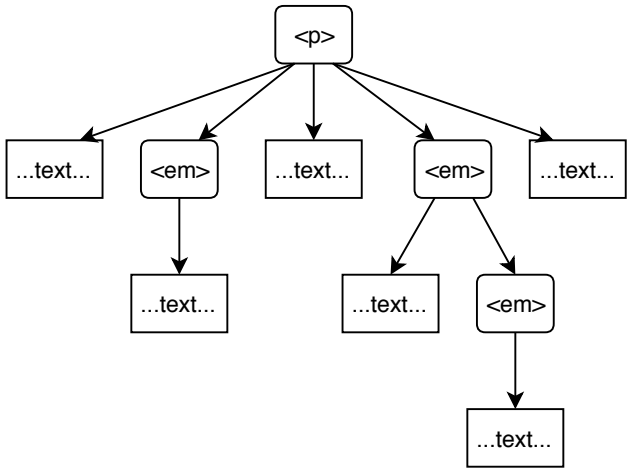

Figure 13: Labelling a Diagram

Another way to use diagrams for formative assessment is to give learners the pieces of the diagram and ask them to arrange them correctly. This is a visual equivalent of a Parsons Problem, and you can provide as much or as little of a skeleton to help them with placement as you think they're ready

for. (I have fond memories of trying to place resistors and capacitors in a circuit diagram in order to get the right voltage at a certain point, and have often seen teachers give learners a fixed set of Scratch blocks and ask them to create a particular drawing using only those blocks.)

Matching problems can be thought of as a special case of labelling in which the "diagram" is a column of text and the labels are taken from the other column. *One-to-one matching* gives the learner two lists of equal length and asks her to pair corresponding items, e.g., "match each piece of code with the output it produces".

Matching

Match each regular expression operator with what it does.

`?`	*start of line*
`*`	*zero or one occurrences*
`+`	*end of line*
`$`	*one or more occurrences*
`^`	*zero or more occurrences*

Many-to-many matching is similar, but the lists aren't the same length, so some items may be matched to several others, while others may not be matched at all. Both kinds require learners to use higher-order thinking skills, but many-to-many are more difficult because learners can't do easy matches first to reduce their search space (i.e., there is a higher cognitive load.)

Matching problems can be implemented by having learners submit lists of matching pairs as text (such as "A3, B1, C2"), but that's clumsy and error-prone. Having them recognize a set of correct pairs in an MCQ is even worse, as it's painfully easy to misread.

Ranking is a special case of matching that is (slightly) more amenable to answering via lists, since our minds are pretty good at detecting errors or anomalies in sequences. Give the learner several items and ask them to order them from fastest to slowest, most robust to most brittle, and so on. The former tends toward recall (e.g., recognizing the names of various sorting algorithms and knowing their properties), while the latter tends more toward reasoning and judgment.

Summarization also requires learners to use higher-order thinking, and gives them a chance to practice a skill that is very useful when reporting bugs rather than fixing them. For example, learners can be asked, "Which sentence best describes how the output of f changes as x varies from 0 to 10?" and then given several options as a multiple choice question.

You can also ask for very short free-form answers to questions in constrained domains, e.g., "What is the key feature of a stable sorting algorithm?" We still can't fully automate checks for these without a frustrating

number of false positives (accepting wrong answers) and false negatives (rejecting correct ones), but they lend themselves well to peer grading (Section 5.3).

12.4 Automatic Grading

Automatic program grading tools have been around longer than I've been alive: the earliest published mention dates from 1960 [Holl1960], and the surveys published in [Douc2005, Ihan2010] mention many specific tools by name. Building such tools is a lot more complex than it might first seem. How are assignments represented? How are submissions tracked and reported? Can learners co-operate? How can submissions be executed safely? [Edwa2014a] is an entire paper devoted to an adaptive scheme for detecting and managing infinite loops and other non-terminating code submissions, and that's just one of the many issues that comes up.

As elsewhere, it's important to distinguish learner satisfaction from learning outcomes. [Magu2018] switched informal programming labs to a weekly machine-evaluated test for a second-year CS course using an auto-grading tool originally developed for programming competitions. Learners didn't like the automated system, but the overall failure rate for the course was halved, and the number of learners gaining first class honors tripled. In contrast, [Rubi2014] also began to use an auto-grader designed for competitions, but saw no significant decrease in their learners' dropout rates; once again, learners made some negative comments about the tool, which the authors attribute to its feedback messages rather than to dislike of auto-grading.

[Srid2016] took a different approach. They used fuzz testing (i.e., randomly-generated test cases) to check whether learner code does the same thing as a reference implementation supplied by the teacher. In the first project of a 1400-learner introductory course, fuzz testing caught errors that were missed by a suite of hand-written test cases for more than 48% of learners, which clearly demonstrates its value.

[Basu2015] gave learners a suite of solution test cases, but learners had to unlock each one by answering questions about its expected behavior before they were allowed to apply it to their proposed solution. For example, suppose learners are writing a function to find the largest adjacent pair of numbers in a list; before being allowed to use the tests associated with this question, they have to choose the right answer to, "What does `largestPair(4, 3, -1, 5, 3, 3)` produce?" (The correct answer is `(5, 3)`.) In a 1300-person university course, the vast majority of learners chose to validate their understanding of test cases this way before attempting to solve problems, and then asked fewer questions and expressed less confusion about assignments.

It's tempting to use off-the-shelf style checking tools to grade learners' code. However, [Nutb2016] initially found no correlation between human-provided marks and style-checker rule violations. Sometimes this was because learners violated one rule many times (thereby losing more points than they should have), and other times it was because they submitted the assignment starter code with few alterations and got more points than they should have.

[Buff2015] presents a well-informed reflection on the whole idea of providing automated feedback. Their starting point is that, "Automated grading systems help learners identify bugs in their code, [but] may inadvertently discourage learners from thinking critically and testing thoroughly and instead encourage dependence on the teacher's tests." One of the key issues they identified is that a learner may thoroughly test their code, but the feature may still not be implemented according to the teacher's specifications. In this case, the "failure" is not caused by a lack of testing, but by a misunderstanding of the requirements, and it is unlikely that more testing will expose the problem. If the auto-grading system doesn't provide insightful, actionable feedback, this experience will only frustrate the learner.

In order to provide that feedback, [Buff2015]'s system identifies which method or methods of the learner's code are executed by failing tests, so that the system can associate failed tests with particular features within the learner's submission. The system decides whether specific hints have been "earned" by seeing whether the learner has tested the associated feature enough, so learners cannot rely on hints instead of doing tests.

[Keun2016a, Keun2016b] classified the messages produced by 69 auto-grading tools. They found that these often do not give feedback on how to fix problems and take the next step. They also found that most teachers cannot easily adapt most of the tools to their needs; like many workflow tools, they tend to enforce their creators' unrecognized assumptions about how institutions work. Their work is ongoing, and their detailed classification scheme is a useful shopping list when looking at tools of this kind.

[Srid2016] discussed strategies for sharing feedback with learners when automatically testing their code. The first is to provide the expected output for the tests—but then learners hard-code output for those inputs (because anything that can be gamed, will be). An alternative is to report the pass/fail results for the learners' code, but only supply the actual inputs and outputs of the tests after the submission date. This can be frustrating, because it tells learners they are wrong, but not why.

A third option is to use a technique called hashing to generate a value that depends on the output, but doesn't reveal it. If the user produces exactly the same output, its hash will be the same as the hash of the correct output, which will unlock the solution, but it is impossible to work backward from the hash to figure out what the output is supposed to be. Hashing is used to

create digital signatures for documents, and requires a bit more work and explanation to set up, but strikes a good balance between revealing answers prematurely and not revealing them when it would help.

12.5 Higher-Level Thinking

Many other kinds of programming exercises are hard for teachers to assess in a class with more than few dozen learners, and equally hard for automated platforms to assess at all. Larger programming projects, or projects in which learners set their own goals, are (hopefully) what classes are building toward. Free-form discussion or twitch coding (Section 8.4) is also valuable, but also doesn't scale.

Code review, on the other hand, is hard to grade automatically in the general case, but can be tackled if learners are given a rubric (e.g., a list of faults to look for) and asked to match particular comments against particular lines of code. For example, the learner can be told that there are two indentation errors and one bad variable name, and asked to point them out; if she is more advanced, she could be given half a dozen kinds of remarks she could make about the code without guidance as to how many of each she should find.

[Steg2016b] is a good starting point for a code style rubric, while [Luxt2009] looks at peer review in programming classes more generally. If you are going to have students do reviews, use calibrated peer review (Section 5.3) so that they have models of what good feedback should look like.

Code Review

Using the rubric provided, mark each line of the code below.

```
01)  def addem(f):
02)      x1 = open(f).readlines()
03)      x2 = [x for x in x1 if x.strip()]
04)      changes = 0
05)      for v in x2:
06)          print('total', total)
07)          tot = tot + int(v)
08)      print('total')
```

1. *poor variable name*
2. *unused variable*
3. *use of undefined variable*
4. *missing return value*

12.6 Exercises

Code and Run (pairs/10)

Create a short C&R exercise; trade with a partner, and see how long it takes each of you to understand and do the other's exercise. Were there any ambiguities or misunderstandings in the exercise description?

Inverting Code and Run (small groups/15)

Form groups of 4–6 people. Have each member of the group create an inverted C&R exercise that requires people to figure out what input produces a particular output. Pick two at random, and see how many different inputs the group can find that satisfy the requirements.

Tracing Values (pairs/10)

Write a short program (10–15 lines); trade with a partner, and trace how the variables in the program change value over time. What differences are there in how you and your partner wrote down your traces?

Refactoring (small groups/15)

Form groups of 3–4 people. Have each person select a short piece of code (10–30 lines long) that they have written that isn't as tidy as it could be. Choose one at random, and have everyone in the group tidy it up independently. How do your cleaned-up versions differ? How well or how poorly would you be able to accommodate all of these variations if marking automatically or in a large class?

Labelling a Diagram (pairs/10)

Draw a diagram showing something that you have explained recently: how browsers fetch data from servers, the relationship between objects and classes, or how data frames are indexed in R. Put the labels on the side, and ask your partner to place them.

Pencil-and-Paper Puzzles (whole class/15)

[Butl2017] describes a set of pencil-and-paper puzzles that can be turned into introductory programming assignments, and found that these assignments are enjoyed by students and encourage meta-cognition. Think of a simple pencil-and-paper puzzle or game you played as a child, and describe how you would turn it into a programming exercise.

Counting Failures (pairs/15)

Any useful estimate of how much time an exercise needs must take into account how frequent failures are and how much time is lost to them. For example, editing text files seems like a simple task, but what about finding those files? Most GUI editors save things to the user's desktop or home directory; if the files used in a course are stored somewhere else, 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 Chapter 3.)

Working with a partner, make a list of "simple" things you have seen go wrong in exercises you have used or taken. How often do they come up? How long do they take learners to fix on their own, or with help? How much time do you currently budget in class to deal with them?

13 Building Community

Many well-intentioned people want the world to be a better place, but don't actually want anything important to change. A lot of grassroots efforts to teach programming fall into this category: they want to teach children and adults how to program so that they can get good jobs, rather than empower them to change the system that has shut them (and people like them) out of those jobs in the past.

If you are going to build a community, the first and most important thing you have to decide is what *you* want: to help people succeed in the world we have, or to give them a way to make a better one. Either way, you have to accept that one person can only do so much. Just as we learn best together, we teach best when we are teaching with other people, and the best way to achieve that is to build a community.

And as Anu Partanen pointed out, sometimes you need to fix several things in order to fix one. Finland's teachers aren't successful in isolation: they are able to achieve outstanding results because their country's citizens truly value equality of opportunity. People (and countries) that try to adopt their teaching methods without ensuring that children (and parents) are well nourished, safe, and treated fairly by the courts will have a more difficult time. This doesn't mean you have to fix all of society's ills in order to teach programming, but it *does* mean that you have to understand and be involved in what happens to your learners outside of your class if you want that class to work.

A framework in which to think about educational communities is situated learning, which focuses on how legitimate peripheral participation leads to people becoming members of a community of practice [Weng2015]. Unpacking those terms, a community of practice is a group of people bound together by interest in some activity, such as knitting or particle physics. Legitimate peripheral participation means doing simple, low-risk tasks that community nevertheless recognizes as valid contributions: making your first scarf, stuffing envelopes during an election campaign, or proof-reading documentation for open source software.

Situated learning focuses on the transition from being a newcomer to

being accepted as a peer by those who are already community members. This typically means starting with simplified tasks and tools, then doing similar tasks with more complex tools, and finally tackling the exercises of advanced practitioners. For example, children learning music may start by playing nursery rhymes on a recorder or ukulele, then play other simple songs on a trumpet or saxophone in a band, and finally start exploring their own musical tastes. Healthy communities of practice understand and support these progressions, and recognize that each step is meant to give people a ramp rather than a cliff. Some of the ways they do this include:

Problem solving: "I'm stuck—Can we work on this design and brainstorm some ideas?"
Requests for information: "Where can I find the code to connect to the server?"
Seeking experience: "Has anyone dealt with a customer in this situation?"
Reusing assets: "I have a proposal for an event website that I wrote for a client last year you can use as a starting point."
Coordination and synergy: "Can we combine our purchases of web hosting to get a discount?"
Building an argument: "How do people in other companies do this? Armed with this information it will be easier to convince my CEO to make some changes."
Growing confidence: "Before I do it, I'll run it through my community first to see what they think."
Discussing developments: "What do you think of the new work tracking system? Does it really help?"
Documenting projects: "We have faced this problem five times now. Let us write it down once and for all."
Visits: "Can we come and see your after-school program? We need to establish one in our city."
Mapping knowledge and identifying gaps: "Who knows what, and what are we missing? What other groups should we connect with?"

Whatever the domain, situated learning emphasizes that learning is a social activity. In order to be effective and sustainable, teaching therefore needs to be rooted in a community; if one doesn't exist, you need to build one. There are at least four types:

Community of action: people focused on a shared goal, such as getting someone elected.
Community of concern: members are brought together by a shared exercise, such as dealing with depression.
Community of interest: focused on a shared love of something like backgammon or knitting.
Community of place: of people who happen to live or work side by side.

Most real communities are mixes of these, such as people in Toronto who like teaching tech; what matters is that you pick something and stick with it.

13.1 Learn, Then Do

The first step in building a community is to decide if you really need to, or whether you would be more effective joining an existing organization. Thousands of groups are already teaching people tech skills, from the 4-H Club and literacy programs to get-into-coding non-profits like Black Girls Code and Bridge. Joining an existing group will give you a head start on teaching, an immediate set of colleagues, and a chance to learn more about how to run things; hopefully, learning those skills will be more important than being able to say that you're the founder or leader of something new.

Whether you join an existing group or set up one of your own, you owe it to yourself and everyone who's going to work with you to find out what's been done before. People have been writing about grassroots organizing for decades; [Alin1989] is probably the best-known work on the subject, while [Brow2007, Midw2010] are practical manuals rooted in decades of practice. If you want to read more deeply, [Adam1975] is a history of the Highlander Folk School, whose approach has been emulated by many successful groups, while [Spal2014] is a guide to teaching adults written by someone with deep personal roots in organizing, and NonprofitReady.org offers free professional development training.

13.2 Three Steps

Everyone who gets involved with your organization, including you, goes through three phases: recruitment, retention, and retirement (from the organization). You don't need to worry about this cycle when you're just getting started, but it *is* worth thinking about as soon as you have more than a couple of non-founders involved.

The first step is recruiting volunteers. Your marketing should help you with this by making your organization findable, and by making its mission and its value to volunteers clear to people who might want to get involved. Share stories that exemplify the kind of help you want as well as stories about the people you're helping, and make it clear that there are many ways to get involved. (We discuss this in more detail in the next section.)

Your best source of new recruits is your own classes: "see one, do one, teach one" has worked well for volunteer organizations for as long as there have *been* volunteer organizations. Make sure that every class or other encounter ends with two sentences explaining how people can help, and that help is welcome. People who come to you this way will know what

you do, and will have recent experience of being on the receiving end of what you offer that they can draw on, which helps your organization avoid collective expert blind spot.

Start Small

As Ben Franklin observed, a person who has performed a favor for someone is more likely to do another favor for that person than they would be if they had received a favor from that person. Asking people to do something small for you is therefore a good step toward getting them to do something larger. One natural way to do this when teaching is to ask people to submit fixes for your lesson materials for typos or unclear wording, or to suggest new exercises or examples. If your materials are written in a maintainable way (Section 6.3), this gives them a chance to practice some useful skills, and gives you an opportunity to start a conversation that might lead to a new recruit.

Recruiting doesn't end when someone first shows up: if you don't follow through, people will come out once or twice, then decide that what you're doing isn't for them and disappear. One thing you can do to get newcomers over this initial hump is to have them take part in group activities before they do anything on their own, both so that they get a sense of how your organization does things, and so that they build social ties that will keep them involved.

Another thing you can do is give newcomers a mentor, and make sure the mentors actually do some proactive mentoring. The most important things a mentor can do are make introductions and explain the unwritten rules, so make it clear to mentors that these are their primary responsibilities, and they are to report back to you every few weeks to tell you what they've done.

The second part of the volunteer lifecycle is retention, which is a large enough topic to deserve a long discussion in Section 13.3. The third and final part is retirement. Sooner or later, everyone moves on (including you). When this happens:

Ask people to be explicit about their departure so that everyone knows they've actually left.

Make sure they don't feel embarrassed or ashamed about leaving or about anything else.

Give them an opportunity to pass on their knowledge. For example, you can ask them to mentor someone for a few weeks as their last contribution, or to be interviewed by someone who's staying with the organization to collect any stories that are worth re-telling.

Make sure they hand over the keys. It's awkward to discover six months after someone has left that they're the only person who knows how to book a playing field for the annual softball game.

Follow up 2–3 months after they leave to see if they have any further thoughts about what worked and what didn't while they were with you, or any advice to offer that they either didn't think to give or were uncomfortable giving on their way out the door.

Thank them, both when they leave and the next time your group gets together.

13.3 Retention

Saul Alinsky once said, "If your people aren't having a ball doing it, there is something very wrong." [Alin1989] Community members shouldn't expect to enjoy every moment of their work with your organization, but if they don't enjoy any of it, they won't stay.

Enjoyment doesn't necessarily mean having an annual party: people may enjoy cooking, coaching, or just working quietly beside others. There are several things every organization should do to ensure that people are getting something they value out of their work:

Ask people what they want rather than guessing. Just as you are not your learners (Section 6.1), you are probably different from other members of your organization. Ask people what they want to do, what they're comfortable doing (which may not be the same thing), what constraints there are on their time, and so on. They might start by saying, "I don't know—anything!" but even a short conversation will probably uncover the fact that they like interacting with people but would rather not be managing the group's finances, or vice versa.

Provide many ways to contribute. The more ways there are for people to help, the more people will be able to help. Someone who doesn't like standing in front of an audience may be able to maintain your organization's website or handle its accounts; someone who doesn't know how to do anything else may be able to proof-read lessons, and so on. The more kinds of tasks you do yourself, the fewer opportunities there are for others to get involved.

Recognize contributions. Everyone likes to be appreciated, so communities should acknowledge their members' contributions both publicly and privately by mentioning them in presentations, putting them on the website, and so on.

Make space. Micromanaging or trying to control everything centrally means people won't feel they have the autonomy to act, which will probably cause them to drift away. In particular, if you're too engaged or too quick on the reply button, people have less opportunity to grow as members and to create horizontal collaborations. As a result, the community will continue to be focused around one or two individuals,

rather than a highly-connected network in which others feel comfortable participating.

Another way to make participation rewarding is to provide training. Organizations require committees, meetings, budgets, grant proposals, and dispute resolution; most people are never taught how to do any of this, any more than they are taught how to teach, but training people to do these things helps your organization run more smoothly, and the opportunity to gain transferable skills is a powerful reason for people to get and stay involved. If you are going to do this, don't try to provide the training yourself (unless it's what you specialize in). Many civic and community groups have programs of this kind, and you can probably make a deal with one of them.

Other groups may be useful in other ways as well, and you may be useful to them—if not immediately, then tomorrow or next year. You should therefore set aside an hour or two every month to find allies and maintain your relationships with them. One way to do this is to ask them for advice: how do they think you ought to raise awareness of what you're doing? Where have they found space to run classes? What needs do they think aren't being met, and would you be able to meet them (either on your own, or in partnership with them)? Any group that has been around for a few years will have useful advice; they will also be flattered to be asked, and will know who you are the next time you call.

Government Matters

It's fashionable in tech circles to disparage government institutions as slow-moving dinosaurs, but in my experience they are no worse than companies of similar size. Your local school board, library, and your city councillor's office may be able to offer space, funding, publicity, connections with other groups that you may not have met yet, help with red tape, and a host of other useful things.

Soup, Then Hymns

Manifestos are fun to write, but most people join a volunteer community to help and be helped rather than to argue over the wording of a grand vision statement. (Most people who prefer the latter are only *interested in arguing. . .) To be effective you should therefore focus on things that are immediately useful, e.g., on what people can create that will be used by other community members right away. Once your organization shows that it can actually achieve small things, people will be more confident that it's worth investing in bigger ones. That's the time to worry about manifestos, since that's the point at which it's important to define values that will guide your growth and operations.*

One important special case of making things rewarding is to pay people. Volunteers can do a lot, but eventually tasks like system administration and accounting need full-time paid staff. When this time comes, you should

either pay people nothing or pay them a proper wage, but not do anything in between. If you pay them nothing, their actual reward for their work is the satisfaction of doing good. If you pay them a token amount, you take that away without giving them the satisfaction of earning a living.

Impostor Syndrome

Impostor syndrome thrives in communities with arbitrary, unnecessary standards, where harsh criticism is the norm, and where secrecy surrounds the actual process of getting work done, so the Ada Initiative has guidelines for communities to go with those given in Section 10.2 for individuals:

Encourage people. This is as simple as it is effective.

Discourage hostility and bickering. Public, hostile, personal arguments are a natural breeding ground for impostor syndrome.

Eliminate hidden barriers to participation. Be explicit about welcoming new students and colleagues, and thoroughly document how someone can participate in projects and events in your research group and at your institution.

