How to Teach Programming (And Other Things)

what everyone in tech ought to know about teaching and learning

Edited by Greg Wilson

ISBN 978-1-365-98428-0

90000

9 781365 984280

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."

Contents

1 Introduction

Hundreds of grassroots groups have sprung up in the past ten years to teach programming, web site design, robotics, and dozens of other things. They exist so that people don't have to teach themselves how to program, but ironically, most of their founders are teaching themselves how to teach.

It doesn't have to be that way. Like good programming practices, good teaching practices can and should be taught and learned. Using them won't automatically make people great teachers, but *will* make them better teachers.

This book is a brief introduction to ideas and techniques I have found useful when teaching programming to free-range learners (i.e., those outside institutional classrooms with required homework and mandated curriculum). It covers:

- how learning works;
- how to design and maintain lessons;
- how to deliver those lessons; and
- how to grow a community of practice around teaching.

All of the material can be freely distributed and re-used under the Creative Commons - Attribution 4.0 license[1]: please see http://third-bit.com/teaching/ to download digital versions, to purchase a printed copy, or to make a contribution.

1.1 Who You Are

Section 6.1 explains how to figure out who your learners are. The four I had in mind when compiling this book are:

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 an after-school program, and while he frequently gives presentations to

[1] https://creativecommons.org/licenses/by/4.0/

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*. Each will use this material differently:

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

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

Samira will use this material 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 than pay lip service to high-quality teaching.

1.2 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[2] 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 hanging drywall, not building the Channel Tunnel.) In the years that followed, I discovered resources like Mark Guzdial's blog[3] 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.

[2] http://carpentries.org

[3] http://computinged.wordpress.com

I started using these ideas in Software Carpentry[4] 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[5] 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.

1.3 Have a Code of Conduct

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

A Code of Conduct cannot stop people from being offensive, any more than laws against theft stop people from stealing. What it *can* do is make expectations and consequences clear. More importantly, it tells people that there *are* rules, and that they can expect a friendly learning experience.

People rarely violate a Code of Conduct in person; they are more likely to online, where they feel less inhibited. If someone challenges you about it, remind them that a Code of Conduct is *not* an infringement on free speech. People have a right to say what they think, but that doesn't mean they have a right to say it wherever and whenever they want. If they want to be disparaging about someone else, they can go and find their own space in which to do it.

1.4 Acknowledgments

This book would not exist without the hard work and feedback of Erin Becker, Azalee Bostroem, Neil Brown, Francis Castro, Warren Code, Ben Cotton, Richie Cotton, Karen Cranston, Katie Cunningham, Natasha Danas, Matt Davis, Neal Davis, Mark Degani, Michael Deutsch, Brian Dillingham, Kathi Fisler, Auriel Fournier, Bob Freeman, Mark Guzdial, Rayna Harris, Ahmed Hasan, Ian Hawke, Felienne Hermans, Kate Hertweck, Toby Hodges, Dan Katz, Christina Koch, Shriram Krishnamurthi, Colleen Lewis, Sue McClatchy, Ian Milligan, Lex Nederbragt, Aleksandra Nenadic, Jeramia Ory, Elizabeth Patitsas, Aleksandra Pawlik, Sorawee Porncharoenwase, Emily Porta, Alex Pounds, Thomas Price, Danielle Quinn, 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, Allegra Via, Anelda van der Walt, Stéfan van der Walt, Belinda Weaver, Hadley Wickham, Jason Williams, John Wrenn, and Andromeda Yelton. I am grateful to them, and to everyone who has used this material over the years.

Breaking the Law

Most of the research reported in this book was publicly funded. Despite that, a lot of it is locked away behind paywalls. At a guess, I broke the

[4]http://carpentries.org
[5]https://carpentries.github.io/instructor-training/

law roughly 250 times to download papers from sites like Sci-Hub. I hope the day is coming soon 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 pre-print servers.*

1.5 Challenges

Each chapter ends with a set of challenge exercises, each of which includes a suggested format (individual, pairs, think-pair-share, small groups, or whole class) and an indication of how long it usually takes. Most can be delivered in other formats, and learners can take longer with most of them if time permits.

The three challenges in this chapter are good pre-assessment questions: 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 minutes)

Write brief answers to the following questions. Put your answers in the shared notes and comment on what others have written, or compare your answers to those of the person beside you.

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 minutes)

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 minutes)

Write brief answers to the following questions. 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?

Part I

Learning

2 Building Mental Models

Objectives

- Explain the cognitive differences between novices and competent practitioners in terms of mental models, and the implications of these differences for teaching.
- Define and differentiate formative and summative assessment.
- Construct multiple-choice questions with plausible distractors that have diagnostic power.

The first task in teaching is to figure out who your learners are and how best to help them. Our approach is based on the 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 variable, and there will always be outliers.) For our purposes, we can simplify that 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. For example, a driver's mental model of how a car works doesn't have to include all of the complexities that a mechanical engineer would be concerned with.

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. 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[1], 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 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 was 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[2] 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 learners 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*

[1] https://en.wikipedia.org/wiki/Not_even_wrong
[2] http://swcarpentry.github.io/shell-novice/

after re-reading them several times, I don't know how they manage to do it.

2.1 Formative Assessment

One of the challenges in building a mental model is to clear away things that *don't* belong. As Mark Twain said, "It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so." Broadly speaking, 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 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 *while they are teaching*. 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 has taught and is ready to move on. One example is a driving exam, which tells the rest of society whether someone is safe behind the wheel.

When the cook tastes the soup, that's formative. When the guests taste the soup, that's summative.
— Michael Scriven, as quoted by Debra Dirksen.

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, these can reveal much more than just tell 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?
a) 52 b) 42 c) 412 d) 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. "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[3] or Stack Overflow[4] to see what people learning the subject elsewhere are confused by.

Teachers should use MCQs or some other kind of formative assessment every 10–15 minutes in order to make sure that the class is 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. (Note that this isn't an attentional limit: [Wils2007] found little support for the often-repeated claim that students can only pay attention for 10–15 minutes.)

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 at. 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

[3]http://www.quora.com
[4]http://stackoverflow.com

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], and more recently, [Hamo2017] reported early work on 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.

MCQs aren't the only kind of formative assessment. Short-answer questions are a viable alternative: if answers are 2–5 words long, there are few enough plausible answers to make scalable assessment possible [Mill2016a]. Whatever you choose, 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.

Humor

Teachers will often put supposedly-silly answers like "a fish!" on MCQs, particularly ones intended for younger learners. However, they don't provide any insight into learners' misconceptions, and most learners don't actually find them funny.

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 Challenges

Your Mental Models (pairs/5 minutes)

What is one mental model you use to understand your work? Write a few sentences describing it, and give feedback on a partner's.

Symptoms of Being a Novice (whole class/5 minutes)

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 minutes)

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., 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 minutes)

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

The correct answer is that the level stays the same: the ice displaces its own weight in water, so it exactly fills the "hole" it has made when it melts. Figuring 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 minutes)

The model of skill development described at the start of this chapter is sometimes called the Dreyfus model[5]. Another commonly-used progression is the four stages of competence[6]:

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 minutes)

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

[5] https://en.wikipedia.org/wiki/Dreyfus_model_of_skill_acquisition

[6] https://en.wikipedia.org/wiki/Four_stages_of_competence

3 Expertise and Memory

Objectives

- Define expertise and explain how it works using a graph metaphor for cognition.
- Explain the difference between repetition and deliberate practice.
- Define and construct concept maps, and explain the benefits of externalizing cognition.
- Differentiate long-term and short-term memory, describe the capacity limits of the latter, and explain the impact of these limits on teaching.

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.

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 graph in which facts are nodes and relationships are arcs. (This is *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.

- Experts are frequently so familiar with their subject that they can no longer imagine what it's like to *not* see the world that way. As a result, they are often less good at teaching the subject than people with less expertise who still remember what it's like to have to learn the things. This phenomenon is called expert blind spot, and while it can be overcome with training, it's part of why there is no correlation between how good someone is at doing research in an area and how good they are at teaching it [Mars2002].
- Densely-connected knowledge graphs are also the basis for experts' fluid representations, i.e., their ability to switch back and forth between different views of a problem [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.
- Finally, this metaphor also explains why experts are better at diagnosis than competent practitioners: more linkages between facts makes it easier to reason backward from symptoms to causes. (And this in turn is why asking programmers to debug during job interviews gives a more accurate impression of their ability than asking them to program.)

The J Word

Experts often betray their blind spot by using the word "just" in explanations, as in, "Oh, it's easy, you just fire up a new virtual machine and then you just install these four patches to Ubuntu and then you just re-write your entire program in a pure functional language." As we discuss 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.

The graph model of knowledge 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.

3.1 Repetition vs. Deliberate Practice

The idea that ten thousand hours of practice will make someone an expert in some field is widely quoted, but reality is more complex. Doing exactly the same thing over and over again is much more likely to solidify bad habits than 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 write an essay about character development in *The Catcher in the Rye* and get feedback on their critique from a teacher.

Give feedback to themselves. At some point, they start critiquing their own work in real time (or nearly so) using the critical 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.

A meta-study conducted in 2014 [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." One explanation for this variation is that deliberate practice works best when the rules for evaluating success are very stable, but is less effective when there are more factors at play (i.e., when it's harder to connect cause to effect).

3.2 Concept Maps

Our tool of choice to represent a knowledge 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, one person's fact may be another person's connection, but one of the benefits of concept mapping is that it makes those differences explicit. The figure below reproduces a concept map taken from the IHMC CMap site[1] showing why the Earth has seasons.

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 first part of the concept map below, but they are only half the story. The expanded concept map shows the *relationships* between those things, which are as important for understanding as the concepts themselves.

Concept maps can be used in many ways:

1. Concept maps aid lesson design by 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.)

2. They aid 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.

[1] https://cmap.ihmc.us/

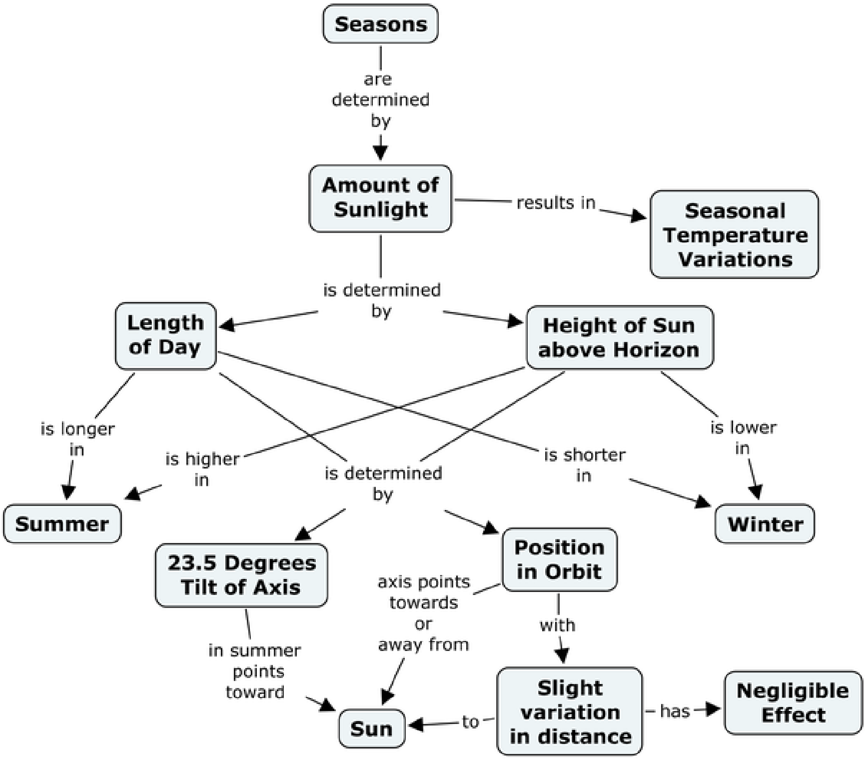


Figure 3.1: Concept Map for Seasons

3. Concept maps also aid communication with learners. While it's possible to give learners a pre-drawn map at the start of a lesson for them to annotate, it's better to draw it piece by piece while teaching to reinforce the ties between what's in the map and what the teacher said. (We will return to this idea in Section 4.1.)

4. Concept maps are also a useful 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." Some teachers are also skeptical of whether novices can effectively map their understanding, since introspection

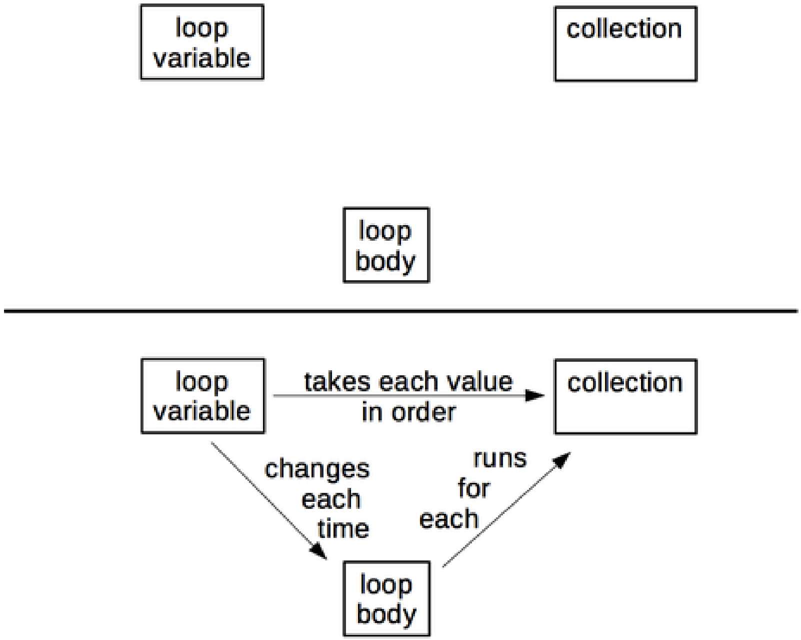


Figure 3.2: Concept Map for a Loop

and explanation of understanding are generally more advanced skills than understanding itself.

> ***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.*

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 pre-defined 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 one-of-a-kind graphics like Abela's decision tree [Abel2009] showing 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.

3.3 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 password, our home address, and what the clown did at our eighth birthday party that scared us so much. It is essentially unbounded: barring injury or disease, we will die before it fills up. However, it is also slow to access—too slow to help us handle hungry lions and disgruntled family members.

Evolution has therefore given us a second system called short-term or working memory. It is much faster, but also much smaller: [Mill1956] estimated that the average adult's working memory could only hold 7 ± 2 items at a time. This is why phone numbers[2] are typically 7 or 8 digits long: back when phones had dials instead of keypads, that was the longest string of numbers most adults could remember accurately for as long as it took the dial to go around and around. It's also why sports teams tend to have about half a dozen members, or be broken down into smaller groups (such as the forwards and backs in rugby).

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

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

This is 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

[2] https://www.quora.com/Why-did-Bell-Labs-create-phone-numbers-of-7-digits-10-digits-Is-there-a-reason-that-dashes-and-brackets-are-used

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. The discussion that follows everyone's realization of how different their mental models of the project's aims and organization are is always interesting...

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 simple 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).

3.4 Challenges

Concept Mapping (pairs/30 minutes)

Draw 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?

The Cost of Multi-Tasking (pairs/10 minutes)

This post[3] from the Learning Scientists describes a simple experiment you can do without preparation or equipment (other than a timer) to demonstrate the mental cost of multi-tasking. Try the activity in pairs, and then report your results to the group.

Noticing Your Blind Spot (small groups/10 minutes)

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 pointed out[4]:

[3] http://www.learningscientists.org/blog/2017/7/28-1
[4] https://twitter.com/elliewix/status/981285432922202113

- 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.
- We're using a single equals sign for assignment.
- Like our commas, there are spaces around the equals sign, which we prefer for readability.
- Our list variable name is plural, which is another personal style choice.

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

Objectives

- Define cognitive load and explain how consideration of it can be used to shape instruction.
- Explain what faded examples are and construct faded examples for use in programming workshops.
- Explain what Parsons Problems are and construct Parsons Problems for use in programming workshops.
- Describe ways they differ from their own students and what effect those differences have on instruction.

In 2006, 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. ([Kirs2006])*

Their paper set off a minor academic firestorm, 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, doesn't actually work very well. This approach—called inquiry-based learning—is intuitively appealing, but Kirschner and colleagues 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 posits 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.

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 that distracts or gets in the way, such as knowing that tabs look like multiple characters but only count as one character when indenting Python code.

According to cognitive load theory, 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 show them a problem and a detailed step-by-step solution, followed by 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.

For example, someone teaching Python might start by explaining how to calculate the total length of a list of words:

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

then ask learners to fill in the blanks in:

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

The next problem might be:

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

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

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

Faded examples work because they introduce the problem-solving strategy piece by piece: at each step, learners have one new problem to tackle, which is less intimidating than a blank screen or a blank sheet of paper (Section 9.10). 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.

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.

Another kind of exercise that can be explained in terms of cognitive load theory is called a Parsons Problem. If you are teaching someone to speak a new language, you could ask them a question, and then give them the words they need to answer the question, but in jumbled order. Their task is to put the words in the right order to answer the question grammatically, which frees them from having to think simultaneously about what to say *and* how to say it.

Similarly, when teaching people to program, you can give them the lines of code they need to solve a problem, and ask them to put them in the right order. This allows them to concentrate on control flow and data dependencies, i.e., on what has to happen before what, without being distracted by variable naming or trying to remember what functions to call. Multiple studies have shown that Parsons Problems take less time for learners to do, but produce equivalent educational outcomes [Eric2017].

> ***Labelled Subgoals***
>
> *[Marg2016, Morr2016] found that students with labelled subgoals solved Parsons Problems for learning loops better than students without, i.e., that giving the steps names helps students learn them (Section L). This can also be explained by cognitive load theory: naming the steps reduces the germane load of figuring out what to do next.*

While faded examples take cognitive load into account in a scalable way, a much older model of learning uses the same ideas on a more personal scale. Cognitive apprenticeship emphasizes the process of a master passing on skills and insights situationally to an apprentice; the master provides models of performance and outcomes, then supports 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 we should have at least a second example when presenting a new idea so that learners can see what to generalize in their schema, and that we should vary the form of the problem to make it clear what are and aren't superficial features (because learners get hung up on those). We should also induce self-explanation, which is discussed in 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.

Learning is therefore more effective when redundant information is *not* presented simultaneously in two different channels. For example, people find it harder to learn from a video that has both narration and on-screen captions than from one that has either the narration or the captions but not both.

The key word in the previous paragraph is "redundant". It turns out that 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, Wies2014] 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.

FIXME (medium): include diagram from https://computinged.wordpress.com/2018/03/23/u more-information-leads-to-worse-performance/

4.2 Pattern Recognition

Section 3.3 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. In fact, supporting this is one of the reasons programmers have been so enthusiastic about design patterns[1]. In practice, though, most pattern catalogs are too large to flick through and too dry to memorize directly. Giving names to a small number of patterns, though, does seem to help with teaching, primarily by giving the learners a richer vocabulary to think and communicate with [Kuit2004, Byck2005, Saja2006]. We will return to this in Section 7.6.

4.3 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 sub-skills 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. Interestingly, this follow-up also reports that people progressed more rapidly when the system rejected errors without doing anything (i.e., left them in the pre-error state).

[1] https://en.wikipedia.org/wiki/Software_design_pattern

4.4 A Final Thought

Cognitive load theory has been criticized as being unfalsifiable: since there's no way to tell in advance of an experiment whether something is germane or not, any result can be justified after the fact by labelling things that hurt performance as extraneous and things that don't germane. However, there is no doubt that instruction based on these principles is effective: for example, [Maso2016] redesigned a conventional introduction to databases 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.

