Ten Quick Tips for Delivering Programming Lessons

Greg Wilson[1,*]
1 RStudio, Inc., Toronto, Ontario M4L 2T9

* greg.wilson@rstudio.com

Abstract

Designing a great lesson is the first 90% of effective teaching; delivering it well is the other 90%. The 10 simple rules outlined in this paper describe practices that anyone teaching programming can adopt immediately and at low cost.

Author Summary

Teaching well is a craft like any other, and success often comes from an accumulation of small improvements rather than from any single large change. This paper describes ten practices you can use when teaching programming (and other subjects). All are easy to adopt, and have proven their value in both institutional classrooms and free-range workshops.

Introduction

Teaching well is a craft like any other, and success often comes from an accumulation of small improvements rather than from any single large change. This paper describes ten practices you can use when teaching programming (and other subjects) that are easy to adopt and have proven their value in both institutional classrooms and free-range workshops. Some have been inspired by [1–3], while others are drawn from the author's experience [4] or operationalize the larger ideas described in [5, 6].

The foundation for these recommendations is the fact that active learning is more effective than passive learning [7, 8]. People who use new knowledge as it comes in by doing exercises or summarizing it learn more than people who just watch or listen. Equally, teaching is more effective when instructors dynamically adjust their teaching based on real-time feedback from their learners, e.g., by providing an alternative explanation of a concept that the class has found difficult or by changing direction to incorporate a question that reveals an unexpected learner interest.

In addition, learners who are intrinsically motivated learn more than those who are extrinsically motivated (or not motivated at all) [9]. When you make a connection between what you are teaching and your learners' goals or demonstrate that you respect their time and priorities, you increase how much they learn.

1 Rule 1: Assess prior knowledge—carefully.

The more you know about your learners before you start teaching, the more you will be able to help them. Inside a formal school system, the prerequisites to your course will tell you something about what they're likely to already know. In a free-range setting,

though, your learners may be much more diverse, so you may want to give them a survey or questionnaire in advance of your class to find out what knowledge and skills they have.

But asking people to rate themselves on a scale from 1 to 5 is pointless because of the Dunning-Kruger effect: the less people know about a subject, the less accurate their estimate of their knowledge is [10]. In addition, people who are members of underrepresented groups will often underrate their skills. Rather than asking people to rate themselves, you should instead ask them how easily they could complete some specific task, such as finding the average of the values in the second column of a table.

Doing this is risky, though, because school trains people to treat anything that looks like an exam as something they have to pass rather than as a chance to shape instruction. If someone answers "I don't know" to even a couple of questions on your pre-assessment, they might conclude that your class is too advanced for them. You could therefore scare off many of the people you most want to help.

2 Rule 2: Use formative assessment every 5–15 minutes.

Instructors always want to get through material than time allows, so we often teach at the speed at which we can talk rather than the speed at which people can learn. Having learners do something every 5–15 slows us down, keeps them engaged, and gives us and them feedback on whether they have actually understood what has just been taught.

In-class checks like this are one kind are called *formative assessments*. Good ones take only a minute or two to complete so that they don't derail the flow of the lesson, and have an unambiguous correct answer so that they can be checked in large classes. Popular kinds of formative assessment in programming classes include:

- Answer a multiple choice question.
- Write a few lines of code.
- Predict what the code on the screen will do when it runs.
- Contribute the next line of code.
- Label a diagram of a data structure.
- Trace the order in which statements are evaluated.

Starting with a formative assessment that reviews a previous lesson is a good way to signal that class has started, and having learners recall older material before tackling something new improves learning outcomes [11]. Similarly, ending the class with such an exercise gives learners a sense of how far they have progressed.

A complement to this rule is to get learners out of their seats every 45–60 minutes. People's brains get tired when they are concentrating, and tired brains can't learn [8]. Caffeine doesn't fix this, so have learners get up and move around for a few minutes every hour in order to reoxygenate their gray cells. This allows those who need a bathroom break to take care of things discreetly[1], and give the instructor a few moments to answer questions and figure out what to teach next.

[1] A colleague once told me that the basic unit of teaching is the bladder. When I said I'd never thought of that, she said, "You've obviously never been pregnant."