As a leader, show your own uncertainties and demonstrate your own learning process. When people see leaders whom they respect struggling or admitting they didn't already know everything when they started, having realistic opinions of their own work becomes easier.

Reward and encourage people for mentoring newcomers. Officially enshrine mentoring as an important criterion in your career advancement process.

Don't make it personal when someone's work isn't up to snuff. When enforcing necessary quality standards, don't make the issue about the person. They aren't wrong or stupid or a waste of space; they've simply done one piece of work that didn't meet your expectations.

13.4 Governance

As [Free1972] pointed out, every organization has a power structure: the only question is whether it's formal and accountable, or informal and unaccountable. Make yours one of the first kind: write and publish the rules governing everything from who's allowed to use the name and logo to who gets to decide whether people are allowed to charge money to teach with whatever materials your group has worked up.

Organizations can govern themselves in many different ways, and a full discussion of the options is outside the scope of this book. For-profit corporations and incorporated non-profits are the two most popular models; the mechanics vary from jurisdiction to jurisdiction, so you should seek

advice locally before doing anything. (This is one of the times when having ties with local government or other like-minded organizations pays off.)

The model I prefer is that of a commons, which is "something managed jointly by a community according to rules they themselves have evolved and adopted". As [Boll2014] emphasizes, all three parts of that definition are essential: a commons isn't just a shared pasture, but also includes the community that shares it and the rules they use to do so.

Most resources, throughout most of human history, have been commons: it is only in the last few hundred years that impersonal markets have pushed them to the margins. In order to do so, free-market advocates have had to convince us we're something we're not (dispassionate calculators of individual advantage) and erase or devalue local knowledge and custom with tragic consequences for us individually and collectively.

Since society has difficulty recognizing commons organizations, and since most of the people you will want to recruit don't have experience with them, you will probably wind up having some sort of board, a director, and other staff. Broadly speaking, your organization can have either a *service board*, whose members also take on other roles in the organization, or a *governance board* whose primary responsibility is to hire, monitor, and if need be fire the director. Board members can be elected by the community or appointed; in either case, it's important to prioritize competence over passion (the latter being more important for the rank and file), and to try to recruit for particular skills such as accounting, marketing, and so on.

Don't worry about drafting a constitution when you first get started: it will only result in endless wrangling about what we're going to do rather than formalization of what you're already doing. When the time does come to formalize your rules, though, make your organization a democracy: sooner or later (usually sooner), every appointed board turns into a mutual agreement society and loses sight of what the community it's meant to serve actually needs. Giving the community power is messy, but is the only way invented so far to ensure that an organization continues to meet people's actual needs.

13.5 Final Thoughts

As [Pign2016] discusses, burnout is a chronic risk in any community activity. If you don't take care of yourself, you won't be able to take care of your community.

Every organization eventually needs fresh ideas and fresh leadership. When that time comes, train your successors and then move on. They will undoubtedly do things you wouldn't have, but the same is true of every generation. Few things in life are as satisfying as watching something you

helped build take on a life of its own. Celebrate that—you won't have any trouble finding something else to keep you busy.

13.6 Exercises

Several of these exercises are taken from [Brow2007], which is an exceptionally useful book on building community organizations.

What Kind of Community? (individual/15)

Re-read the discussion in the introduction of types of communities and decide which type or types your group is, or aspires to be.

People You May Meet (small groups/30)

As an organizer, part of your job is sometimes to help people find a way to contribute despite themselves. In small groups, pick three of the people below and discuss how you would help them become a better contributor to your organization.

Anna knows more about every subject than everyone else put together—at least, she thinks she does. No matter what you say, she'll correct you; no matter what you know, she knows better.

Catherine has so little confidence in her own ability that she won't make any decision, no matter how small, until she has checked with someone else.

Frank believes that knowledge is power, and enjoys knowing things that other people don't. He can make things work, but when asked how he did it, he'll grin and say, "Oh, I'm sure you can figure it out."

Hediyeh is quiet. She never speaks up in meetings, even when she knows that what other people are saying is wrong. She might contribute to the mailing list, but she's very sensitive to criticism, and will always back down rather than defending her point of view.

Kenny has discovered that most people would rather shoulder his share of the work than complain about him, and he takes advantage of it at every turn. The frustrating thing is that he's so damn *plausible* when someone finally does confront him. "There have been mistakes on all sides," he says, or, "Well, I think you're nit-picking."

Melissa means well, but somehow something always comes up, and her tasks are never finished until the last possible moment. Of course, that means that everyone who is depending on her can't do their work until *after* the last possible moment. . .

Raj is rude. "It's just the way I talk," he says, "If you can't hack it, maybe you should find another team." His favorite phrase is, "That's stupid," and he uses obscenity in every second sentence.

Values (small groups/45)

Answer the following questions on your own, and then compare your answers to those given by other members of your group.

1. What are the values your organization expresses?
2. Are these the values you want the organization to express?
3. If not, what values would you like it to express?
4. What are the specific behaviors that demonstrate those values?
5. What are some key behaviors that would demonstrate the values you would like for your group?
6. What are the behaviors that would demonstrate the opposite of those values?
7. What are some key behaviors that would demonstrate the opposite of the values you want to have?

Meeting Procedures (small groups/30)

Answer the following questions on your own, and then compare your answers to those given by other members of your group.

1. How are your meetings run?
2. Is this how you want your meetings to be run?
3. Are the rules for running meetings explicit or just assumed?
4. Are these the rules you want?
5. Who is eligible to vote/make decisions?
6. Is this who you want to be vested with decision-making authority?
7. Do you use majority rule, make decisions by consensus, or use some other method?
8. Is this the way you want to make decisions?
9. How do people in a meeting know when a decision has been made?
10. How do people who weren't at a meeting know what decisions were made?
11. Is this working for your group?

Size (small groups/20)

Answer the following questions on your own, and then compare your answers to those given by other members of your group.

1. How big is your group?
2. Is this the size you want for your organization?
3. If not, what size would you like it to be?
4. Do you have any limits on the size of membership?
5. Would you benefit from setting such a limit?

Staffing (small groups/30)

Answer the following questions on your own, and then compare your answers to those given by other members of your group.

1. Do you have paid staff in your organization?
2. Or is it all-volunteer?
3. Should you have paid staff?
4. Do you want/need more or less staff?
5. What do you call the staff (e.g., organizer, director, coordinator, etc.)?
6. What do the staff members do?
7. Are these the primary roles and functions that you want the staff to be filling?
8. Who supervises your staff?
9. Is this the supervision process and responsibility chain that you want for your group?
10. What is your staff paid?
11. Is this the right salary to get the needed work done and to fit within your resource constraints?
12. What benefits does your group provide to its staff (health, dental, pension, short and long-term disability, vacation, comp time, etc.)?
13. Are these the benefits that you want to give?

Money (small groups/30)

Answer the following questions on your own, and then compare your answers to those given by other members of your group.

1. Who pays for what?
2. Is this who you want to be paying?
3. Where do you get your money?
4. Is this how you want to get your money?
5. If not, do you have any plans to get it another way?
6. If so, what are they?
7. Who is following up to make sure that happens?
8. How much money do you have?
9. How much do you need?
10. What do you spend most of your money on?
11. Is this how you want to spend your money?

Becoming a Member (small groups/45)

Answer the following questions on your own, and then compare your answers to those given by other members of your group.

1. How does someone join?
2. Does this process work for your organization?
3. What are the membership criteria?
4. Are these the membership criteria you want?
5. Are people required to agree to any rules of behavior upon joining?
6. Are these the rules for behavior you want?
7. Are there membership dues?

Borrowing Ideas (whole class/15)

Many of our ideas about how to build a community have been shaped by our experience of working in open source software development. [Foge2005] (which is available online) is a good guide to what has and hasn't worked for those communities, and the Open Source Guides site has a wealth of useful information as well. Choose one section of the latter, such as "Finding Users for Your Project" or "Leadership and Governance", read it through, and give a two-minute presentation to the group of one idea from it that you found useful or that you strongly disagreed with.

Who Are You? (small groups/20)

The National Oceanic and Atmospheric Administration (NOAA) has published a short, amusing, and above all useful guide to dealing with disruptive behaviors. It categorizes those behaviors under labels like "talkative", "indecisive", and "shy", and outlines strategies for handling each. In groups of 3–6, read the guide and decide which of these descriptions best fits you. Do you think the strategies described for handling people like you are effective? Are other strategies equally or more effective?

14 Marketing

It's hard to get people with academic or technical backgrounds to take marketing seriously, not least because it's perceived as being about spin and misdirection. In reality, it is the craft of seeing things from other people's perspective, understanding their wants and needs, and finding ways to meet them. This should sound familiar: many of the techniques introduced in Chapter 6 do exactly this for lessons. This chapter will look at how to apply similar ideas to the larger problem of getting people to understand and support what you're doing.

14.1 What Are You Offering to Whom?

The first step is to figure out what you are offering to whom, i.e., what actually brings in the volunteers, funding, and other support you need to keep going. As [Kuch2011] points out, the answer is often counter-intuitive. For example, most scientists think their papers are their product, but it's actually their grant proposals, because those are what brings in money. Their papers are the advertising that persuades people to fund those proposals, just as albums are now what persuades people to buy musicians' concert tickets and t-shirts.

You may not be a scientist, so suppose instead that your group is offering weekend programming workshops to people who are re-entering the workforce after taking several years out to look after young children. If your learners are paying enough for your workshops to cover your costs, then the learners are your customers and the workshops are the product. If, on the other hand, the workshops are free, or the learners are only paying a token amount (to cut the no-show rate), then your actual product may be some mix of:

- your grant proposals,
- the alumni of your workshops that the companies sponsoring you would like to hire,
- the half page summary of your work in the mayor's annual report to city council that shows how she's supporting the local tech sector, or

- the personal satisfaction that your volunteer instructors get from teaching.

As with the lesson design process in Chapter 6, you should try to create personas to describe people who might be interested in what you're doing and figure out which of their needs your program will meet. You should also write a set of elevator pitches, each aimed at a different potential stakeholder. A widely-used template for these pitches looks like this:

1. For *target audience*
2. who *dissatisfaction with what's currently available*
3. our *category*
4. provide *key benefit*.
5. Unlike *alternatives*
6. our program *key distinguishing feature.*

Continuing with the weekend workshop example, we could use this pitch for participants:

> *For* people re-entering the workforce after taking time out to raise children *who* still have regular childcare responsibilities*, our* introductory programming workshops *provide* weekend classes with on-site childcare. *Unlike* online classes*, our program* gives participants a chance to meet people who are at the same stage of life.

but this one for companies that we want to donate staff time for teaching:

> *For* a company that wants to recruit entry-level software developers *that* is struggling to find mature, diverse candidates *our* introductory programming workshops *provide* a pool of potential recruits in their thirties that includes large numbers of people from underrepresented groups. *Unlike* college recruiting fairs*, our program* connects companies directly with a diverse audience.

If you don't know why different potential stakeholders might be interested in what you're doing, ask them. If you do know, ask them anyway: answers can change over time, and it's a good way to discover things that you might have missed.

Once you have written these pitches, you should use them to drive what you put on your organization's web site and in other publicity material, since it will help people figure out as quickly as possible whether you and they have something to talk about. (You probably *shouldn't* copy them verbatim, since many people in tech have seen this template so often that their eyes will glaze over if they encounter it again.)

As you are writing these pitches, remember that people are not just economic animals. A sense of accomplishment, control over their own lives, and being part of a community motivates them just as much as money.

People may volunteer to teach with you because their friends are doing it; similarly, a company may say that they're sponsoring classes for economically disadvantaged high school students because they want a larger pool of potential employees further down the road, but the CEO might actually be doing it simply because it's the right thing to do.

14.2 Branding and Positioning

A brand is someone's first reaction to a mention of a product; if the reaction is "what's that?", you don't have a brand yet. Branding is important because people aren't going to help with something they don't know about or don't care about.

Most discussion of branding today focuses on ways to build awareness online. Mailing lists, blogs, and Twitter all give you ways to reach people, but as the volume of (mis)information steadily increases, the attention people pay to each interruption decreases. As this happens, positioning becomes more important. Sometimes called "differentiation", it is what sets your offering apart from others, i.e., it's the "unlike" section of your elevator pitches. When you reach out to people who are already familiar with your field, you should emphasize your positioning, since it's what will catch their attention.

There are other things you can do to help build your brand. One is to use props: a robot car that one of your students made from scraps she found around the house, the website another student made for his parents' retirement home, or anything else that makes what you're doing seem real. Another is to make a short video—no more than a few minutes long—showcasing the backgrounds and accomplishments of your students. The aim of both is to tell a story: while people always ask for data, stories are what they believe.

Notice, though that these examples assume people have access to the money, materials, and/or technology needed to create these products. Many don't—in fact, those serving economically disadvantaged groups almost certainly don't. As Rosario Robinson says, "Free works for those that can afford free." In those situations, stories become even more important, because they can be shared and re-shared without limit.

Foundational Myths

One of the most compelling stories a person or organization can tell is why and how they got started. Are you teaching what you wish someone had taught you but didn't? Was there one particular person you wanted to help, and that opened the floodgates? If there isn't a section on your website starting, "Once upon a time," think about adding one.

Whatever else you do, make your organization findable in online searches: [DiSa2014b] discovered that the search terms parents were likely

to use for out-of-school computing classes didn't actually find those classes. There's a lot of folklore about how to make things findable under the label "SEO" (for "search engine optimization"); given Google's near-monopoly powers and lack of transparency, most of it boils down to trying to stay one step ahead of algorithms designed to prevent people from gaming rankings.

Unless you're very well funded, the best you can do is to search for yourself and your organization on a regular basis and see what comes up, then read these guidelines from Moz and do what you can to improve your site. Keep this cartoon in mind: people don't (initially) want to know about your org chart or get a virtual tour of your site; they want your address, parking information, and above all, some idea of what you teach, when you teach it, how to get in touch, and how it's going to change their life.

Offline findability is equally important for new organizations. Many of the people you hope to reach might not be online as often as you, and some won't be online at all. Notice boards in schools, local libraries, drop-in centers, and grocery stores are still an effective way to reach them.

Build Alliances

As discussed in Chapter 13, building alliances with other groups that are doing things related to what you're doing pays off in many ways. One of those is referrals: if someone approaches you for help, but would be better served by some other organization, take a moment to make an introduction. If you've done this several times, add something to your website to help the next person find what they need. The organizations you are helping will soon start to help you in return.

14.3 The Art of the Cold Call

Building a web site and hoping that people find it is one thing; calling people up or knocking on their door without any sort of prior introduction is another. As with standing up and teaching, though, it's a craft that can be learned like any other, and there are a few simple rules you can follow:

Establish a point of connection such as "I was speaking to X" or "You attended bootcamp Y". This must be specific: spammers and headhunters have trained us all to ignore anything that starts, "I recently read your website".

Create a slight sense of urgency by saying something like, "We're booking workshops right now." Be cautious with this, though; as with the previous recommendation, the web's race to the bottom has conditioned people to discount anything that sounds like a hustle.

Explain how you are going to help make their lives better. A pitch like "Your students will be able to do their math homework much faster if you let us tutor them" is a good attention-getter.

Be specific about what you are offering. "Our usual two-day curriculum includes. . . " helps listeners figure out right away whether a conversation is worth pursuing.

Make yourself credible by mentioning your backers, your size, how long you've been around, or your instructors's backgrounds.

Tell them what your terms are. Do you charge money? Do they need to cover instructors' travel costs? Can they reserve seats for their own staff?

Write a good subject line. Keep it short, avoid ALL CAPS, words like "sale" or "free" (which increase the odds that your message will be treated as spam), and never! use! exclamation! marks!

Keep it short, since the purest form of respect is to treat other people as if their time was as valuable as your own.

The email template below puts all of these points in action. It has worked pretty well: we found that about half of emails were answered, about half of those wanted to talk more, and about half of those led to workshops, which means that 10–15% of targeted emails turned into workshops. That's much better than the 2–3% response rate most organizations expect with cold calls, but can still be pretty demoralizing if you're not used to it.

Mail Out of the Blue

Hi NAME,

I hope you don't mind mail out of the blue, but I wanted to follow up on our conversation at the tech showcase last week to see if you would be interested having us run an instructor training workshop - we're scheduling the next batch over the next couple of weeks.

This one-day class will introduce your volunteer teachers to a handful of key practices that are grounded in education research and proven useful in practice. The class has been delivered dozens of times on four continents, and will be hands-on: short lessons will alternate with individual and group practical exercises, including practice teaching sessions.

If this sounds interesting, please give me a shout - I'd welcome a chance to talk ways and means.

Thanks,

NAME

14.4 A Final Thought

As [Kuch2011] says, if you can't be first in a category, create a new category that you can be first in; if you can't do that, join an existing group or think about doing something else entirely. This isn't defeatist: if someone else is already doing what you're doing better than you, there are probably lots of other equally useful things you could be doing instead.

14.5 Exercises

Write an Elevator Pitch for a City Councillor (individual/10)

This chapter described an organization that offers weekend programming workshops for people re-entering the workforce after taking a break to raise children. Write an elevator pitch for that organization aimed at a city councillor whose support the organization needs.

Write Elevator Pitches for Your Organization (individual/30)

Identify two groups of people your organization needs support from, and write an elevator pitch aimed at each one.

Email Subjects (pairs/10)

Write the subject lines (and only the subject lines) for three email messages: one announcing a new course, one announcing a new sponsor, and one announcing a change in project leadership. Compare your subject lines to a partner's and see if you can merge the best features of each while also shortening them.

Identify Causes of Passive Resistance (small groups/30)

People who don't want change will sometimes say so out loud, but will also often use various forms of passive resistance, such as just not getting around to it over and over again, or raising one possible problem after another to make the change seem riskier and more expensive than it's actually likely to be. Working in small groups, list three or four reasons why people might not want your teaching initiative to go ahead, and explain what you can do with the time and resources you have to counteract each.

Why Learn to Program? (individual/15)

Revisit the "Why Learn to Program?" exercise in Section 1.7. Where do your reasons for teaching and your learners' reasons for learning align? Where are they not aligned? How does that affect your marketing?

Appealing to Your Learners (think-pair-share/15)

Adult learners are different from children and teens: in general, they are better at managing their time, they're learning because they want to or need to, and they bring a lot of previous experience of learning into the room, so they tend to be better at knowing when they're struggling productively and when they're just struggling.

Working in pairs, write a one-paragraph pitch for a class on web design that touches on these points, and then compare your pair's pitch with those of other pairs.

Conversational Programmers (think-pair-share/15)

A conversational programmer is someone who needs to know enough about computing to have a meaningful conversation with a programmer, but isn't going to program themselves. [Wang2018] found that most learning resources don't address this group's needs. Working in pairs, write a pitch for a half-day workshop intended to help people that fit this description, and then share your pair's pitch with the rest of the class.

15 Partnerships

Section 13.1 said that the first step in building a community is to decide if you really need to, or whether you would be more effective joining an existing organization. Either way, the organization you're part of will eventually need to work with other, more established groups: schools, community programs, churches, the courts, and companies. This chapter presents a handful of strategies for figuring out how to do that, and when it's worthwhile.

Unlike most of the rest of this book, this chapter is drawn more from things I have seen than from things I have done. Most of my attempts to get large institutions to change have been unproductive (which is part of why I left a university position to re-start Software Carpentry in 2010). While contributions to any part of this book are welcome, I would be particularly grateful to hear what you have to say about the issues discussed below.

15.1 Working With Schools

Everyone is afraid of the unknown and of embarrassing themselves. As a result, most people would rather fail than change. For example, Lauren Herckis looked at why university faculty don't adopt better teaching methods. She found that the main reason is a fear of looking stupid in front of their students; secondary reasons were concern that the inevitable bumps in changing teaching methods would affect course evaluations, and a desire to continue emulating the lecturers who had inspired them. It's pointless to argue about whether these issues are "real" or not: faculty believe they are, so any plan to work with faculty needs to address them.

[Bark2015] did a two-part study of how computer science educators adopt new teaching practices as individuals, organizationally, and in society as a whole. They asked and answered three key questions:

1. *How do faculty hear about new teaching practices?* They intentionally seek them out because they're motivated to solve a problem (particularly student engagement), are made aware through deliberate

initiatives by their institutions, pick them up from colleagues, or get them from expected *and unexpected* interactions at conferences (either teaching-related or technical).

2. *Why do they try them out?* Sometimes because of institutional incentives (e.g., they innovate to improve their chances of promotion), but there is often tension at research institutions where rhetoric about the importance of teaching is largely disbelieved. Another important reason is their own cost/benefit analysis: will the innovation save them time? A third is that they are inspired by role models—again, this largely affects innovations aimed to improve engagement and motivation rather than learning outcomes—and a fourth is trusted sources, e.g., people they meet at conferences who are in the same situation that they are and reported successful adoption.

   But faculty had concerns, and those concerns were often not addressed by people advocating changes. The first was Glass's Law: any new tool or practice initially slows you down. Another is that the physical layout of classrooms makes many new practices hard: discussion groups just don't work in theater-style seating.

   But the most telling result was this: "Despite being researchers themselves, the CS faculty we spoke to for the most part did not believe that results from educational studies were credible reasons to try out teaching practices." This is consistent with other findings: even people whose entire careers are devoted to research will disregard education research.

3. *Why do they keep using them?* As [Bark2015] says, "Student feedback is critical," and is often the strongest reason to continue using a practice, even though we know that students' self-reports don't correlate strongly with learning outcomes. (Note that student attendance in lectures is seen as an indicator of engagement.) Another reason to retaining a practice is institutional requirements, although if this is the motivation, people will often drop the practice and regress to whatever they were doing before when the explicit incentive or monitoring is removed.

The good news is, you can tackle these problems systematically. [Baue2015] looked at adoption of new medical techniques within the US Veterans Administration. They found that evidence-based practices in medicine take an average of 17 years to be incorporated into routine general practice, and that only about half of such practices are ever widely adopted. This depressing finding and others like it spurred the growth of implementation science, which is the scientific study of ways to get people to actually adopt better evidence-based practices.

As Chapter 13 said, the starting point is to find out what the people you're trying to help believe they need. For example, [Yada2016] summarizes feedback from K-12 teachers on the preparation and support they want;

while it may not all be applicable to your setting, having a cup of tea with a few people and listening before you speak can make a world of difference.