Part of the problem 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[2] 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 emphasizes the social aspects of learning, particularly those 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].)

---

[2] http://www.learning-theories.com/

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[3] characterizes learners as visual, auditory, or kinesthetic according to whether they like to see things, hear things, or do things. This scheme is easy to understand, but as [DeBr2015] explains, it is almost certainly false. Unfortunately, that hasn't stopped a large number of companies from marketing products based on it to parents and school boards.

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.

4.5 Challenges

Create a Faded Example (pairs/30 minutes)

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.

Create a Parsons Problem (pairs/20 minutes)

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.

[3] https://en.wikipedia.org/wiki/Learning_styles#Learning_modalities

Minimal Manuals (individual/20 minutes)

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).

Critiquing Graphics (individual/15 minutes)

[Maye2009] presents six principles for designing good instructional graphics. 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 lesson you have recently taught (or recently been taught) that uses slides or other static presentations, and rate its graphics as "low", "medium", or "high" according to these six criteria.

Cognitive Apprenticeship (pairs/15 minutes)

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.

5 Individual Learning

Objectives

- Explain what metacognition is and why it is important to learning.
- Explain what near and far transfer are, and correctly identify which one occurs most often.
- Name and explain six strategies learners can use to accelerate their learning.
- Explain why working long hours reduces productivity.
- Define calibrated peer review and explain its benefits for learning.
- List common myths about computing education.

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].

The big prize is transfer of learning, which occurs when something we have learned in one field helps us learn more quickly in another. 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—nothing we mean by "learning" could occur if it didn't—but as [Barn2002] showed, discussion of far transfer is muddied by conflicting definitions. [Sala2017] recently analyzed many studies and concluded:

> *...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.*

What this means in practice is that learning to program won't help you play chess and vice versa.

5.1 Six Strategies

Psychologists use a variety of approaches to study learning, but have wound up making the same recommendations about what actually works [Mark2018] The Learning Scientists[1] have catalogued six core strategies and summarized them in a set of downloadable posters[2]. 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). Like any other skill, recall of specific information improves with practice, so taking practice tests, or summarizing the details of a topic from memory and then checking what was remembered and what wasn't, improved outcomes in real situations. 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

[1] http://www.learningscientists.org/
[2] http://www.learningscientists.org/downloadable-materials

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. Your score on the second version shows you how well you are 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.

If you are studying with a partner, you can each pick an idea, then try to find a series of connections that lead from one to the other. For example, 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, so there's a five-step chain from Saskatchewan to statistics.

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.

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].

One structured approach way to do this is the ADEPT method[3]: 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.

Dual Coding

The last of the Learning Scientists[4]' six core strategies 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 redundant information is *not* presented simultaneously in two different channels [Maye2003], because then the brain has to expend effort to check each channel against the other.)

[3] https://betterexplained.com/articles/adept-method/
[4] http://www.learningscientists.org/

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 others 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. Instead, I'd show up for class around noon, unshaven and yawning, and casually mention how I'd been up 'til 6:00 a.m. hacking away at some monster bug or other.

Looking back, I can't remember who I was trying to impress. Instead, what I remember is how much of the code I wrote in those all-nighters I threw away once I'd had some sleep, and how much damage the bugs I created in those bleary-eyed stupors 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. Scientific study of the issue 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.

These facts have been reproduced and verified through hundreds of experiments over the course of more than a century. The data behind them is as solid as the data linking smoking to lung cancer. However, while most smokers will admit that their habit is killing them, people in the software industry still talk and act as if they were somehow exempt from these findings.

It's very easy to go backward when programming: it only takes me a couple of minutes to create a bug that will take hours to track down later—or days, if someone else is unlucky enough to have to track it down. This is summarized in Robinson's first rule:

> 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.

It's hard to quantify the productivity of programmers, testers, and UI designers, but five eight-hour days per week has been proven to maximize long-term total output in every industry that has ever been studied. There's no reason to believe that software development is any different, or that student programming is different from full-time programming in industry.

Ah, you say, that's "long-term total output". What about short bursts now and then, like pulling an all-nighter to meet a deadline? Well, that's 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, the level at which people are usually judged incapable of independent living.

The catch in all of this 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). They certainly don't realize that they're passing parameters into function calls the wrong way around, or that what they're typing in will all have to be deleted and re-done tomorrow, when it will take longer than it would have if they'd just gone home and gotten a good night's sleep.

And despite what many people want to believe about themselves, none of us are good at multi-tasking. What we can become good at, though 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 chew while reading; with practice, we can also take notes while listening, but we *can't* study while doing other things.

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—I have so many assignments to do!", you say. "And they're all due at once! I *have* to work extra hours to get them all done!" No: in order to be productive, you have to do two things: prioritize, and focus. The first is important because people are naturally very good at spending hours on things that don't need to be done, and then finding themselves with too little time for the things that actually count. It can actually be expressed as an algorithm:

1. Make a list of the things you have to do.
2. Weed out everything that you don't need to do right away. Notice that I said "need", not "want": if you want to mess around with a new blogging tool, that's fine, but that's play time, not work time, and we're talking about getting work done.
3. Sorting what's left so that the most important tasks are at the top. (I don't worry about getting the stuff below the first three or four lines into exact order, since I'm going to re-check my list before I get to them anyway.)
4. Make sure you have everything you need to see the first task through: the assignment description, a comfortable chair, etc. Don't give yourself an excuse to interrupt your own work: the world will provide enough of those.
5. Shut down your email and turn off your cell phone. Don't panic, it's only for an hour—most people can't stay focused longer than that, and

anyway, you'll need to stretch your muscles and get rid of that coffee you drank.

6. Set an alarm to go off in sixty minutes, and *focus*. Don't switch tasks in that hour unless you absolutely have to. Instead, if you remember an email message that needs to be sent, or discover a couple of tests that really should be written, add a note to your to-do list. (This is one reason I keep mine in a lab notebook: the few seconds it takes to pick up a pen and jot something down gives my hands a rest from the keyboard.)

7. When your hour is up, take a break: check mail (but don't reply to anything that isn't urgent), go to the washroom, stretch a little, and then re-order your to-do list and start the next round.

If any task on your list is more than an hour long, break it down into into smaller pieces and prioritize those separately. Keep in mind that the future is approaching at a fixed rate of one day every 24 hours: if something's going to take sixty hours to do, you'd better allow at least ten working days for it (assuming you're only interrupted 25% of the time), which means you'd better tackle the first piece two working weeks before the deadline. And since breaking large tasks down into small ones takes time, don't be embarrassed about putting "plan XYZ" in your to-do list.

The point of all this organization and preparation is to get yourself 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, you will 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.

Designed to Fail

The effect of interruptions on productivity begs the question: if timeslicing is bad, why are schools set up to require you to do it all the time*?) Doing nothing but the project course eight hours a day for three weeks would be more efficient. However, it would be harder on instructors, and would be difficult to integrate with courses in subjects like math and languages that take time to soak in.*

Making lists and setting one-hour alarms will seem a little earnest at first, but trust me: your friends will stop mocking you once they see that you're able to finish your assignments and still have time to play some badminton and catch a movie. They may even start to imitate you.

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 real (and as we have recently seen, can be damaging to society as a whole), 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. 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 Appendix J shows a sample to get you started. To use it, rank yourself and each of your teammates, then calculate and compare scores.

5.4 Final Thoughts

A lot of what people believe about programming isn't true, or isn't proven [Oram2010, Stef2017], and the same is true for computing education. [Guzd2015a] list of the top 10 myths is:

1. The lack of women in Computer Science is just like all the other fields of science, technology, engineering, and medicine.
2. To get more women in CS, we need more female CS faculty.
3. A good CS teacher is a good lecturer.
4. Clickers and the like are an add-on for a good teacher.
5. Student evaluations are the best way to evaluate teaching.
6. Good teachers personalize education for students' learning styles.
7. High schools just can't teach CS well, so they shouldn't do it at all.
8. The real problem is to get more CS curriculum into the hands of teachers.

9. All I need to do to be a good CS teacher is model good software development practice, because my job is to produce excellent software engineers.

10. Some people are just born to program.

The last of these is the most pervasive and most damaging. As discussed in Section 10.3, Elizabeth Patitsas and others have shown that grades in computing classes are *not* bimodal [Pati2016], i.e., there isn't one group that gets it and another that doesn't. Many people who would be excellent programmers have convinced themselves that they just don't have what it takes to do so; if all you do is dispel that belief, you will have made the world a slightly better place.

5.5 Challenges

Learning Strategies (individual/20 minutes)

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).

Convergent Evolution (pairs/15 minutes)

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 minutes)

[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 minutes)

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 minutes)

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

Changing Your Mind (small groups/20 minutes)

Working in groups of 3–6, go through Guzdial's list of myths in computing education [Guzd2015a]. Which ones do you believe? Why? What would convince you to change your mind? Does [Guzd2015a] present or link to such evidence?

Calibrated Peer Review (pairs/20 minutes)

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?

Part II

Designing

6 A Lesson Design Process

Objectives

- Describe the steps in backward lesson design and explain why it generally produces better lessons than the more common forward development process.
- Define "teaching to the test" and explain why backward lesson design is *not* the same thing.
- Construct and critique five-part learner personas.
- Construct good learning objectives and critique learning objectives with reference to Bloom's Taxonomy and/or Fink's Taxonomy.

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. 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.

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, and the outcomes of those summative assessments directly affect the teachers' pay and promotion.

[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 as [Scot1998] pointed out, large organizations usually value uniformity over productivity.) We will return to this topic in Chapter 8.

One of the most influential papers in the history of software engineering was "A Rational Design Process: How and Why to Fake It" [Parn1986]. In it, the authors pointed out that in real life we move back and forth between gathering requirements, interface design, programming, and testing, but when we write up our work it's important to describe it as if we did these steps one after another so that other people can retrace our steps. The same is true of lesson design: while we may change our mind about what we want to teach based on something that occurs to us while we're writing an MCQ, we want the notes we leave behind to present things in the order described above.

Agile Teaching

Agile development is a catch-all term for software development processes that rely on very short iterations and constant course correction rather than lots of up-front design. It has many advocates, some very passionate, and several attempts to adapt its ideas to teaching have been described [Chun2004, DSou2015, Duva2018]. To date, though, the results are *indistinguishable from what good teachers do using classical methods when given enough resources and highly engaged learners; as with much of the use of agile in software development, it's a relabelling of things that people already know they ought to do.*

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: the person's general background, what they already know, what *they* think they want to do (as opposed to what someone who already understands the subject thinks), how the course will help them, and 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?"

Deciding What to Teach

There are two ways to decide what to teach: pick material and then find an audience, or decide on an audience and then figure out what they want to learn. Either way, [Guzd2016] offers essential guidance:

1. *Connect to what learners know.*
2. *Keep cognitive load low.*
3. *Be honest (i.e., use authentic tasks).*
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 [Pete2017], data science [Dahl2018], and cell biology [Ritz2018] will also improve outcomes.

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” describes the lesson’s content, not the attributes of successful students.
- “The learner will have a better appreciation for good programming practices” doesn’t start with an active verb or define the level of learning, and the subject of learning has no context and is not specific.
- “The learner will understand how to program in R” starts with an active verb, but doesn’t define the level of learning, and the subject of learning is still too vague for assessment.
- “The learner will write one-page data analysis scripts to read, filter, summarize, and print results for tabular data using R and R Studio.” 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:

Remembering: Exhibit memory of previously learned material by recalling facts, terms, basic concepts, and answers.

Understanding: Demonstrate understanding of facts and ideas by organizing, comparing, translating, interpreting, giving descriptions, and stating main ideas.

Applying: Solve problems to new situations by applying acquired knowledge, facts, techniques and rules in a different way.

Analyzing: Examine and break information into parts by identifying motives or causes. Make inferences and find evidence to support generalizations.

Evaluating: Present and defend opinions by making judgments about information, validity of ideas, or quality of work based on a set of criteria.

Creating: Compile information together in a different way by combining elements in a new pattern or proposing alternative solutions.

The list below shows some of the verbs typically used in learning objectives written for each level:

Remembering: recognize, list, describe, name, find.

Understanding: interpret, summarize, paraphrase, classify, explain.

Applying: build, identify, use, plan, select.

Analyzing: compare, contrast, simplify.

Evaluating: check, choose, critique, prove, rate.

Creating: design, construct, improve, adapt, maximize, solve.

[Masa2018] found that even experienced educators sometimes 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 synthesis and evaluation. (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 has six categories:

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:

- *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.*
- *Be familiar with W3Schools[1] and other free tutorials for HTML and CSS, and know what search terms to use to find answers on Stack Overflow[2].*

[1] https://www.w3schools.com/
[2] https://stackoverflow.com/

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[3] 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[4], 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 almost twenty years, but most teachers still don't write and share lessons the way that they write and share encyclopedia entries. When asked why not, teachers raise many objections[5]:

- *The most important thing about a lesson isn't having it, but writing it, because that gives you a chance to figure out what you think about the topic.* The same is true of software, and somehow we get up-and-coming programmers to use and improve libraries rather than building their own stuff from scratch.

- *It's just more trouble than it's worth, because it's always easier in the short term to write something from scratch than to learn your way around someone else's material.* And yet most teachers use textbooks, and most actors perform other people's plays, and...

- *It doesn't pay off for most teachers because they only teach any particular lesson once a year (or once a quarter).* Infrequent teaching ought to push people *toward* re-use, not away from it.

- *Working at scale results in a more neutral point of view (the average of the contributors' personal views), but in many fields, lessons are valuable precisely because they're one person's opinion.* This may be true for literature, but not for basic algebra. And if the difference is one of teaching method rather than content, there's no reason there couldn't be half a dozen different shared lessons on polynomials, each approaching the topic in a different way, rather than as many as there are teachers.

[3] http://docs.google.com

[4] http://github.com

[5] http://blog.mrmeyer.com/2016/why-secondary-teachers-dont-want-a-github-for-lesson-plans/

- *There's no onboarding process to teach people the mechanics of distributed ad hoc large-scale collaboration.* This is undoubtedly a contributing factor, but teachers get more training in how to develop lessons than most programmers get in how to take part in an open source project and lack of a formal onboarding process hasn't slowed down Wikipedia.
- *Collaboration on lesson development gets squeezed out by more important things* (where "important" means "to the principal or chair"). Again, this should push people *toward* collaboration (possibly under official radar), since every minute they don't spend writing a lesson is a minute they can use to satisfy the principal or chair.
- *Schools' firewalls prevents people from working on shared materials.* This may be true for some teachers, but not for all, and most teachers in industrialized countries have access to a computer at home these days.
- *The stakes are too high for teachers who are going to be evaluated on their teaching.* Again, this may be true for some teachers, but isn't universal.
- *No measurable outcome will show improvement, so there's no incentive to do it.* The same is true of open source software, but while only a small minority of programmers contribute, that's still enough for it to thrive.
- *It's a generational thing: as digital natives, tomorrow's teachers will just naturally do it.* Millenials don't actually act that differently from their elders, and "not yet" arguments are as unfalsifiable as the claims by members of millenarian movements that the apocalypse is definitely coming any day now.
- *You can't run regression tests on a lesson, so there's no easy way to tell if my changes have broken something that you wrote.* Again, Wikipedia proves that this isn't insuperable.

As discussed in Section 13.4, commons-based lesson development and maintenance actually works very well. [Benk2005] described four models of collaboration based on the degree of design and sharing:

High design, low sharing: The teacher can pull together whatever curriculum she feels is appropriate to create a high-quality learning experience.

High design, high sharing: the minister checks his watch and knows what every child is learning at that precise moment across the whole country.

Low design, low sharing: teachers are underpaid and poorly trained, and cobble together whatever they can to get through a course.

Low design, high sharing: Wikipedia.

[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 the context of the author's full course, since visitors may be new teachers and struggle 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[6]. He argues that sites like Stack Overflow[7] 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 future of learning—particularly online learning—may lie in guided tours of community-curated Q&A repositories rather than in things we would recognize as "lessons" today.

6.4 Challenges

Create Learner Personas (small groups/30 minutes)

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

Classify Learning Objectives (pairs/10 minutes)

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 minutes)

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 minutes)

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 minutes)

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

[6] https://hapgood.us/2016/05/13/choral-explanations/
[7] https://stackoverflow.com/

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 minutes)

Betsy Leondar-Wright coined the phrase "inessential weirdness[8]" to describe things groups do that aren't really necessary, but which alienate people who aren't already members of that group. Sumana Harihareswara later used this notion as the basis for a talk on inessential weirdnesses in open source software[9]. 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?

Evaluating Lessons (pairs/20 minutes)

[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?

---

[8] http://www.classmatters.org/2006_07/its-not-them.php
[9] https://www.harihareswara.net/sumana/2016/05/21/0

Promoting Learning (individual/15 minutes)

According to [Merr2013], learning is promoted when learners:

1. Acquire skill in the context of real-world problems (problem-centered).
2. Activate existing knowledge and skill as a foundation for new skill (activation).
3. Observe a demonstration of the skill to be learned (demonstration).
4. Apply their newly acquired skill to solve problems (application).
5. Reflect on, discuss, and defend their newly acquired skill (integration).

Look at a lesson you have recently taught or learned from. Which of these five did it embody? Which were missing? How could they be added?

PRIMM (individual/15 minutes)

One approach to introducing new ideas in computing is PRIMM[10]: **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.

What's Your Viewpoint? (individual/10 minutes)

[Tedr2008] summarizes the history and views of three traditions in computing:

- **Mathematical:**

  Assumptions: Programs (algorithms) are abstract objects, they are correct or incorrect, as well as more or less efficient.

  Aims: Coherent theoretical structures and systems.

  Strengths: Rigorous, results are certain, utilized in other traditions.

  Weaknesses: Limited to axiomatic systems.

  Methods: Analytic, deductive (and inductive).

- **Engineering:**

  Assumptions: Programs (processes) affect the world, they are more or less effective and reliable.

  Aims: Constructing useful, efficient, and reliable systems; solving problems.

  Strengths: Able to work under great uncertainty, flexible, progress is tangible.

  Weaknesses: Rarely follows rigid, preordained procedures; poor generalizability.

[10]http://blogs.kcl.ac.uk/cser/2017/09/01/primm-a-structured-approach-to-teaching-programming/

Methods: Empirical, constructive.

- **Empirical:**

  Assumptions: Programs can model information processes, models are more or less accurate.

  Aims: Investigating and explaining phenomena, solving problems.

  Strengths: Combines deduction and induction, cumulative.

  Weaknesses: Incommensurability of results, uncertainty about what counts as proper science.

  Methods: Empirical, inductive and deductive.

Which of these best describes your approach, and why?

7 Pedagogical Content Knowledge

Objectives

- Learners can summarize research reporting how well (or poorly) students are doing in introductory computing classes today.
- Learners can explain at least three factors that influence how well or how quickly people learn how to program, and give examples of each.
- Learners can define the term "notional machine" and give an example of what one is.
- Learners can summarize shortcomings in the ways that students typically test their software and the pros and cons of using unit tests to grade student programs.
- Learners can explain when and how to use program visualization to teach introductory programming courses.

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. But we do know some things, and this chapter attempts to sum them up and explain their practical implications.

Most of what this chapter presents comes from studying school children and computer science undergraduates at university, both because those are the populations that researchers have easiest access to, and because those are the ages at which people most often learn to program. (As a reminder of this, we use the word "student" in this chapter instead of "learner".) Much less is known about how adults learn to programming in free-range settings or about people who *aren't* intending to be computer scientists, but what we do know is reported here.

[Ihan2016] summarizes the methods most often used to mine and analyze data in these studies. As in all empirical research, it is important to remember that correlation is not causation, and that theories may change as more and better data becomes available.