3 Rule 3: Use sticky notes to monitor progress, distribute attention, and create a backlog.

Sticky notes are my favorite teaching tool, and I'm not alone in loving their versatility, portability, stickability, foldability, and subtle yet alluring aroma [12]. Give each learner two sticky notes of different colors, such as orange and green. If someone has completed an exercise and wants it checked, or if they feel that they are following the lesson, they put the green sticky note somewhere the instructor can see. If they run into a problem and need help, they put up the orange one. This works much better than having people raise their hands: it's more discreet (which means they're more likely to actually do it), they can keep working while their flag is raised rather than trying to type one-handed, and the instructor can quickly see from the front of the room what state the class is in.

Sticky notes can also be used to ensure that attention is fairly distributed. Instructors naturally focus their attention on learners who are making eye contact and asking lots of questions—in other words, on extroverts. This creates two feedback loops: the extroverts ask even more questions because they're getting attention, while other learners stop trying to engage because they're not. To prevent this, have each learner write their name on a sticky note and post it on their laptop or somewhere equally visible. Each time the instructor calls on them or answers one of their questions, they take their sticky note down. Once all the sticky notes are down, everyone puts theirs up again. This technique makes it easy for the instructor to see who they haven't spoken with recently, which helps them avoid unconscious bias and preferential interaction. It also shows learners how attention is being distributed so that when they *are* called on, they won't feel like they're being picked on.

Finally, you may not have time to answer all of your learners' questions, or might not actually know the answers. To handle this, write questions on sticky notes and post them on the wall behind you, then look over this backlog during breaks and decide which questions you want to tackle. This gives you a chance to prioritize based on what's most relevant (and what you actually know). It also helps build trust: many people have learned that "I'll address that later" means "I hope you'll forget that you asked." If they see you trying to tackle a few of the questions that have come up, they will forgive you for not getting to the rest.

4 Rule 4: Use live coding and make mistakes.

A slide deck is like taking a journey by train: the ride may be smooth, but the route can't be changed on the fly. Writing code in front of the class while learners following along, on the other hand, is like going off road in a four-wheel drive: it may be bumpier and messier, but it's a lot easier to explore things that catch the learners' attention.

This practice is called *live coding*. It encourages active learning—people are using knowledge immediately when acquiring it—and allows the instructor to adapt to their actual audience. If a learner asks a "what if" question, it's a lot easier to respond if you are writing code in real time rather than presenting slides. Following learners' lead also signals that you respect your learners' time and interests; this improves engagement, which in turn improves learning outcomes [9]. It also slows you down and reminds you just how much extraneous material your learners have to wade through.

But it's possible to have too much of a good thing. If someone asks a question, try it out. If they or someone else asks a follow-up, try that too, but then come back to your main point. Otherwise, you can easily be pulled so far off course that you don't reach the most important points of your lesson.

Finally, live coding presents natural opportunities to demonstrate something that is often left out of textbooks. Novices spend most of their time trying to figure out what's

broken and how to fix it, but lessons rarely devote that amount of time to analyzing and correcting errors. Watching the instructor make a mistake, figure out what went wrong, and then make and test fixes is immediately valuable. Doing this also shows learners that it's all right to make mistakes of their own: if you aren't embarrassed about making and talking about mistakes, learners will be more comfortable doing so too.

5 Rule 5: Use a wide variety of exercise types.

Most programming classes rely primarily on code & run exercises in which learners write software that behaves in a tightly-specified way. To keep learners engaged, and to give them opportunities to practice other skills and higher-level reasoning, you should also use:

Parsons Problems, which give them the lines of code needed to solve a problem, but in jumbled order [13–15]. Parsons Problems reduce cognitive load by allowing learners to focus on control flow without simultaneously having to recall syntax.

Multiple choice questions whose wrong answers have been chosen to probe for specific misconceptions. For example, learners who have worked with spreadsheets may believe that after executing `a=10`, `b=a`, and `a=20`, the value of `b` will be 20.

Debugging, completion, and **extension** exercises in which learners must fix, finish, or extend an existing program. These all model authentic tasks (i.e., the kinds of things programmers spend most of their time doing in real life).

Tracing execution order or **tracing values**, in which the learner lists the order in which the statements in a program are executed or the values that one or more variables take on as the program runs, which are essential program comprehension skills.

Code reviews in which learners score a program against a rubric supplied by the instructor. In these exercises, learners typically start with a perfect score and lose points for false positives as well as false negatives so that they don't simply mark every statement as being wrong in all possible ways.