Once you know what people need, the next step is to make changes incrementally, within institutions' own frameworks. [Nara2018] describes an intensive three-year bachelor's program based on tight-knit cohorts and administrative support that tripled graduation rates. Elsewhere, [Hu2017] describes impact of introducing a six-month certification program for existing high school teachers who want to teach computing (as opposed to the older two-year/five-course program). The number of computing teachers had been stable from 2007 to 2013, but quadrupled after introduction of the new certification program, without diluting quality: new-to-computing teachers seemed to be as effective as teachers with more computing training at teaching the introductory Exploring Computer Science course. The authors report, "How much CS content students self-reported learning in ECS appears to be based on how much they believed they knew before taking ECS, and appears to have no correlation to their teacher's CS background."

More broadly, [Borr2014] categorizes ways to make change happen in higher education. The categories are defined by whether the change is individual or to the system as a whole, and whether it is prescribed (top-down) or emergent (bottom-up). The person trying to make the changes—and make them stick—has a different role in each situation, and should pursue different strategies accordingly.

The paper goes on to explain each of the methods in detail, while [Hend2015a, Hend2015b] present the same ideas in more actionable form. Coming in from outside, you will probably fall into the Individual/Emergent category to start with, since you will be approaching teachers one by one and trying to make change happen bottom-up. If this is the case, the strategies Borrego and Henderson recommend center around having teachers reflect on their teaching individually or in groups. Since they may know more about teaching than you do, this often comes down to doing live coding sessions with them so that they know how to program themselves, and to demonstrate whatever curriculum you may already have.

15.2 Working Outside Schools

Schools and universities aren't the only places people go to learn programming; over the past few years, a growing number have turned to intensive bootcamp programs. These are typically one to six months long, run by private firms for profit, and target people who are retraining to get into tech. Some are very high quality, but others exist primarily to separate people (often from low-income backgrounds) from their money [McMi2017].

[Thay2017] interviewed 26 alumni of such bootcamps that provide a second chance for those who missed computing education opportunities

earlier (though the authors phrasing this as "missed earlier opportunities" makes some pretty big assumptions when it comes to people from underrepresented groups). Bootcamp students face great personal costs and risks: significant time, money, and effort spent before, during, and after bootcamps, and career change could take students a year or more. Several interviewees felt that their certificates were looked down on by employers; as some said, getting a job means passing an interview, but interviewers often won't share their reasons for rejection, so it's hard to know what to fix or what else to learn. Many resorted to internships (paid or otherwise) and spent a lot of time building their portfolios and networking. The three informal barriers they most clearly identified were knowledge (or rather, jargon), impostor syndrome, and a sense of not fitting in.

[Burk2018] dug into this a bit deeper by comparing the skills and credentials that tech industry recruiters are looking for to those provided by 4-year degrees and bootcamps. They interviewed 15 hiring managers from firms of various sizes and ran some focus groups, and found that recruiters uniformly emphasized soft skills (especially teamwork, communication, and the ability to continue learning). Many companies required a 4-year degree (though not necessarily in computer science), but many also praised bootcamp graduates for being older or more mature and having more up-to-date knowledge.

If you are approaching one of these groups, your best strategy could well be to emphasize what you know about teaching rather than what you know about tech, since many of their founders and staff have programming backgrounds but little or no training in education. The first few chapters of this book have played well with this audience in the past, and [Lang2016] describes evidence-based teaching practices that can be put in place with minimal effort and at low cost. These may not have the most impact, but scoring a few early wins helps build support for larger and riskier efforts.

15.3 Final Thoughts

It is impossible to change large institutions on your own: you need allies, and to get allies, you need tactics. The most useful guide I have found is [Mann2015], which catalogs more than four dozen methods you can use, and organizes them according to whether they're best deployed early on, later, throughout the change cycle, or when you encounter resistance. A handful of their patterns include:

Small Successes: To avoid becoming overwhelmed by the exercises and all the things you have to do when you're involved in an organizational change effort, celebrate even small successes.

In Your Space: Keep the new idea visible by placing reminders throughout the organization.

Token: To keep a new idea alive in a person's memory, hand out tokens that can be identified with the topic being introduced.

Champion Skeptic: Ask strong opinion leaders who are skeptical of your new idea to play the role of "official skeptic". Use their comments to improve your effort, even if you don't change their minds.

Conversely, [Farm2006] has ten tongue-in-cheek rules for ensuring that a new tool isn't adopted, all of which apply to new teaching practices as well:

1. Make it optional.
2. Economize on training.
3. Don't use it in a real project.
4. Never integrate it.
5. Use it sporadically.
6. Make it part of a quality initiative.
7. Marginalize the champion.
8. Capitalize on early missteps.
9. Make a small investment.
10. Exploit fear, uncertainty, doubt, laziness, and inertia.

The most important strategy is to be willing to change your goals based on what you learn from the people you are trying to help. It could well be that tutorials showing them how to use a spreadsheet will help them more quickly and more reliably than an introduction to JavaScript. I have often made the mistake of confusing things I was passionate about with things that other people ought to know; if you truly want to be a partner, always remember that learning and change have to go both ways.

15.4 Exercises

Collaborations (small groups/30)

Answer the following questions on your own, and then compare your answers to those given by other members of your group.

1. Do you have any agreements or relationships with other groups?
2. Do you want to have relationships with any other groups?
3. How would having (or not having) collaborations help you to achieve your goals?
4. What are your key collaborative relationships?
5. Are these the right collaborators for achieving your goals?
6. With what groups or entities would you like your organization to have agreements or relationships?

Educationalization (whole class/10)

[Laba2008] explores why the United States and other countries keep pushing the solution of social problems onto educational institutions, and why that continues not to work. As he points out, "[Education] has done very little to promote equality of race, class, and gender; to enhance public health, economic productivity, and good citizenship; or to reduce teenage sex, traffic deaths, obesity, and environmental destruction. In fact, in many ways it has had a negative effect on these problems by draining money and energy away from social reforms that might have had a more substantial impact." He goes on to write:

> *So how are we to understand the success of this institution in light of its failure to do what we asked of it? One way of thinking about this is that education may not be doing what we ask, but it is doing what we want. We want an institution that will pursue our social goals in a way that is in line with the individualism at the heart of the liberal ideal, aiming to solve social problems by seeking to change the hearts, minds, and capacities of individual students. Another way of putting this is that we want an institution through which we can express our social goals without violating the principle of individual choice that lies at the center of the social structure, even if this comes at the cost of failing to achieve these goals. So education can serve as a point of civic pride, a showplace for our ideals, and a medium for engaging in uplifting but ultimately inconsequential disputes about alternative visions of the good life. At the same time, it can also serve as a convenient whipping boy that we can blame for its failure to achieve our highest aspirations for ourselves as a society.*

How do efforts to teach computational thinking and digital citizenship in schools fit into this framework?

Institutional Adoption (whole class/15)

Re-read the list of motivations to adopt new practices given in Section 15.1. Which of these apply to you and your colleagues? Which are irrelevant to your context? Which do you emphasize if and when you interact with people working in formal educational institutions?

Making It Fail (small groups/15)

Working in small groups, re-read the list of ways to ensure new tools aren't adopted given in Section 15.3. Which of these have you seen done recently? Which have you done yourself? What form did they take?

Mentoring (whole class/15)

The Institute for African-American Mentoring in Computer Science has published a brief set of guidelines for mentoring doctoral students, which you can download from http://iaamcs.org/guidelines. Take a few minutes to read the guidelines individually, and then go through them as a class and rate your efforts for your own group as +1 (definitely doing), -1 (definitely not doing), and 0 (not sure or not applicable).

16 Why I Teach

When I first started teaching at the University of Toronto, some of my students asked me why I was doing it. 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 whether you want to 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 is,* we *are the grownups.*
> *Twenty years from now, though, we'll be heading for retirement and* you *will be in charge. That may sound like a long time when you're nineteen, but 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 learn how to do these things now, you won't be ready to do them when you have to.*

It was all true, but it wasn't the whole story. I don't want people to make the world a better place so that I can retire in comfort. I want them to do it because it's the greatest adventure of our time. A hundred and fifty years ago, most societies practiced slavery. A hundred years ago, my grandmother 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 were still ordering electroshock therapy to "cure" homosexuals. 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.

This didn't happen by chance. It happened because millions of people 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.

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

> *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 do what I do. 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. So:

Start where you are.
Use what you have.
Help who you can.

Thank you for reading. I hope we can learn something together some day.

Bibliography

[Abba2012] Janet Abbate. *Recoding Gender: Women's Changing Participation in Computing*. MIT Press, 2012. Describes the careers and accomplishments of the women who shaped the early history of computing, but have all too often been written out of that history.

[Abel1996] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996. One of the most widely cited introductions to programming ever written.

[Abel2009] Andrew Abela. Chart suggestions - a thought starter. http://extremepresentation.typepad.com/files/choosing-a-good-chart-09.pdf, 2009. A graphical decision tree for choosing the right type of chart.

[Adam1975] Frank Adams and Myles Horton. *Unearthing Seeds of Fire: The Idea of Highlander*. Blair, 1975. A history of the Highlander Folk School and its founder, Myles Horton.

[Aike1975] 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):439–444, 1975. An early landmark study showing that taking notes improved retention.

[Aiva2016] Efthimia Aivaloglou and Felienne Hermans. How kids code and how we know. In *Proc. 2016 International Computing Education Research Conference (ICER'16)*. Association for Computing Machinery (ACM), 2016. Presents an analysis of 250,000 Scratch projects.

[Alha2018] Sohail Alhazmi, Margaret Hamilton, and Charles Thevathayan. CS for All: Catering to diversity of master's students through assignment choices. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Reports

improvement in learning outcomes and student satisfaction in a course for students from a variety of academic backgrounds which allowed them to choose -related assignments.

[Alin1989] Saul D. Alinsky. *Rules for Radicals: A Practical Primer for Realistic Radicals*. Vintage, 1989. A widely-read guide to community organization written by one of the 20th Century's great organizers.

[Alqa2017] Basma S. Alqadi and Jonathan I. Maletic. An empirical study of debugging patterns among novice programmers. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports patterns in the debugging activities and success rates of novice programmers.

[Alvi1999] Jennifer Alvidrez and Rhona S. Weinstein. Early teacher perceptions and later student academic achievement. *Journal of Educational Psychology*, 91(4):731–746, 1999. An influential study of the effects of teachers' perceptions of students on their later achievements.

[Ambr2010] 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. Summarizes what we know about education and why we believe it's true, from cognitive psychology to social factors.

[Ande2001] Lorin W. Anderson and David R. Krathwohl, editors. *A Taxonomy for Learning, Teaching, And Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, 2001. A widely-used revision to Bloom's Taxonomy.

[Armo2008] Michal Armoni and David Ginat. Reversing: A fundamental idea in computer science. *Computer Science Education*, 18(3):213–230, Sep 2008. Argues that the notion of reversing things is an unrecognized fundamental concept in computing education.

[Atki2000] Robert K. Atkinson, Sharon J. Derry, Alexander Renkl, and Donald Wortham. Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research*, 70(2):181–214, Jun 2000. A comprehensive survey of worked examples research at the time.

[Avel2013] 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), Aug 2013. Reports the

effectiveness of surgical checklist implementations in the UK and Africa.

[Bacc2013] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. 2013 International Conference on Software Engineering (ICSE'13)*, May 2013. A summary of work on code review.

[Bari2017] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. Do developers read compiler error messages? In *Proc. 2017 International Conference on Software Engineering (ICSE'17)*. Institute of Electrical and Electronics Engineers (IEEE), May 2017. Reports that developers do read error messages and doing so is as hard as reading source code: it takes 13-25% of total task time.

[Bark2014] Lecia Barker, Christopher Lynnly Hovey, and Leisa D. Thompson. Results of a large-scale, multi-institutional study of undergraduate retention in computing. In *Proc. 2014 Frontiers in Education Conference (FIE'14)*. Institute of Electrical and Electronics Engineers (IEEE), Oct 2014. Reports that meaningful assignments, faculty interaction with students, student collaboration on assignments, and (for male students) pace and workload relative to expectations drive retention in computing classes, while interactions with teaching assistants or with peers in extracurricular activities have little impact.

[Bark2015] Lecia Barker, Christopher Lynnly Hovey, and Jane Gruning. What influences CS faculty to adopt teaching practices? In *Proc. 2015 Technical Symposium on Computer Science Education (SIGCSE'15)*. Association for Computing Machinery (ACM), 2015. Describes how computer science educators adopt new teaching practices.

[Basi1987] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, Dec 1987. An early and influential summary of the effectiveness of code review.

[Basu2015] Soumya Basu, Albert Wu, Brian Hou, and John DeNero. Problems before solutions: Automated problem clarification at scale. In *Proc. 2015 Conference on Learning @ Scale (L@S'15)*. Association for Computing Machinery (ACM), 2015. Describes a system in which students have to unlock

test cases for their code by answering MCQs, and presents data showing that this is effective.

[Batt2018] Lina Battestilli, Apeksha Awasthi, and Yingjun Cao. Two-stage programming projects: Individual work followed by peer collaboration. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Reports that learning outcomes were improved by two-stage projects in which students work individually, then re-work the same problem in pairs.

[Baue2015] Mark S. Bauer, Laura Damschroder, Hildi Hagedorn, Jeffrey Smith, and Amy M. Kilbourne. An introduction to implementation science for the non-specialist. *BMC Psychology*, 3(1), Sep 2015. Explains what implementation science is, using examples from the US Veterans Administration to illustrate.

[Beck2013] Leland Beck and Alexander Chizhik. Cooperative learning instructional methods for CS1: Design, implementation, and evaluation. *ACM Transactions on Computing Education*, 13(3):10:1–10:21, Aug 2013. Reports that cooperative learning enhances learning outcomes and self-efficacy in CS1.

[Beck2014] Victoria Beck. Testing a model to predict online cheating—much ado about nothing. *Active Learning in Higher Education*, 15(1):65–75, Jan 2014. Reports that cheating is no more likely in online courses than in face-to-face courses.

[Beck2016] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. Effective compiler error message enhancement for novice programming students. *Computer Science Education*, 26(2-3):148–175, Jul 2016. Reports that improved error messages helped novices learn faster.

[Beni2017] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. Meaningful identifier names: The case of single-letter variables. In *Proc. 2017 International Conference on Program Comprehension (ICPC'17)*. Institute of Electrical and Electronics Engineers (IEEE), May 2017. Reports that use of single-letter variable names doesn't affect ability to modify code, and that some single-letter variable names have implicit types and meanings.

[Benn2000] Patricia Benner. *From Novice to Expert: Excellence and Power in Clinical Nursing Practice*. Pearson, 2000. A classic study of clinical judgment and the development of expertise.

[Benn2007a] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32, Jun 2007. Reports that 67% of students pass CS1, with variation from 5% to 100%.

[Benn2007b] Jens Bennedsen and Carsten Schulte. What does "objects-first" mean?: An international study of teachers' perceptions of objects-first. In *Proc. 2007 Koli Calling Conference on Computing Education Research (Koli'07)*, pages 21–29, 2007. Teases out three meanings of "objects first" in computing education.

[Berg2012] Joseph Bergin, Jane Chandler, Jutta Eckstein, Helen Sharp, Mary Lynn Manns, Klaus Marquardt, Marianna Sipos, Markus Völter, and Eugene Wallingford. *Pedagogical Patterns: Advice for Educators*. CreateSpace, 2012. A catalog of design patterns for teaching.

[Biel1995] Katerine Bielaczyc, Peter L. Pirolli, and Ann L. Brown. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and Instruction*, 13(2):221–252, Jun 1995. Reports that training learners in self-explanation accelerates their learning.

[Bigg2011] John Biggs and Catherine Tang. *Teaching for Quality Learning at University*. Open University Press, 2011. A step-by-step guide to lesson development, delivery, and evaluation for people working in higher education.

[Bink2012] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, May 2012. Reports that reading and understanding code is fundamentally different from reading prose, and that experienced developers are relatively unaffected by identifier style, but beginners benefit from the use of camel case (versus pothole case).

[Blik2014] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, 23(4):561–599, Oct 2014. Reports an attempt to categorize novice programmer behavior using machine learning that found interesting patterns on individual assignments.

[Bloo1984] Benjamin S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6):4–16, Jun 1984. Reports that students tutored one-to-one using mastery learning techniques perform two standard deviations better than those who learned through conventional lecture.

[Boha2011] Mark Bohay, Daniel P. Blakely, Andrea K. Tamplin, and Gabriel A. Radvansky. Note taking, review, memory, and comprehension. *American Journal of Psychology*, 124(1):63, 2011. Reports that note-taking improves retention most at deeper levels of understanding.

[Boll2014] David Bollier. *Think Like a Commoner: A Short Introduction to the Life of the Commons*. New Society Publishers, 2014. A short introduction to a widely-used model of governance.

[Borr2014] Maura Borrego and Charles Henderson. Increasing the use of evidence-based teaching in STEM higher education: A comparison of eight change strategies. *Journal of Engineering Education*, 103(2):220–252, Apr 2014. Categorizes different approaches to effecting change in higher education.

[Bria2015] Samuel A. Brian, Richard N. Thomas, James M. Hogan, and Colin Fidge. Planting bugs: A system for testing students' unit tests. In *Proc. 2015 Conference on Innovation and Technology in Computer Science Education (ITiCSE'15)*. Association for Computing Machinery (ACM), 2015. Describes a tool for assessing students' programs and unit tests and finds that students often write weak tests and misunderstand the role of unit testing.

[Broo2016] Stephen D. Brookfield and Stephen Preskill. *The Discussion Book: 50 Great Ways to Get People Talking*. Jossey-Bass, 2016. Describes fifty different ways to get groups talking productively.

[Brop1983] Jere E. Brophy. Research on the self-fulfilling prophecy and teacher expectations. *Journal of Educational Psychology*, 75(5):631–661, 1983. A early, influential study of the effects of teachers' perceptions on students' achievements.

[Brow2007] Michael Jacoby Brown. *Building Powerful Community Organizations: A Personal Guide to Creating Groups that Can Solve Problems and Change the World*. Long Haul Press, 2007. A practical guide to creating effective organizations in and for communities.

[Brow2017] Neil C. C. Brown and Amjad Altadmri. Novice java programming mistakes. *ACM Transactions on Computing Education*,

17(2), May 2017. Summarizes the authors' analysis of novice programming mistakes.

[Brow2018] Neil C. C. Brown and Greg Wilson. Ten quick tips for teaching programming. *PLoS Computational Biology*, 14(4), April 2018. A short summary of what we actually know about teaching programming and why we believe it's true.

[Buff2015] Kevin Buffardi and Stephen H. Edwards. Reconsidering automated feedback: A test-driven approach. In *Proc. 2015 Technical Symposium on Computer Science Education (SIGCSE'15)*. Association for Computing Machinery (ACM), 2015. Describes a system that associates failed tests with particular features in a learner's code so that learners cannot game the system.

[Burg2015] Sheryl E. Burgstahler. *Universal Design in Higher Education: From Principles to Practice*. Harvard Education Press, second edition, 2015. Describes how to make online teaching materials accessible to everyone.

[Burk2018] Quinn Burke, Cinamon Bailey, Louise Ann Lyon, and Emily Greeen. Understanding the software development industry's perspective on coding boot camps versus traditional 4-year colleges. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Compares the skills and credentials that tech industry recruiters are looking for to those provided by 4-year degrees and bootcamps.

[Butl2017] Zack Butler, Ivona Bezakova, and Kimberly Fluet. Pencil puzzles for introductory computer science. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Describes pencil-and-paper puzzles that can be turned into CS1/CS2 assignments, and reports that they are enjoyed by students and encourage meta-cognition.

[Byck2005] Pauli Byckling, Petri Gerdt, and Jorma Sajaniemi. Roles of variables in object-oriented programming. In *Proc. 2005 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. Association for Computing Machinery (ACM), 2005. Presents single-variable design patterns common in novice programs.

[Camp2016] Jennifer Campbell, Diane Horton, and Michelle Craig. Factors for success in online CS1. In *Proc. 2016 Conference on Innovation and Technology in Computer Science Education (ITiCSE'16)*. Association for Computing Machinery (ACM),

2016. Compares students who opted in to an online CS1 class online with those who took it in person in a flipped classroom.

[Cao2017a] Yingjun Cao and Leo Porter. Evaluating student learning from collaborative group tests in introductory computing. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports significant short-term gains but no long-term gains for students doing exams collaboratively.

[Cao2017b] Yingjun Cao and Leo Porter. Impact of performance level and group composition on student learning during collaborative exams. In *Proc. 2017 Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that collaborative exams benefit middling students more than high or low-performing students, and that homogeneous groups benefit more than heterogeneous ones.

[Carr1987] John Carroll, Penny Smith-Kerker, James Ford, and Sandra Mazur-Rimetz. The minimal manual. *Human-Computer Interaction*, 3(2):123–153, Jun 1987. The foundational paper on minimalist instruction.

[Carr2014] John Carroll. Creating minimalist instruction. *International Journal of Designs for Learning*, 5(2), Nov 2014. A look back on the author's work on minimalist instruction.

[Cart2017] Adam Scott Carter and Christopher David Hundhausen. Using programming process data to detect differences in students' patterns of programming. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Shows that students of different levels approach programming tasks differently, and that these differences can be detected automatically.

[Casp2007] Michael E. Caspersen and Jens Bennedsen. Instructional design of a programming course. In *Proc. 2007 International Computing Education Research Conference (ICER'07)*. Association for Computing Machinery (ACM), 2007. Goes from a model of human cognition to three learning theories, and from there to the design of an introductory object-oriented programming course.

[Cele2018] Mehmet Celepkolu and Kristy Elizabeth Boyer. Thematic analysis of students' reflections on pair programming in CS1.

In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Reports that pair programming has the same learning gains side-by-side programming but higher student satisfaction.

[Ceti2016] Ibrahim Cetin and Christine Andrews-Larson. Learning sorting algorithms through visualization construction. *Computer Science Education*, 26(1):27–43, Jan 2016. Reports that people learn more from constructing algorithm visualizations than they do from viewing visualizations constructed by others.

[Chen2009] Nicholas Chen and Maurice Rabb. A pattern language for screencasting. In *Proc. 2009 Conference on Pattern Languages of Programs (PLoP'09)*. Association for Computing Machinery (ACM), 2009. A brief, well-organized collection of tips for making screencasts.