Jargon

Like any specialty, computing education research has its jargon. The term CS1 is often used to mean an introductory semester-long programming course in which students meet variables, loops, and functions for the first time, while CS2 refers to a second semester-long course that covers basic data structures like stacks and queues. A CS1 course is often useful for undergraduates in other disciplines, though as we will discuss below, it's more effective if it uses relevant examples. A CS2 course designed for *computer science students is usually less relevant for artists, ecologists, and other end-user programmers, but is sometimes the only next step available.*

7.1 What's Our Baseline?

How hard is it to learn to program? It's easy to ask students how much they have learned—universities do it all the time—but study after study has shown that students' teaching evaluations don't correlate with actual learning outcomes [Star2014]. Instead, we have to turn to data.

A decade apart, [Benn2007] and [Wats2014] sought to answer this question by looking at how many students pass their first computer science course. They got very similar answers: two-thirds of post-secondary students pass their introductory course, with some variations depending on class size and so on. There were no significant differences over time or based on language (although failure rates were highest for courses using C and C++).

The more important question is, *How well are they learning?* [McCr2001] was a multi-site international study of how well students can program after their introductory course, replicated more than a decade later by [Utti2013]. The original study reported, "...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."

Switching domains, [Park2015] collected data from an online HTML editor during an introductory web page development course. Nearly all students 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, while 35% related to the simpler tag syntax determining *how* HTML elements are nested.

How much does prior experience matter? [Wilc2018] compared the performance and confidence of students with and without prior programming experience in CS1 and CS2. They found that students with prior experience outscored students without by 6% on exams and 10% in CS1, but those differences disappeared by the end of CS2. Female students with prior exposure outperformed their male peers in all areas, but were consistently less confident in their abilities.

How do newcomers think about programming? [Simo2006] asked students in introductory CS and economics classes to explain how they would sort a list of numbers. The majority of the CS students could describe a plausible algorithm, while less than a third of other students could. However, *fewer* CS students provided a correct answer after their first course because they were trying to put in too many code-level details.

Are they mastering concepts or just mechanics? [Muhl2016] analyzed 350 concept maps drawn by students and compared those who had done a CS course and those who had not, and found that the maps drawn by those who had looked more like the maps experts would draw. For example, "program" was a central concept in both sets of concept maps, but the next most central concepts for those with prior CS exposure were "class" and "data structure", while for those without, they were "processor" and "data".

7.2 What Do Students Misunderstand?

The "superbug" in coding is the belief that the computer understands intention the way that a human being would [Pea1986]. As paradoxical as it sounds, it's very important to teach people that programs are meaningless, i.e., that calling a variable "cost" doesn't guarantee that it actually contains a cost.

Another big misconception has to do with correctness. [Koli2008] found that students view a program's correctness as the sum of the correctness of its parts. They rarely consider programs incorrect, but are instead much more likely to consider them "partially correct" if they have any correct operations. (This is probably a result of thinking in terms of grading schemes.)

A third misconception has to do with the nature of variables. [Kohn2017] found that novices often believe that evaluation is delayed, i.e., that in a program like this:

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

the result will be 90 rather than 75. This is actually correct for spreadsheets, which many learners will have used before they start writing code, so it's worth include exercises early on to diagnose and correct this misconception.

The most comprehensive exploration of novice misconceptions to date is [Qian2017]. This paper divides programming knowledge into:

Syntactic knowledge: Language features, such as the use of quotation marks to define strings.

Conceptual knowledge: Programming constructs, such as how loops work.

Strategic knowledge: How to tackle specific kinds of problems.

They point out that, "...problems in syntactic knowledge are often easy to detect and fix. Perhaps that is why they are often noted as the most frequent mistakes novices make..." and then summarize some common conceptual errors:

- Variables can only hold one value at a time.
- Order in assignment statements matters (5=A and A=5).
- Name of a variable may affect its value or use (i.e., the superbug described earlier).
- Pretty much anything to do with variable scope.
- Printing "X is 1" changes the value of X.
- A conditional inside a loop will be executed whenever the condition is true, even outside the loop.

They also summarize a few strategic errors, such as failing to initialize counter variables, but the more useful part is their list of factors contributing to misconceptions:

- Task complexity and cognitive load.
- Students being confused by the special technical meaning of jargon terms (such as "or" meaning "either or both" rather than "one or the other").
- Variable assignment looking like an algebraic expression (although [Alta2015] found this to be less important than many educators think).
- Inadequate patterns and strategies, characterized by phrases like, "I don't know where to start" or "I can't think at that level of abstraction".
- Confusing syntax, such as using + for addition and concatenation.
- Teachers' explanations. For example, saying that a variable is like a box may imply that many things can be put in it.

Some of their recommendations for fixing this will be familiar by now:

- Present lots of examples.
- Teach programming strategies explicitly.
- Use better tools, such as blocks-based languages like Scratch.
- Provide tools for visualizing program execution so that learners can see what happens when their code runs.
- Better error messages (which are discussed below).
- Automated assessment tools to give more and better feedback earlier (which we discuss in Chapter 11).

7.3 What Are We Teaching Them Now?

What topics do introductory courses cover? [Luxt2017] surveyed the topics included in a variety of introductory programming courses, and analyzed those courses' assessments to see which bits of syntax and semantics students actually had to master. After identifying and counting dozens of individual topics, they broke them down into a dozen categories:

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%)

But this paper does more than catalog concepts: it presents dependency graphs showing how they are connected. For example, it's impossible to explain how operator precedence works without first explaining a few operators, and hard 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.

FIXME (medium): reproduce Rich2017 diagrams

Do computing courses have to include programming? No. [Shel2017] reports that having students work in small groups on computational creativity exercises improves grades at several levels. Some of these exercises include:

- 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.
- Devise a three-step encoding scheme to transfer the alphabet letters into digits and encode questions for other teams to compete to decode.
- Design a calendar for a planet with two suns, four different cultural groups with different resource constraints and industrial needs.

Each exercise comes with an explicit list of reveal insights (which the authors call "light bulb moments") and some prompts for reflection after the exercise is done, such as:

- Did your group have to redesign segments in order to meet the testing requirements?
- If so, identify the reasons for why the initial designs failed; if not, identify the reasons why the initial designs succeeded.
- What considerations were (or should have been) considered during the initial designs in order to meet the testing requirements?

As Silicon Valley finally (grudgingly) acknowledges that insight and reasoning ability matter more than mastery of obscure technical details for programmers at all stages of their careers, non-coding work like this will become more important.

7.4 Do Languages Matter?

The short answer is "yes": novices learn to program faster using blocks-based tools like Scratch that make syntax errors impossible. And its interface encourages exploration in a way that typing text does not; like all good tools, Scratch can be learned accidentally.

Scratch works well because it has been designed and refined over many years with usability and learnability as its primary goals [Malo2010]. We know a great deal about how it is used and misused; for example, [Aiva2016] analyzed over 250,000 Scratch projects and found that:

- Most are small, but it's hard to say whether this is because the authors are young or because of the medium.

- There is very little use of abstractions like custom blocks (the equivalent of procedures), but programs that use them at all tend to use a fair number of them (11 to 12), which could be a signal that they are being used by people who have already mastered abstraction. (The fact that most procedures are only called once may be another sign of this.)

- Simple "stack" blocks (i.e., linear control flow) are by far the most common. Cloned scripts within a single project are also common, and about 28% of projects have some unreachable code (i.e., something that will never run because it's 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.

[Mlad2017] studied 207 students 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.

Blocks are not a panacea. [Grov2017] studied 100 middle-school children and found that it was easier for them to assemble programs with blocks than with text. However, hard concepts are still hard: repeat-until loops with variables that change value inside the loop (rather than the loop doing exactly the same thing each time) and `or` (which is often interpreted as "one or the other" rather than "one or both") were both still difficult.

What about the transition from blocks to text? [Wein2017] studied students using a tool that allowed them to switch between blocks and text for programming. They found that students tend to migrate from blocks to text over time, but there are interesting exceptions. In two thirds of the cases where 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 in blocks mode, or because syntax errors with unfamiliar new commands are not possible. Learners also shifted from text to blocks when adding complex control (e.g., an `if` with an `else`), either because syntax errors are harder, or because the flow of control is immediately visible. 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."

What about explicit typing? Programmers argue a lot about whether variables' data types should have to be declared or not. One recent non-educational finding is [Gao2017], which selected fixed bugs from public JavaScript projects, checked out the code from version control just prior to the fix, manually added type annotations to the buggy code, and then tested whether strongly-typed variants of JavaScript reported an error. They found that about 15% of bugs are caught, 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] and other studies show that declared types do add some complexity to programs, but it pays off fairly quickly by acting as documentation hints for a method's use, in particular by preventing questions about what we have and and what we can do with it.

What about object-oriented programming? Objects and classes are power tools for experienced programmers, but power tools aren't always suitable for beginners. [Mill2016b] found that most students had difficulty with the `self` object in Python (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 a study of 86 high school students. Only 45% of students understood when to use `this`, only 60% understood when not to, only 24% could define it clearly, and these figures probably overestimate understanding because respondents might simply be reciting memorized answers. They also looked at 48 high school teachers, and drily observe that they, "...expressed a considerable lack of clarity in accurately characterizing the correctness of students' answers." And [Mill2014] found that novice programmers often refer to an object when they mean an attribute or property of that object or vice versa. Students seem to make errors of this kind more often for identifying attributes (like "Canada") than for descriptive attributes (like "Canadian"), probably because they think of nouns and adjectives as things and pointers to things respectively.

Does it have to hurt this much? No. [Stef2013] has shown that programming language designers needlessly make programming 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 other 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.

Is it going to get better? Not any time soon. [Pere2013] compared Git's actual operation with its users' conceptual model, highlighting and explaining the many errors and confusion that result from the differences. [Pere2016] then used that work to design a more user-friendly alternative to Git. The result? "In sharing our research with colleagues...we have discovered a significant polarization. Experts, who are deeply familiar with the product, have learned its many intricacies, developed complex, customized workflows, and regularly exploit its most elaborate features, are often defensive and resistant to the suggestion that the design has flaws. In contrast, less intensive users, who have given up on understanding the product, and rely on only a handful of memorized commands, are so frustrated by their experience that an analysis like ours seems to them belaboring the obvious."

7.5 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 is it true? Early studies like [Lawr2006] tried to find out, but didn't distinguish between short and long names. More recently, [Hofm2017] found that using full words in variable names led to an average of 19% faster comprehension compared to letters and abbreviations, with no significant difference in speed between single letters and abbreviations, but didn't look at *which* names were abbreviated.

For that, we have to turn to [Beni2017], which found that using single-letter variable names doesn't affect novice programmers' ability to modify code. This may be because novices' programs are shorter than professionals', but it may also be 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. (`e` doesn't have a strong implicit meaning, but "exception" wasn't one of the options in the study.)

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 (although again, they didn't explore *which* variables), and that beginners found CamelCase easier to read than pothole_case. This is surprising because word spacing improves readability in conventional tasks. Digging deeper, "...camel casing produces more accurate results. However, this correctness comes at a cost as the camel-case style significantly increases the time needed to correctly detect the correct identifier."

More recently, [Floy2017] reports an fMRI study of 29 people that found that the same brain regions are involved in reading code and prose, and that while the two are distinct activities, they become more similar as people develop expertise.

7.6 How Do Students Program?

[Solo1984, Solo1986] pioneered the exploration of novice and expert programming strategies. The key finding is that experts have both the ability to plan a program and sufficient syntactic knowledge to implement it. Novices lack both, but we often mistakenly focus on gaps in the latter. For example, bugs are often related to planning errors (i.e., lack of a strategy for solving the problem) rather than to lack of knowledge about the language. Teachers should therefore emphasize the "how" of program construction as much as the "what". Having lots of plans or goals when programming isn't always a good thing—as [Spoh1985] found, merging plans and/or goals can yield bugs because of goals being dropped or fragmented—but not having plans is always harmful.

> ***Harder or Less Familiar?***
>
> *[Solo1986] introduced the Rainfall Problem, which is simple to state: 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.*
>
> *The Rainfall Problem has been used in many subsequent studies of programming. For example, [Fisl2014] found that students made fewer low-*

level errors when solving the problem in a pure functional language, but [Sepp2015] still found that success rates were disappointingly low: very few studies found even half of students able to solve it correctly. However, [Simo2013] argues that the Rainfall Problem is harder for novices than it used to be because they're not used to handling keyboard input, and "run until you see a sentinel" isn't a pattern today's novice programmers are familiar with. Direct comparison with past cohorts may therefore be unfair.

The most important recommendation in this chapter is therefore to teach solution patterns, i.e., show learners over and over again *how* to tackle problems. [Mull2007b] is just one of many studies proving the benefits of this.

One of the most useful (and sadly under-used) pieces of work I've found helpful in describing programming plans to novices is the single-variable design patterns in [Kuit2004, Byck2005, Saja2006]. Consistent with everything we know about worked examples and subgoals, they found that labelling the parts of students' programs gave students a vocabulary to think with, and implicitly a set of programming plans for constructing code of their own. Their patterns are:

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.

The Roles of Variables website[1] has examples of all of these, and I use these terms frequently in my own teaching. (I also wonder what a tool like Scratch would look like if users had to create variables with these roles, rather than plain variables.)

[1] http://www.cs.joensuu.fi/~saja/var_roles/

Does step size matter? Maybe. [Blik2014] found that, "more experienced students were more likely to adopt an incremental coding strategy (trying to debug and advance their code without external help through myriad trial-and-error attempts), whereas novices would update their code in larger batches, copying and adapting code from sample programs and other external sources." However, when they looked at whether the amount of tinkering correlated with course performance, the answer was negative despite repeated re-slicing of the data to try to find an effect. They then looked at changes in the frequency of updates rather than update size, hypothesize that low-performing students wouldn't change their update patterns, but high-performing students would, and again struck out.

On the other hand, [Cart2017] showed that students at different levels approach programming tasks differently, and that these differences can be detected automatically. Their model categorizes student activity in a two-dimensional space where one axis is the student's current activity (e.g., editing or debugging) and the other is the correctness of the student's most recently compiled program. This gives states like "editing syntactically correct code, last debug successful", and allows them to construct activity sequences like, "Running a semantically incorrect program outside of debug mode." The authors caution that a given sequence of state transitions could correspond to several different problem-solving activities, but found that high-performing students spent a lot of time in testing modes, while low-performing students spent much more time working on code with errors.

[Kaze2017] analyzed character-level edit and execution data from participants in an undergraduate course to see if incremental development and procrastination correlate with solution correctness, completion time, or total work time. Projects where the author started editing earlier were more likely to submit their projects earlier and to earn higher scores for correctness, and starting to write tests earlier was also associated with higher correctness scores. However, the authors found no significant relationship between incremental test writing or incremental checking of work and higher scores.

Does order matter? Probably. [Ihan2011] describes tool for 2D Parsons Problems (i.e., ones in which code can be dragged horizontally as well as vertically). They found that experienced programmers often drag the method signature to the beginning, then add the majority of the control flow (i.e., loop statements, assignments, conditional statements), 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; one of the benefits of live coding (Section 8.3) is that it gives them a chance to see the sequence that more advanced programmers actually use.

7.7 What Mistakes Do Learners Make?

The short answer is, "Teachers don't know as much as many of us think we do." [Brow2014] looked at eighteen types of errors, from mismatched parentheses to discarding the result of a non-`void` method. 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.

[Alta2015] then looked at what errors novices actually make in Java. Unsurprisingly, mistakes that produce compiler errors are fixed much faster than ones that don't. 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. However, some mistakes are made many times, like invoking methods with the wrong arguments (e.g., passing a string instead of an integer). Interestingly, another common error is reaching the end of a non-`void`‘ method without returning a value. Python and other languages permit this, and return `none` if no value is explicitly specified; so far as I know, nobody has done a study to see if this default behavior masks faults in novices' mental models.

These findings aren't specific to any particular language: [Herm2016] gave 61 novice Scratch programmers a comprehension task, and found that students working with smelly code did not take more time to solve problems, but had lower correctness rate. Similarly, [Keun2017] looked at code quality issues in students' Java programs using a subset of [Steg2016a]'s rubric for code quality. They found that students usually don't fix issues, particularly issues related to modularization. One caution from their work 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.

[Edwa2017] studied nearly 10 million static analysis errors in over 500,000 program submissions—things like checking for `null` after a pointer is used instead of before. They found that formatting and Javadoc issues are the most common, and that coding flaws at any point when developing a solution resulted in significantly lower scores on the assignment. Students produce fewer errors with experience, but the errors that are most frequent are consistent between both computer science majors and non-majors and across experience levels.

7.8 How Do Students Test and Debug?

A decade ago, [McCa2008] wrote, "It is surprising how little page space is devoted to bugs and debugging in most introductory programming textbooks." They describe many reasons for bugs that can be demonstrated in class or checked with formative assessment exercises, including:

Using the natural-language meaning of terms like `while` (meaning "at the same time" rather than "as long as a condition is true").

Off-by-one errors when counting or indexing, e.g., looking up from the last element of a list, or down from the first).

Putting code in the wrong place, such as placing a `print` statement inside a loop when it should only be executed once after the loop finishes.

(How) do students debug? [Fitz2008, Murp2008] looked at how undergraduate students debugged their code. Most students who were good debuggers were good programmers, but not all good programmers were good at debugging. Those who were traced execution, wrote tests, re-read

the spec, and used a debugger. However, tracing was sometimes used ineffectively: for example, a student might put the same `print` statement in both parts of an `if-else`. Students would also comment out lines that were actually correct in an attempt to isolate the problem, and didn't seem to realize when they were stuck.

(How well) do students test their code? [Bria2015] describes a tool that scores a student's program by how many teacher-provided tests cases pass, and conversely scores the student's test cases by how many of the bugs in a model solution deliberately seeded with errors they catch. They found that students' tests often have low coverage (i.e., they don't test most of the code) and that students misunderstand the "unit" part of unit tests: their tests often exercise many things at once, which makes it hard to pinpoint the causes of errors.

[Edwa2014b] dug a little deeper, with sobering results. They had students write their own software tests, which were graded in part on branch coverage (i.e., how many of the possible paths through the code their tests exercised). They then looked at all of the bugs in all of the students' code submissions combined and identified those detected by each student-written test suite. The result was that students' tests had an average of 95.4% branch coverage on their own code, but only detected an average of 13.6% of the faults present in the entire program population. What's more, 90% of the students' tests were very similar, which indicates that students mostly write tests to confirm that code is doing what it's supposed to rather than to uncover situations where it isn't.

[Alqa2017] collected data from 142 novices doing their second programming course to categorize their debugging activities. They picked eight bugs, such as accidentally creating a loop without a body (because of a misplaced semi-colon) or an array bounds error, and then created one program for each bug that contained only that bug, and that could be fixed by modifying a single line. Unsurprisingly, students with more experience solved the problems significantly faster, but times varied widely: 4–10 minutes is a typical range, and overall times ranged from 4.5 minutes to two hours, which means that some learners will need 2–3 times longer than others to get through the same material.

Multiple studies have shown that reading code is the most effective way to find bugs [Basi1987, Keme2009, Bacc2013]. The value of tracing code has been studied less often for professional programmers, but it is a key skill for learners. [List2004] tested students' ability 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. Many students were weak at these tasks, which suggests that not being able to trace program execution is part of the explanation for poor programming ability.

[List2009] returned to this subject, and found once again that students who perform well at writing code can usually also trace and explain code. [Harr2018] later found that the gap between being able to trace code and being able to write it has largely closed by CS2, but that students who still have a gap (in either direction) are likely to do poorly in the course.

[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 pause their "execution" of the example to generate an explanation of what's going on, and crucially, these people learn faster. This suggests that as well as asking learners to trace the execution of programs, we should show them programs' output and ask them to explain why it is what it is.