6 Rule 6: Have learners take notes, and review them.

Fifty years ago, when being able to summarize a speech or record the minutes of a meeting was considered an essential white-collar skill, it was common for high school instructors to require learners to take notes and hand them in for grading. This has fallen out of fashion, but taking notes is still beneficial: it forces learners to organize and reflect on information as it's coming in, which in turn increases the likelihood that they will transfer it to long-term memory [16, 17],

To help learners improve their note-taking, have them take a minute at the end of each class to write one thing they learned on one side of a card and one question they still have or something they're confused about on the other. Review these *minute cards* and looking for patterns only takes a few minutes, and tells you what you need to clarify at the start of the next lesson.

Another technique is to make 4–6 learners the official note takers for each class. They must summarize the information presented and find answers to backlog questions that the instructor didn't get to (Rule 4). Their notes are then graded and shared with the class (e.g., by being posted on the course website). In many cases, their notes will

be more useful than what you might have put together, since they will record what they and their peers actually need to know rather than what you think they ought to need to know.

7 Rule 7: Have learners work in pairs and discuss in groups.

Pair programming is a software development practice in which one person (the driver) does the typing while the other (the navigator) offers comments and suggestions, and the two switch roles several times per hour. It is effective in professional work [18], and benefits in teaching include increased success rate in introductory courses, better software, and higher learner confidence in their solutions. There is also evidence that learners from underrepresented groups benefit even more than others [19–21]. It is particularly helpful with mixed-ability classes, since pairs are more homogeneous than individuals. However, when you use pairing, put everyone in pairs: if you only pair learners who are struggling, they will feel singled out.

Peer instruction attempts to provide the benefits of individual instruction in a scalable way by interleaving formative assessment with small-group discussion. After a brief introduction to a new topic, the instructor gives learners a multiple choice question or some other formative assessment (Rule 2). The learners vote on their answers, then spend several minutes discussing the question, during which they will fill in gaps in each other's knowledge and clear up each other's misunderstandings. This technique makes group discussion the focus of learning, and its effectiveness has been proven by multiple studies [22–24].

8 Rule 8: Present relevant diagrams incrementally to complement other material.

Our brains have separate channels for processing visual and linguistic information, so people learn best when complementary material is presented simultaneously through these channels [25, 26]. In simple terms, this means that you should present diagrams or other relevant images for you to talk about rather than slabs of text that duplicate what you are saying. Diagrams are even more effective if they are built up piece by piece rather than shown all at once. When this is done, learners' brains will correlate the arrival of new visual information with the arrival of new linguistic information. Presentation of either later on will then help trigger recall of the other.

All graphics should be directly relevant to the course material. [27] distinguished *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 (which are directly relevant to the instructional goal). Learners who received any kind of graphic gave material higher satisfaction ratings than those who didn't get graphics, but only learners who got instructive graphics actually performed better.

9 Rule 9: Teach together.

Co-teaching describes two instructors working together in the same classroom [28]:

Team teaching: Both instructors deliver a single stream of content in tandem, taking turns like musicians taking solos.

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

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

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

Parallel teaching: The class is divided in two and the instructors present the same material simultaneously to each.

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

Team teaching is particularly beneficial in day-long workshops: it give each instructor a chance to rest and think about what they are going to do next. If you and a partner are co-teaching:

- Take 2–3 minutes before the start of each class to confirm who's teaching what.
- 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 learners will be distracted by more frequent switch-ups.
- The person who isn't teaching shouldn't interrupt, offer corrections or elaborations, or do anything else to distract from what the person teaching is doing or saying, but may ask leading questions if the learners seem lethargic or unsure of themselves.

Most importantly, take a few minutes when the class is over to congratulate or commiserate with each other: in teaching as in life, shared misery is lessened and shared joy increased.

10 Rule 10: Include everyone.

Inclusivity is a policy of including people who might otherwise be excluded or marginalized. In computing, it means making a positive effort to be more welcoming to women, under-represented racial or ethnic groups, people with various sexual orientations, the elderly, those facing physical challenges, the formerly incarcerated, the economically disadvantaged, and everyone else who doesn't fit Silicon Valley's affluent white/Asian male demographic.

The first and most important step toward fixing this is to stop using a *deficit model*, i.e., to stop thinking that the members of under-represented groups lack something and are therefore responsible for not getting ahead. Believing that puts the burden on people who already have to do extra work to overcome structural inequities and (not coincidentally) gives those who benefit from the current arrangements an excuse not to look at themselves too closely.

Misogyny in video games, the use of "cultural fit" in hiring to excuse conscious or unconscious bias, a culture of silence around harassment, and the growing inequality in society that produces preparatory privilege are not any one person's fault, but fixing them is everyone's responsibility. As an instructor, you have more power than most;

Frame Shift Consulting's workshop materials offer excellent practical advice on how to be a good ally [29], and are probably more important than anything else in this paper.