[Chen2017] Nick Cheng and Brian Harrington. The code mangler: Evaluating coding ability without writing any code. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that student performance on exercises in which they undo code mangling correlates strongly with performance on traditional assessments.

[Cher2007] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard: How and why software developers use drawings. In *Proc. 2007 Conference on Human Factors in Computing Systems (CHI'07)*. Association for Computing Machinery (ACM), 2007. Reports that developers draw diagrams to aid discussion rather than to document designs.

[Cher2009] Sapna Cheryan, Victoria C. Plaut, Paul G. Davies, and Claude M. Steele. Ambient belonging: How stereotypical cues impact gender participation in computer science. *Journal of Personality and Social Psychology*, 97(6):1045–1060, 2009. Reports that subtle environmental clues have a measurable impact on the interest that people of different genders have in computing.

[Chet2014] Raj Chetty, John N. Friedman, and Jonah E. Rockoff. Measuring the impacts of teachers II: Teacher value-added and student outcomes in adulthood. *American Economic Review*, 104(9):2633–2679, Sep 2014. Reports that good teachers have a small but measurable impact on student outcomes.

[Chi1989] Michelene T. H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145–182, Apr 1989. A seminal paper on the power of self-explanation.

[Coco2018] Center for Community Organizations. The "problem" woman of colour in the workplace. https://coconet.org/problem-woman-colour-nonprofit-organizations/, 2018. Outlines the experience of many women of color in the workplace.

[Coll1991] Allan Collins, John Seely Brown, and Ann Holum. Cognitive apprenticeship: Making thinking visible. *American Educator*, 6:38–46, 1991. Describes an educational model based on the notion of apprenticeship and master guidance.

[Coom2012] Norman Coombs. *Making Online Teaching Accessible*. Jossey-Bass, 2012. An accessible guide to making online lessons accessible.

[Covi2017] Martin V. Covington, Linda M. von Hoene, and Dominic J. Voge. *Life Beyond Grades: Designing College Courses to Promote Intrinsic Motivation*. Cambridge University Press, 2017. Explores ways of balancing intrinsic and extrinsic motivation in institutional education.

[Craw2010] Matthew B. Crawford. *Shop Class as Soulcraft: An Inquiry into the Value of Work*. Penguin, 2010. A deep analysis of what we learn about ourselves by doing certain kinds of work.

[Crou2001] Catherine H. Crouch and Eric Mazur. Peer instruction: Ten years of experience and results. *American Journal of Physics*, 69(9):970–977, Sep 2001. Reports results from the first ten years of peer instruction in undergraduate physics classes, and describes ways in which its implementation changed during that time.

[Csik2008] Mihaly Csikszentmihaly. *Flow: The Psychology of Optimal Experience*. Harper, 2008. An influential discussion of what it means to be fully immersed in a task.

[Cumm2011] Stephen Cummins, Liz Burd, and Andrew Hatch. Investigating shareable feedback tags for programming assignments. *Computer Science Education*, 21(1):81–103, Mar 2011. Describes the use of tagging for peer feedback in introductory programming courses.

[Cunn2017] Kathryn Cunningham, Sarah Blanchard, Barbara J. Ericson, and Mark Guzdial. Using tracing and sketching to solve programming problems. In *Proc. 2017 Conference on International Computing Education Research (ICER'17)*. Association for Computing Machinery (ACM), 2017. Found that writing new values near variables' names as they change is the most effective tracing technique.

[Cutt2017] Quintin Cutts, Charles Riedesel, Elizabeth Patitsas, Elizabeth Cole, Peter Donaldson, Bedour Alshaigy, Mirela Gutica, Arto Hellas, Edurne Larraza-Mendiluze, and Robert McCartney. Early developmental activities and computing proficiency. In *Proc. 2017 Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)*. Association for Computing Machinery (ACM), 2017. Surveyed adult computer users about childhood activities and found strong correlation between confidence and computer use based on reading on one's own and playing with construction toys with no moving parts (like Lego).

[Dahl2018] Sarah Dahlby Albright, Titus H. Klinge, and Samuel A. Rebelsky. A functional approach to data science in CS1. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Describes the design of a CS1 class built around data science.

[DeBr2015] Pedro De Bruyckere, Paul A. Kirschner, and Casper D. Hulshof. *Urban Myths about Learning and Education*. Academic Press, 2015. Describes and debunks some widely-held myths about how people learn.

[Deb2018] Debzani Deb, Muztaba Fuad, James Etim, and Clay Gloster. MRS: Automated assessment of interactive classroom exercises. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Reports that doing in-class exercises with realtime feedback using mobile devices improved concept retention and student engagement while reducing failure rates.

[Derb2006] Esther Derby and Diana Larsen. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, 2006. Describes how to run a good project retrospective.

[DiSa2014a] Betsy DiSalvo, Mark Guzdial, Amy Bruckman, and Tom McKlin. Saving face while geeking out: Video game testing as a justification for learning computer science. *Journal of*

the Learning Sciences, 23(3):272–315, Jul 2014. Found that 65% of male African-American participants in a game testing program went on to study computing.

[DiSa2014b] Betsy DiSalvo, Cecili Reid, and Parisa Khanipour Roshan. They can't find us. In *Proc. 2014 Technical Symposium on Computer Science Education (SIGCSE'14)*. Association for Computing Machinery (ACM), 2014. Reports that the search terms parents were likely to use for out-of-school CS classes didn't actually find those classes.

[Dida2016] David Didau and Nick Rose. *What Every Teacher Needs to Know About Psychology*. John Catt Educational, 2016. An informative, opinionated explanation of what modern psychology has to say about teaching.

[Douc2005] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming. *Journal on Educational Resources in Computing*, 5(3):4–es, Sep 2005. Reviews the state of auto-graders at the time.

[DuBo1986] Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, Feb 1986. Introduces the idea of a notional machine.

[Edwa2014a] Stephen H. Edwards, Zalia Shams, and Craig Estep. Adaptively identifying non-terminating code when testing student programs. In *Proc. 2014 Technical Symposium on Computer Science Education (SIGCSE'14)*. Association for Computing Machinery (ACM), 2014. Describes an adaptive scheme for detecting non-terminating student coding submissions.

[Edwa2014b] Stephen H. Edwards and Zalia Shams. Do student programmers all tend to write the same software tests? In *Proc. 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. Association for Computing Machinery (ACM), 2014. Reports that students wrote tests for the happy path rather than to detect hidden bugs.

[Endr2014] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do API documentation and static typing affect API usability? In *Proc. 2014 International Conference on Software Engineering (ICSE'14)*. ACM Press, 2014. Shows that types do add complexity to programs, but it pays off fairly quickly by acting as documentation hints for a method's use.

[Ensm2003] Nathan L. Ensmenger. Letting the "computer boys" take over: Technology and the politics of organizational transformation.

International Review of Social History, 48(S11):153–180, Dec 2003. Describes how programming was turned from a female into a male profession in the 1960s.

[Ensm2012] Nathan L. Ensmenger. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. MIT Press, 2012. Traces the emergence and rise of computer experts in the 20th Century, and particularly the way that computing became male-gendered.

[Eppl2006] Martin J. Eppler. A comparison between concept maps, mind maps, conceptual diagrams, and visual metaphors as complementary tools for knowledge construction and sharing. *Information Visualization*, 5(3):202–210, Jun 2006. Compares concept maps, mind maps, conceptual diagrams, and visual metaphors as learning tools.

[Epst2002] Lewis Carroll Epstein. *Thinking Physics: Understandable Practical Reality*. Insight Press, 2002. An entertaining problem-based introduction to thinking like a physicist.

[Eric2015] Barbara J. Ericson, Steven Moore, Briana B. Morrison, and Mark Guzdial. Usability and usage of interactive features in an online ebook for CS teachers. In *Proc. 2015 Workshop in Primary and Secondary Computing Education (WiPSCE'15)*, pages 111–120. Association for Computing Machinery (ACM), 2015. Reports that learners are more likely to attempt Parsons Problems than nearby multiple choice questions in an ebook.

[Eric2016] K. Anders Ericsson. Summing up hours of any type of practice versus identifying optimal practice activities. *Perspectives on Psychological Science*, 11(3):351–354, May 2016. A critique of a meta-study of deliberate practice based on the latter's overly-broad inclusion of activities.

[Eric2017] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. Solving Parsons Problems versus fixing and writing code. In *Proc. 2017 Koli Calling Conference on Computing Education Research (Koli'17)*. Association for Computing Machinery (ACM), 2017. Reports that solving 2D Parsons problems with distractors takes less time than writing or fixing code but has equivalent learning outcomes.

[Farm2006] Eugene Farmer. The gatekeeper's guide, or how to kill a tool. *IEEE Software*, 23(6):12–13, Nov 2006. Ten tongue-in-cheek rules for making sure that a new software tool doesn't get adopted.

[Fehi2008] Chris Fehily. *SQL: Visual QuickStart Guide*. Peachpit Press, third edition, 2008. An introduction to SQL that is both a good tutorial and a good reference guide.

[Fell2001] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001. An introduction to computing that focuses on the program design process.

[Finc2007] Sally Fincher and Josh Tenenberg. Warren's question. In *Proc. 2007 International Computing Education Research Conference (ICER'07)*. Association for Computing Machinery (ACM), 2007. A detailed look at a particular instance of transferring a teaching practice.

[Finc2012] Sally Fincher, Brad Richards, Janet Finlay, Helen Sharp, and Isobel Falconer. Stories of change: How educators change their practice. In *Proc. 2012 Frontiers in Education Conference (FIE'12)*. Institute of Electrical and Electronics Engineers (IEEE), Oct 2012. A detailed look at how educators actually adopt new teaching practices.

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

[Fisl2014] Kathi Fisler. The recurring rainfall problem. In *Proc. 2014 International Computing Education Research Conference (ICER'14)*. Association for Computing Machinery (ACM), 2014. Reports that students made fewer low-level errors when solving the Rainfall Problem in a functional language.

[Fitz2008] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, Jun 2008. Reports that good undergraduate debuggers are good programmers but not necessarily vice versa, and that novices use tracing and testing rather than causal reasoning.

[Foge2005] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, 2005. The definite guide to managing open source software development projects.

[Foor1998] Barbara R. Foorman, David J. Francis, Jack M. Fletcher, Christopher Schatschneider, and Paras Mehta. The role of

instruction in learning to read: Preventing reading failure in at-risk children. *Journal of Educational Psychology*, 90(1):37–55, 1998. Reports that children learn to read faster when taught with phonics rather than other approaches.

[Ford2016] Denae Ford, Justin Smith, Philip J. Guo, and Chris Parnin. Paradise unplugged: Identifying barriers for female participation on Stack Overflow. In *Proc. 2016 International Symposium on Foundations of Software Engineering (FSE'16)*. Association for Computing Machinery (ACM), 2016. Reports that lack of awareness of site features, feeling unqualified to answer questions, intimidating community size, discomfort interacting with or relying on strangers, and perception that they shouldn't be slacking were seen as significantly more problematic by female Stack Overflow contributors rather than male ones.

[Fran2018] Pablo Frank-Bolton and Rahul Simha. Docendo discimus: Students learn by teaching peers through video. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Reports that students who make short videos to teach concepts to their peers have a significant increase in their own learning compared to those who only study the material or view videos.

[Free1972] Jo Freeman. The tyranny of structurelessness. *The Second Wave*, 2(1), 1972. Points out that every organization has a power structure: the only question is whether it's accountable or not.

[Frie1995] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer*. MIT Press, fourth edition, 1995. An introduction to programming using Scheme.

[Frie2016] Marilyn Friend and Lynne Cook. *Interactions: Collaboration Skills for School Professionals*. Pearson, eighth edition, 2016. A textbook on how teachers can work with other teachers.

[Galp2002] Vashti Galpin. Women in computing around the world. *ACM SIGCSE Bulletin*, 34(2), Jun 2002. Looks at female participation in computing in 35 countries.

[Gao2017] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in JavaScript. In *Proc. 2017 International Conference on Software Engineering (ICSE'17)*. Institute of Electrical and Electronics Engineers (IEEE), May 2017. Reports that static typing would catch about 15% of errors in JavaScript packages.

[Gauc2011] Danielle Gaucher, Justin Friesen, and Aaron C. Kay. Evidence that gendered wording in job advertisements exists and sustains gender inequality. *Journal of Personality and Social Psychology*, 101(1):109–128, 2011. Reports that gendered wording in job recruitment materials can maintain gender inequality in traditionally male-dominated occupations.

[Gawa2007] Atul Gawande. The checklist. *The New Yorker*, Dec 2007. Describes the life-saving effects of simple checklists.

[Gawa2011] Atul Gawande. Personal best. *The New Yorker*, Oct 2011. Describes how having a coach can improve practice in a variety of fields.

[Gelm2002] Andrew Gelman and Deborah Nolan. *Teaching Statistics: A Bag of Tricks*. Oxford University Press, 2002. A collection of tips and examples for teaching statistics.

[Gick1987] Mary L. Gick and Keith J. Holyoak. The cognitive basis of knowledge transfer. In S. J. Cormier and J. D. Hagman, editors, *Transfer of Learning: Contemporary Research and Applications*, pages 9–46. Elsevier, 1987. Finds that transference only comes with mastery.

[Gorm2014] Cara Gormally, Mara Evans, and Peggy Brickman. Feedback about teaching in higher ed: Neglected opportunities to promote change. *Cell Biology Education*, 13(2):187–199, Jun 2014. Summarizes best practices for providing instructional feedback, and recommends some specific strategies.

[Gree2014] Elizabeth Green. *Building a Better Teacher: How Teaching Works (and How to Teach It to Everyone)*. W. W. Norton & Company, 2014. Explains why educational reforms in the past fifty years has mostly missed the mark, and what we should do instead.

[Grif2016] Jean M. Griffin. Learning by taking apart. In *Proc. 2016 Conference on Information Technology Education (SIGITE'16)*. ACM Press, 2016. Reports that people learn to program more quickly by deconstructing code than by writing it.

[Grov2017] Shuchi Grover and Satabdi Basu. Measuring student learning in introductory block-based programming. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that middle-school children using blocks-based programming find loops, variables, and Boolean operators difficult to understand.

[Gull2004] Ned Gulley. In praise of tweaking. *interactions*, 11(3):18, May 2004. Describes an innovative collaborative coding contest.

[Guo2013] Philip J. Guo. Online python tutor. In *Proc. 2013 Technical Symposium on Computer Science Education (SIGCSE'13)*. Association for Computing Machinery (ACM), 2013. Describes the design and use of a web-based execution visualization tool.

[Guo2014] Philip J. Guo, Juho Kim, and Rob Rubin. How video production affects student engagement. In *Proc. 2014 Conference on Learning @ Scale (L@S'14)*. Association for Computing Machinery (ACM), 2014. Measured learner engagement with MOOC videos and reports that short videos are more engaging than long ones and that talking heads are more engaging than tablet drawings.

[Guzd2013] Mark Guzdial. Exploring hypotheses about media computation. In *Proc. 2013 International Computing Education Research Conference (ICER'13)*. Association for Computing Machinery (ACM), 2013. A look back on ten years of media computation research.

[Guzd2015a] Mark Guzdial. *Learner-Centered Design of Computing Education: Research on Computing for Everyone*. Morgan & Claypool Publishers, 2015. Argues that we must design computing education for everyone, not just people who think they are going to become professional programmers.

[Guzd2015b] Mark Guzdial. Top 10 myths about teaching computer science. https://cacm.acm.org/blogs/blog-cacm/189498-top-10-myths-about-teaching-computer-science/fulltext, 2015. Ten things many people believe about teaching computing that simply aren't true.

[Guzd2016] Mark Guzdial. Five principles for programming languages for learners. https://cacm.acm.org/blogs/blog-cacm/203554-five-principles-for-programming-languages-for-learners/fulltext, 2016. Explains how to choose a programming language for people who are new to programming.

[Haar2017] Lassi Haaranen. Programming as a performance - live-streaming and its implications for computer science education. In *Proc. 2017 Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)*. Association for Computing Machinery (ACM), 2017. An early look at live streaming of coding as a teaching technique.

[Hake1998] Richard R. Hake. Interactive engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66(1):64–74, Jan 1998. Reports the use of a concept inventory to measure the benefits of interactive engagement as a teaching technique.

[Hamo2017] Sally Hamouda, Stephen H. Edwards, Hicham G. Elmongui, Jeremy V. Ernst, and Clifford A. Shaffer. A basic recursion concept inventory. *Computer Science Education*, 27(2):121–148, Apr 2017. Reports early work on developing a concept inventory for recursion.

[Hank2011] Brian Hanks, Sue Fitzgerald, Renée McCauley, Laurie Murphy, and Carol Zander. Pair programming in education: a literature review. *Computer Science Education*, 21(2):135–173, Jun 2011. Reports increased success rates and retention with pair programming, with some evidence that it is particularly beneficial for women, but finds that scheduling and partner compatibility can be problematic.

[Hann2009] Jo Erskine Hannay, Tore Dybå, Erik Arisholm, and Dag I. K. Sjøberg. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122, Jul 2009. A comprehensive meta-analysis of research on pair programming.

[Hann2010] Jo Erskine Hannay, Erik Arisholm, Harald Engvik, and Dag I. K. Sjøberg. Effects of personality on pair programming. *IEEE Transactions on Software Engineering*, 36(1):61–80, Jan 2010. Reports weak correlation between the "Big Five" personality traits and performance in pair programming.

[Hansen2015] John D. Hansen and Justin Reich. Democratizing education? examining access and usage patterns in massive open online courses. *Science*, 350(6265):1245–1248, Dec 2015. Reports that MOOCs are mostly used by the affluent.

[Harm2016] Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. Distractors in Parsons Problems decrease learning efficiency for young novice programmers. In *Proc. 2016 International Computing Education Research Conference (ICER'16)*. Association for Computing Machinery (ACM), 2016. Shows that adding distractors to Parsons Problems does not improve learning outcomes but increases solution times.

[Harr2018] Brian Harrington and Nick Cheng. Tracing vs. writing code: Beyond the learning hierarchy. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Associ-

ation for Computing Machinery (ACM), 2018. Finds that the gap between being able to trace code and being able to write it has largely closed by CS2, and that students who still have a gap (in either direction) are likely to do poorly in the course.

[Hazz2014] Orit Hazzan, Tami Lapidot, and Noa Ragonis. *Guide to Teaching Computer Science: An Activity-Based Approach*. Springer, second edition, 2014. A textbook for teaching computer science at the K-12 level with dozens of activities.

[Hend2015a] Charles Henderson, Renée Cole, Jeff Froyd, Debra Friedrichsen, Raina Khatri, and Courtney Stanford. *Designing Educational Innovations for Sustained Adoption*. Increase the Impact, 2015. A detailed analysis of strategies for getting institutions in higher education to make changes.

[Hend2015b] Charles Henderson, Renée Cole, Jeff Froyd, Debra Friedrichsen, Raina Khatri, and Courtney Stanford. Designing educational innovations for sustained adoption (executive summary). http://www.increasetheimpact.com/resources.html, 2015. A short summary of key points from the authors' work on effecting change in higher education.

[Hend2017] Carl Hendrick and Robin Macpherson. *What Does This Look Like In The Classroom?: Bridging The Gap Between Research And Practice*. John Catt Educational, 2017. A collection of responses by educational experts to questions asked by classroom teachers, with prefaces by the authors.

[Henr2010] Joseph Henrich, Steven J. Heine, and Ara Norenzayan. The weirdest people in the world? *Behavioral and Brain Sciences*, 33(2-3):61–83, Jun 2010. Points out that the subjects of most published psychological studies are Western, educated, industrialized, rich, and democratic.

[Hest1992] David Hestenes, Malcolm Wells, and Gregg Swackhamer. Force concept inventory. *The Physics Teacher*, 30(3):141–158, Mar 1992. Describes the Force Concept Inventory's motivation, design, and impact.

[Hick2018] Marie Hicks. *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing*. MIT Press, 2018. Describes how Britain lost its early dominance in computing by systematically discriminating against its most qualified workers: women.

[Hofm2017] Johannes Hofmeister, Janet Siegmund, and Daniel V. Holt. Shorter identifier names take longer to comprehend. In *Proc.*

2017 Conference on Software Analysis, Evolution and Reengineering (SANER'17). Institute of Electrical and Electronics Engineers (IEEE), Feb 2017. Reports that using words for variable names makes comprehension faster than using abbreviations or single-letter names for variables.

[Holl1960] Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, Oct 1960. A brief note describing what may have been the world's first auto-grader.

[Hu2017] Helen H. Hu, Cecily Heiner, Thomas Gagne, and Carl Lyman. Building a statewide computer science teacher pipeline. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that a six-month program for high school teachers converting to teach CS quadruples the number of teachers without noticeable reduction of student outcomes and increases teachers' belief that anyone can program.

[Hust2012] Therese Huston. *Teaching What You Don't Know*. Harvard University Press, 2012. A pointed, funny, and very useful exploration of exactly what the title says.

[Ihan2010] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proc. 2010 Koli Calling Conference on Computing Education Research (Koli'10)*. Association for Computing Machinery (ACM), 2010. Reviews auto-grading tools of the time.

[Ihan2011] Petri Ihantola and Ville Karavirta. Two-dimensional Parson's Puzzles: The concept, tools, and first observations. *Journal of Information Technology Education: Innovations in Practice*, 10:119–132, 2011. Describes a 2D Parsons Problem tool and early experiences with it that confirm that experts solve outside-in rather than line-by-line.

[Ihan2016] Petri Ihantola, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, Daniel Toll, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, and Andrew Petersen. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proc. 2016 Conference on Innovation and Technology in Computer Science Education (ITiCSE'16)*. Association for Computing Machinery (ACM), 2016. A survey of methods used in mining and analyzing programming data.

[Ijss2000] Wijnand A. IJsselsteijn, Huib de Ridder, Jonathan Freeman, and Steve E. Avons. Presence: Concept, determinants, and measurement. In Bernice E. Rogowitz and Thrasyvoulos N. Pappas, editors, *Proc. 2000 Conference on Human Vision and Electronic Imaging*. SPIE, Jun 2000. Summarizes thinking of the time about real and virtual presence.