7.9 Do Error Messages Matter?

The answer to the question in the section title is "yes". [Hugh2010] traced student responses to non-literal errors, such as reports of missing parentheses that are in fact caused by missing + for string concatenation. They found that 8% of compilation errors and 100% of runtime exceptions in novices' first Java programs were caused by string formatting problems exacerbated by poor error reports about non-literal errors. The problem is that misleading error messages erode novices' trust: poor error messages are effectively behavioral conditioning to ignore *all* error messages. A common response from students is to move on and revisit the problem later, which is occasionally effective.

Can we do better? [Beck2016] tried writing better error messages for the Java compiler, 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.
```

students would see:

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

Novices given these made fewer errors overall and fewer repeated errors.

[Beck2018b] measured the effect of enhanced compiler error messages by having students remove syntax errors from non-compiling code they did not write. They found a significant positive effect on the overall number of errors rectified, as well as the number of certain specific error types, but no significant effect on the number of non-compiling submissions or student scores. These results suggest that the apparently contradictory findings of other recent studies are not actually in conflict: enhanced error messages may be effective, but also that the signal is relatively weak.

[Bari2017] went further and used eye tracking to show that developers really do read error messages—people spend 13–25% of their time on task doing this. Doing so is as difficult as reading source code, and how difficult it is to read the error messages strongly predicts task performance. The inescapable conclusion is that it really is important to get this right, and that error messages should be usability tested the same way as any other interface.

Since interpreting and responding to error message is important, instructors should give learners exercises that do just that. [Marc2011] has a rubric for responses to error messages that can be useful in grading such assignments:

1. Learner deletes the problematic code wholesale.
2. Learner makes a change that is unrelated to the error message and does not help.

3. Learner makes a change that is unrelated to the error message but correctly addresses a different error or makes progress in some other way.

4. Learner's change shows that they have understood the error message (though perhaps not wholly) and is trying to take an appropriate action (though perhaps not well).

5. Learner's change fixes the actual error (though other errors might remain).

[Sirk2012] catalogued the errors that students made when using a simple execution visualization tool, several of which echo the findings of [Mill2014, Mill2016b, Rago2017] discussed above. These mistakes can all motivate the design of particular exercises, and are presented in one of the challenges below.

7.10 Does Program Visualization Matter?

The idea of visualizing programs is perennially popular, but that doesn't mean it's effective. We have known for over 20 years that people learn more from constructing visualizations of algorithms than they do from viewing visualizations constructed by others [Stas1998, Ceti2016]. That said, [Guo2013] (a web-based tool for visualizing the execution of Python programs) and Loupe[2] (which shows how JavaScript's event loop works) are both great teaching *aids*.

Does drawing pictures help learners understand programs? Yes. [Cunn2017] replicates an earlier study of the kinds of sketching students do when tracing code execution, and correlations between different kinds and effectiveness. Different rates of sketching for different problems indicated that students use it to externalize cognition; as the study says, "Students frequently re-wrote pieces of code from questions on their scratch sheets, perhaps to manage issues related to the split attention effect." 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.

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. This is once again the kind of study that should lead to tool improvements: to the best of my knowledge, nobody provides a debugger that shows variables in rows with successive values laid out in columns using the horizontal axis for time.

Flowcharts

One often-overlooked result from [Scan1989] is that students understand flowcharts better than pseudocode if both are equally well structured. *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.*

[2] http://latentflip.com/loupe/

7.11 How Can We Help Them?

[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 here is positive reporting bias or the way we're teaching right now is almost the worst way possible and anything would be an improvement. It's also worth remembering that like almost all of the studies discussed in this chapter, they were only looking at university classes: their findings may not generalize to other groups.

With all those caveats in mind, they found ten things instructors can do to improve outcomes. (The figures after each intervention are the number of studies reported and the average improvement they report.)

Collaboration (20/34%): Activities that encourage student collaboration either in classrooms or labs.

Content Change (36/34%): At least parts of the teaching material was changed or updated.

Contextualization (17/40%): Activities where course content and activities were aligned towards a specific context such as games or media.

CS0 (7/43%): 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 (9/18%): A game-themed component was introduced to the course.

Grading Scheme (11/29%): A change in the grading schema; 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 (7/45%): Activities with increased group work commitment such as team-based learning and cooperative learning.

Media Computation (10/48%): Activities explicitly declaring the use of media computation (Chapter 10).

Peer Support (23/34%): Support by peers in form of pairs, groups, hired peer mentors or tutors.

Other Support (9/33%): An umbrella term for all support activities, e.g. increased teacher hours, additional support channels, etc.

This list highlights the importance of cooperative learning, and [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.

Pair programming is the most obvious kind of cooperative learning in a coding class, but there are lots of other things learners can do together. If they are writing programs to draw things using turtle graphics, for example, one can play the part of the loop, the second can be the conditional, and the third can be the turtle that is moving (or not) based on the instructions given by the other two.

One way to help learners is to give them consistent feedback that points them in the right direction. The most useful resource I have found for this is the code quality rubric developed in [Steg2014, Steg2016a] (which is online at [Steg2016b]). Its categories include variable names, comments, code layout, control flow, modularization, and so on; two examples of its levels are:

Names

1. names appear unreadable, meaningless, or misleading
2. names accurately describe the intent of the code, but can be incomplete, lengthy, misspelled or inconsistent use of casing
3. names accurately describe the intent of the code, and are complete, distinctive, concise, correctly spelled and consistent use of casing
4. all names in the program use a consistent vocabulary

Control Flow

1. there is deep nesting; code performs more than one task per line; unreachable code is present
2. flow is complex or contains many exceptions or jumps; parts of code are duplicated
3. flow is simple and contains few exceptions or jumps; duplication is very limited
4. in the case of exceptions or jumps, the most common path through the code is clearly visible

7.12 How Should We Design Lessons?

What is our goal? 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.

[Sorv2014] lays out three cognitively plausible frameworks for the design of a first programming course, all of which lead to something like the notional machine described above. The first is Motivate-Isolate-Practice-Integrate:

Motivate: start with a "large" project (by the learner's standards).

Isolate: highlight necessary skills.

Practice: work on individual skills.

Integrate: bring skills to bear on the project.

The second, Head Straight for Objects, emphasizes object-oriented programming early: the teacher introduces ideas that objects can easily represent, like shapes or people's roles, then shows how code can capture these ideas. The third approach, Explicit Program Dynamics, teaches programming by explaining how programs run. Copying values vs. copying references, stack frames, and other parts of the program's state are first-class concepts, and visualization is typically used extensively and in varied ways.

7.13 Final Thoughts

As the introduction said, we don't know as much about how people learn to program as we do about how they learn other things, and much of what we do know may only apply to certain groups. But that's not the same as saying that we don't know anything, and good lessons and teaching should draw on what knowledge we have. Conferences like SIGCSE[3], ITiCSE[4] and ICER[5] are home to a steady stream of rigorous, insightful studies with immediate practical application; while much of that work is sadly hidden behind exclusionary paywalls, sites like Sci-Hub[6] can help you find what you need.

7.14 Challenges

Checking for Common Errors (individual/20 minutes)

This list of common errors is taken from [Sirk2012]. Pick three, and write an exercise 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.

---

[3] http://sigcse.org/
[4] http://iticse.acm.org/
[5] https://icer.hosting.acm.org
[6] http://sci-hub.tw

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 minutes)

[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 minutes)

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

Rate Your Tools (individual/15 minutes)

[Koll2016] proposes a set of heuristics to be used in evaluating programming systems for novices, based in large part on the authors' work developing several generations of such systems [Koll2015]. The heuristics are listed below; rate the programming system you are using in your teaching as "low", "medium", "high", or "not applicable" for each one.

Engagement: The system should engage and motivate the intended audience of learners. It should stimulate learners' interest or sense of fun.

Non-threatening: The system should not appear threatening in its appearance or behaviour. Users should feel safe in the knowledge that they can experiment without breaking the system, or losing data.

Minimal language redundancy: The programming language should minimize redundancy in its language constructs and libraries.

Learner-appropriate abstractions: The system should use abstractions that are at the appropriate level for the learner and task. Abstractions should be driven by pedagogy, not by the underlying machine.

Consistency: The model, language and interface presentation should be consistent—internally, and with each other. Concepts used in the programming model should be represented in the system interface consistently.

Visibility: The user should always be aware of system status and progress. It should be simple to navigate to parts of the system displaying other relevant data, such as other parts of a program under development.

Secondary notations: The system should automatically provide secondary notations where this is helpful, and users should be allowed to add their own secondary notations where practical.

Clarity: The presentation should maintain simplicity and clarity, avoiding visual distractions. This applies to the programming language and to other interface elements of the environment.

Human-centric syntax: The program notation should use human-centric syntax. Syntactic elements should be easily readable, avoiding terminology obscure to the target audience.

Edit-order freedom: The interface should allow the user freedom in the order they choose to work. Users should be able to leave tasks partially finished, and come back to them later.

Minimal viscosity: The system should minimize viscosity in program entry and manipulation. Making common changes to program text should be as easy as possible.

Error-avoidance: Preference should be given to preventing errors over reporting them. If the system can prevent, or work around an error, it should.

Feedback: The system should provide timely and constructive feedback. The feedback should indicate the source of a problem and offer solutions.

Roles of Variables (pairs/15 minutes)

Take a short program you have written (5–15 lines) and classify each of its variables using the categories defined by Sajaniemi et al. 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 minutes)

Which of the three approaches described in [Sorv2014] (Section 7.3) do you use when teaching? Or is your approach best described in some other way?

What Are You Teaching? (individual/10 minutes)

Compare the topics you teach to the list developed in [Luxt2017] (Section 7.3). Which topics do you cover? What extra topics do you cover that aren't in their list?

Beneficial Activities (individual/10 minutes)

Look at the list of interventions developed by [Viha2014] (Section 7.11). Which of these things do you already do in your classes? Which ones could you easily add? Which ones are irrelevant?

Visualizations (individual/10 minutes)

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 minutes)

The Professional Development for CS Principles Teaching[7] site includes a detailed list of student misconceptions and challenges[8]. 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?

[7] http://www.pd4cs.org/
[8] http://www.pd4cs.org/mc-index/

Part III

Teaching

8 Teaching as a Performance Art

Objectives

- Define *jugyokenkyu* and lateral knowledge transfer and explain their relationship to each other.
- Describe and enact at least three techniques for giving and receiving feedback on teaching performance.
- Explain at least two ways in which using a rubric makes feedback more effective.
- Describe live coding and explain its advantages as a teaching practice for programming workshops.
- Do and critique live coding.

From politicians to researchers and teachers themselves, educational reformers have designed systems to find and promote those people can teach and eliminate those who cannot. But the assumption that some people are born teachers is wrong, which is why reforms based on it have repeatedly failed. 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[1].

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, a textbook publisher, or the Internet, but each teacher has to figure out for herself how to combine that with the theory she learned in education school to deliver an actual lesson in an actual classroom for actual students.

Demonstration lessons, in which one teacher is in front of a room full of students while other teachers observe, seem like a way to solve this. However, Fincher and colleagues studied how teaching practices are actually transferred using both a detailed case study [Finc2007] and analysis of change stories [Finc2012]. The abstract of the latter paper sums up their findings:

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

[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.

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

- content knowledge, such as the "what" of programming;

[1] https://en.wikipedia.org/wiki/W._Edwards_Deming

- general pedagogical knowledge, i.e., an understanding of the psychology of learning; and
- the pedagogical content knowledge (PCK) that connects the two. PCK is things like what examples to use when teaching how parameters are passed to a function, or what misconceptions about wildcard expansion are most common. For example, an instructor could write variable names and values on paper plates and then stack and unstack them to show how the call stack works.

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. The CS Teaching Tips[2] site is collecting PCK for teaching programming; I hope that one day we will have catalogs like [Ojos2015], teacher training materials like [Hazz2014], or more personal collections like [Gelm2002] to help us all do it better.

8.1 Feedback

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

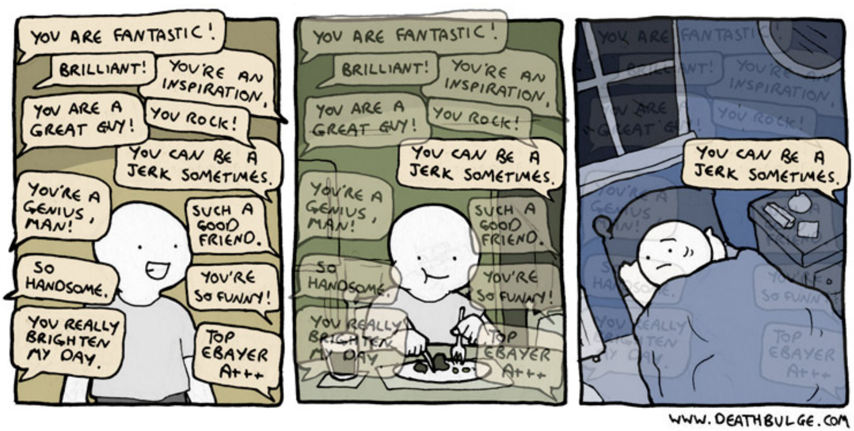


Figure 8.1: Feedback Feelings (copyright © Deathbulge 2013)

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.

[2] http://csteachingtips.org/

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 2×2 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.

A more sophisticated rubric developed for assessing 5–10 minute videos of programming instruction is given in Appendix I. 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 I 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.

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.

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, and a few people have experimented with using live coding at conferences or online in similar ways.

Tells

Everyone has nervous habits. For example, many of us 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.

8.2 How to Practice Teaching

The best way to improve your teaching 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 2×2 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 9.2): teachers find it a lot easier to give each other informal feedback if they'd have some practice doing so and have a shared rubric to set expectations.

8.3 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 following 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. 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, but most textbooks and tutorials spend little time on 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[3] 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 "real" mistakes and making them deliberately, but that often feels forced. 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.

[3] http://software-carpentry.org

Take It Slow

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

If the output of your command or code makes what you just typed disappear from view, scroll back up so learners can see it again. 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 schemes 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.

Accessibility Aids Help Everyone

Tools like Mouseposé[4] (for Mac) and PointerFocus[5] (for Windows) will highlight the position of your mouse cursor on the screen, and screen recording software tools like Camtasia[6] 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.

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

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

Multiple Personalities

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

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 9.5). 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.

[4] https://boinx.com/mousepose/overview/
[5] http://www.pointerfocus.com/
[6] https://www.techsmith.com/video-editor.html

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.

Improvise After You Know the Material

The first time you teach a new lesson, stick fairly closely to the whatever 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 (DI) is a teaching method centered around meticulous curriculum design delivered through prescribed script—i.e., it's like an actor reciting lines rather than the improvisatory approach we advocate. [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 challenge 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 9.8).

8.4 Challenges

Give Feedback (whole class/20 minutes)

1. Watch [Wils2016] 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 2×2 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 minutes)

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 2×2 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 minutes)

Watch the videos of live coding done poorly[7] and live coding done well[8] and summarize your feedback on both using the usual 2×2 grid. These videos assume learners know what a shell variable is, know how to use the `head` command, and are familiar with the contents of the data files being filtered.

See Then Do (pairs/30 minutes)

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.

[7] https://youtu.be/bXxBeNkKmJE
[8] https://youtu.be/SkPmwe_WjeY

- 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 minutes)

1. Read the description of tells at the end of Section 8.1, 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 minutes)

The CS Teaching Tips[9] 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] http://csteachingtips.org/

9 In the Classroom

Objectives

- Describe how to handle a Code of Conduct violation.
- Explain the benefits and drawbacks of co-teaching.
- Explain why teachers should *not* introduce new pedagogical practices in a short workshop.
- Name, describe, and enact four teaching practices that are appropriate to use in programming workshops for adults, and give a pedagogical justification for each.

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.

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 Appendix B. 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 organizer of your class as soon as you can and describe what happened. Remember, a Code of Conduct is meaningless without a procedure for enforcing it.

9.2 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 are better than teaching alone because they create opportunities for lateral knowledge transfer; we use team teaching, teach and assist, and alternative teaching most often in programming workshops. 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 9.5) 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 9.5), 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; in that moment, no one will understand how pleased you are at helping someone understand how loops work, or how disappointed you are that you just couldn't get software to install on that one learner's laptop, than the person you just taught with.

9.3 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 pre-assessment, 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[1] [Krug1999]: the less people know about a subject, the less accurate their estimate of their knowledge is. Rather than asking people to rate their knowledge from 1 to 5, you can try to ask them how easily they could complete some specific tasks, but that still runs the risk of scaring them away. Appendix K presents a short pre-assessment 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.

False Beginners *FIXME (medium): false beginners*

[1] https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

9.4 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 class 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.

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.

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.

9.5 Take Notes Together

Many studies have shown that taking notes while learning improves retention [Aike1975, Boha2011], because taking notes forces you to organize and reflect on material as it's coming in, which in turn increases the likelihood that you will transfer it to long-term memory in a usable way (Chapter 3).

Our experience, and some recent research findings, lead us to believe that taking notes *collaboratively* helps learning even more [Ornd2015, Yang2015], even though taking notes on a computer is generally less effective than taking notes using pen and paper [Muel2014]. Some of the arguments in favor of shared note-taking 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.

Shared note-taking is usually mentioned positively in after-class feedback from workshops I have taught. However, it's also common for participants to report that they find it distracting, as it's one more thing they have to keep an eye on. I think the positives outweigh the negatives, but if you are only working with a particular group once, heed the advice in Section 9.11 and let them do whatever they're used to.

We usually use Etherpad[2] or Google Docs[3] for collaborative note-taking. The former makes it easy to see who's written what, while the latter scales better and allows people to add images to the notes. Whichever is chosen, classes also use it to share snippets of code and small datasets, and as a way for learners to show 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 respond individually (e.g., 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.

9.6 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.

9.6.1 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.

9.6.2 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.

9.6.3 As Minute Cards

We frequently use sticky notes as minute cards: before each break, learners take a minute to write one positive thing on the green sticky note (e.g., one thing they've learned that they think will be useful), and one thing they found too fast, too slow, confusing, or irrelevant on the red one. They can

[2] http://etherpad.org
[3] https://docs.google.com

use the red sticky note for questions that haven't yet been answered. 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.

9.7 Peer Instruction

No matter how good a teacher is, she can only say one thing at a time. How then can she clear up many different misconceptions in a reasonable time? The best solution developed so far is a technique called peer instruction. Originally created by Eric Mazur at Harvard, it has been studied extensively in a wide variety of contexts, including programming [Crou2001, Port2013].

Peer instruction 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[4] 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.

Peer instruction is essentially a way to provide one-to-one mentorship in a scalable way. Despite this, we usually do not use it in either programming workshops or teacher training workshops because it is new to most learners (Section 9.11). If you are working with the same group for an extended period, however, [Port2016] showed that students value peer instruction.

[4] https://www.youtube.com/watch?v=2LbuoxAy56o

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).

9.8 Never a Blank Page

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

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

9.9 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.*

We have experimented with 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, and learners often struggle to switch back and forth between two different sets of keyboard shortcuts for things like copying and pasting.

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[5] enter widespread use, that is less and less important.

9.10 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 Etherpad, but its 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.)

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.

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 Appendix H.

[5] https://glitch.com/

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 9.4). 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 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.

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.

9.11 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.

9.12 Feedback

You can't improve your lessons or your teaching if you don't know how well you're doing, but finding out turns out to be surprisingly difficult:

Ask learners if the class was useful. Study after study has shown that there is no correlation between how highly learners rate a course and how much they actually learn [Uttl2017], and most people working in education are now aware of that.

Give them an exam at the end. 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. In addition, any kind of final exam dramatically changes 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. That's difficult to do in school, and practically impossible with free-range learners. In addition, the people who didn't get anything out of the workshop are probably less likely to take part in follow-up, so feedback gathered this way will be subject to self-selection bias.

See if they keep using what they learned. This is probably the most meaningful way to gauge the value of what you have taught; the problem again is how to do it. (Installing spyware on their computers is generally frowned upon.)

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

Before embarking on any major assessment, find out what your learners, colleagues, and sponsors will accept as evidence. It's not enough to be right—you also have to be believed.

9.13 Challenges

Create a Questionnaire (individual/20 minutes)

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

One of Your Own (whole class/15 minutes)

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 minutes)

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 minutes)

Watch this video[6] 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?

[6] https://www.youtube.com/watch?v=vgkahOzFH2Q

10 Motivation and Demotivation

Objectives

- Explain the difference between intrinsic and extrinsic motivation.
- Name and describe three ways in which teachers can demotivate learners.
- Define impostor syndrome and describe ways in which to combat it.
- Explain what stereotype threat and fixed vs. growth mindset are, summarize the strength of evidence supporting each, and describe their implications for teaching.
- Describe and enact three things teachers can do to make their classes more accessible.
- Describe and enact three things teacher can do to make their classes more inclusive.

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 intrinsic motivation, which is what we feel when we find something personally rewarding, from extrinsic motivation, which we feel when we do something to avoid punishment or earn a reward. Both are factors in 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].

> ***The Problem of Grades***
>
> *I've never had an audience in my life. My audience is a rubric.*
> *– quoted by Matt Tierney*[1]
>
> *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.*

[1] https://twitter.com/figuralities/status/987330064571387906

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":

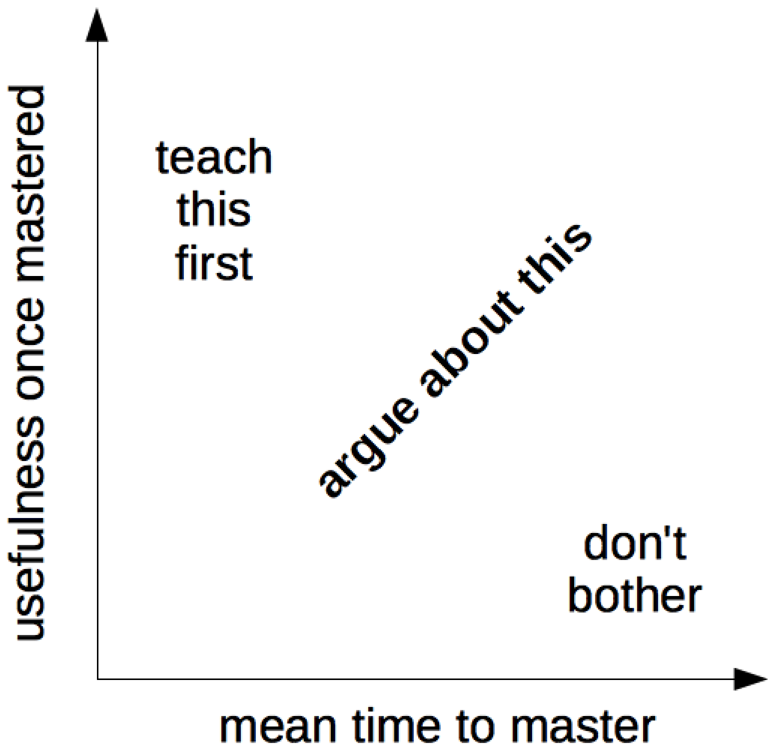


Figure 10.1: What to Teach

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 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 courses around data science, image processing, and biology [Dahl2018, Meys2018, Ritz2018].

[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 9.2): 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 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 challenge 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 that they have no way to change a situation, they may learn not to even try to change things when 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[2] 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. 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 10.6, and doing so improves outcomes for everyone [Sved2016].

10.2 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 7.1, [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.

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[3] 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.
- *Ask your friends what they think of you.* Usually, other people have a more realistic (higher) opinion of your work. Your friends can remind of you major accomplishments you have completely forgotten about.
- *Go to an in-person Impostor Syndrome session at a conference, from your workplace training program, or your school.* There's nothing like being in a room full of people you respect and discovering that 90% of them have impostor syndrome.

[2]https://stackoverflow.com/
[3]https://www.usenix.org/blog/impostor-syndrome-proof-yourself-and-your-community

- *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.
- *Re-orient ourselves around your values and worth.* When called upon to step up and show your work, think about your core values and how your work reflects them.

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 also has guidelines for communities:

- *Simply encourage people.*
- *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 in your team and department 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.

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 8.3, 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.

10.3 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, and the clearest examples in computing are gender-related. Depending on whose numbers you trust, only 12–18% of programmers are women, and those figures have actually been getting worse over the last 20 years. There are many reasons for this [Marg2003, Marg2010], and [Stee2011] summarizes what we know about stereotype threat in general and presents some strategies for mitigating it in the classroom.

> ***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] reviews all three books; mentioning any of this makes many Silicon Valley brogrammers uncomfortable to the point of belligerence, which is a good reason to do it.*

While there's lots of evidence that unwelcoming climates demotivate members of under-represented groups, 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[4] about the replicability of key studies. What *is* clear is that we need to avoid thinking in terms of a deficit model, i.e., that the members of under-represented groups are the ones who should change because they have some deficit, such as lack of prior experience. Instead, we should use acknowledge that the system that produces these disparities needs to change.

In particular, it's easy to use language that suggests that some people are natural programmers and others aren't, but Guzdial has called this the biggest myth about teaching computer science[5]. [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*

[4] https://www.psychologytoday.com/blog/rabble-rouser/201512/is-stereotype-threat-overcooked-overstated-and-oversold

[5] http://cacm.acm.org/blogs/blog-cacm/189498-top-10-myths-about-teaching-computer-science/fulltext

categorize the distributions. A random half of participants were primed to think about the fact that CS grades are commonly thought to be bimodal; these participants were more likely to label ambiguous distributions as "bimodal". Participants were also more likely to label distributions as bimodal if they believed that some students are innately predisposed to do better at CS. These results suggest that bimodal grades are instructional folklore in CS, caused by confirmation bias and instructor beliefs about their students.

10.4 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[6] 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.

10.5 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[7] 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[8] (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

[6] https://educhatter.wordpress.com/2017/03/26/growth-mindset-is-the-theory-flawed-or-has-gm-been-debased-in-the-classroom/

[7] https://en.wikipedia.org/wiki/Screen_reader

[8] https://en.wikipedia.org/wiki/Curb_cut

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*[9] (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. The W3C Accessibility Initiative's checklist for presentations[10] is a good starting point focused primarily on assisting the visually impaired, while Liz Henry's blog post about accessibility at conferences[11] has a good checklist for people with mobility issues, and this interview with Chad Taylor[12] is a good introduction to issues faced by the hearing impaired.

Know how well you're doing. For example, sites like WebAIM[13] 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.

---

[9] https://en.wikipedia.org/wiki/Nothing_About_Us_Without_Us

[10] https://www.w3.org/WAI/training/accessible

[11] https://modelviewculture.com/pieces/unlocking-the-invisible-elevator-accessibility-at-tech-conferences

[12] https://modelviewculture.com/pieces/qa-making-tech-events-accessible-to-the-deaf-community

[13] http://webaim.org/

Conduct Revisited

We said in Sections 1.3 and 9.1 that classes should enforce a Code of Conduct like the one in Appendix B. In a sense, 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.*

10.6 Inclusivity

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

Ask learners to email you before the workshop to explain how they believe the training could help them achieve their goals.

Review 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.

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 to bridge computing and African-American cultural capital in computing education. Community representation enrolls cultural capital to highlight students' social identities, histories, and community networks. This can take the form of after-school mentors or role models who support computer literacies and are from students' neighborhoods, or of activities that use community narratives and histories as a foundation for a computing project. Computational integration situates computing in culturally situated practices and local sources of wealth generation; examples include "heritage algorithms" that allow indigenous and vernacular designs to be reverse engineered in a visual programming environment or the use of remix practices from hip-hop culture on websites where users post and share their computational artifacts.

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.

10.7 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, but as Benjamin Doxtdator has pointed out[14], those claims are built on shaky ground. Even if they were true, the kind of education I'm interested in won't prepare people for the jobs of the future: it will empower them to create jobs that are worth doing. And as Mark Guzdial has pointed out[15], there are lots of other reasons to learn 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. To use computers better.
7. As a medium in which to learn problem-solving.

Section 16 explains why I teach; in part, what motivates me 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.

[14] http://www.longviewoneducation.org/field-guide-jobs-dont-exist-yet/

[15] https://computinged.wordpress.com/2017/10/18/why-should-we-teach-programming-hint-its-not-to-learn-problem-solving/

10.8 Challenges

Authentic Tasks (pairs/15 minutes)

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 2×2 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 minutes)

Paloma Medina identifies six core needs[16] 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 minutes)

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 minutes)

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.

[16] https://www.palomamedina.com/biceps

Demotivational Experiences (think-pair-share/15 minutes)

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 minutes)

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

Who Decides? (whole class/15 minutes)

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

Credibility (individual/15 minutes)

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

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

Trustworthiness , or 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 , or 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.

Common Stereotypes (pairs/10 minutes)

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.

[17] https://mobile.nytimes.com/2016/04/10/upshot/why-talented-black-and-hispanic-students-can-go-undiscovered.html

Gender-Neutral Writing (pairs/10 minutes)

This short article[18] by Jean Hollis Weber presents half a dozen tips for making technical writing gender neutral. Take a lesson you have recently taught or been taught, go through it with a partner, and see how many small changes you would make to conform to these rules.

Saving Face (individual/10 minutes)

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?

Why Learn to Program? (individual/20 minutes)

Read Guzdial's list of reasons to learn to program[19], then draw a 3×3 grid whose X and Y 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 your learners (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?

How It Goes Wrong (small groups/15 minutes)

The Center for Community Organizations[20] created a graphical representation[21] of what many women of color experience working in the nonprofit sector. Working in small groups, go through it and discuss which parts of it you and your colleagues have experienced or have seen occur in the groups and organizations you work with.

After the Fact (whole class/15 minutes)

[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.

[18] https://techwhirl.com/gender-neutral-technical-writing/

[19] https://computinged.wordpress.com/2017/10/18/why-should-we-teach-programming-hint-its-not-to-learn-problem-solving/

[20] https://coco-net.org/problem-woman-colour-nonprofit-organizations/

[21] https://coco-net.org/wp-content/uploads/2018/03/WoC-in-Organizations-Tool-FINAL-EN.pdf

Counting Failures (pairs/15 minutes)

Any useful estimate of how long something takes to master must take into account how frequent failures are and how much time is lost to them. For example, editing 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 gone wrong in classes you have taught 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?

Tracing the Cycle (small groups/15 minutes)

[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.

Inclusivity Strategies (small groups/15 minutes)

[Tann2013] enumerates twenty-one teaching strategies to promote student engagement and cultivate inclusivity in classes:

- Give learners opportunities to think and talk about the subject.

  1. Wait a few seconds after asking a question before selecting someone to answer it.
  2. Allow learners time to write.
  3. Use think-pair-share (Section 9.10).
  4. Do not try to do too much.

- Encourage, demand, and actively manage the participation of all learners.

  5. Hand raising.
  6. Multiple hands, multiple voices.
  7. Call on learners randomly.
  8. Assign reporters for small groups.
  9. Whip around.
  10. Monitor learner participation.

- Build an inclusive and fair community.

  11. Learn or have access to learners' names.
  12. Integrate culturally diverse and relevant examples.
  13. Work in stations or small groups.
  14. Use varied active-learning strategies.

15. Be explicit about promoting access and equity for all learners.

- Cultivate divergent thinking.

16. Ask open-ended questions.
17. Do not judge responses.
18. Use praise with caution.
19. Establish classroom community norms.

- Teach all of the learners in your classroom.

20. Teach them from the moment they arrive.
21. Collect assessment evidence from every learner, every class.

Working in small groups, collect examples of specific cases in which you have seen these recommendations violated. Are there cases where the members of your group disagree on whether a rule was broken or not?

11 Teaching Online

Objectives

- Explain why expectations for massive online courses were unrealistic.
- Explain what personalized learning is, and how the term is sometimes misused.
- Describe several key practices of successful automated courses.
- Summarize at least four features that make instructional videos engaging.
- Describe the pros and cons of using off-the-shelf style checking tools to give students feedback on their programs.

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[1] 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" are two different things. 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.

[1] http://teachingmachin.es/timeline.html

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 decentralized, but the term was quickly co-opted by creators of centralized, automated, video-based courses. The former are now sometimes referred to as "cMOOCs" to distinguish them from the less threatening "xMOOCs" offered by institutions that see the lack of a grading scheme as the first step toward anarchy in the streets. (The latter type of course is also sometimes called a "MESS", for Massively Enhanced Sage on the Stage.)

Two strength 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—or 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 their instructional 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[2] *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[3] *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 far more information, far more quickly, than ever before*—provided, of course, that a search engine considers it worth indexing, that their internet service provider and government don't block it, and that the truth isn't drowned in a sea of attention-sapping disinformation.
- *Learners can access far more people than ever before as well*—again, provided that they aren't driven offline by harassment or marginalized because they don't conform to the social norms of whichever group is talking loudest.
- *Courses can reach far more learners than before too*—but only if those learners actually have access to the required technology, can afford to use it, and aren't being used as a way to redistribute wealth from the have-nots to the haves [McMi2017].
- *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 aren't in a position to object to the use of ubiquitous surveillance in the classroom.

[Marg2015, Mill2016a, Nils2017] describe ways to take advantage of 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 offline 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 9.5), serve as classroom scribes, or contribute problems to shared problem sets (Section 5.3).

[2] https://www.rand.org/pubs/research_briefs/RB9994.html
[3] http://educationnext.org/in-schools-teacher-quality-matters-most-coleman/

- Encourage or require learners to do some of their work in small groups (2–6 people) 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.

- Create, publicize, and enforce a code of conduct so that everyone can actually (as opposed to theoretically) take part in online discussions (Section 1.3).

- 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.

- Remember that, disabilities aside, learners can read faster than you can talk, so use video to engage rather than instruct. 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 that screenshots with text.

- Remember that the goal when teaching novices is to identify and clear up misconceptions (Chapter 2). If early data shows that learners are struggling with some parts of a lesson, create extra 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 (LMS) like Moodle[4] or Sakai[5], or assemble something yourself using Slack[6] or Zulip[7] for chat, Google Hangouts[8] or appear.in[9] for video conversations, and WordPress[10], Google Docs[11], 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[12], and find its arcane conventions and interface offputting. Similarly, while this book lives in a GitHub[13] 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.

[4]http://moodle.org
[5]https://www.sakaiproject.org/
[6]http://slack.com
[7]https://zulipchat.com/
[8]http://hangouts.google.com
[9]https://appear.in/
[10]https://wordpress.org/
[11]http://docs.google.com
[12]https://en.wikipedia.org/wiki/Internet_Relay_Chat
[13]http://github.com

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 your lecture (Section 9.5); in online courses, you can put your nodes 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 the 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[14]" 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 lets those who have something to say 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, 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, DI scripts 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

[14] https://en.wikipedia.org/wiki/Two_Concepts_of_Liberty

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."

[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], 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.

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.

If you are teaching programming, you will often use screencasts instead of slides, since they have many of the same advantages as live coding (Section 8.3). [Chen2009] offers useful tips for creating and critiquing screencasts and other videos. The figure below shows the patterns they present and the relationships between them.

11.3 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 autograding.

[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 are 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 common and 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

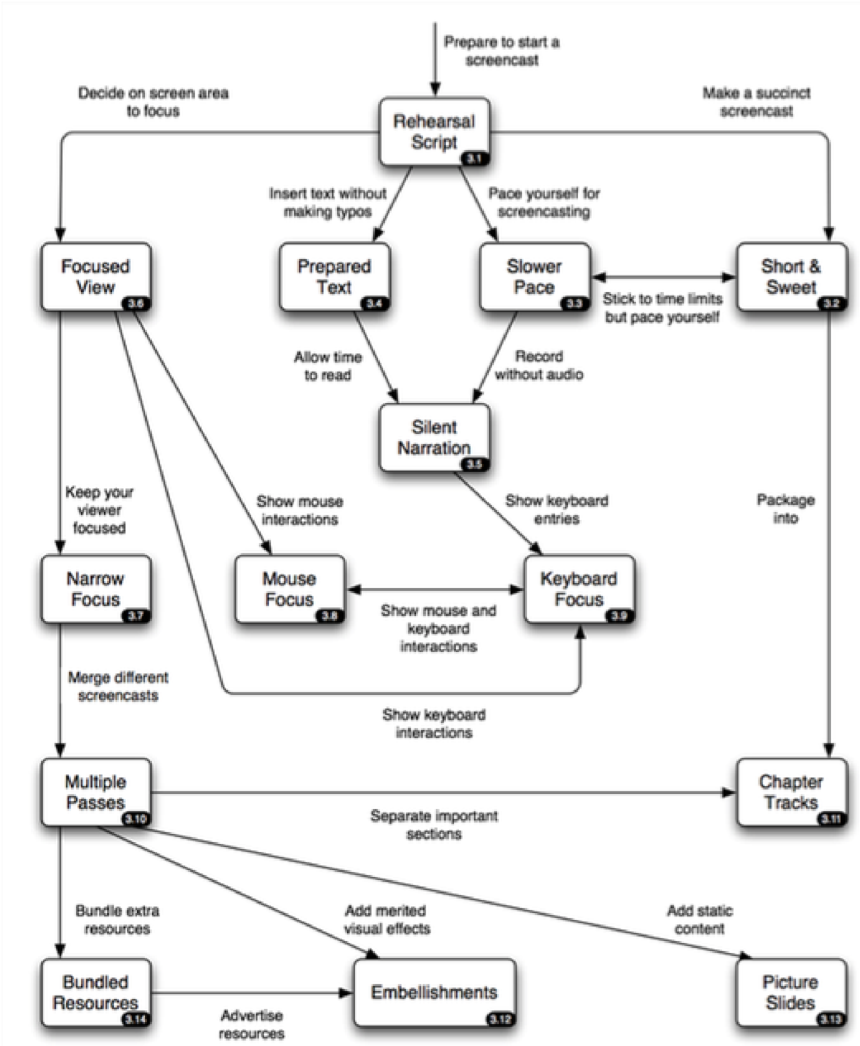


Figure 11.1: Patterns for Screencasting from [Chen2009]

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. The authors modified the autograder's rules to reflect this, and then weighted infrequent and overly-frequent violations of a particular feature. Unsurprisingly, after all these tweaks there was a stronger positive correlation with manual assessment.

[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 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 autograding 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; as with many workflow tools, they tend to bake in their creators' unconscious or 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 provide hashed output, so learners can tell if their output is correct without knowing what the output is unless they reproduce it. This requires a bit more work and explanation, but strikes a good balance between revealing answers prematurely and not revealing them when it would help.

11.4 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, exercises have to be more detailed because they're anticipating questions: teaching live is the intersection of what everyone needs to know (expanded on demand), while teaching online is the union of what everyone needs to know (because you can't).

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 9.7), 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.

11.5 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 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 final 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. While the discussion in Chapter 11.3 assumed that grading had to be fully automatic in order to scale to large classes, that doesn't have to be the case. [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 location in coding assignment (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 that 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 can be equally effective [Kulk2013]. 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 8.3, and when combined with collaborative note-taking (Section 9.5) it can come pretty close to approximating an in-class experience.

Looking even further ahead, [Ijss2000] identified four levels of online presence, from realism (where we can't tell the difference) through immersion (we forget the difference) and involvement (we're engaged, but aware that it's online) to suspension of disbelief (where the participant is 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

Just as combining online and in-person instruction can be more effective than either on its own, combining online and in-person presence can outperform either. 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. 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

11.6 Challenges

Give Feedback (whole class/20 minutes)

Watch this screencast[15] 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 2×2 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 Appendix I.)

Classifying Online Coding Lessons (individual/15 minutes)

Use this derivative of [Kim2017]'s rubric for evaluating online coding tutorials to classify your favorite online coding tutorial:

Personalization: age, educational status, coding experience.

Utilization: application to authentic tasks, pointers to subsequent knowledge.

Content: variables, arithmetic, logical, conditionals, loops, arrays, functions, objects.

Organization: bottom-up, need-driven.

Context: lecture-based, project-based, storyline.

Actionability: learners write code.

Feedback: output correctness, code structure, code style.

Transfer: how to use, when to use, why to use.

Support: additional materials for self-monitoring.

Two-Way Video (pairs/10 minutes)

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 minutes)

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.

---

[15] https://youtu.be/xcnoHaxXvdQ

Computer Science: technological implementation.

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

12 Exercise Types

Objectives

- Describe four types of formative assessment exercises for programming classes.

Every good carpenter has a set of screwdrivers, and every good teacher has different kinds of formative assessment exercises to check what her 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 worked and your learners are ready to move on, or whether you need to go back and re-explain things. It then explores 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[1] [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)`?*
>
> *a) subtraction, function call, assignment*
> *b) function call, subtraction, assignment*
> *c) function call, then assignment and subtraction simultaneously*
> *d) 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.

[1] http://web-cat.org/questionbank/

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 C&R

You are in the directory `/home/greg`. Which of the following files is not *in that directory?*

a) `autumn.csv`
b) `fall.csv`
c) `spring.csv`
d) `winter.csv`

C&Rs help learners practice the skills they most want to learn, but they can be hard to assess: learners can find very creative ways to get the right answer, and it's demoralizing to give them a false negative. One way to reduce how often this occurs 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're 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. Doing this helps learners realize just how hard it is to write comprehensive tests, which in turn may give them a bit more sympathy for how hard their teachers work.

Inverting C&R

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.) 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 calculate the sums of the positive and negative values in a list. (You will need to add colons in appropriate places as well.)