[Irib2009] Alicia Iriberri and Gondy Leroy. A life-cycle perspective on online community success. *ACM Computing Surveys*, 41(2):1–29, Feb 2009. Reviews research on online communities organized according to a five-stage lifecycle model.

[Juss2005] Lee Jussim and Kent D. Harber. Teacher expectations and self-fulfilling prophecies: Knowns and unknowns, resolved and unresolved controversies. *Personality and Social Psychology Review*, 9(2):131–155, May 2005. A survey of the effects of teacher expectations on student outcomes.

[Kaly2003] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. The expertise reversal effect. *Educational Psychologist*, 38(1):23–31, Mar 2003. Reports that instructional techniques that work well with inexperienced learners lose their effectiveness or have negative consequences when used with more experienced learners.

[Kaly2015] Slava Kalyuga and Anne-Marie Singh. Rethinking the boundaries of cognitive load theory in complex learning. *Educational Psychology Review*, 28(4):831–852, Dec 2015. Argues that cognitive load theory is basically micro-management within a broader pedagogical context.

[Kang2016] Sean H. K. Kang. Spaced repetition promotes efficient and effective learning. *Policy Insights from the Behavioral and Brain Sciences*, 3(1):12–19, Jan 2016. Summarizes research on spaced repetition and what it means for classroom teaching.

[Kapu2016] Manu Kapur. Examining productive failure, productive success, unproductive failure, and unproductive success in learning. *Educational Psychologist*, 51(2):289–299, Apr 2016. Looks at productive failure as an alternative to inquiry-based learning and approaches based on cognitive load theory.

[Karp2008] Jeffrey D. Karpicke and Henry L. Roediger. The critical importance of retrieval for learning. *Science*, 319(5865):966–968, Feb 2008. Reports that repeated testing improves recall of word lists from 35% to 80%, even when learners can still access the material but are not tested on it.

[Kauf2000] Deborah B. Kaufman and Richard M. Felder. Accounting for individual effort in cooperative learning teams. *Journal*

of Engineering Education, 89(2), 2000. Reports that self-rating and peer ratings in undergraduate courses agree, that collusion isn't significant, that students don't inflate their self-ratings, and that ratings are not biased by gender or race.

[Keme2009] Chris F. Kemerer and Mark C. Paulk. The impact of design and code reviews on software quality: An empirical study based on PSP data. *IEEE Transactions on Software Engineering*, 35(4):534–550, Jul 2009. Uses individual data to explore the effectiveness of code review.

[Kepp2008] Jeroen Keppens and David Hay. Concept map assessment for teaching computer programming. *Computer Science Education*, 18(1):31–42, Mar 2008. A short review of ways concept mapping can be used in CS education.

[Kern1978] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, second edition, 1978. An early and influential description of the Unix programming philosophy.

[Kern1983] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice-Hall, 1983. An influential early description of Unix.

[Kern1988] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988. The book that made C a popular programming language.

[Kern1999] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999. A programming style manual written by two of the creators of modern computing.

[Keun2016a] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proc. 2016 Conference on Innovation and Technology in Computer Science Education (ITiCSE'16)*. Association for Computing Machinery (ACM), 2016. Reports that auto-grading tools often do not give feedback on what to do next, and that teachers cannot easily adapt most of the tools to their needs.

[Keun2016b] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises - extended version. Technical Report UU-CS-2016-001, Utrecht University, 2016. An extended look at feedback messages from auto-grading tools.

[Kim2017] Ada S. Kim and Andrew J. Ko. A pedagogical analysis of online coding tutorials. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that online coding tutorials largely teach similar content, organize content bottom-up, and provide goal-directed practices with immediate feedback, but are not tailored to learners' prior coding knowledge and usually don't tell learners how to transfer and apply knowledge.

[King1993] Alison King. From sage on the stage to guide on the side. *College Teaching*, 41(1):30–35, Jan 1993. An early proposal to flip the classroom.

[Kirk1994] Donald L. Kirkpatrick. *Evaluating Training Programs: The Four Levels*. Berrett-Koehle, 1994. Defines a widely-used four-level model for evaluating training.

[Kirs2006] Paul A. Kirschner, John Sweller, and Richard E. 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):75–86, Jun 2006. Argues that inquiry-based learning is less effective for novices than guided instruction.

[Kirs2013] Paul A. Kirschner and Jeroen J. G. van Merriënboer. Do learners really know best? Urban legends in education. *Educational Psychologist*, 48(3):169–183, Jul 2013. Argues that three learning myths—digital natives, learning styles, and self-educators—all reflect the mistaken belief that learners know what is best for them, and cautions that we may be in a downward spiral in which every attempt by education researchers to rebut these myths confirms their opponents' belief that learning science is pseudo-science.

[Kirs2018] Paul A. Kirschner, John Sweller, Femke Kirschner, and Jimmy Zambrano R. From cognitive load theory to collaborative cognitive load theory. *International Journal of Computer-Supported Collaborative Learning*, Apr 2018. Extends cognitive load theory to include collaborative aspects of learning.

[Koed2015] 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. In *Proc. 2015 Conference on Learning @ Scale (L@S'15)*. Association for Computing Machinery (ACM), 2015. Measures the benefits of doing rather than watching.

[Koeh2013] Matthew J. Koehler, Punya Mishra, and William Cain. What is technological pedagogical content knowledge (TPACK)? *Journal of Education*, 193(3):13–19, 2013. Refines the discussion of PCK by adding technology, and sketches strategies for building understanding of how to use it.

[Kohn2017] Tobias Kohn. Variable evaluation: An exploration of novice programmers' understanding and common misconceptions. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that students often believe in delayed evaluation or that entire equations are stored in variables.

[Koll2015] Michael Kölling. Lessons from the design of three educational programming environments. *International Journal of People-Oriented Programming*, 4(1):5–32, Jan 2015. Compares three generations of programming environments intended for novice use.

[Kran2015] Steven G. Krantz. *How to Teach Mathematics*. American Mathematical Society (AMS), third edition, 2015. Advice and opinions drawn from the author's personal experience of teaching mathematics.

[Krau2016] Robert E. Kraut and Paul Resnick. *Building Successful Online Communities: Evidence-Based Social Design*. MIT Press, 2016. Sums up what we actually know about making thriving online communities and why we believe it's true.

[Krug1999] Justin Kruger and David Dunning. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, 77(6):1121–1134, 1999. The original report on the Dunning-Kruger effect: the less people know, the less accurate their estimate of their knowledge.

[Kuch2011] Marc J. Kuchner. *Marketing for Scientists: How to Shine in Tough Times*. Island Press, 2011. A short, readable guide to making people aware of, and care about, your work.

[Kuit2004] Marja Kuittinen and Jorma Sajaniemi. Teaching roles of variables in elementary programming courses. *ACM SIGCSE Bulletin*, 36(3):57, Sep 2004. Presents a few patterns used in novice programming and the pedagogical value of teaching them.

[Kulk2013] Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R. Klemmer. Peer and self assessment in massive online

classes. *ACM Transactions on Computer-Human Interaction*, 20(6):1–31, Dec 2013. Shows that peer grading can be as effective at scale as expert grading.

[Laba2008] David F. Labaree. The winning ways of a losing strategy: Educationalizing social problems in the United States. *Educational Theory*, 58(4):447–460, Nov 2008. Explores why the United States keeps pushing the solution of social problems onto educational institutions, and why that continues not to work.

[Lach2018] Michael Lachney. Computational communities: African-American cultural capital in computer science education. *Computer Science Education*, pages 1–22, Feb 2018. Explores use of community representation and computational integration to bridge computing and African-American cultural capital in CS education.

[Lang2013] James M. Lang. *Cheating Lessons: Learning from Academic Dishonesty*. Harvard University Press, 2013. Explores why students cheat, and how courses often give them incentives to do so.

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

[Lazo1993] Ard W. Lazonder and Hans van der Meij. The minimal manual: Is less really more? *International Journal of Man-Machine Studies*, 39(5):729–752, Nov 1993. Reports that the minimal manual approach to instruction outperforms traditional approaches regardless of prior experience with computers.

[Leak2017] Mackenzie Leake and Colleen M. Lewis. Recommendations for designing CS resource sharing sites for all teachers. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Explores why CS teachers don't use resource sharing sites and recommends ways to make them more appealing.

[Lee2013] Cynthia Bailey Lee. Experience report: CS1 in MATLAB for non-majors, with media computation and peer instruction. In *Proc. 2013 Technical Symposium on Computer Science Education (SIGCSE'13)*. Association for Computing Machinery (ACM), 2013. Describes an adaptation of media computation to a first-year MATLAB course.

[Lee2017] Cynthia Bailey Lee. What can i do today to create a more inclusive community in CS? http://bit.ly/2oynmSH, 2017. A practical checklist of things instructors can do to make their computing classes more inclusive.

[Lemo2014] Doug Lemov. *Teach Like a Champion 2.0: 62 Techniques that Put Students on the Path to College*. Jossey-Bass, 2014. Presents 62 classroom techniques drawn from intensive study of thousands of hours of video of good teachers in action.

[Lewi2015] Colleen M. Lewis and Niral Shah. How equity and inequity can emerge in pair programming. In *Proc. 2015 International Computing Education Research Conference (ICER'15)*. Association for Computing Machinery (ACM), 2015. Reports a study of pair programming in a middle-grade classroom in which less equitable pairs were ones that sought to complete the task quickly.

[List2004] Raymond Lister, Otto Seppälä, Beth Simon, Lynda Thomas, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, and Kate Sanders. A multi-national study of reading and tracing skills in novice programmers. In *Proc. 2004 Conference on Innovation and Technology in Computer Science Education (ITiCSE'04)*. Association for Computing Machinery (ACM), 2004. Reports that students are weak at both predicting the outcome of executing a short piece of code and at selecting the correct completion for short pieces of code.

[List2009] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin*, 41(3):161, Aug 2009. Replicates earlier studies showing that students who cannot trace code usually cannot explain code and that students who tend to perform reasonably well at code writing tasks have also usually acquired the ability to both trace code and explain code.

[Litt2004] Dennis Littky. *The Big Picture: Education Is Everyone's Business*. Association for Supervision & Curriculum Development (ASCD), 2004. Essays on the purpose of education and how to make schools better.

[Luxt2009] Andrew Luxton-Reilly. A systematic review of tools that support peer assessment. *Computer Science Education*,

19(4):209–232, Dec 2009. Surveys peer assessment tools that may be of use in computing education.

[Luxt2017] Andrew Luxton-Reilly, Jacqueline Whalley, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, and Simon. Developing assessments to determine mastery of programming fundamentals. In *Proc. 2017 Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)*. Association for Computing Machinery (ACM), 2017. Synthesizes work from many previous works to determine what CS instructors are actually teaching, how those things depend on each other, and how they might be assessed.

[Macn2014] Brooke N. Macnamara, David Z. Hambrick, and Frederick L. Oswald. Deliberate practice and performance in music, games, sports, education, and professions: A meta-analysis. *Psychological Science*, 25(8):1608–1618, Jul 2014. A meta-study of the effectiveness of deliberate practice.

[Magu2018] Phil Maguire, Rebecca Maguire, and Robert Kelly. Using automatic machine assessment to teach computer programming. *Computer Science Education*, pages 1–18, Feb 2018. Reports that weekly machine-evaluated tests are a better predictor of exam scores than labs (but that students didn't like the system).

[Majo2015] Claire Howell Major, Michael S. Harris, and Tod Zakrajsek. *Teaching for Learning: 101 Intentionally Designed Educational Activities to Put Students on the Path to Success*. Routledge, 2015. Catalogs a hundred different kinds of exercises to do with students.

[Malo2010] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):1–15, Nov 2010. Summarizes the design of the first generation of Scratch.

[Mann2015] Mary Lynn Manns and Linda Rising. *Fearless Change: Patterns for Introducing New Ideas*. Addison-Wesley, 2015. A catalog of patterns for making change happen in large organizations.

[Marc2011] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proc. 2011 Technical Symposium on Computer Science Education (SIGCSE'11)*. Association for Computing Machinery (ACM), 2011. Looks

at edit-level responses to error messages, and introduces a useful rubric for classifying user responses to errors.

[Marg2003] Jane Margolis and Allan Fisher. *Unlocking the Clubhouse: Women in Computing*. MIT Press, 2003. A groundbreaking report on the gender imbalance in computing, and the steps Carnegie-Mellon took to address the problem.

[Marg2010] Jane Margolis, Rachel Estrella, Joanna Goode, Jennifer Jellison Holme, and Kim Nao. *Stuck in the Shallow End: Education, Race, and Computing*. MIT Press, 2010. Dissects the school structures and belief systems that lead to underrepresentation of African American and Latinx students in computing.

[Marg2012] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proc. 2012 International Computing Education Research Conference (ICER'12)*, pages 71–78. ACM Press, 2012. Reports that labelled subgoals improve outcomes and transference when learning about mobile app development.

[Marg2015] Anoush Margaryan, Manuela Bianco, and Allison Littlejohn. Instructional quality of massive open online courses (MOOCs). *Computers & Education*, 80:77–83, Jan 2015. Reports that instructional design quality in MOOCs poor, but that the organization and presentation of material is good.

[Marg2016] Lauren E. Margulieux, Richard Catrambone, and Mark Guzdial. Employing subgoals in computer programming education. *Computer Science Education*, 26(1):44–67, Jan 2016. Reports that labelled subgoals improve learning outcomes in introductory computing courses.

[Mark2018] Rebecca A. Markovits and Yana Weinstein. Can cognitive processes help explain the success of instructional techniques recommended by behavior analysts? *NPJ Science of Learning*, 3(1), Jan 2018. Points out that behavioralists and cognitive psychologists differ in approach, but wind up making very similar recommendations about how to teach, and gives two specific examples.

[Mars2002] Herbert W. Marsh and John Hattie. The relation between research productivity and teaching effectiveness: Complementary, antagonistic, or independent constructs? *Journal of Higher Education*, 73(5):603–641, 2002. One study of many showing there is zero correlation between research ability and teaching effectiveness.

[Mart2017] Christopher Martin, Janet Hughes, and John Richards. Learning dimensions: Lessons from field studies. In *Proc. 2017 Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)*. Association for Computing Machinery (ACM), 2017. Outlines dimensions along which to evaluate lessons.

[Masa2018] Susana Masapanta-Carrión and J. Ángel Velázquez-Iturbide. A systematic review of the use of Bloom's Taxonomy in computer science education. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Reports that even experienced educators have trouble agreeing on the correct classification for a question or idea using Bloom's Taxonomy.

[Maso2016] Raina Mason, Carolyn Seton, and Graham Cooper. Applying cognitive load theory to the redesign of a conventional database systems course. *Computer Science Education*, 26(1):68–87, Jan 2016. Reports how redesigning a database course using cognitive load theory reduced exam failure rate while increasing student satisfaction.

[Maye2003] Richard E. Mayer and Roxana Moreno. Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist*, 38(1):43–52, Mar 2003. Shows how research into how we absorb and process information can be applied to the design of instructional materials.

[Maye2004] Richard E. Mayer. Teaching of subject matter. *Annual Review of Psychology*, 55(1):715–744, Feb 2004. An overview of how and why teaching and learning are subject-specific.

[Maye2009] Richard E. Mayer. *Multimedia Learning*. Cambridge University Press, second edition, 2009. Presents a cognitive theory of multimedia learning.

[Mazu1996] Eric Mazur. *Peer Instruction: A User's Manual*. Prentice-Hall, 1996. A guide to implementing peer instruction.

[McCa2008] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: A review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, Jun 2008. Summarizes research about why bugs occur, why types there are, how people debug, and whether we can teach debugging skills.

[McCr2001] Michael McCracken, Tadeusz Wilusz, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Ko-

likant, Cary Laxer, Lynda Thomas, and Ian Utting. A multinational, multi-institutional study of assessment of programming skills of first-year CS students. In *Proc. 2001 Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*. Association for Computing Machinery (ACM), 2001. Reports that most students still struggle to solve even basic programming problems at the end of their introductory course.

[McDo2006] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. Pair programming improves student retention, confidence, and program quality. *Communications of the ACM*, 49(8):90–95, Aug 2006. A summary of research showing that pair programming improves retention and confidence.

[McGu2015] Saundra Yancey McGuire. *Teach Students How to Learn: Strategies You Can Incorporate Into Any Course to Improve Student Metacognition, Study Skills, and Motivation*. Stylus Publishing, 2015. Explains how metacognitive strategies can improve learning.

[McMi2017] Tressie McMillan Cottom. *Lower Ed: The Troubling Rise of For-Profit Colleges in the New Economy*. The New Press, 2017. Lays bare the dynamics of the growing educational industry to show how it leads to greater inequality rather than less.

[McTi2013] Jay McTighe and Grant Wiggins. Understanding by design framework. http://www.ascd.org/ASCD/pdf/siteASCD/publications/UbD_WhitePaper03
2013. Summarizes the backward instructional design process.

[Metc2016] Janet Metcalfe. Learning from errors. *Annual Review of Psychology*, 68(1):465–489, Jan 2016. Summarizes work on the hypercorrection effect in learning.

[Meys2018] Mark Meysenburg, Tessa Durham Brooks, Raychelle Burks, Erin Doyle, and Timothy Frey. DIVAS: Outreach to the natural sciences through image processing. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Describes early results from a programming course for science undergrads built around image processing.

[Midw2010] Midwest Academy. *Organizing for Social Change: Midwest Academy Manual for Activists*. The Forum Press, fourth edition, 2010. A training manual for people building progressive social movements.

[Mill1956] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956. The original paper on the limited size of short-term memory.

[Mill2013] Kelly Miller, Nathaniel Lasry, Kelvin Chu, and Eric Mazur. Role of physics lecture demonstrations in conceptual learning. *Physical Review Special Topics - Physics Education Research*, 9(2), Sep 2013. Reports a detailed study of what students learn during demonstrations and why.

[Mill2015] David I. Miller and Jonathan Wai. The bachelor's to ph.d. STEM pipeline no longer leaks more women than men: a 30-year analysis. *Frontiers in Psychology*, 6, Feb 2015. Shows that the "leaky pipeline" metaphor stopped being accurate some time in the 1990s.

[Mill2016a] Michelle D. Miller. *Minds Online: Teaching Effectively with Technology*. Harvard University Press, 2016. Describes ways that insights from neuroscience can be used to improve online teaching.

[Mill2016b] Craig S. Miller and Amber Settle. Some trouble with transparency: An analysis of student errors with object-oriented Python. In *Proc. 2016 International Computing Education Research Conference (ICER'16)*. Association for Computing Machinery (ACM), 2016. Reports that students have difficulty with `self` in Python.

[Milt2018] Kate M. Miltner. Girls who coded: Gender in twentieth century U.K. and U.S. computing. *Science, Technology, & Human Values*, May 2018. A review of three books about how women were systematically pushed out of computing.

[Miya2018] Toshiya Miyatsu, Khuyen Nguyen, and Mark A. McDaniel. Five popular study strategies: Their pitfalls and optimal implementations. *Perspectives on Psychological Science*, 13(3):390–407, May 2018. Explains how learners mis-use common study strategies and what they should do instead.

[Mlad2017] Monika Mladenović, Ivica Boljat, and Žana Žanko. Comparing loops misconceptions in block-based and text-based programming languages at the K-12 level. *Education and Information Technologies*, Nov 2017. Reports that K-12 students have fewer misconceptions about loops using Scratch than using Logo or Python, and fewer misconceptions about nested loops with Logo than with Python.

[Morr2016] Briana B. Morrison, Lauren E. Margulieux, Barbara J. Ericson, and Mark Guzdial. Subgoals help students solve Parsons

Problems. In *Proc. 2016 Technical Symposium on Computer Science Education (SIGCSE'16)*. Association for Computing Machinery (ACM), 2016. Reports that students using labelled subgoals solve Parsons Problems better than students without labelled subgoals.

[Muel2014] Pam A. Mueller and Daniel M. Oppenheimer. The pen is mightier than the keyboard. *Psychological Science*, 25(6):1159–1168, Apr 2014. Presents evidence that taking notes by hand is more effective than taking notes on a laptop.

[Muhl2016] Andreas Mühling. Aggregating concept map data to investigate the knowledge of beginning CS students. *Computer Science Education*, 26(2-3):176–191, Jul 2016. Analyzed concepts maps drawing by students with prior CS experience and those without to compare their mental models.

[Mull2007a] Derek A. Muller, James Bewes, Manjula D. Sharma, and Peter Reimann. Saying the wrong thing: Improving learning with multimedia by including misconceptions. *Journal of Computer Assisted Learning*, 24(2):144–155, Jul 2007. Reports that including explicit discussion of misconceptions significantly improves learning outcomes: students with low prior knowledge benefit most and students with more prior knowledge are not disadvantaged.

[Mull2007b] Orna Muller, David Ginat, and Bruria Haberman. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proc. 2007 Technical Symposium on Computer Science Education (SIGCSE'07)*. Association for Computing Machinery (ACM), 2007. Reports that explicitly teaching solution patterns improves learning outcomes.

[Murp2008] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: The good, the bad, and the quirky - a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, 40(1):163, Feb 2008. Reports that many CS1 students use good debugging strategies, but many others don't, and students often don't recognize when they are stuck.

[Nara2018] Sathya Narayanan, Kathryn Cunningham, Sonia Arteaga, William J. Welch, Leslie Maxwell, Zechariah Chawinga, and Bude Su. Upward mobility for underrepresented students. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery

(ACM), 2018. Describes an intensive 3-year bachelor's program based on tight-knit cohorts and administrative support that tripled graduation rates.

[Nath2003] Mitchell J. Nathan and Anthony Petrosino. Expert blind spot among preservice teachers. *American Educational Research Journal*, 40(4):905–928, Jan 2003. Early work on expert blind spot.

[Nils2017] Linda B. Nilson and Ludwika A. Goodson. *Online Teaching at Its Best: Merging Instructional Design with Teaching and Learning Research*. Jossey-Bass, 2017. A guide for college instructors that focuses on online teaching.

[Nord2017] Emily Nordmann, Colin Calder, Paul Bishop, Amy Irwin, and Darren Comber. Turn up, tune in, don't drop out: The relationship between lecture attendance, use of lecture recordings, and achievement at different levels of study. https://psyarxiv.com/fd3yj, 2017. Reports on the pros and cons of recording lectures.