```
positive = 0
return positive
if v > 0
positive += 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 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.

```
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 sets the upper left triangle of the image to the color provided.

```
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 challenge 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.

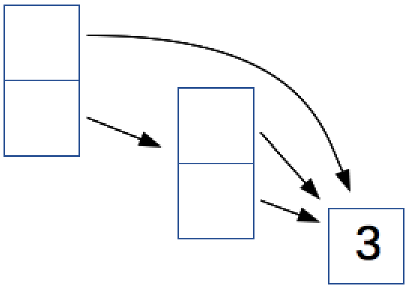
```
result = []
for v in values:
    if len(v) > threshold:
        result.append(v)
```

12.3 Diagrams

Having students draw concept maps and other diagrams gives insight into how they're thinking (Section 3.2), 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



Label the diagram above to show which structures the variables x, y, and z refer to after executing:

```
x = 3
y = [x, x]
z = [x, y]
```

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 are 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 Higher-Level Thinking

Most other popular programming exercises are hard for teachers to give feedback on in a class with more than half a 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 8.3) 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.

Code Review

Using the rubric provided, mark each line of the code below.

```
def addem(f):
    x1 = open(f).readlines()
    x2 = [x for x in x1 if x.strip()]
    changes = 0
    for v in x2:
        print('total', total)
        tot = tot + int(v)
    print('total')
```

1. *poor variable name*
2. *unused variable*
3. *use of undefined variable*
4. *missing return value*

12.5 Challenges

Code & Run (pairs/10 minutes)

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 C&R (small groups/15 minutes)

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 minutes)

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 minutes)

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 minutes)

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 minutes)

[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.

Part IV

Organizing

13 Building Community

Objectives

- Explain what situated learning is and identify its key elements.
- Explain how to decide whether to try to create a new community or join an existing effort.
- Outline a three-step plan for recruiting, retaining, and retiring volunteers and other organization participants.
- Explain the difference between a service board and a governance board, and judge which kind an organization has.
- Judge whether a meeting is well organized and well run or not.
- Describe the key steps in conducting a post mortem review, and the key outcomes to expect from doing one.

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. If you choose the latter, 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[1], you have to recognize that some things can't be fixed piecemeal. 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. Countries that try to adopt their teaching methods without also ensuring that children (and adults) are well nourished, safe, and treated fairly by the courts are doomed to fail. In my experience, people who try to teach programming without also tackling these issues are doomed to fail as well.

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].

[1] https://www.theatlantic.com/national/archive/2011/12/what-americans-keep-ignoring-about-finlands-school-success/250564/

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 challenges 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[2]:

[2] https://www.feverbee.com/types-of-community-and-activity-within-the-community/

Community of action: people focused on a shared goal, such as getting someone elected.

Community of concern: members are brought together by a shared challenge, 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[3] and literacy programs[4] to get-into-coding non-profits like Black Girls Code[5]. 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. The only thing it *won't* give you is the ego gratification and control that comes from being a founder.

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.

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.)

[3] http://www.4-h-canada.ca/
[4] https://www.frontiercollege.ca/
[5] http://www.blackgirlscode.com/

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[6] 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 its own section. 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.

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.

---

[6] https://en.wikipedia.org/wiki/Ben_Franklin_effect

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.

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.

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. The community can continue to be "hub and spoke", 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***
>
> *Have you ever spoken to someone from the public relations office at your local college or school board, or in your city councillor's office? If not, what are you waiting for?*

> ***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 things—even small things—people will be more confident that it's worth thinking about bigger issues.*

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. This is often a difficult transition for grassroots organizations, but is outside the scope of this book.

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. The most important thing to keep in mind is that countries and corporations are only two of many governance models, and that a commons is often a better model for volunteer teaching organizations. A commons 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. Both have had tragic consequences for us individually and communally, and now for our whole planet.

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 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 also help you keep the meeting moving (as in, "That's very interesting, but we have five other topics to get through in the next fifteen minutes, so could you please make your point?"), so 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.

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.

Put someone in charge. "In charge" means keeping the meeting moving, glaring at people who are muttering to one another or checking email, and telling people who are talking too much to get to the point. It does *not* mean doing all the talking; in fact, whoever is in charge will usually talk less than anyone else, just as a referee usually kicks the ball less often than the players.

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. 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. Be sure to make it clear that assistive technology is exempt from this rule.

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.

13.6 The Post Mortem

The most valuable part of your project isn't the software you write, or the grade you're given. It's the project's post mortem. Literally, this is an examination of a deceased person; in a software project, it's a look back at what went right, and what went wrong.

The aim of a post mortem is to help the team and its members do better next time by giving everyone a chance to reflect on what they've just accomplished. It 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.

Post mortems are pretty easy to run—just add the following to the rules for running a meeting:

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 (if the course is too large, or the instructor is lazy) 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 "Good" and "Bad" 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).

13.7 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.

13.8 Challenges

Several of these exercises are taken from [Brow2007], which is an exceptionally useful book on building community organizations.

What Kind of Community? (individual/15 minutes)

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 minutes)

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 as casually as minor characters in Tarantino films.

Values (small groups/45 minutes)

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 minutes)

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 minutes)

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 minutes)

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 minutes)

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 minutes)

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 minutes)

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[7]) is a good guide to what has and hasn't worked for those communities, and the Open Source Guides site[8] 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 minutes)

The National Oceanic and Atmospheric Administration (NOAA) has published a short, amusing, and above all useful guide to dealing with disruptive behaviors[9]. 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?

---

[7] http://producingoss.com/
[8] https://opensource.guide/
[9] https://coast.noaa.gov/ddb/story_html5.html

14 Marketing

Objectives

- Explain what marketing actually is.
- Explain the value of what they are offering to different potential stakeholders.
- State what a brand is and what their organization's is.

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. There's a lot of folklore about how to do this 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[1] and do what you can to improve your site. Keep

[1] https://moz.com/learn/seo/on-page-factors

this cartoon[2] 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. . . " helpers 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!

[2] https://xkcd.com/773/

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 is puts all of these points in action. It has worked pretty well, but "pretty well" is relative: we found that about half of emails were answered, about half of those answers were, "Sure, let's talk more," and about half of those led to workshops, which means that 10–15% of targeted emails to people we had some sort of connection with 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, think about doing something else entirely. This isn't as defeatist as it sounds: 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 Challenges

Write an Elevator Pitch for a City Councillor (individual/10 minutes)

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 minutes)

Identify two groups of people your organization needs support from, and write an elevator pitch aimed at each one.

Identify Causes of Passive Resistance (small groups/30 minutes)

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 minutes)

Revisit the "Why Learn to Program?" challenge in Section 10.8. Where do your reasons for teaching and your learners' reasons for learning align? Where are they not aligned? How does that affect your marketing?

Email Subjects (pairs/10 minutes)

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.

15 Partnerships

Objectives

- Explain why teachers in schools and universities do and don't adopt better teaching practices.
- Summarize methods that can be used to effect changes in educational institutions.

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[1] 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[2]. 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

[1] http://carpentries.org

[2] https://www.insidehighered.com/news/2017/07/06/anthropologist-studies-why-professors-dont-adopt-innovative-teaching-methods

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 changes 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 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 meed 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:

Individuals/ Prescribed	**I. Disseminating: Curriculum & Pedagogy** **Change Agent Role**: tell/teach individuals about new teaching conceptions and/or practices and encourage their use. **Methods**: Diffusion, Implementation.
Individuals/ Emergent	**II. Developing: Reflective Teachers** **Change Agent Role**: Encourage/support individuals to develop new teaching conceptions and/or practices. **Methods**: Scholarly Teaching, Faculty Learning Communities.
System/ Prescribed	**III. Enacting: Policy** **Change Agent Role**: Enact new environment features that require/encourage new teaching conceptions and/or practices. **Methods**: Quality Assurance, Organizational Development.
System/ Emergent	IV. Developing: Shared Vision **Change Agent Role**: Empower/support stakeholders to collectively develop new environmental features that encourage new teaching conceptions and/or practices. **Methods**: Learning Organizations, Complexity Leadership

The paper goes on to explain each of the methods in detail (and [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 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 under-represented 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 to get allies, you need tactics. The most useful guide I have found is [Mann2015], which catalogs more than four dozens 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 challenges 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 the 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 Challenges

Collaborations (small groups/30 minutes)

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 minutes)

[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 minutes)

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 minutes)

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?

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 still practiced slavery. A hundred years ago, my grandmother wasn't legally a person[1] 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[2]", George Orwell wrote:

[1] https://en.wikipedia.org/wiki/The_Famous_Five_(Canada)
[2] http://www.resort.com/~prime8/Orwell/whywrite.html

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.

[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 to present information.

[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, who inspired many other social change organizations.

[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. An empirical study of how Scratch is used based on analysis of 250,000 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 between variant domain-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 novices 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 142 novice programmers.

[Alta2015] Amjad Altadmri and Neil C.C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proc. 2015 Technical Symposium on Computer Science Education (SIGCSE'15)*. Association for Computing Machinery (ACM), 2015. Summarizes the types of errors made by thousands of novice Java 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. An excellent overview of what we know about education and why we believe it's true, covering everything 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 landmark 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 computer science 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 on surgical checklist implementations and effects 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. Used eye tracking to show that developers do read error messages, that doing so is as hard as reading source code, and that it takes 13-25% of total task time.

[Bark2005] Lecia J. Barker, Kathy Garvin-Doxas, and Eric Roberts. What can computer science learn from a fine arts approach to teaching? In *Proc. 2005 Technical Symposium on Computer Science Education (SIGCSE'05)*, pages 421–425. Association for Computing Machinery (ACM), 2005. Looks at techniques that computing education can learn from fine arts education.

[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. Data from a large survey of US institutions finds that meaningful assignments, faculty interaction with students, student collaboration on assignments, and (for male students) pace and workload relative to expectations drive retention, but interactions with teaching assistants or with peers in extracurricular activities don't.

[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 findings from a two-part study of how computer science educators adopt new teaching practices.

[Barn2002] Susan M. Barnett and Stephen J. Ceci. When and where do we apply what we learn?: A taxonomy for far transfer. *Psychological Bulletin*, 128(4):612–637, 2002. Identifies nine different dimensions of "far transfer" and shows that discussion of its existence is muddied by confusion among them.

[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 work on 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 called OK in which students have to unlock test cases for their code by correctly answering MCQs, and presents some 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. Found that learning outcomes were improved by two-stage projects (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. A three-year study found that cooperative learning enhances learning outcomes and self-efficacy in CS1.

[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. Found that improved error messages helped novices learn faster.

[Beck2018b] Brett A. Becker, Kyle Goslin, and Graham Glanville. The effects of enhanced compiler error messages on a syntax error debugging test. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Had students remove syntax errors from code they did not write, and found a positive effect on number of errors fixed, but no overall effect on the number of non-compiling submissions or student scores, which may reconcile apparently contradictory results of earlier studies, and cautions that effects are weak.

[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. An empirical study showing 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.

[Benk2005] Yochai Benkler. Common wisdom: Peer production of educational materials. http://www.benkler.org/Common_Wisdom.pdf, 2005. An influential early analysis of commons-based peer production that seems optimistic in retrospect.

[Benn2000] Patricia Benner. *From Novice to Expert: Excellence and Power in Clinical Nursing Practice*. Pearson, 2000. A classic study of clinical judgment and how expertise develops.

[Benn2007] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32, Jun 2007. A multi-site survey that found 67% of students pass CS1, with wide variation (5% to 100%).

[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. Showed that training learners in self-explanation accelerated 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. A series of studies found that reading and understanding code is fundamentally different from reading prose, 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. An attempt to categorize novice programmer behavior using machine learning that was largely unsuccessful at the course level, but found interesting patterns on individual assignments.

[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. Presents a study showing 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 one of the most widely used kinds of governance in human societies throughout history.

[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.

[Bouv2016] Dennis Bouvier, Ellie Lovellette, John Matta, Bedour Alshaigy, Brett A. Becker, Michelle Craig, Jana Jackova, Robert McCartney, Kate Sanders, and Mark Zarb. Novice programmers and the problem description effect. In *Proc. 2016 Conference on Innovation and Technology in Computer Science Education (ITiCSE'16)*, pages 103–118. Association for Computing Machinery (ACM), 2016. A large study that found that providing context for problems didn't make a difference to outcomes in introductory computing courses, but did affect motivation.

[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 (with unit tests) and unit tests (using programs with deliberate errors) and finds that students often write weak tests (low coverage) and misunderstand the role of unit tests.

[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 of students on their later 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. An excellent practical introduction to creating effective organizations in and for communities written by someone with decades of experience doing exactly that.

[Brow2014] Neil C.C. Brown and Amjad Altadmri. Investigating novice programming mistakes. In *Proc. 2014 International Computing Education Research Conference (ICER'14)*. Association for Computing Machinery (ACM), 2014. Uses data from over 100,000 students to show that educators know less than they think about what mistakes novice programmers actually make.

[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. Explain a framework for identifying whether a student has adequately tested a specific feature of their code that is failing an instructor's tests, analyzes an auto-grader's feedback using the framework, and finds that false positives (revealing unearned hints) were roughly four times more common than true positives (earned hints revealed), while earned hints were no more likely to be revealed than they were to be obscured.

[Burg2015] Sheryl E. Burgstahler. *Universal Design in Higher Education: From Principles to Practice*. Harvard Education Press, second edition, 2015. A detailed description of 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. A preliminary comparison of the skills and credentials that tech industry recruiters are looking for to those provided by 4-year degrees and bootcamps that found hiring managers uniformly emphasize soft skills (especially the ability to continue learning and teamwork/communication), and that while they often require 4-year degree (but not necessarily CS), they also praise bootcamp graduates for being more mature and having up-to-date knowledge and hands-on skills.

[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 a set of pencil-and-paper puzzles that can be turned into CS1/CS2 assignments, and reports a study showing that these assignments 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. Compared students who opted in to anonline CS1 class online with those who took it in person in a flipped classroom, and found that completion of unmarked practice exercises correlated with exam scores for both (but completion rate of rehearsal exercises by online students was significantly lower than lecture attendance for in-person), that test anxiety and self-efficacy were significant factors only for the online section, and that intrinsic value was a factor only for the flipped section (once results were controlled for prior programming experience).

[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. Found significant short-term gains for students doing exams collaboratively, but not long-term gains.

[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. Found that collaborative exams benefited middling students more than high or low-performing students, and that homogeneous groups benefited more than heterogeneous groups.

[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 for 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 manuals.

[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. Builds on earlier work to show that students of different levels approach programming tasks differently, and 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 the SOAR model of human cognition to three learning theories (cognitive load, cognitive apprenticeship, and worked examples), and from there to the design of an introductory object-oriented programming course using worked examples, scaffolding, faded guidance, cognitive apprenticeship, and an emphasis on patterns to aid schema creation and improve learning.

[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. Compared classical pair programming (two coders, one machine) with side-by-side (two coders, two machines) and found that the former had the same learning gains 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. Another study showing that people learn more from constructing visualizations of algorithms 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. Describes exercises in which students reconstruct mangled code, and shows that student performance on these 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. Found that developers draw diagrams more often to aid discussion than to document designs (though that may be an artifact of the cost of translating whiteboards into permanent storage).

[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. Four studies showing that subtle environmental clues have a measurable difference 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. The latest in a long series of studies showing that good teachers have a measurable but small impact on 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.

[Chun2004] Andy Hon Wai Chun. The agile teaching/learning methodology and its e-learning platform. In *Proc. 2004 International Conference on Web-Based Learning (ICWL'04)*, pages 11–18. Springer, 2004. Describes an effort to adapt the ideas of agile software development to teaching and an online platform to support it, the results of which look very much like what good teachers using classical methods do anyway.

[Coco2018] Center for Community Organizations. The "problem" woman of colour in the workplace. https://coco-net.org/problem-woman-colour-nonprofit-organizations/, 2018. A diagram explaining the experience of many women of color in the worksplace.

[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.

[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 evidence of benefit from the first ten years of peer instruction in undergraduate physics classes, and describes some ways in which the implementation of peer instruction changed during that time.

[Crou2004] Catherine Crouch, Adam P. Fagen, J. Paul Callan, and Eric Mazur. Classroom demonstrations: Learning tools or entertainment? *American Journal of Physics*, 72(6):835–838, 2004. Found that learners who observe a demonstration do not learn better than those who did not see the demonstration.

[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, and reports that high- and low-performing students are more likely to share their work with tags than middle-performing students.

[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. Replicates an earlier study of how students sketch when tracing code execution, found that writing new values near variables' names as they change is the most effective technique (reflecting a notional machine in which old values become unavailable).

[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 two childhood activities: reading on one's own, and playing with construction toys with no moving parts like Lego.

[DSou2015] Manoj Joseph D'Souza and Paul Rodrigues. Extreme pedagogy: An agile teaching-learning methodology for engineering education. *Indian Journal of Science and Technology*, 8(9):828, May 2015. Another attempt apply agile development to teaching that doesn't recommend anything good teachers don't already do.

[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 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 a study showing that doing in-class exercises with realtime feedback using mobile devices improved concept retention and student engagement while reducing failure rates.

[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 the Glitch game testing program went on to study computing in part because the gaming aspects enabled them to save face with their peers.

[Dida2016] David Didau and Nick Rose. *What Every Teacher Needs to Know About Psychology*. John Catt Educational, 2016. An informative, opinionated survey 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. Introduced the idea of a notional machine.

[Duva2018] Shannon Duvall, Dugald Ralph Hutchings, and Robert C. Duvall. Scrumage: A method for incorporating multiple, simultaneous pedagogical styles in the classroom. In *Proc. 2018 Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery (ACM), 2018. Describes an adaptation of Scrum practices to teaching.