[Nutb2016] Stephen Nutbrown and Colin Higgins. Static analysis of programming exercises: Fairness, usefulness and a method for application. *Computer Science Education*, 26(2-3):104–128, May 2016. Describes ways auto-grader rules were modified and grades weighted to improve correlation between automatic feedback and manual grades.

[Nuth2007] Graham Nuthall. *The Hidden Lives of Learners*. NZCER Press, 2007. Summarizes a lifetime of work looking at what students actually do in classrooms and how they actually learn.

[Ojos2015] Bobby Ojose. *Common Misconceptions in Mathematics: Strategies to Correct Them*. UPA, 2015. A catalog of K-12 misconceptions in mathematics and what to do about them.

[Ornd2015] 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):340–351, 2015. Reports that taking notes together online is more effective than solo note-taking.

[Pape1993] Seymour A. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, second edition, 1993. The foundational text on how computers can underpin a new kind of education.

[Pare2008] Dwayne E. Paré and Steve Joordens. Peering into large lectures: Examining peer and expert mark agreement using

peerScholar, an online peer assessment tool. *Journal of Computer Assisted Learning*, 24(6):526–540, Oct 2008. Shows that peer grading by small groups can be as effective as expert grading once accountability features are introduced.

[Park2015] Thomas H. Park, Brian Dorn, and Andrea Forte. An analysis of HTML and CSS syntax errors in a web development course. *ACM Transactions on Computing Education*, 15(1):1–21, Mar 2015. Describes the errors students make in an introductory course on HTML and CSS.

[Park2016] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. Replication, validation, and use of a language independent CS1 knowledge assessment. In *Proc. 2016 International Computing Education Research Conference (ICER'16)*. Association for Computing Machinery (ACM), 2016. Describes construction and replication of a second concept inventory for basic computing knowledge.

[Parn1986] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, Feb 1986. Argues that using a rational design process is less important than looking as though you had.

[Parn2017] Chris Parnin, Janet Siegmund, and Norman Peitek. On the nature of programmer expertise. In *Psychology of Programming Interest Group Workshop 2017*, 2017. An annotated exploration of what "expertise" means in programming.

[Pars2006] Dale Parsons and Patricia Haden. Parson's programming puzzles: A fun and effective learning tool for first programming courses. In *Proc. 2006 Australasian Conference on Computing Education (ACE'06)*, pages 157–163. Australian Computer Society, 2006. The first description of Parson's Problems.

[Pati2016] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. Evidence that computer science grades are not bimodal. In *Proc. 2016 International Computing Education Research Conference (ICER'16)*. Association for Computing Machinery (ACM), 2016. Presents a statistical analysis and an experiment which jointly show that grades in computing classes are not bimodal.

[Pea1986] Roy D. Pea. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1):25–36, Feb 1986. First named the "superbug" in coding: most newcomers think the computer understands what they want, in the same way that a human being would.

[Pete2017] John Peterson and Greg Haynes. Integrating computer science into music education. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Describes a DSL for music composition that can be used to introduce coding ideas into introductory music classes.

[Petr2016] Marian Petre and André van der Hoek. *Software Design Decoded: 66 Ways Experts Think*. MIT Press, 2016. A short illustrated overview of how expert software developers think.

[Pign2016] Alessandra Pigni. *The Idealist's Survival Kit: 75 Simple Ways to Prevent Burnout*. Parallax Press, 2016. A guide to staying sane and healthy while doing good.

[Port2013] Leo Porter, Mark Guzdial, Charlie McDowell, and Beth Simon. Success in introductory programming: What works? *Communications of the ACM*, 56(8):34, Aug 2013. Summarizes the evidence that peer instruction, media computation, and pair programming can significantly improve outcomes in introductory programming courses.

[Port2016] Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Bailey Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. A multi-institutional study of peer instruction in introductory computing. In *Proc. 2016 Technical Symposium on Computer Science Education (SIGCSE'16)*. Association for Computing Machinery (ACM), 2016. Reports that students in introductory programming classes value peer instruction, and that it improves learning outcomes.

[Qian2017] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming. *ACM Transactions on Computing Education*, 18(1):1–24, Oct 2017. Summarizes research on student misconceptions about computing.

[Rago2017] Noa Ragonis and Ronit Shmallo. On the (mis)understanding of the "this" reference. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that most students do not understood when to use `this`, and that teachers are also often not clear on the subject.

[Raws2014] Katherine A. Rawson, Ruthann C. Thomas, and Larry L. Jacoby. The power of examples: Illustrative examples enhance conceptual learning of declarative concepts. *Educational Psychology Review*, 27(3):483–504, Jun 2014. Reports that

presenting examples helps students understand definitions, so long as examples and definitions are interleaved.

[Ray2014] Eric J. Ray and Deborah S. Ray. *Unix and Linux: Visual QuickStart Guide*. Peachpit Press, fifth edition, 2014. An introduction to Unix that is both a good tutorial and a good reference guide.

[Rice2018] Gail Taylor Rice. *Hitting Pause: 65 Lecture Breaks to Refresh and Reinforce Learning*. Stylus Publishing, 2018. Justifies and catalogs ways to take a pause in class to help learning.

[Rich2017] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, Cheryl Moran, and Diana Franklin. K-8 learning trajectories derived from research literature. In *Proc. 2017 International Computing Education Research Conference (ICER'17)*. Association for Computing Machinery (ACM), 2017. Presents learning trajectories for K-8 computing classes for Sequence, Repetition, and Conditions gleaned from the literature.

[Ritz2018] Anna Ritz. Programming the central dogma: An integrated unit on computer science and molecular biology concepts. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Describes an introductory computing course for biologists whose problems are drawn from the DNA-to-protein processes in cells.

[Robe2017] Eric Roberts. Assessing and responding to the growth of computer science undergraduate enrollments: Annotated findings. cs.stanford.edu/people/eroberts/ResourcesForTheCSCapacityCrisis/files/Ann 2017. Summarizes findings from a National Academies study about computer science enrollments.

[Robi2005] Evan Robinson. Why crunch mode doesn't work: 6 lessons. http://www.igda.org/articles/erobinson_crunch.php, 2005. Summarizes research on the effects of overwork and sleep deprivation.

[Rohrer2015] Doug Rohrer, Robert F. Dedrick, and Sandra Stershic. Interleaved practice improves mathematics learning. *Journal of Educational Psychology*, 107(3):900–908, 2015. Reports that interleaved practice is more effective than monotonous practice when learning.

[Rubi2013] Marc J. Rubin. The effectiveness of live-coding to teach introductory programming. In *Proc. 2013 Technical Symposium on Computer Science Education (SIGCSE'13)*, pages 651–656.

Association for Computing Machinery (ACM), 2013. Reports that live coding is as good as or better than using static code examples.

[Rubi2014] Manuel Rubio-Sánchez, Päivi Kinnunen, Cristóbal Pareja-Flores, and J. Ángel Velázquez-Iturbide. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 31:453–460, Feb 2014. Describes use of an auto-grader for student assignments.

[Saja2006] Jorma Sajaniemi, Mordechai Ben-Ari, Pauli Byckling, Petri Gerdt, and Yevgeniya Kulikova. Roles of variables in three programming paradigms. *Computer Science Education*, 16(4):261–279, Dec 2006. A detailed look at the authors' work on roles of variables.

[Sala2017] Giovanni Sala and Fernand Gobet. Does far transfer exist? Negative evidence from chess, music, and working memory training. *Current Directions in Psychological Science*, 26(6):515–520, Oct 2017. A meta-analysis showing that far transfer rarely occurs.

[Sand2013] Kate Sanders, Jaime Spacco, Marzieh Ahmadzadeh, Tony Clear, Stephen H. Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, and Elizabeth Patitsas. The Canterbury QuestionBank: Building a repository of multiple-choice CS1 and CS2 questions. In *Proc. 2013 Conference on Innovation and Technology in Computer Science Education (ITiCSE'13)*. Association for Computing Machinery (ACM), 2013. Describes development of a shared question bank for introductory CS, and patterns for multiple choice questions that emerged from entries.

[Scaf2017] Christopher Scaffidi. Workers who use spreadsheets and who program earn more than similar workers who do neither. In *Proc. 2017 Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17)*. Institute of Electrical and Electronics Engineers (IEEE), 2017. Reports that workers who aren't software developers but who program make higher wages than comparable workers who do not.

[Scan1989] David A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6(5):28–36, Sep 1989. Reports that students understand flowcharts better than pseudocode if both are equally well structured.

[Scho1984] Donald A. Schön. *The Reflective Practitioner: How Professionals Think In Action*. Basic Books, 1984. A groundbreaking

look at how professionals in different fields actually solve problems.

[Scot1998] James C. Scott. *Seeing Like a State: How Certain Schemes to Improve the Human Condition Have Failed*. Yale University Press, 1998. Argues that large organizations consistently prefer uniformity over productivity.

[Sent2018] Sue Sentance, Erik Barendsen, and Carsten Schulte, editors. *Computer Science Education: Perspectives on Teaching and Learning in School*. Bloomsbury Press, 2018. A collection of academic survey articles on teaching computing.

[Sepp2015] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. Do we know how difficult the Rainfall Problem is? In *Proc. 2015 Koli Calling Conference on Computing Education Research (Koli'15)*. ACM Press, 2015. A meta-study of the Rainfall Problem.

[Shap2007] Jenessa R. Shapiro and Steven L. Neuberg. From stereotype threat to stereotype threats: Implications of a multi-threat framework for causes, moderators, mediators, consequences, and interventions. *Personality and Social Psychology Review*, 11(2):107–130, MAY 2007. Explores the ways the term "stereotype threat" has been used.

[Shel2017] Duane F. Shell, Leen-Kiat Soh, Abraham E. Flanigan, Markeya S. Peteranetz, and Elizabeth Ingraham. Improving students' learning and achievement in CS classrooms through computational creativity exercises that integrate computational and creative thinking. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that having students work in small groups on computational creativity exercises improves learning outcomes.

[Simo2013] Simon. Soloway's Rainfall Problem has become harder. In *Proc. 2013 Conference on Learning and Teaching in Computing and Engineering*. Institute of Electrical and Electronics Engineers (IEEE), Mar 2013. Argues that the Rainfall Problem is harder for novices than it used to be because they're not used to handling keyboard input, so direct comparison with past results may be unfair.

[Sirk2012] Teemu Sirkiä and Juha Sorva. Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *Proc. 2012 Koli Calling Conference on Computing Education Research (Koli'12)*. Association for Computing Machinery (ACM), 2012. Analyzes data from

student use of an execution visualization tool and classifies common mistakes.

[Sisk2018] Victoria F. Sisk, Alexander P. Burgoyne, Jingze Sun, Jennifer L. Butler, and Brooke N. Macnamara. To what extent and under which circumstances are growth mind-sets important to academic achievement? Two meta-analyses. *Psychological Science*, page 095679761773970, Mar 2018. Reports meta-analyses of the relationship between mind-set and academic achievement, and the effectiveness of mind-set interventions on academic achievement, and finds that overall effects are weak for both, but some results support specific tenets of the theory.

[Skud2014] Ben Skudder and Andrew Luxton-Reilly. Worked examples in computer science. In *Proc. 2014 Australasian Computing Education Conference, (ACE'14)*, 2014. A summary of research on worked examples as applied to computing education.

[Smar2018] Benjamin L. Smarr and Aaron E. Schirmer. 3.4 million real-world learning management system logins reveal the majority of students experience social jet lag correlated with decreased performance. *Scientific Reports*, 8(1), Mar 2018. Reports that students who have to work outside their natural body clock cycle do less well.

[Smit2009] Michelle K. Smith, William B. Wood, Wendy K. Adams, Carl E. Wieman, Jennifer K. Knight, N. Guild, and T. T. Su. Why peer discussion improves student performance on in-class concept questions. *Science*, 323(5910):122–124, Jan 2009. Reports that student understanding increases during discussion in peer instruction, even when none of the students in the group initially know the right answer.

[Solo1984] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, Sep 1984. Proposes that experts have programming plans and rules of programming discourse.

[Solo1986] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, Sep 1986. Analyzes programming in terms of choosing appropriate goals and constructing plans to achieve them, and introduces the Rainfall Problem.

[Sond2012] Harald Søndergaard and Raoul A. Mulder. Collaborative learning through formative peer review: Pedagogy, programs and potential. *Computer Science Education*, 22(4):343–367,

Dec 2012. Surveys literature on student peer assessment, distinguishing grading and reviewing as separate forms, and summarizes features a good peer review system needs to have.

[Sorv2013] Juha Sorva. Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2):1–31, Jun 2013. Reviews literature on programming misconceptions, and argues that instructors should address notional machines as an explicit learning objective.

[Sorv2014] Juha Sorva and Otto Seppälä. Research-based design of the first weeks of CS1. In *Proc. 2014 Koli Calling Conference on Computing Education Research (Koli'14)*. Association for Computing Machinery (ACM), 2014. Proposes three cognitively plausible frameworks for the design of a first CS course.

[Sorv2018] Juha Sorva. Misconceptions and the beginner programmer. In Sue Sentance, Erik Barendsen, and Carsten Schulte, editors, *Computer Science Education: Perspectives on Teaching and Learning in School*. Bloomsbury Press, 2018. Summarizes what we know about what novices misunderstand about computing.

[Spal2014] Dan Spalding. *How to Teach Adults: Plan Your Class, Teach Your Students, Change the World*. Jossey-Bass, 2014. A short guide to teaching adult free-range learners informed by the author's social activism.

[Spoh1985] James C. Spohrer, Elliot Soloway, and Edgar Pope. A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction*, 1(2):163–207, Jun 1985. One of the first cognitively plausible analyses of how people program, which proposes a goal/plan model.

[Srid2016] Sumukh Sridhara, Brian Hou, Jeffrey Lu, and John DeNero. Fuzz testing projects in massive courses. In *Proc. 2016 Conference on Learning @ Scale (L@S'16)*. Association for Computing Machinery (ACM), 2016. Reports that fuzz testing student code catches errors that are missed by handwritten test suite, and explains how to safely share tests and results.

[Stam2013] Eliane Stampfer and Kenneth R. Koedinger. When seeing isn't believing: Influences of prior conceptions and misconceptions. In *Proc. 2013 Annual Meeting of the Cognitive Science Society (CogSci'13)*, 2013. Explores why giving children more information when they are learning about fractions can lower their performance.

[Stam2014] Eliane Stampfer Wiese and Kenneth R. Koedinger. Investigating scaffolds for sense making in fraction addition and comparison. In *Proc. 2014 Annual Conference of the Cognitive Science Society (CogSci'14)*, 2014. Looks at how to scaffold learning of fraction operations.

[Star2014] Philip Stark and Richard Freishtat. An evaluation of course evaluations. *ScienceOpen Research*, Sep 2014. Yet another demonstration that teaching evaluations don't correlate with learning outcomes, and that they are frequently statistically suspect.

[Stas1998] John Stasko, John Domingue, Mark H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998. A survey of program and algorithm visualization techniques and results.

[Stee2011] Claude M. Steele. *Whistling Vivaldi: How Stereotypes Affect Us and What We Can Do*. W. W. Norton & Company, 2011. Explains and explores stereotype threat and strategies for addressing it.

[Stef2013] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, 13(4):1–40, Nov 2013. Reports that curly-brace languages are as hard to learn as a language with randomly-designed syntax, but others are easier.

[Stef2017] Andreas Stefik, Patrick Daleiden, Diana Franklin, Stefan Hanenberg, Antti-Juhani Kaijanaho, Walter Tichy, and Brett A. Becker. Programming languages and learning. https://quorumlanguage.com/evidence.html, 2017. Summarizes what we actually know about designing programming languages and why we believe it's true.

[Steg2014] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Towards an empirically validated model for assessment of code quality. In *Proc. 2014 Koli Calling Conference on Computing Education Research (Koli'14)*. Association for Computing Machinery (ACM), 2014. Presents a code quality rubric for novice programming courses.

[Steg2016a] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Designing a rubric for feedback on code quality in programming courses. In *Proc. 2016 Koli Calling Conference on Computing Education Research (Koli'16)*. Association for Computing Machinery (ACM), 2016. Describes several iterations of a code quality rubric for novice programming courses.

[Steg2016b] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Rubric for feedback on code quality in programming courses. http://stgm.nl/quality, 2016. Presents a code quality rubric for novice programming.

[Stoc2018] Jean Stockard, Timothy W. Wood, Cristy Coughlin, and Caitlin Rasplica Khoury. The effectiveness of direct instruction curricula: A meta-analysis of a half century of research. *Review of Educational Research*, page 003465431775191, Jan 2018. A meta-analysis that finds significant positive benefit for Direct Instruction.

[Sung2012] Eunmo Sung and Richard E. Mayer. When graphics improve liking but not learning from online lessons. *Computers in Human Behavior*, 28(5):1618–1625, Sep 2012. Reports that students who receive any kind of graphics give significantly higher satisfaction ratings than those who don't, but only students who get instructive graphics perform better than groups that get no graphics, seductive graphics, or decorative graphics.

[Sved2016] Maria Svedin and Olle Bälter. Gender neutrality improved completion rate for all. *Computer Science Education*, 26(2-3):192–207, Jul 2016. Reports that redesigning an online course to be gender neutral improves completion probability in general, but decreases it for students with a superficial approach to learning.

[Tedr2008] Matti Tedre and Erkki Sutinen. Three traditions of computing: What educators should know. *Computer Science Education*, 18(3):153–170, Sep 2008. Summarizes the history and views of three traditions in computing: mathematical, scientific, and engineering.

[Tew2011] Allison Elliott Tew and Mark Guzdial. The FCS1: A language independent assessment of CS1 knowledge. In *Proc. 2011 Technical Symposium on Computer Science Education (SIGCSE'11)*. Association for Computing Machinery (ACM), 2011. Describes development and validation of a language-independent assessment instrument for CS1 knowledge.

[Thay2017] Kyle Thayer and Andrew J. Ko. Barriers faced by coding bootcamp students. In *Proc. 2017 International Computing Education Research Conference (ICER'17)*. Association for Computing Machinery (ACM), 2017. Reports that coding bootcamps are sometimes useful, but quality is varied, and formal and informal barriers to employment remain.

[Ubel2017] 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, 2017. A brief exploration of why MOOCs haven't lived up to initial hype.

[Urba2014] 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):1029–1038, Mar 2014. Reports a study showing that the introduction of surgical checklists did not have a significant effect on operative outcomes.

[Utti2013] Ian Utting, Juha Sorva, Tadeusz Wilusz, Allison Elliott Tew, Michael McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael E. Caspersen, and Yifat Ben-David Kolikant. A fresh look at novice programmers' performance and their teachers' expectations. In *Proc. 2013 Conference on Innovation and Technology in Computer Science Education (ITiCSE'13)*. ACM Press, 2013. Replicates an earlier study showing how little students learn in their first programming course.

[Uttl2017] 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*, 54:22–42, Sep 2017. Summarizes studies showing that how students rate a course and how much they actually learn are not related.

[Varm2015] Roli Varma and Deepak Kapur. Decoding femininity in computer science in india. *Communications of the ACM*, 58(5):56–62, apr 2015. Reports female participation in computing in India.

[Vell2017] Mickey Vellukunnel, Philip Buffum, Kristy Elizabeth Boyer, Jeffrey Forbes, Sarah Heckman, and Ketan Mayer-Patel. Deconstructing the discussion forum: Student questions and computer science learning. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Found that students mostly ask constructivist and logistical questions in forums, and that the former correlate with grades.

[Viha2014] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proc. 2014 International Computing Education Research Conference*

(ICER'14). Association for Computing Machinery (ACM), 2014. Consolidates studies of CS1-level teaching changes and finds media computation the most effective, while introducing a game theme is the least effective.

[Wall2009] Thorbjorn Walle and Jo Erskine Hannay. Personality and the nature of collaboration in pair programming. In *Proc. 2009 International Symposium on Empirical Software Engineering and Measurement (ESER'09)*. Institute of Electrical and Electronics Engineers (IEEE), Oct 2009. Reports that pairs with different levels of a given personality trait communicated more intensively.

[Wang2018] April Y. Wang, Ryan Mitts, Philip J. Guo, and Parmit K. Chilana. Mismatch of expectations: How modern learning resources fail conversational programmers. In *Proc. 2018 Conference on Human Factors in Computing Systems (CHI'18)*. Association for Computing Machinery (ACM), 2018. Reports that learning resources don't really help conversational programmers (those who learn coding to take part in technical discussions).

[Ward2015] James Ward. *Adventures in Stationery: A Journey Through Your Pencil Case*. Profile Books, 2015. A wonderful look at the everyday items that would be in your desk drawer if someone hadn't walked off with them.

[Wats2014] Christopher Watson and Frederick W. B. Li. Failure rates in introductory programming revisited. In *Proc. 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. Association for Computing Machinery (ACM), 2014. A larger version of an earlier study that found an average of one third of students fail CS1.

[Watt2014] Audrey Watters. *The Monsters of Education Technology*. CreateSpace, 2014. A collection of essays about the history of educational technology and the exaggerated claims repeatedly made for it.

[Wein2017a] David Weintrop and Nathan Holbert. From blocks to text and back: Programming patterns in a dual-modality environment. In *Proc. 2017 Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery (ACM), 2017. Reports that students using a dual-mode blocks and text coding environment tend to migrate from blocks to text over time, and that two thirds of the shifts from text to blocks were followed by adding a new type of command.

[Wein2017b] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education*, 18(1):1–25, Oct 2017. Reports that students learn faster and better with blocks than with text.

[Wein2018] Yana Weinstein, Christopher R. Madan, and Megan A. Sumeracki. Teaching the science of learning. *Cognitive Research: Principles and Implications*, 3(1), Jan 2018. A tutorial review of six evidence-based learning practices.

[Weng2015] Etienne Wenger-Trayner and Beverly Wenger-Trayner. Communities of practice: A brief introduction. http://wenger-trayner.com/intro-to-cops/, 2015. A brief summary of what communities of practice are and aren't.

[Wibu2016] Karin Wiburg, Julia Parra, Gaspard Mucundanyi, Jennifer Green, and Nate Shaver, editors. *The Little Book of Learning Theories*. CreateSpace, second edition, 2016. Presents brief summaries of various theories of learning.