[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 (essentially backoff with a ceiling).

[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. Describes an experiment using 101 CS2 programs in which students wrote their own software tests and graded in part on the branch coverage, and found that while students achieved an average branch coverage of 95.4% on their own solutions, their test suites only detected an average of 13.6% of the faults present in the entire program population, and there was a high degree of similarity among 90% of the student test suites, indicating that students wrote tests for the happy path rather than to detect hidden bugs.

[Edwa2017] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. Investigating static analysis errors in student java programs. In *Proc. 2017 Conference on International Computing Education Research (ICER'17)*. Association for Computing Machinery (ACM), 2017. Describes patterns found in static analysis reports of student code submissions, finding that coding errors at any point during development correlate with lower grades in the final submission.

[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, in particular by preventing questions about what we have and and what we can do with it.

[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 Ericson, Steven Moore, Briana 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. Found that learners were more likely to attempt Parsons Problems than nearby multiple choice questions in an ebook.

[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. Found that solving 2D Parsons problems with distractors took less time than writing or fixing code, and had equivalent learning performance and retention.

[Farm2006] Eugene Farmer. The gatekeeper's guide, or how to kill a tool. *IEEE Software*, 23(6):12–13, Nov 2006. Ten tongue-in-cheeck 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.

[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. Found that students made fewer low-level errors when solving Soloway's 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. A study of 21 undergraduate programmers found that they could find 70% of bugs and of those fix 97%, that good debuggers were good programmers but not necessarily vice versa, and that novices use forward reasoning (tracing and testing) rather than causal reasoning.

[Floy2017] Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *Proc. 2017 International Conference on Software Engineering (ICSE'17)*. Institute of Electrical and Electronics Engineers (IEEE), May 2017. Reports an fMRI study of 29 people that found that the same brain regions are involved in reading code and prose, and the two are distinct activities that become more similar as people develop expertise.

[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. A study showing that children learn to read faster when taught letter-sound correspondences (phonics) 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. Found five barriers to contribution on Stack Overflow that are seen as significantly more problematic by women than by men: 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.

[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. 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.

[Free1972] Jo Freeman. The tyranny of structurelessness. *The Second Wave*, 2(1), 1972. Pointed out that every organization has a power structure: the only question is whether it's accountable or not.

[Frie2016] Marilyn Friend and Lynne Cook. *Interactions: Collaboration Skills for School Professionals*. Pearson, eighth edition, 2016. A standard textbook on how teachers can work with other teachers.

[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. Finds that static typing would catch about 15% of publicly-reported 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 a trio of studies showing that gendered wording commonly employed 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 wide variety of fields.

[Gelm2002] Andrew Gelman and Deborah Nolan. *Teaching Statistics: A Bag of Tricks*. Oxford University Press, 2002. A collection of useful motivating 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. Found 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 the best practices for providing instructional feedback, and recommends specific strategies for providing feedback.

[Gree2014] Elizabeth Green. *Building a Better Teacher: How Teaching Works (and How to Teach It to Everyone)*. W. W. Norton & Company, 2014. A well-written look at why educational reforms in the past 50 years have mostly missed the mark, and what we should be doing instead.

[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 a study of 100 middle-school children using blocks-based programming that finds that loops, variables, and Boolean operators are 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. Measures engagement by how long learners watch MOOC videos and finds that shorter videos are more engaging, that talking heads are more engaging than tablet drawings, and makes recommendations for future videos.

[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 10 years of media computation research.

[Guzd2015a] Mark Guzdial. *Learner-Centered Design of Computing Education: Research on Computing for Everyone*. Morgan & Claypool Publishers, 2015. An evidence-based argument 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. Itemizes ten things that 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 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. Finds increased success rates and retention, and some evidence that pairing is particularly beneficial for women, but 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 E. Hannay, Erik Arisholm, Harald Engvik, and Dag I.K. Sjoberg. Effects of personality on pair programming. *IEEE Transactions on Software Engineering*, 36(1):61–80, Jan 2010. Found weak correlation between the "Big Five" personality traits and performance in pair programming.

[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 while increasing the time spent solving them.

[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)*. Association 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 comprehensive 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 actually 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; very British in its focus, but contains many gems.

[Herm2016] Felienne Hermans and Efthimia Aivaloglou. Do code smells hamper novice programming? a controlled experiment on scratch programs. In *Proc. 2016 International Conference on Program Comprehension (ICPC'16)*. Institute of Electrical and Electronics Engineers (IEEE), May 2016. A controlled experiment with 61 novice Scratch programmers found that students working with smelly code did worse overall, and that long methods decreased system understanding, while duplication made it harder for students to modify programs.

[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. Found that using words for variable names made comprehension 19% faster than using abbreviations or single-letter names for variables, but did not discriminate which variable names could or could not be shortened.

[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. Introducing a six-month program for existing high school teachers converting to teach CS quadrupled the number of teachers without noticeable reduction of student outcomes, but with an increase in the teachers' belief that anyone could program.

[Hugh2010] Michael C. Hughes, Matthew C. Jadud, and Ma. Mercedes T. Rodrigo. String formatting considered harmful for novice programmers. *Computer Science Education*, 20(3):201–228, Sep 2010. Found that 8% of compilation errors and 100% of runtime exceptions in novices' first Java programs were caused by string formatting problems exacerbated by poor error reports about non-literal errors.

[Hust2012] Therese Huston. *Teaching What You Don't Know.* Harvard University Press, 2012. A pointed, funny, and very useful book that explores exactly what the title suggests.

[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 which 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 presence (both real and virtual).

[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. Surveyed what we did and didn't know about the effects of teacher expectations on student outcomes at the time.

[Kaly2003] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. The expertise reversal effect. *Educational Psychologist*, 38(1):23–31, Mar 2003. Instructional techniques that work well with inexperienced learners lose their effectiveness or have negative consequences when used with more experienced learners.

[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.

[Karp2008] Jeffrey D. Karpicke and Henry L. Roediger. The critical importance of retrieval for learning. *Science*, 319(5865):966–968, Feb 2008. Found that repeated testing improved recall of word lists from 35% to 80%, even when learners still had access to the material (but were 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. Compares confidential peer ratings and grades on several axes for two undergraduate engineering courses, and found that self-rating and peer ratings agreed, that collusion wasn't significant, that students didn't inflate their self-ratings, or that ratings were biased by gender or race.

[Kaze2017] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. Quantifying incremental development practices and their relationship to procrastination. In *Proc. 2017 International Computing Education Research Conference (ICER'17)*. Association for Computing Machinery (ACM), 2017. Analyzed edit and execution data from an undergrad course and found earlier start time and earlier edit of tests correlated with higher scores and earlier submission times, but found no relationship with incremental testing or checking.

[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. Short and not very specific literature review of ways concept mapping could 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. Classifies feedback messages from 69 auto-grading tools and finds that they often do not give feedback on how to fix problems and take the next step, 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.

[Keun2017] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. In *Proc. 2017 Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)*. Association for Computing Machinery (ACM), 2017. Looked at student code quality using Blackbox data, and found that students usually don't fix issues, in particular issues related to modularization, and that use of tooling does not have much effect on the occurrence of issues.

[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. Found that online coding tutorials largely taught similar content, organized content bottom-up, and provided goal-directed practices with immediate feedback, but were not tailored to learners' prior coding knowledge and usually didn't inform learners how to transfer and apply learned 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.

[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—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.

[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. A small study of novice misconceptions about variables that finds they often believe in delayed evaluation or that entire equations are stored in variables.

[Koli2008] Yifat Ben-David Kolikant and M. Mussai. "so my program doesn't run!" definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education*, 18(2):135–151, Jun 2008. Reports an empirical study found that students view program correctness as the sum of the correctness of its constituent operations, and so think of programs as "partially correct" rather than incorrect if they have any correct operations (probably learned from grading schemes).

[Koll2015] Michael Kölling. Lessons from the design of three educational programming environments. *International Journal of People-Oriented Programming*, 4(1):5–32, Jan 2015. A detailed comparison of three generations of programming environments intended for novice use: Blue, BlueJ, and Greenfoot (including the Stride frame-based editor).

[Koll2016] Michael Kölling and Fraser McKay. Heuristic evaluation for novice programming systems. *ACM Transactions on Computing Education*, 16(3):1–30, Jun 2016. Proposes 13 heuristics to use to evaluate programming systems for novices and evaluates them using inter-expert comparison of Greenfoot (Java), Scratch, and Visual Basic.

[Kran2015] Steven G. Krantz. *How to Teach Mathematics*. American Mathematical Society, 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 looks at 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.

[Lawr2006] Dawn Lawrie, Christopher Morrell, Henry Feild, and Dave Binkley. What's in a name? a study of identifiers. In *Proc. 2006 International Conference on Program Comprehension (ICPC'06)*. Institute of Electrical and Electronics Engineers (IEEE), 2006. Found that full-word variable names are easier to understand in general, but in many cases, short variable names aren't an impediment.

[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. An independent study of Carroll's "minimal manual" approach that found that the minimalist approach outperformed the traditional approach 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. Interviewed 17 CS teachers to find out why they don't use resource sharing sites and recommended that sites have good landing pages (ask "what is your current role?" and "what course and grade level are you interested in?"), display all resources in the context of the author's full course (since visitors may be new teachers and struggle connect the dots themselves), and allow anonymous posts on discussion forums (to reduce fear of looking foolish in front of peers).

[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. Studied pair programming in a Grade 6 classroom and found that 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. Tested students' ability to predict the outcome of executing a short piece of code and their ability to select the correct completion for short pieces of code and found many were weak at both.

[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, 2004. A personal exploration of 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, and presents several case studies showing how their classification can be applied.

[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. Switched informal labs to a weekly machine-evaluated test using HackerRank and found the latter was the best predictor of exam scores; students didn't like the system, but overall failure rate halved and the number of students getting first class honors tripled.

[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 under-representation 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. Showed 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. Examined 76 MOOCs and found that instructional design quality was poor, but organization and presentation of material was 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. Describes three controlled experiments showing 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 behaviorists and cognitive psychologists differ in approach, but wind up making very similar recommendations about how to teach, and gives two specific examples (guided notes and response cards).

[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: closed vs. open, cultural relevance, recognition, space to play, driver shift, risk reward, grouping, and session shape.

[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. A literature survey of uses of Bloom's Taxonomy in CS education that finds that even experienced educators have trouble agreeing on the correct classification for a question or idea.

[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. Redesigned a conventional introduction to databases course based on cognitive load theory to remove split attention and redundancy effects, provide worked examples and sub-goals, and re-sequence content, and reduced exam failure rate by 34% on an identical exam 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 showing that 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 based in part on the author's own research.

[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 as of the time of publication about why bugs occur, why types occur, how people debug, and whether/how we can teach debugging skills.

[McCr2001] Michael McCracken, Tadeusz Wilusz, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, and Ian Utting. A multi-national, 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. An international study that found 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 this 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_White 2013. A short white paper summarizing the backward instructional design process.

[Merr2013] M. David Merrill. *First Principles of Instruction: Identifying and Designing Effective, Efficient and Engaging Instruction*. Pfeiffer, 2013. Describes how to apply learning science principles.

[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.

[Mill2014] Craig S. Miller. Metonymy and reference-point errors in novice programming. *Computer Science Education*, 24(2-3):123–152, Jul 2014. Reports two experiments showing that use of metonymy (referring to something structurally related to an actual subject) may explain some novice programming errors, and that students are more likely to produce reference-point errors that involve identifying attributes than descriptive attributes.

[Mill2016a] Michelle D. Miller. *Minds Online: Teaching Effectively with Technology*. Harvard University Press, 2016. Desribes 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. Study that found most students had difficulty with self in Python, either by omitting it in method definitions or by not using it when referencing object attributes, and that such errors were more common than other types of errors.

[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.

[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. Study of 207 K-12 students found fewer misconceptions about loops using Scratch than using Logo or Python, and fewer misconceptions about nested loops with Logo than with Python.

[Morr2015] Briana Morrison, Lauren E. Margulieux, and Mark Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *Proc. 2015 International Computing Education Research Conference (ICER'15)*, pages 21–29. Association for Computing Machinery (ACM), 2015. Reports some unexpected non-effects from labelled subgoals in computing education.

[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. Study found that students with labelled subgoals solved Parsons Problems for learning while loops 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 350 concept maps and compared those drawn by students with prior CS experience and those without, finding that OO concepts featured more prominently in the former and procedural concepts more prominently in the latter, and that while "program" was the central concept in both sets, "class" and "data structure" were next in the former, while "processor" and "data" were next in the latter.

[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. A study of 364 physics students found that including explicit discussion of misconceptions significantly improved learning outcomes, with students with low prior knowledge benefiting most without students with more prior knowledge being 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. Describes a multi-year study showing that explicitly teaching solution patterns improved 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. A study of 21 CS2 students found that many used good debugging strategies, but many others did not, and students often didn't recognize when they were stuck.

[NRC2015] National Research Council. *Reaching Students: What Research Says About Effective Instruction in Undergraduate Science and Engineering*. The National Academies Press, 2015. A summary of evidence-based findings in undergraduate STEM education.

[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.

[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 sequel to the first author's widely-regarded guide for college-level 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. Studies 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. Initially no correlation between human provided marks and style-checker rule violations, e.g. because students violated one rule many times or submitted skeleton code with few modifications, so altered autograder rules and weighted grades to get a stronger correlation with manual assessment.

[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 K12 misconceptions in mathematics and what to do about them.

[Oram2010] Andy Oram and Greg Wilson, editors. *Making Software: What Really Works and Why We Believe It*. O'Reilly Media, 2010. Explores the question posed in the title, drawing on educational research for answers.

[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. Finds that taking notes together online is more effective than solo note-taking.

[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. Using data from an intro web development course, 20% of student errors related to the rules dictating when HTML elements can be nested, while 35% related to the (simpler) syntax determining how elements are nested, and nearly all of the errors that were detected through validation were eventually corrected.

[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. Argued 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 descriptions 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, i.e., there is no geek gene.

[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.

[Pere2013] Santiago Perez De Rosso and Daniel Jackson. What’s wrong with git? a conceptual design analysis. In *Proc. 2013 International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!’13)*. Association for Computing Machinery (ACM), 2013. An analysis comparing Git’s actual operation with its users’ conceptual model, highlighting the errors and confusion that result from the differences.

[Pere2016] Santiago Perez De Rosso and Daniel Jackson. Purposes, concepts, misfits, and a redesign of git. In *Proc. 2016 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’16)*. Association for Computing Machinery (ACM), 2016. Follows up earlier work with a usability-based redesign of Git, tested by a user study, but finds that most people have either mastered Git and are defensive about its quirks, or think that its poor usability is so obvious as to not need belaboring.

[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.

[Port2011] Leo Porter, Cynthia Bailey Lee, Beth Simon, Quintin Cutts, and Daniel Zingaro. Experience report: A multi-classroom report on the value of peer instruction. In *Proc. 2011 Conference on Innovation and Technology in Computer Science Education (ITiCSE'11)*, pages 138–142, 2011. Studies the (positive) value of peer instruction in introductory computing classes.

[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. Large study showing 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 and places it in the framework of conceptual change theory.

[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. Studied 86 high school students and 48 high school teachers, found that only 45% of students understood when to use this, only 60% understood when not to, and only 24% could define it clearly (all these figures probably being overestimates because respondents could simply be reciting), and that teachers were also often not clear on these subjects.

[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.

[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.

[Repe2017] Alexander Repenning. Moving beyond syntax: Lessons from 20 years of blocks programing in AgentSheets. *Journal of Visual Languages and Sentient Systems*, 3(1):68–91, Jul 2017. Summarizes what the author has learned from building a blocks-based programming system.

[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. Reviews 100 articles to find learning trajectories for K-8 computing classes for Sequence, Repetition, and Conditions, presented diagrammatically.

[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.

[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 a study of how students use the Mooshask autograder (originally designed for programming competitions): students made some negative comments about it, which the authors attribute to Mooshak's feedback messages rather than to dislike of autograding, and there was no significant decrease in dropout rates of users.

[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. Meta-analyses 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 the authors' experience developing a shared question bank with entries for various languages and topics related to introductory CS, and 12 patterns for MCQs that emerged from entries.

[Scan1989] David A. Scanlan. Structured flowcharts outperform pseudocode: an experimental comparison. *IEEE Software*, 6(5):28–36, Sep 1989. A study that found 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.

[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 Soloway's 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 as measured by grades.

[Simo2006] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Commonsense computing: What students know before we teach (episode 1: Sorting). In *Proc. 2006 International Computing Education Research Conference (ICER'06)*. Association for Computing Machinery (ACM), 2006. Found that most CS1 students could describe a plausible algorithm to sort numbers in natural language on the first day of class, compared to only one third of general students, but that the CS students' ability decreased after ten weeks of instruction because they were trying too hard to include code-level details.

[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 Soloway's rainfall problem is harder for novices than it used to be because they're not used to handling keyboard input, so direct comparison 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 strength of the relationship between mind-set and academic achievement, and of effectiveness of mind-set interventions on academic achievement, and finds that overall effects were weak for both, 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 mind-set interventions.

[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 a study showing that student understanding actually does increase 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 (generic program fragments that represent stereotypical actions) and rules of programming discourse (capture the conventions in programming and methods of plan composition), and presents rules from two studies that support this model.

[Solo1986] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, Sep 1986. Groundbreaking work that analyzed programming in terms of choosing appropriate goals and constructing plans to achieve them, focused on mechanisms and explanations, and introduced 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, the cognitive theory of mental models, constructivist theory of knowledge and learning, phenomenographic research on experiencing programming, and the theory of threshold concepts in relation to the concept of a "notional machine", which emerges as a major challenge in introductory programming education, 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: motivate-isolate-practice-integrate, head straight for objects, and explicit program dynamics.

[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 proposed 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. Found that fuzz testing student code caught errors that were missed by handwritten test suite (and includes discussion of how to share test cases and outputs).

[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. Exploring why giving children more information when they are learning about fractions can lower performance.

[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 suspect from a statistical point of view.

[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. A careful study showing that the curly-brace languages are as hard to learn as a language with randomly-designed syntax, but others are better.

[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. A brief summary of 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. Synthesized 401 suggestions for code quality and 178 statements from three instructors to generate the first draft of 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 rounds of iteration to improve 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. Code quality rubric developed by the authors.

[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. Meta-analysis finds significant positive benefit for Direct Instruction (strong form, involving teaching from a script).

[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. Study showing that students who receive any kind of graphics give significantly higher satisfaction ratings than those who don't get graphics, but only students who get instructive graphics perform significantly 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. A study involving over a thousand students found that redesigning an online course to be gender neutral improved completion probability by 7%, in particular for men, but decreased for students with a high score for superficial learning approach, especially among women.

[Tann2013] Kimberly D. Tanner. Structure matters: Twenty-one teaching strategies to promote student engagement and cultivate classroom equity. *Cell Biology Education*, 12(3):322–331, Sep 2013. Presents and explains 21 strategies in five groups for making classes more engaging and equitable.

[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 (analytical, theoretical, or formalist), scientific (empirical), and engineering (technological).

[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, where validation involved using a pseudocode version of the test as well as versions using real programming languages (Java, MATLAB, and Python) and think aloud interview data, student responses to the FCS1, and student CS1 exam scores.

[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. Interviews with 26 coding bootcamp alumni found mixed results: bootcamps are sometimes useful, but formal and informal barriers remain, and quality is varied.

[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, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael 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 the 2001 study showing how little students seem to 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. A summary of studies shown that how students rate a course and how much they actually learn are not related.

[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 of the 10 approaches studied, and introducing a game theme the least effective.

[Wall2009] Thorbjorn Walle and Jo E. 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. Found that pairs with different levels of a given personality trait communicated more intensively.