[Wigg2005] Grant Wiggins and Jay McTighe. *Understanding by Design*. Association for Supervision & Curriculum Development (ASCD), 2005. A lengthy presentation of reverse instructional design.

[Wilc2018] Chris Wilcox and Albert Lionelle. Quantifying the benefits of prior programming experience in an introductory computer science course. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Reports that students with prior experience outscore students without in CS1, but there is no significant difference in performance by the end of CS2; also finds that female students with prior exposure outperform their male peers in all areas, but are consistently less confident in their abilities.

[Wilk2011] Richard Wilkinson and Kate Pickett. *The Spirit Level: Why Greater Equality Makes Societies Stronger*. Bloomsbury Press, 2011. Presents evidence that inequality harms everyone, both economically and otherwise.

[Will2010] Daniel T. Willingham. *Why Don't Students Like School?: A Cognitive Scientist Answers Questions about How the Mind Works and What It Means for the Classroom*. Jossey-Bass, 2010. A cognitive scientist looks at how the mind works in the classroom.

[Wils2007] Karen Wilson and James H. Korn. Attention during lectures: Beyond ten minutes. *Teaching of Psychology*, 34(2):85–89,

Jun 2007. Reports little support for the claim that students only have a 10–15 minute attention span (though there is lots of individual variation).

[Wils2016] Greg Wilson. Software Carpentry: Lessons learned. *F1000Research*, Jan 2016. A history and analysis of Software Carpentry.

[Wlod2017] Raymond J. Wlodkowski and Margery B. Ginsberg. *Enhancing Adult Motivation to Learn: A Comprehensive Guide for Teaching All Adults*. Jossey-Bass, 2017. The standard reference for understanding adult motivation.

[Yada2016] Aman Yadav, Sarah Gretter, Susanne Hambrusch, and Phil Sands. Expanding computer science education in schools: Understanding teacher experiences and challenges. *Computer Science Education*, 26(4):235–254, Dec 2016. Summarizes feedback from K-12 teachers on what they need by way of preparation and support.

[Yang2015] Yu-Fen Yang and Yuan-Yu Lin. Online collaborative note-taking strategies to foster EFL beginners' literacy development. *System*, 52:127–138, Aug 2015. Reports that students using collaborative note taking when learning English as a foreign language do better than those who don't.

A License

This is a human-readable summary of (and not a substitute for) the license. Please see https://creativecommons.org/licenses/by/4.0/legalcode for the full legal text.

This work is licensed under the Creative Commons Attribution 4.0 International license (CC-BY-4.0).

You are free to:

- **Share**—copy and redistribute the material in any medium or format
- **Remix**—remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution**—You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions**—You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

B Citation

Please cite this work as:

> *Greg Wilson (ed.):* Teaching Tech Together. *Lulu.com, 2018, 978-0-9881137-0-1, http://teachtogether.tech/*.

C Joining Our Community

This appendix describes how you can become part of our community by using, sharing, and improving this material.

C.1 Contributor Covenant

The contributor covenant laid out below governs contributions to this book. It is adapted from the Contributor Covenant version 1.4; please see s:conduct for a sample code of conduct for use in classes and other learning situations.

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socioeconomic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances

- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this covenant, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This covenant applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at gvwilson@third-bit.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the covenant in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

C.2 Using This Material

This material has been used in many ways, from a multi-week online class to an intensive in-person workshop. It's usually possible to cover large parts of Chapter 2 to Chapter 6, Chapter 8, and Chapter 10 in two long days.

In Person

This is the most effective way to deliver this training, but also the most demanding. Participants are physically together. When they need to practice teaching in small groups, some or all of them go to nearby breakout spaces. Participants use their own tablets or laptops to view online material during the class and for shared note-taking (Section 9.7), and use pen and paper or whiteboards for other exercises. Questions and discussion are done aloud.

If you are teaching in this format, you should use sticky notes as status flags so that you can see who needs help, who has questions, and who's ready to move on (Section 9.8). You should also use them to distribute attention so that everyone gets a fair share of the instructor's time, and as minute cards to encourage learners to reflect on what they've just learned and to give you actionable feedback while you still have time to act on it.

Online in Groups

In this format, 10–40 learners are together in 2–6 groups of 4–12, but those groups are geographically distributed. Each group uses one camera and microphone to connect to the video call, rather than each person 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 via text online.

The entire class does shared note-taking together, and also uses the shared notes 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.)

Online as Individuals

The natural extension of being online in groups is to be online as individuals. As with online groups, the instructor will do most of the talking and learners will mostly participate via text chat. Good audio is once again more important than good video, and participants should use text chat to signal that they want to speak next (s:meetings).

Having participants online individually makes it more difficult to draw and share concept maps (Section 3.5) or give feedback on teaching (Sec-

tion 8.5). Instructors should therefore rely more on exercises with written results that can be put in the shared notes, such as giving feedback on stock videos of people teaching.

Multi-Week Online

This was the first format used, and I no longer recommend it: while spreading the class out gives people time to reflect and tackle larger exercises, it also greatly increases the odds that they'll have to drop out because of other demands on their time.

The class meets every week for an hour via video conferencing. Each meeting may be held twice to accommodate learners' time zones and schedules. Participants use shared note-taking as described above for online group classes, post homework online between classes, and comment on each other's work. (In practice, comments are relatively rare: people strongly prefer to discuss material in the weekly meetings.)

C.3 Contributing and Maintaining

This book is a community resource: contributions of all kinds are welcome, from suggestions for improvements to errata and new material. All contributors must abide by the contributor covenant presented above; by submitting your work, you are agreeing that it may incorporated in either original or edited form and release it under the same license as the rest of this material (s:license). If your material is incorporated, we will add you to the acknowledgments (Section 1.6) unless you request otherwise.

- The source for this book is stored on GitHub at https://github.com/gvwilson/teachtogether.tech/. If you know how to use Git and GitHub and would like to change, fix, or add something, please submit a pull request that modifies the LaTeX source in the `tex` directory. If you would like to preview your changes, please read the instructions in the `BUILD.md` file in the root directory of the project.
- If you simply want to report an error, ask a question, or make a suggestion, please file an issue at https://github.com/gvwilson/teachtogether.tech/. You need to have a GitHub account in order to do this, but do not need to know how to use Git.
- If you do not wish to create a GitHub account, please email your contribution to gvwilson@third-bit.com with either "T3" or "Teaching Tech Together" somewhere in the subject line. We will try to respond within a week.

Please note that we also welcome improvements to our build process, tooling, and typography, and are always grateful for more diagrams; please see the file `BUILD.md` in the root directory of the book's GitHub repository

at https://github.com/gvwilson/teachtogether.tech/ for more information. Finally, we always enjoy hearing how people have used this material: please let us know if you have a story you would like to share.

A Teaching Commons

Section 13.4 defined a commons as something managed jointly by a community according to rules they themselves have evolved and adopted. Open source software and Wikipedia are both successful examples; the question is, why don't teachers build lessons collaboratively in the same way? People have proposed a variety of reasons, but I don't think any of them hold up to close scrutiny.

Software Carpentry is proof by implementation that a teaching commons can produce and maintain high-quality lessons that hundreds of people can use [Wils2016]. I hope you will choose to help us do the same for this book. If you are new to working this way:

Start small. Fix a typo, clarify the wording of an exercise, correct or update a citation, or suggest a better example or analogy to illustrate some point.

Join the conversation. Have a look at the issues and proposed changes that other people have already filed and add your comments to them. It's often possible to improve improvements, and it's a good way to introduce yourself to the community and make new friends. (To make this as easy as possible, we tag some issues and proposed changes as "Suitable for Newcomers" or "Help Wanted".)

Discuss, then edit. If you want to propose a large change, such as reorganizing or splitting an entire chapter, please file an issue that outlines your proposal and your reasoning and tag it with "Proposal". We encourage everyone to add comments to these issues so that the whole discussion of what and why is in the open and can be archived. If the proposal is accepted, the actual work may then be broken down into several smaller issues or changes that can be tackled independently.

D Code of Conduct

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socioeconomic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

D.1 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- using welcoming and inclusive language,
- being respectful of differing viewpoints and experiences,
- gracefully accepting constructive criticism,
- focusing on what is best for the community, and
- showing empathy towards other community members.

Examples of unacceptable behavior by participants include:

- the use of sexualized language or imagery and unwelcome sexual attention or advances,
- trolling, insulting/derogatory comments, and personal or political attacks,
- public or private harassment,
- publishing others' private information, such as a physical or electronic address, without explicit permission, and
- other conduct which could reasonably be considered inappropriate in a professional setting

D.2 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in

response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

D.3 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

D.4 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by emailing the project. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

D.5 Attribution

This Code of Conduct is adapted from the Contributor Covenant version 1.4.

E Glossary

Absolute beginner: Someone who has never encountered concepts or material before. The term is used in distinction to *false beginner*.

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.

Automaticity: The ability to do a task without concentrating on its low-level details.

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

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*.

Bloom's Taxonomy: A six-part hierarchical classification of understand whose levels are *knowledge*, *comprehension*, *application*, *analysis*, *synthesis*, and *evaluation* that has been widely adopted. See also *Fink's Taxonomy*.

Brand: The associations people have with a product's name or identity.

Calibrated peer review: Having students compare their reviews of sample work with an instructor's reviews before being allowed to review their peers' work.

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

Co-teaching: Teaching with another instructor in the classroom.

Cognitive apprenticeship: A theory of learning that emphasizes the process of a master passing on skills and insights situationally to an apprentice.

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 such as knitters, musicians, or programmers. See also *legitimate peripheral participation*.

Community representation: Using cultural capital to highlight students' social identities, histories, and community networks in learning activities.

Computational integration: Using computing to re-implement pre-existing cultural artifacts, e.g., creating variants of traditional designs using computer drawing tools.

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

Computational thinking: Thinking about problem-solving in ways inspired by programming (though the term is used in many other ways).

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 holds that knowledge is distributed, that learning is the process of navigating, growing, and pruning connections, and which emphasizes the social aspects of learning made possible by the Internet

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*.

Contributing student pedagogy: Having students produce artifacts to contribute to other students' learning.

Conversational programmer: Someone who needs to know enough about computing to have a meaningful conversation with a programmer, but isn't going to program themselves.

CS0: An introductory college-level course on computing aimed at non-majors with little or no prior experience of programming.

CS1: An introductory college-level computer science course, typically one semester long, that focuses on variables, loops, functions, and other basic mechanics.

CS2: A second college-level computer science course that typically introduces basic data structures such as stacks, queues, and dictionaries.

Deficit model: The idea that some groups are under-represented in computing (or some other field) because their members lack some attribute or quality.

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

Demonstration lesson: A staged lesson in which one teacher presents material to a class of actual students while other teachers observe in order to learn new teaching techniques.

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

Direct instruction: A teaching method centered around meticulous curriculum design delivered through prescribed script.

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

Ego depletion: The impairment of self control that occurs when it is exercised intensively or for long periods.

Elevator pitch: A short description of an idea, project, product, or person that can be delivered and understood in just a few seconds.

End-user programmer: Someone who does not consider themselves a programmer, but who nevertheless writes and debugs software, such as an artist creating complex macros for a drawing tool.

End-user teacher: By analogy with *end-user programmer*, someone who is teaching frequently, but whose primary occupation is not teaching, who has little or no background in pedagogy, and who may work outside institutional classrooms.

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.

Expertise reversal effect: The way in which instruction that is effective for novices becomes ineffective for competent practitioners or experts.

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

Extrinsic motivation: Being driven by external rewards such as payment or fear of punishment. See also *intrinsic motivation*.

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

False beginner: Someone who has studied a language before but is learning it again. False beginners start at the same point as true beginners (i.e., a pre-test will show the same proficiency) but can move much more quickly.

Far transfer: Transfer of learning or proficiency between widely-separated domains, e.g., improvement in math skills as a result of playing chess.

Fink's Taxonomy: A six-part non-hierarchical classification of understanding first proposed in [Fink2013] whose categories are *foundational knowledge*, *application*, *integration*, *human dimension*, *caring*, and *learning how to learn*. See also *Bloom's Taxonomy*.

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*.

Flipped classroom: One in which learners watch recorded lessons on their own time, while class time is used to work through problem sets and answer questions.

Flow: The feeling of being fully immersed in an activity; frequently associated with high productivity.

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*.

Free-range learner: Someone learning outside an institutional classrooms with required homework and mandated curriculum. (Those who use the term occasionally refer to students in classrooms as "battery-farmed learners", but we don't, because that would be rude.)

Functional programming: A style of programming in which data structures cannot be modified once they have been created, and in which functions that operate on other functions are widely used for abstraction.

Fuzz testing: A software testing technique based on generating and submitting random data.

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*.

Guided notes: Instructor-prepared notes that cue students to respond to key information in a lecture or discussion.

Hashing: Generating a condensed pseudo-random digital key from data; any specific input produces the same output, but different inputs are highly likely to produce different outputs.

Hypercorrection effect: The more strongly someone believed that their answer on a test was right, the more likely they are not to repeat the error once they discover that in fact they were wrong.

Implementation science: The study of how to translate research findings to everyday clinical practice.

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*.

Intrinsic motivation: Being driven by enjoyment of a task or the satisfaction of doing it for its own sake. See also *extrinsic motivation*.

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.

Learned helplessness: A situation in which people who are repeatedly subjected to negative feedback that they have no way to escape learn not to even try to escape when they could.

Learner persona: 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*.

Marketing: The craft of seeing things from other people's perspective, understanding their wants and needs, and finding ways to meet them

Mental model: A simplified representation of the key elements and relationships of some problem domain that is good enough to support problem solving.

Metacognition: Thinking about thinking.

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).

Near transfer: Transfer of learning or proficiency between closely-related domains, e.g., improvement in understanding of decimals as a result of doing exercises with fractions.

Notional machine: A general, simplified model of how a particular family of programs executes.

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.

Parsons 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*.

Personalized learning: Automatically tailoring lessons to meet the needs of individual students.

Plausible distractor: A wrong or less-than-best answer to a multiple-choice question that looks like it could be right. See also *diagnostic power*.

Positioning: What sets one brand apart from other, similar brands.

Preparatory privilege: The advantage of coming from a background that provides more preparation for a particular learning task than others.

Pull request: A set of proposed changes to a GitHub repository that can be reviewed, updated, and eventually merged.

Read-cover-retrieve: A study practice in which the learner covers up key facts or terms during a first pass through material, then checks their recall on a second pass.

Reflective practice: see *deliberate practice*.

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*.

Split-attention effect: The decrease in learning that occurs when learners must divide their attention between multiple concurrent presentations of the same information (e.g., captions and a voiceover).

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

Subgoal labelling: Giving names to the steps in a step-by-step description of a problem-solving process.

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.

Think-pair-share: A collaboration method in which each person thinks individually about a question or problem, then pairs with a partner to pool ideas, and then have one person from each pair present to the whole group.

Transfer-appropriate processing: The improvement in recall that occurs when practice uses activities similar to those used in testing.

Twitch coding: Having a group of people decide moment by moment or line by line what to add to a program next.

Working memory: see *short-term memory*.

F Meetings, Meetings, Meetings

Most people are really bad at meetings: they don't have an agenda going in, they don't take minutes, they waffle on or wander off into irrelevancies, they repeat what others have said or recite banalities simply so that they'll have said something, and they hold side conversations (which pretty much guarantees that the meeting will be a waste of time). Knowing how to run a meeting efficiently is a core skill for anyone who wants to get things done. (Knowing how to take part in someone else's meeting is just as important, but gets far less attention—as a colleague once said, everyone offers leadership training, nobody offers followership training.) The most important rules for making meetings efficient are not secret, but are rarely followed:

Decide if there actually needs to be a meeting. If the only purpose is to share information, have everyone send a brief email instead. Remember, you can read faster than anyone can speak: if someone has facts for the rest of the team to absorb, the most polite way to communicate them is to type them in.

Write an agenda. If nobody cares enough about the meeting to write a point-form list of what's supposed to be discussed, the meeting itself probably doesn't need to happen.

Include timings in the agenda. Agendas can also help you prevent early items stealing time from later ones if you include the time to be spent on each item in the agenda. Your first estimates with any new group will be wildly optimistic, so revise them upward for subsequent meetings. However, you shouldn't plan a second or third meeting because the first one ran over-time: instead, try to figure out why you're running over and fix the underlying problem.

Prioritize. Every meeting is a micro-project, so work should be prioritized in the same way that it is for other projects: things that will have high impact but take little time should be done first, and things that will take lots of time but have little impact should be skipped.

Make one person responsible for keeping things moving. One person should be tasked with keeping items to time, chiding people who are

having side conversations or checking email, and asking people who are talking too much to get to the point. This person should *not* do all the talking; in fact, whoever is in charge will talk less in a well-run meeting than most other participants.

Require politeness. No one gets to be rude, no one gets to ramble, and if someone goes off topic, it's the chair's job to say, "Let's discuss that elsewhere."

No technology (unless it's required for accessibility reasons). Insist that everyone put their phones, tablets, and laptops into politeness mode (i.e., closes them). If this is too stressful, let participants hang on to their electronic pacifiers but turn off the network so that they really *are* using them just to take notes or check the agenda.

No interruptions. Participants should raise a finger, put up a sticky note, or make one of the other gestures people make at high-priced auctions instead if they want to speak next. If the speaker doesn't notice you, the person in charge ought to.

Record minutes. Someone other than the chair should take point-form notes about the most important pieces of information that were shared, and about every decision that was made or every task that was assigned to someone.

Take notes. While other people are talking, participants should take notes of questions they want to ask or points they want to make. (You'll be surprised how smart it makes you look when it's your turn to speak.)

End early. If your meeting is scheduled for 10:00-11:00, you should aim to end at 10:55 to give people time to get where they need to go next.

As soon as the meeting is over, the minutes should be circulated (e.g., emailed to everyone or posted to a wiki):

People who weren't at the meeting can keep track of what's going on. You and your fellow students all have to juggle assignments from several other courses while doing this project, which means that sometimes you won't be able to make it to team meetings. A wiki page, email message, or blog entry is a much more efficient way to catch up after a missed meeting or two than asking a team mate, "Hey, what did I miss?"

Everyone can check what was actually said or promised. More than once, I've looked over the minutes of a meeting I was in and thought, "Did I say that?" or, "Wait a minute, I didn't promise to have it ready then!" Accidentally or not, people will often remember things differently; writing it down gives team members a chance to correct mistaken or malicious interpretations, which can save a lot of anguish later on.

People can be held accountable at subsequent meetings. There's no point making lists of questions and action items if you don't follow up on them later. If you're using a ticketing system, the best thing to do is to create a ticket for each new question or task right after the meeting, and

update those that are being carried forward. That way, your agenda for the next meeting can start by rattling through a list of tickets.

[Brow2007] and [Broo2016] have lots of good advice on running meetings, and if you want to "learn, then do", an hour of training on chairing meetings is the most effective place to start.

Sticky Notes and Interruption Bingo

Some people are so used to the sound of their own voice that they will insist on talking half the time no matter how many other people are in the room. One way to combat this is to give everyone three sticky notes at the start of the meeting. Every time they speak, they have to take down one sticky note. When they're out of notes, they aren't allowed to speak until everyone has used at least one, at which point everyone gets all of their sticky notes back. This ensures that nobody talks more than three times as often as the quietest person in the meeting, and completely changes the dynamics of most groups: people who have given up trying to be heard because they always get trampled suddenly have space to contribute, and the overly-frequent speakers quickly realize just how unfair they have been.

Another useful technique is called interruption bingo. Draw a grid, and label the rows and columns with the participants' names. Each time someone interrupts someone else, add a tally mark to the appropriate cell. Halfway through the meeting, take a moment to look at the results. In most cases, you will see that one or two people are doing all of the interrupting, often without being aware of it. After that, saying, "All right, I'm adding another tally to the bingo card," is often enough to get them to throttle back. (Note that this technique is intended to manage interruptions, not speaking time. It may be completely appropriate for people with more knowledge of a subject to speak about it more often in a meeting, but it is almost never appropriate to repeatedly cut people off.)

F.1 Online Meetings

Chelsea Troy's discussion of why online meetings are often frustrating and unproductive makes an important point: in most online meetings, the first person to speak during a pause gets the floor. The result? "If you have something you want to say, you have to stop listening to the person currently speaking and instead focus on when they're gonna pause or finish so you can leap into that nanosecond of silence and be the first to utter something. The format. . . encourages participants who want to contribute to say more and listen less."

The solution is to run a text chat beside the video conference where people can signal that they want to speak, and have the moderator select people from the waiting list. If the meeting is large or argumentative, this

can be reinforced by having everyone mute themselves, and only allowing the moderator to unmute people.

F.2 The Post Mortem

Every project should end with a post mortem in which you reflect on what you just accomplished and what you could o better next time. Its aim is *not* to point the finger of shame at individuals, although if that has to happen, the post mortem is the best place for it.

A post mortem is run like any other meeting, but with a few additional guidelines [Derb2006]:

Get a moderator who wasn't part of the project and doesn't have a stake in it. Otherwise, the meeting will either go in circles, or focus on only a subset of important topics. In the case of student projects, this moderator might be the course instructor, or a TA.

Set aside an hour, and only an hour. In my experience, nothing useful is said in the first ten minutes of anyone's first post mortem, since people are naturally a bit shy about praising or damning their own work. Equally, nothing useful is said after the first hour: if you're still talking, it's probably because one or two people have a *lot* they want to get off their chests.

Require attendance. Everyone who was part of the project ought to be in the room for the post mortem. This is more important than you might think: the people who have the most to learn from the post mortem are often least likely to show up if the meeting is optional.

Make two lists. When I'm moderating, I put the headings "Do Again" and "Do Differently" on the board, then do a lap around the room and ask every person to give me one item (that hasn't already been mentioned) for each list.

Comment on actions, rather than individuals. By the time the project is done, some people simply won't be able to stand one another. Don't let this sidetrack the meeting: if someone has a specific complaint about another member of the team, require him to criticize a particular event or decision. "He had a bad attitude" does *not* help anyone improve their game.

Once everyone's thoughts are out in the open, organize them somehow so that you can make specific recommendations about what to do next time. This list is one of the two major goals of the post mortem (the other being to give people a chance to be heard).

G A Little Bit of Theory

One of the exercises in educational research is deciding what we 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 my teaching. The first is cognitivism, which focuses 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 is situated learning, which focuses on bringing people into a community, and recognizes that teaching and learning are always rooted in who we are and who we aspire to be. We will discuss it in more detail in Chapter 13.

The Learning Theories website and [Wibu2016] have good summaries of these and other perspectives. 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 holds that knowledge is distributed, that learning is the process of navigating, growing, and pruning connections, and which emphasizes the social aspects of learning made possible by the Internet). It would help if their names were less similar, but setting that aside, none of them can tell us how to teach on their own because 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.

The first approach is called “phonics”, and the second, “whole language”. The whole language approach may seem upside down, but more than a billion people have learned to read and write Chinese and similar ideogrammatic languages in exactly this way. The only way to tell which approach works best for most children, most of the time, is to try them both out. These studies have to be done carefully, because so many other variables can have an impact on rules. For example, the teacher’s enthusiasm for the teaching method may matter more than the method itself, since children will model their teacher’s excitement for a subject. (With all of that taken into account, phonics does seem to be better than other approaches [Foor1998].)

As frustrating as the maybes and howevers in education research are, this kind of painstaking work is essential to dispel myths that can get in the way of better teaching. One well-known myth is that people are visual, auditory, or kinesthetic learners, and that teaching is more effective when lessons are designed according to whether they like to see things, hear things, or do things. This idea is easy to understand, but as [DeBr2015] explains, it is almost certainly false. Unfortunately, that hasn’t stopped companies from marketing products based on it to parents, school boards, and the general public.

Similarly, 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. Just as we need to clear away our learners’ misconceptions in order to help them learn, we need to clear away our own about teaching if we are to teach more effectively.

G.1 Notional Machines

The term computational thinking is bandied about a lot, in part because people can agree it’s important while meaning very different things by it. I find it more useful to think in terms of getting learners to understand a notional machine. The term was introduced in [DuBo1986], and means abstraction of the structure and behavior of a computational device. According to [Sorv2013], a notional machine:

- is an idealized abstraction of computer hardware and other aspects of the runtime environment of programs;
- serves the purpose of understanding what happens during program execution;
- is associated with one or more programming paradigms or languages, and possibly with a particular programming environment;
- enables the semantics of program code written in those paradigms or languages (or subsets thereof) to be described;

- gives a particular perspective to the execution of programs; and
- correctly reflects what programs do when executed.

For example, my notional machine for Python is:

1. Running programs live in memory, which is divided between a call stack and a heap.
2. Memory for data is always allocated from the heap.
3. Every piece of data is stored in a two-part structure: the first part says what type the data is, and the second part is the actual value.
4. Atomic data like Booleans, numbers, and character strings are stored directly in the second part. These values are never modified after they are created.
5. The scaffolding for collections like lists and sets are also stored in the second part, but they store references to other data rather than storing those values directly. The scaffolding may be modified after it is created, e.g., a list may be extended or new key/value pairs may be added to a dictionary.
6. When code is loaded into memory, Python parses it and converts it to a sequence of instructions that are stored like any other data. (This is why it's possible to alias functions and pass them as parameters.)
7. When code is executed, Python steps through the instructions, doing what each tells it to in turn.
8. Some instructions make Python read data, operate on it, and create new data.
9. Other instructions make Python jump to other instructions instead of executing the next one in sequence; this is how conditionals and loops work.
10. Yet another instruction tells Python to call a function, which means temporarily switching from one blob of instructions to another.
11. When a function is called, a new stack frame is pushed on the call stack.
12. Each stack frame stores variables' names and references to data. (Function parameters are just another kind of variable.)
13. When a variable is used, Python looks for it in the top stack frame. If it isn't there, it looks in the bottom (global) frame.
14. When the function finishes, Python erases its stack frame and switches from its blob of instructions back to the blob that called it. If there isn't a "beforehand", the program has finished.

I don't try to explain all of this at once, but I draw on this mental model over and over again as I draw pictures, trace execution, and so on. After about 25 hours of class and 100 hours of work on their own time, I expect adult learners to be able to understand most of it.

H Lesson Design Template

Designing a good course is as hard as designing good software. To help you, this appendix summarizes a process based on evidence-based teaching practices:

- It lays out a step-by-step progression to help you figure out what to think about in what order.
- It provides spaced deliverables so you can re-scope or redirect effort without too many unpleasant surprises.
- Everything from Step 2 onward goes into your final course, so there is no wasted effort.
- Writing sample exercises early lets you check that everything you want your students to do actually works.

This backward design process was developed independently by [Wigg2005, Bigg2011, Fink2013]. We have slimmed it down by removing steps related to meeting curriculum guidelines and other institutional requirements.

Note that the steps are described in order of increasing detail, but the process itself is always iterative. You will frequently go back to revise earlier work as you learn something from your answer to a later question or realize that your initial plan isn't going to play out the way you first thought.

H.1 Step 1: Brainstorming

The first step is to throw together some rough ideas so that you and your colleagues can make sure your thoughts about the course are aligned. To do this, write some point-form answers to three or four of the questions listed below. You aren't expected to answer all of them, and you may pose and answer others if you think it's helpful, but you should always include a couple of answers to the first.

1. What problem(s) will student learn how to solve?
2. What concepts and techniques will students learn?
3. What technologies, packages, or functions will students use?

4. What terms or jargon will you define?
5. What analogies will you use to explain concepts?
6. What heuristics will help students understand things?
7. What mistakes or misconceptions do you expect?
8. What datasets will you use?

You may not need to answer every question for every course, and you will often have questions or issues we haven't suggested, but couple of hours of thinking at this stage can save days of rework later on.

Deliverable: a rough scope for the course that you have agreed with your colleagues.

H.2 Step 2: Who Is This Course For?

"Beginner" and "expert" mean different things to different people, and many factors besides pre-existing knowledge influence who a course is suitable for. The second step in designing a course is therefore to figure out who your audience is. To do this, you should either create some learner personas (Section 6.1), or (preferably) reference ones that you and your colleagues have drawn up together.

After you are done brainstorming, you should go through these personas and decide which of them your course is intended for, and how it will help them. While doing this, you should make some notes about what specific prerequisite skills or knowledge you expect students to have above and beyond what's in the persona.

Deliverable: brief summaries of who your course will help and how.

H.3 Step 3: What Will Learners Do Along the Way?

The best way to make the goals in Step 1 firmer is to write full descriptions of a couple of exercises that students will be able to do toward the end of the course. Writing exercises early is directly analogous to test-driven development: rather than working forward from a (probably ambiguous) set of learning objectives, designers work backward from concrete examples of where their students are going. Doing this also helps uncover technical requirements that might otherwise not be found until uncomfortably late in the lesson development process.

To complement the full exercise descriptions, you should also write brief point-form descriptions of one or two exercises per lecture hour to show how quickly you expect learners to progress. (Again, these serve as a good reality check on how much you're assuming, and help uncover technical requirements.) One way to create these "extra" exercises is to make a point-form list of the skills needed to solve the major exercises and create an exercise that targets each.

Deliverable: 1–2 fully explained exercises that use the skills the student is to learn, plus half a dozen point-form exercise outlines.

Note: be sure to include solutions with example code so that you can check that your software can do everything you need.

H.4 Step 4: How Are Concepts Connected?

In this stage, you put the exercises in a logical order then derive a point-form course outline for the entire course from them. This is also when you will consolidate the datasets your formative assessments have used.

Deliverable: a course outline.

Notes:

- The final outline should be at the lecture and formative assessment level, e.g., one major bullet point for each hour of work with 3–4 minor bullet points for the episodes in that hour.
- It's common to change assessments in this stage so that they can build on each other.
- You are likely to discover things you forgot to list earlier during this stage, so don't be surprised if you have to double back a few times.

H.5 Step 5: Course Overview

You can now write a course overview consisting of:

- a one-paragraph description (i.e., a sales pitch to students)
- half a dozen learning objectives
- a summary of prerequisites

Doing this earlier often wastes effort, since material is usually added, cut, or moved around in earlier steps.

Deliverable: course description, learning objectives, and prerequisites.

H.6 Reminder

As noted at the start, this process is described as a sequence, but in practice you will loop back repeatedly as each stage informs you of something you overlooked.

I Checklists for Events

[Gawa2007] 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 [Avel2013, Urba2014], 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.

I.1 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 instructors.
5. Arrange space, including breakout rooms if needed.
6. Choose dates. If it is in person, book travel.
7. Get names and email addresses of attendees from host(s).
8. Make sure they are added to the registration system.

I.2 Setting Up

1. Set up a web page with details on the workshop, including date, location, and a list of what participants need 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, and a description of any prerequisite tasks.

I.3 At the Start of the Event

1. Remind everyone of the code of conduct.
2. Collect attendance.
3. Distribute sticky notes.
4. Collect any relevant online account IDs.

I.4 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.

I.5 Travel Kit

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

- sticky notes
- cough drops
- comfortable shoes
- a small notepad
- 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 the notes, or a tablet or other device
- an insulated cup for tea/coffee
- spare glasses/contacts
- a notebook and pen
- a portable WiFi hub (in case the room's network isn't working)
- extra whiteboard markers
- a laser pointer
- a packet of wet wipes (because spills happen)
- USB drives with installers for various operating systems
- running shoes, a bathing suit, a yoga mat, or whatever else you exercise in or with

J Presentation Rubric

This rubric is designed to assess 5–10 minute recordings of people teaching with slides, live coding, or a mix of both. You can use it as a starting point for creating a rubric of your own. Rate each item as "Yes", "Iffy", "No", or "Not Applicable".

- Opening
  - Exists (use N/A for other responses if not)
  - Good length (10–30 seconds)
  - Introduces self
  - Introduces topics to be covered
  - Describes prerequisites
- Content
  - Clear goal/narrative arc
  - Inclusive language
  - Authentic tasks/examples
  - Teaches best practices/uses idiomatic code
  - Steers a path between the Scylla of jargon and the Charybdis of over-simplification
- Delivery
  - Clear, intelligible voice (use "Iffy" or "No" for strong accent)
  - Rhythm: not too fast or too slow, no long pauses or self-interruption, not obviously reading from a script
  - Self-assured: does not stray into the icky tarpit of uncertainty or the dungheap of condescension
- Slides
  - Exist (use N/A for other responses if not)
  - Slides and speech complement one another (dual coding)
  - Readable fonts and colors/no overwhelming slabs of text
  - Frequent change (something happens on screen at least every 30 seconds)

  - Good use of graphics
- Live Coding
  - Used (use N/A for other responses if not)
  - Code and speech complement one another (i.e., instructor doesn't just read code aloud)
  - Readable fonts and colors/right amount of code on the screen at a time
  - Proficient use of tools
  - Highlights key features of code
  - Dissects errors
- Closing
  - Exists (use N/A for other responses if it doesn't)
  - Good length (10–30 seconds)
  - Summarizes key points
  - Outlines next steps
- Overall
  - Points clearly connected/logical flow
  - Make the topic interesting (i.e., not boring)
  - Knowledgeable

K Teamwork Rubric

This rubric is designed to assess individual performance within a team. You can use it as a starting point for creating a rubric of your own. Rate each item as “Yes”, “Iffy”, “No”, or “Not Applicable”.

- Communication
  - Listens attentively to others without interrupting.
  - Clarifies with others have said to ensure understanding.
  - Articulates ideas clearly and concisely.
  - Gives good reasons for ideas.
  - Wins support from others.
- Decision Making
  - Analyzes problems from different points of view.
  - Applies logic in solving problems.
  - Offers solutions based on facts rather than “gut feel” or intuition.
  - Solicits new ideas from others.
  - Generates new ideas.
  - Accepts change.
- Collaboration
  - Acknowledges issues that the team needs to confront and resolve.
  - Encourages ideas and opinions even when they differ from his/her own.
  - Works toward solutions and compromises that are acceptable to all involved.
  - Shares credit for success with others.
  - Encourages participation among all participants.
  - Accepts criticism openly and non-defensively.
  - Cooperates with others.
- Self-Management
  - Monitors progress to ensure that goals are met.
  - Puts top priority on getting results.

– Defines task priorities for work sessions.
– Encourages others to express their views even when they are contrary.
– Stays focused on the task during meetings.
– Uses meeting time efficiently.
– Suggests ways to proceed during work sessions.

L Pre-Assessment Questionnaire

This questionnaire is designed to help teachers gauge the prior knowledge of learners in an introductory JavaScript programming workshop. You can use it as a starting point for creating a rubric of your own.

- Which of these best describes your previous experience with programming in general?
  - I have none.
  - I have written a few lines now and again.
  - I have written programs for my own use that are a couple of pages long.
  - I have written and maintained larger pieces of software.
- Which of these best describes your previous experience with programming in JavaScript?
  - I have none.
  - I have written a few lines now and again.
  - I have written programs for my own use that are a couple of pages long.
  - I have written and maintained larger pieces of software.
- Which of these best describes how easily you could write JavaScript to find the largest number in a list?
  - I wouldn't know where to start.
  - I could struggle through by trial and error with a lot of web searches.
  - I could do it quickly with little or no use of external help.
- Which of these best describes how easily you could write JavaScript to capitalize all of the titles in a web page?
  - I wouldn't know where to start.
  - I could struggle through by trial and error with a lot of web searches.
  - I could do it quickly with little or no use of external help.
- Why do you want to take this training course?

M Design Notes

This design follows the backward design process described in Chapter 6.

M.1 Brainstorming

These questions and answers provide a rough scope for the material.

1. What problems will learners learn how to solve?
    1. How people learn and what that tells us about how best to teach them (educational psychology, cognitive load, study skills).
    2. How to design and deliver instruction in computing skills (backward curriculum design, some pedagogical content knowledge for computing).
    3. How to deliver lessons (teaching as a performance art, live coding, motivation and demotivation, and automation).
    4. How to grow a teaching community (community organization and marketing).
2. What is out of scope?
    1. How to teach children or people with special learning needs. Much of what's in this material applies to those learners, but they have different or extra needs.
    2. How to rigorously assess the impact of training. Informal self-assessment will be included, but we will not try to explain how to do publishable scientific research in education.
    3. How to design and deliver entire degree programs and other extended curriculum. Again, much of what's in this material applies, but the extra needs of large-scale curriculum design is out of scope.
3. What concepts and techniques will learners encounter?
    1. 7 ± 2 and chunking.
    2. Authentic tasks with tangible artifacts.

3. Bloom's Taxonomy, Fink's Taxonomy, and Piaget's development stage theory.
4. Branding.
5. Cognitive development from novice to competent to expert.
6. Cognitive load.
7. Collaborative lesson development.
8. Concept mapping.
9. Designing assessments with diagnostic power.
10. Dunning-Kruger effect.
11. Expert blind spot.
12. Externalized cognition.
13. Fixed vs. growth mindset (and critiques of it).
14. Formative vs. summative assessment.
15. Governance models of community organizations.
16. Inquiry-based learning (and critiques of it).
17. Intrinsic vs. extrinsic motivation.
18. *Jugyokenkyu* (lesson study).
19. Learner personas.
20. Legitimate peripheral participation in a community of practice.
21. Live coding (teaching as a performance art).
22. Pedagogical content knowledge (PCK) and technological pedagogical and content knowledge (TPACK).
23. Peer instruction.
24. Reflective (deliberate) practice.
25. Backward design.
26. Stereotype threat (and critiques of it).
27. Working memory vs. persistent memory.

4. What mistakes or misconceptions will they have?

1. Children and adults learn the same way.
2. Computing education should be for and about computer science.
3. Programming skill is innate.
4. Student evaluations of courses are indicative of learning outcomes.
5. Teaching ability is innate.
6. The best way to teach is to throw people in at the deep end.
7. The best way to teach is to use "real" tools right from the start.
8. Visual-auditory-kinesthetic (VAK) learning styles are real.
9. Women just don't like programming or innately have less aptitude.

10. Getting a (better) job is the main reason someone should learn how to program.

5. In what contexts will this material be used?

    1. Primary: an intensive weekend workshop for people in tech who want to volunteer with grassroots get-into-coding initiatives.
    2. Secondary: self-study or guided study by such people.
    3. Secondary: a one-semester undergraduate course for computer science majors interested in education.

M.2 Intended Audience

These learner personas clarify what readers are interested in and what can be assumed about their prior knowledge.

Emily trained as a librarian, and now works as a web designer and project manager in a small consulting company. In her spare time, she helps run web design classes for women entering tech as a second career. She is now recruiting colleagues to run more classes in her area using the lessons that she has created, and wants to know how to grow a volunteer teaching organization.

Moshe is a professional programmer with two teenage children whose school doesn't offer programming classes. He has volunteered to run a monthly after-school programming club, and while he frequently gives presentations to colleagues, he has no experience designing lessons. He wants to learn how to build effective lessons in collaboration with others, and is interested in turning his lessons into a self-paced online course.

Samira is an undergraduate in robotics who is thinking about becoming a full-time teacher after she graduates. She wants to help teach weekend workshops for undergraduate women, but has never taught an entire class before, and feels uncomfortable teaching things that she's not an expert in. She wants to learn more about education in general in order to decide if it's for her.

Gene is a professor of computer science whose research area is operating systems. They have been teaching undergraduate classes for six years, and increasingly believe that there has to be a better way. The only training available through their university's teaching and learning center relates to posting assignments and grades in the learning management system, so they want to find out what else they ought to be asking for.

Common elements:

- A variety of technical backgrounds and skills.
- May or may not have some teaching experience.

- No formal training in teaching, lesson design, or community organization.
- More likely to teach in free-range settings often than in institutional classrooms with required homework, final exams, and externally-mandated curriculum.
- Focused on teenagers and adults rather children.
- Limited time and resources (either because they are volunteers, or because their institution considers teaching a secondary responsibility).

Learning contexts:

Emily will take part in a weekly online reading group with her volunteers.

Moshe will cover part of this book in a two-day weekend workshop and study the rest on his own.

Samira will use this book in a one-semester undergraduate course with assignments, a project, and a final exam.

Gene will read the book on their own in their office or while commuting, wishing all the while that universities did more to support high-quality teaching.

M.3 Exercises

These formative exercises summarize what learners will be able to do with their new knowledge. The finished book will include others as well.

1. Create multiple choice questions whose incorrect answers have diagnostic power.
2. Give feedback on a recorded teaching episode and compare points with expert feedback.
3. Create a Parsons Problem.
4. Create a short debugging exercise.
5. Create a short execution tracing exercise.
6. Explain their personal motivation for teaching.
7. Explain their community of practice's aims and conventions.
8. Explain the difference between an oversight board and a governance board.
9. Design an hour-long lesson using backward design.
10. Describe the pros and cons of standardized testing.
11. Create learner personas for their intended students.
12. Write and critique learning objectives for an hour-long lesson.
13. Write and critique a short value proposition for a class they intend to offer.
14. Teach a short lesson using live coding and critique a recording of it.
15. Create a short video lesson and critique it.
16. Construct a short series of faded examples that illustrate a problem-solving pattern in programming.

17. Describe ways in which they differ from their intended learners.
18. Create and critique an elevator pitch for a course they intend to teach.
19. Write a "cold call" email to solicit support for what they intend to teach.
20. Create a concept map for a topic they intend to teach.
21. Explain six strategies students can use to learn more effectively.
22. Analyze and critique the accessibility of a short online lesson.
23. Analyze and critique the inclusivity of a short lesson.
24. Create and critique a non-programming exercise to use in a programming class.
25. Describe the relative merits of block-based and text-based environments for introductory programming classes for adults.
26. Create a one-to-one matching exercise for use in a class they intend to teach.
27. Create a diagram labelling exercise for use in a class they intend to teach.
28. Write and submit an improvement or extension to an existing lesson and review a peer's submission.
29. Describe the pros and cons of collaborative note-taking.
30. Describe the pros and cons of gamification in online learning.
31. Create and critique a short questionnaire for assessing learners' prior knowledge.
32. Conduct a demonstration lesson using peer instruction.
33. Describe ways in which computing is unwelcoming to or unaccepting of people from diverse backgrounds.
34. Demonstrate several ways to ensure that an instructor's attention is fairly distributed through a class.
35. Describe the pros and cons of in-person, automated, and hybrid teaching strategies.
36. Write and critique automated tests for a short programming exercise.

M.4 Outline

Each major section can be covered in detail in 2–3 weeks in a conventional classroom format, or in less detail in one full day in an intensive workshop format.

1. Introduction
2. Learning
    1. Building Mental Models
    2. Expertise and Memory
    3. Cognitive Load
    4. Effective Learning
3. Designing

    1. A Lesson Design Process
    2. Pedagogical Content Knowledge
4. Delivering
    1. Teaching as a Performance Art
    2. Live Coding
    3. Motivation and Demotivation
    4. Automation
    5. Hybrid Models
5. Organizing
    1. Awareness
    2. Operations
    3. Building Community

M.5 Course Overview

Brief Description

Teaching isn't magic: good teachers are simply people who have learned how to design lessons to achieve concrete goals, how to get and use feedback from learners, and how to work well with other teachers. This book will show you how to do these things and more, and will introduce you to some of the research that explains why some things work and some things don't. It is primarily intended for people in tech with no formal training in teaching who want to help adults learn how to create web sites, write programs, and analyze data, but the ideas apply equally well to other groups in other settings.

Learning Objectives

Learners will be able to...

1. Explain the cognitive changes that occur as people go from novice to competent to expert and how best to teach each group.
2. Explain how to design, construct, and maintain lessons in a systematic, collaborative way.
3. Design exercises to help correct key misconceptions that learners have about computing.
4. Summarize key elements of pedagogical content knowledge related to computing and other technical skills.
5. Compare and contrast teaching with other performance arts and participate in structured critiques of live teaching.

6. Compare and contrast interactive teaching, automated teaching, and hybrid models.
7. Describe factors that motivate or demotivate adult learners and how to take those into account when teaching.
8. Describe ways in which members of different groups are made to feel unwelcome or excluded in computing and what can be done to make computing more inclusive.
9. Explain the purpose and value of their teaching and of their community of practice.
10. Be a productive member of a community of teaching practice.

Prerequisites

Some exercises will assume a small amount of programming knowledge: readers should know how to loop over the elements of a list, how to take action using an if-else statement, and how to write and call a simple function.