[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 a 2007 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. There's more criticism than prescription, but the former is well-informed and sharp-edged.

[Wein2017] 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. Studied students using a dual-mode blocks and text coding environment and found that students 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.

[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 key learning practices: spaced practice, interleaving, retrieval practice, elaboration, concrete examples, and dual coding.

[Weng2015] Etienne Wenger-Trayner and Beverly Wenger-Trayner. Communities of practice: A brief introduction. http://wenger-trayner.com/intro-to-cops/, 2015. A good 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. Brief summaries of various theories of learning.

[Wies2014] 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. Exploring why giving children more information when they are learning about fractions can lower performance.

[Wigg2005] Grant Wiggins and Jay McTighe. *Understanding by Design*. Association for Supervision & Curriculum Development, 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. Compares performance and confidence of students with and without prior programming experience in CS1 and CS2, and finds with students outscoring without students by more than 6% on exams and 10% on programming quizzes in CS1, but 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. Found 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, which more than lives up to the "comprehensive" in its name.

[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 K12 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. Found that students using collaborative note taking learned English as a foreign language better than those who didn'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.

B Code of Conduct

We are dedicated to providing a harassment-free learning experience for everyone, regardless of gender, sexual orientation, disability, physical appearance, body size, race, or religion. We do not tolerate harassment any form, including offensive communication, sexual images in public spaces, deliberate intimidation, stalking, following, harassing photography or recording, sustained disruption of talks or other events, inappropriate physical contact, or unwelcome sexual attention.

Be kind to others. Do not insult or put down other attendees. Behave professionally. Remember that sexist, racist, or exclusionary jokes are not appropriate.

People asked to stop any harassing behavior are expected to comply immediately. Anyone violating these rules may be asked to leave the classroom at the sole discretion of the instructors.

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

This code of conduct is a modified version of that used by PyCon, which in turn is forked from a template written by the Ada Initiative and hosted on the Geek Feminism Wiki[1].

[1] http://geekfeminism.wikia.com/wiki/Conference_anti-harassment/Policy

C Citation

Please cite this work as:

> *Greg Wilson (ed.):* How to Teach Programming (And Other Things)*. Second edition, Lulu.com, 2017, 978-1-365-98428-0, http://third-bit.com/teaching/.*

D Using This Material

This material has using in many formats, from a multi-week online class to an intensive in-person workshop. It's usually possible to cover large parts of Chapters 2–6, Chapter 8, and Chapter 10 in two long days.

D.1 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 9.5), 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 9.6.1). You should also Use them to distribute attention so that everyone gets a fair share of the instructor's time (Section 9.6.2), 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 (Section 9.6.3).

D.2 Online in Groups

In this format, learners are together in 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 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.)

D.3 Online as Individuals

FIXME (medium): describe format

D.4 Multi-Week Online

This was the first format we used, and we 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 real-world interruptions will require people to drop out.

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.)

E How to Contribute

This book is a community resource; contributions of all kinds are welcome, from suggestions for improvements to errata and new material. Please note that all contributors must abide by the Code of Conduct (Appendix B), and that by submitting your work, you are agreeing that I may incorporate it in either original or edited form and release it under the same license as the rest of this material (Appendix A). I will add you to the acknowledgments (Section 1.4) unless you request otherwise.

- The source for this book is stored on GitHub at https://github.com/gvwilson/thirdbit. If you know how to use GitHub, and would like to add or fix something, please send us a pull request. If you simply want to report an error, ask a question, or make a suggestion, please file an issue.
- If you don't know how to use GitHub, please email your contribution to mailto:gvwilson@third-bit.com. I will try to respond within a week.

Finally, I always enjoy hearing how people have used this material. Please let me know if you have a story you would like to share.

F Glossary

Agile development An approach to software development that emphasizes short iterations, incremental delivery, and close collaboration between customers and developers.

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*.

Branch coverage The degree to which a set of tests exercise all possible branches of control structures like `if/else` statements.

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 which emphasizes its social aspects, particularly as enabled by the Internet and other technologies.

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

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

Contributing student pedagogy Having students produce artifacts to contribute to other students' learning.

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.

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.

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*.

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.)

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 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.

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.

Reflecting listening The practice of paraphrasing a speaker's point back to them immediately after hearing it in order to confirm understanding.

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.

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.

Understanding by design see *backward design*.

Working memory see *short-term memory*.

G 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 check-in points so you can re-scope or redirect effort.
- The end product specifies deliverables clearly so you can finish development without major 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.

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.

Checkin: a rough scope for the course that you have agreed with your colleagues.

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.

Checkin: brief summaries of who your course will help and how.

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[1]: 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.

Checkin: 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.

[1] https://en.wikipedia.org/wiki/Test-driven_development

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.

Checkin: a course outline.

Note:

- 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.

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.

Checkin: course description, learning objectives, and prerequisites.

Note: see the appendix for a discussion of how to write good learning objectives.

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.

H 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.

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.

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 pre-requisite tasks.

At the Start of the Event

1. Remind everyone of the code of conduct.
2. Collect attendance.
3. Distribute sticky notes.
4. Collect any relevant online account IDs.

At the End of the Event

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

Travel Kit

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

- sticky notes
- cough drops
- 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

I 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.

Opening	**Yes**	**Iffy**	**No**	**N/A**
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	○	○	○	○

J Teamwork Rubric

This rubric is designed to assess individual performance within a team.

Communication	**Yes**	**Iffy**	**No**	**N/A**
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.	○	○	○	○

K 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.

1. 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.
2. 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.
3. 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.
4. 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.
5. Why do you want to take this training course?

L Ten Quick Tips for Teaching Programming

This material was originally co-written with Dr. Neil Brown of King's College London, and appeared in [Brow2018]. We are grateful to Barbara Ericson and Leo Porter for their comments on a draft of this article, and to Lauren Margulieux for providing the subgoal example.

Research from educational psychology suggests that teaching and learning are subject-specific activities [Maye2004]: learning programming has a different set of challenges and techniques than learning physics or learning to read and write. Computing is a younger discipline than mathematics, physics, or biology, and while there have been correspondingly fewer studies of how best to teach it, there is a growing body of evidence about what works and what doesn't. This paper presents ten quick tips that should be the foundation of any teaching of programming, whether formal or informal.

These tips will be useful to anyone teaching programming at any level and to any audience. A larger list aimed primarily at K-12 audiences can be found at the CS Teaching Tips site[1].

Remember That There is No Geek Gene

Guzdial [Guzd2015b] refers to the belief that some people are born programmers and others aren't as "computing's most enduring and damaging myth." This is often "confirmed" by looking at university grade distributions, which are commonly held to be bimodal: a low-scoring hump of those that will never get it and a high-scoring hump of those that have the right stuff. Our first and most important tip is that this is wrong: competence at programming is not innate, but is rather a learned skill that can be acquired and improved with practice.

The most powerful evidence for this comes from Patitsas et al. [Pati2016]. They examined grade distributions in introductory computing courses at a large university, and found that only 5.8% were actually multi-modal. More damningly, they found that computer science faculty were more likely to see distributions as bimodal if they thought those grades came from a programming class than if they believed the grades came from some other kind of class, and that those faculty were even more likely to see the distributions as bimodal if they believed that some students are innately predisposed to do well in computer science.

Beliefs such as this are known to have powerful effects on education outcomes [Alvi1999, Brop1983, Juss2005]. If instructors believe that "some

[1] http://csteachingtips.org/

kids get it and some kids don't", they will (consciously or unconsciously) invest less in those whom they put in the second category. When combined with cultural stereotypes about who is and isn't a "natural programmer", the downward spiral of under-achievement that results from differential attention may be partly responsible for the gender imbalance in computing.

Use Peer Instruction

One-on-one tutoring is perhaps the ideal form of teaching: all of a teacher's attention can be focused on one student, and they can completely customize their teaching for that person and tailor individual feedback and corrections based on two-way dialogue with them. In realistic settings, however, one teacher must usually teach several, tens, or even hundreds of students at once. How can teachers possibly hope to clear up many learners' different misconceptions in these larger settings in a reasonable time?

The best method developed so far for larger-scale classrooms is called Peer Instruction. Originally created by Eric Mazur at Harvard [Mazu1996], it has been studied extensively in a wide variety of contexts, including programming [Port2011, Port2013]. In simplified form, peer instruction proceeds in several phases:

1. The instructor gives learners a brief introduction to the topic.

2. The instructor then gives learners a multiple choice question that probes for misconceptions rather than simple factual recall. A programming example is given below which relates to integer comparison and loops. The multiple choice question must be well designed. There is no point asking a trivial question that all students will get right or one with meaningless wrong answers which no student will pick. The ideal questions are those where 40–60% of students are likely to get the right answer first time [NRC2015, p23], and where every wrong answer corresponds to a misconception that will cause it to be picked by at least some students.

3. Learners then vote on the answer to the question individually, thus formalizing their initial prediction.

4. Learners are then given several minutes to discuss those answers with one another in small groups (typically 2–4 students) and then reconvene and vote again.

5. Then the instructor can act on the latest answers:

    - If all the learners have the right answer, the instructor can move on.
    - If some of the wrong answers remain popular after group discussion, the instructor can addresses those specific misconceptions directly or engage in class-wide discussion.

Peer instruction is essentially a way to provide one-to-one mentorship in a scalable way. Group discussion significantly improves learners' understanding because it forces them to clarify their thinking, which can be enough to call out gaps in reasoning. Re-polling the class then lets the instructor know if they can move on, or if further explanation is necessary.

While it significantly outperforms lecture-based instruction in most situations, it can be problematic if ability levels differ widely (as they often do in introductory programming classes because of varied prior experience). Pair programming can be used to mitigate this.

```
for (int i = 1; i < 10; i++) {
  if (i < 3 || i >= 8) {
    System.out.println("Yes");
  }
}
```

How many times will the above code print out the word Yes?

1. *10*
2. *5*
3. *4*
4. *3*

Use Live Coding

Rather than using slides, instructors should create programs in front of their learners [Rubi2013]. This is more effective for multiple reasons:

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

2. It facilitates unintended knowledge transfer: students learn more than the instructor consciously intends to teach by watching *how* instructors do things. The extra knowledge may be high-level (e.g., whether a program is written top-down or bottom-up) or fairly low-level (e.g., learning useful editor shortcuts).

3. It slows the instructor down: if the instructor has to type in the program as they go along, they can only go twice as fast as their learners, rather than ten-fold faster as they could with slides – which risks leaving everyone behind.

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

5. Watching instructors make mistakes shows learners that it's all right to make mistakes of their own [Bark2005]. Most people model the behavior of their teachers: if the instructor isn't embarrassed about making and talking about mistakes, learners will be more comfortable doing so too.

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

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

2. Instructors can spend too much time typing in boilerplate code that is needed by the lesson, but not directly relevant to it (such as library import statements). Not only does this slow things down, it can distract learners from the intended thrust of a lesson. As Willingham [Will2010] says, “Memory is the residue of thought”; if the instructor spends their time typing boilerplate, that may be all that learners take away. This can be avoided by starting with a partial skeleton that includes the boilerplate, or having it on hand to copy and paste when needed. (Of the two, we prefer the former, since learners may not be able to keep up with copying and pasting.)

Note that live coding does not always have to start with a blank screen: instructors may give students some starter code that relies solely on concepts they have already mastered, and then extend it or modify it with live coding. Instructors who use live coding should ensure that learners have reference material available after lectures, such as a textbook, but should also recognize that students of all ages increasingly turn to Q&A sites such as Stack Overflow for information.

Have Students Make Predictions

When instructors are using live coding, they usually run the program several times during its development to show what it does. Surprising research from peer instruction in physics education shows that learners who observe a demonstration *do not learn better* than those who did not see the demonstration [Crou2004], and in fact many learners misremember the outcome of demonstrations afterwards [Mill2013]. In other words, demonstrations can actually be useless or actively harmful.

The key to making demonstrations more effective is to make learners predict the outcome of the demonstration before performing it. Crucially, their prediction should be in some way recorded or public, e.g. by a show of hands, by holding up a cue card (A/B/C/D), or by talking to their neighbor. We speculate that the sting of being publicly wrong leads learners to pay more attention and to reflect on what they are learning; regardless of whether this hypothesis is true, instructors should be careful not to punish or criticize students who predicted wrongly, but rather to use those incorrect predictions as a spur to further exploration and explanation.

Use Pair Programming

Pair programming is a software development practice in which two programmers share one computer. One person (called the driver) does the typing, while the other (called the navigator) offers comments and suggestions. The two switch roles several times per hour. Pair programming is a good practice in real-life programming [Hann2009], and also a good way to teach [McDo2006]. Partners can not only help each other out during practical exercises, but can also clarify each other’s misconceptions when the solution is presented.

Both parties involved in pair programming learn while doing it. The weaker gets individual instruction from the stronger, while the stronger learns by explaining, and by being forced to reconsider things which they may not have thought about in a while. When pair programming is used it is important to put everyone in pairs, not just the learners who may be struggling, so that no one feels singled out. It's also important to have people switch roles within each pair three or four times per hour, so that the stronger personality in each pair does not dominate the session.

Use Worked Examples With Labelled Subgoals

Learning to program involves learning the syntax and semantics of a programming language, but also involves learning how to construct programs. A good way to guide students through constructing programs is the use of worked examples: step-by-step guides showing how to solve an existing problem.

Instructors usually provide many similar programming examples for learners to practice on. But since learners are novices, they may not see the similarity between examples: finding the highest rainfall from a list of numbers and finding the first surname alphabetically from a list of names may seem like quite different problems to learners, even though more advanced programmers would recognize them as isomorphic.

Margulieux and Morrison et al. [Morr2015, Morr2016, Marg2012] have shown that students perform better when worked examples are broken down into steps (or subgoals) which are given names (or labels) – an example is given below. Subgoal labels provide a structure which allow learners to see the similarities between coding problems and to communicate with their peers and instructors more efficiently. Learners can then apply the labels to future tasks that they attempt themselves.

Conventional Materials

```
1. Click on "AccelerometerSensor1"
2. Drag out a when AccelerometerSensor1.AccelerationChanged block
3. Click on "cowbellSound"
4. Drag out call cowbellSound.Play and connect it after
   when AccelerometerSensor1.AccelerationChanged
```

Subgoal Labelled Materials

```
Handle Events from My Blocks
1. Click on "AccelerometerSensor1"
2. Drag out a when AccelerometerSensor1.AccelerationChanged block

Set Output from My Blocks
3. Click on "cowbellSound"
4. Drag out call cowbellSound.Play and connect it after
   when AccelerometerSensor1.AccelerationChanged
```

Stick to One Language

A principle that applies across all areas of education is that transference only comes with mastery [Gick1987]. Courses should therefore stick to one language until learners have progressed far enough with it to be able to distinguish the forest from the trees. While an experienced programmer can, for example, take what they know about loops and function calls in one language and re-use that understanding in a language with a different syntax or semantics, a newcomer does not yet know which elements of their knowledge are central and which are accidental. Attempting to force transference too early—e.g., requiring them to switch from Python to JavaScript in order to do a web programming course early in their education—will confuse learners and erode their confidence.

Use Authentic Tasks

Guzdial et al. found that having learners manipulate images, audio, and video in their early programming assignments increased retention in two senses: learners remembered more of the material when re-tested after a delay, and were more likely to stay in computing programs [Guzd2013]. This is a particular instance of a larger observation: learners find authentic tasks more engaging than abstracted examples.

A classic question in computing (and mathematics) education is whether problems are better with context (e.g., find the highest student grade) or without (e.g. find the maximum of the list of numbers). Bouvier et al. [Bouv2016] examined this with a multi-university study and found no difference between the two. They suggest that since it makes no difference, other considerations (such as motivation) should be given priority.

One caution about choosing context is that context can inadvertently exclude some people while drawing others in. For example, many educators use computer games as a motivating example for programming classes, but some learners may associate them with violence and racial or gender stereotypes, or simply find them unenjoyable. Whatever examples are chosen, the goal must be to move learners as quickly as possible from "hard and boring" to "easy and exciting" [Repe2017].

To help students accomplish a visible and satisfying result quickly, instructors can provide some pre-written software libraries or source code that start students closer to the end goal. The idea that students must start from scratch and write all the code they need themselves is the relic of a bygone era of home microcomputers (and it was not true even then). Pick the task that you actually want to the students to engage in, and provide everything else pre-made.

Remember That Novices Are Not Experts

This principle is tautological, but it is easily forgotten. Novices program differently than experts [Parn2017], and need different approaches or tools. If you ask a professional programmer to iterate over a list of integers and produce the average, they can write the code within seconds, using stored knowledge of the exact pattern required. A novice will approach this problem totally differently: they need to remember the syntax for the different parts, they need to know how to iterate over a list, how to use an accumulator variable, and so on.

Novices may need to spend time thinking about an algorithm on paper (something expert programmers rarely need, as they have usually memorized most common algorithmic patterns). They may need to construct examples in guided steps. They may struggle to debug. Debugging usually involves contrasting what is happening to what should be happening, but a novice's grasp on what should be happening is usually fragile.

Novices do not become professionals simply by doing what professionals do at a slower pace. We do not teach reading by taking a classic novel and simply proceeding more slowly. We teach by using shorter books with simpler words and larger print. So in programming, we must take care to use small, self-contained tasks at a level suitable for novices, with tools that suit their needs, and without scoffing.

Don't Just Code

Our final tip for teaching programming is that you don't have to program to do it. Faced with the challenges of learning syntax, semantics, algorithms, and design, examples that seem small to instructors can still easily overwhelm novices. Breaking the problem down into smaller single-concept pieces can reduce the cognitive load to something manageable.

For example, a growing number of educators are including Parsons Problems in their pedagogic repertoire [Pars2006, Morr2016]. Rather than writing programs from scratch, learners are given the lines of code they need to solve a problem, but in jumbled order. Re-ordering them to solve the problem correctly allows them to concentrate on mastering control flow without having to devote mental energy to recalling syntax or the specifics of library functions. They are also liked by learners; Ericson et al. [Eric2015] found that learners were more likely to attempt Parsons Problems than nearby multiple choice questions in an ebook.

Conclusion

The ten tips presented here are backed up by scientific research. Like any research involving human subjects, studies of computing education must necessarily be hedged with qualifiers. However, we do know a great deal, and are learning more each year. Conferences like SIGCSE[2]), ITiCSE[3] and ICER[4] present a growing number of rigorous, insightful studies with immediate practical application. Future work may overturn or qualify some of our ten tips, but they form a solid basis for any educational effort to the best of our current knowledge.

We offer one final observation: do not forget the human element. Programmers have a reputation for pouring scorn on certain programming tools (e.g., pouring scorn on spreadsheets), or for gatekeeping (e.g., stating that you cannot learn programming if you did not start young). If you are teaching someone to program, the last thing you want to do is make them feel like they can't succeed or that any existing skill they have (no matter when or how acquired) is worthless. Make your learners feel that they can be a programmer, and they just might become one.

[2] http://sigcse.org/
[3] http://iticse.acm.org/
[4] https://icer.hosting.acm.org

M Design Notes

This design follows the five-step backward design process described in this book [Wigg2005, Bigg2011, Fink2013].

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.

Intended Audience

These learner personas clarify what readers are interested in and what can be assumed about their prior knowledge.

- *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.
- *Moshe* is a professional programmer with two teenage children whose school doesn't offering programming classes. He has volunteered to run an after-school program, 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.
- *Emily* trained as a librarian, and how 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.
- *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).
  - Diverse learning contexts:
    * Moshe will do a two-day weekend workshop covering much less material and then study on his own.
    * Emily will study on her own while leading an online reading group for her volunteers.
    * Samira will take a one-semester undergraduate course with assignments, a project, and a final exam.
    * Gene will study on their own (like Emily) or take an intensive workshop (like Moshe).

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.

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

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.