[30] describes things instructors can do to make their lessons more accessible to learners facing physical challenges, while [31] is a brief, practical guide to inclusivity in the classroom. The practices it describes help everyone, not just members of marginalized groups, and include:

Ask learners to email you beforehand to explain how they believe what they're about to learn will 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 people learn, not the advantages or disadvantages they had when they started.

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

One other way to support learners from marginalized groups is to have people sign up 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 afterward: if people come with their friends or colleagues, they can work together to use what they've learned.

Conclusion

One final note: if you are teaching an evening class after working for a full day, you and your learners will both appreciate it if you brush your teeth and put on a clean shirt before you start teaching. Cough drops will also help you keep your voice and fend off whatever colds the learners brought with them, and your back will be grateful tomorrow that you wore comfortable shoes today.

References

1. Huston T. Teaching What You Don't Know. Harvard University Press; 2012.
2. Lang JM. Small Teaching: Everyday Lessons from the Science of Learning. Jossey-Bass; 2016.
3. Lemov D. Teach Like a Champion 2.0: 62 Techniques that Put Students on the Path to College. Jossey-Bass; 2014.
4. Wilson G. Software Carpentry: lessons learned. F1000Research. 2016;doi:10.12688/f1000research.3-62.v2.
5. Brown NCC, Wilson G. Ten Quick Tips for Teaching Programming. PLOS Computational Biology. 2018;14(4). doi:10.1371/journal.pcbi.1006023.
6. Devenyi GA, Emonet R, Harris RM, Hertweck KL, Irving D, Milligan I, et al. Ten Simple Rules for Collaborative Lesson Development. PLoS Computational Biology. 2018;14(3). doi:10.1371/journal.pcbi.1005963.
7. Ambrose SA, Bridges MW, DiPietro M, Lovett MC, Norman MK. How Learning Works: Seven Research-Based Principles for Smart Teaching. Jossey-Bass; 2010.

8. National Academies of Sciences, Engineering, and Medicine. How People Learn II: Learners, Contexts, and Cultures. National Academies Press; 2018.

9. Wlodkowski RJ, Ginsberg MB. Enhancing Adult Motivation to Learn: A Comprehensive Guide for Teaching All Adults. Jossey-Bass; 2017.

10. Kruger J, Dunning D. Unskilled and Unaware of it: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments. Journal of Personality and Social Psychology. 1999;77(6):1121–1134. doi:10.1037/0022-3514.77.6.1121.

11. Weinstein Y, Sumeracki M, Caviglioli O. Understanding How We Learn: A Visual Guide. Routledge; 2018.

12. Ward J. Adventures in Stationery: A Journey Through Your Pencil Case. Profile Books; 2015.

13. Parsons D, Haden P. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In: 2006 Australasian Conference on Computing Education (ACE'06). Australian Computer Society; 2006. p. 157–163.

14. Morrison BB, Margulieux LE, Ericson BJ, Guzdial M. Subgoals Help Students Solve Parsons Problems. In: 2016 Technical Symposium on Computer Science Education (SIGCSE'16). Association for Computing Machinery (ACM); 2016.

15. Ericson BJ, Margulieux LE, Rick J. Solving Parsons Problems versus Fixing and Writing Code. In: 2017 Koli Calling Conference on Computing Education Research (Koli'17). Association for Computing Machinery (ACM); 2017.

16. Aiken EG, Thomas GS, Shennum WA. Memory for a Lecture: Effects of Notes, Lecture Rate, and Informational Density. Journal of Educational Psychology. 1975;67(3):439–444. doi:10.1037/h0076613.

17. Bohay M, Blakely DP, Tamplin AK, Radvansky GA. Note Taking, Review, Memory, and Comprehension. American Journal of Psychology. 2011;124(1):63. doi:10.5406/amerjpsyc.124.1.0063.

18. Hannay JE, Dybå T, Arisholm E, Sjøberg DIK. The Effectiveness of Pair Programming: A Meta-analysis. Information and Software Technology. 2009;51(7):1110–1122. doi:10.1016/j.infsof.2009.02.001.

19. McDowell C, Werner L, Bullock HE, Fernald J. Pair Programming Improves Student Retention, Confidence, and Program Quality. Communications of the ACM. 2006;49(8):90–95. doi:10.1145/1145287.1145293.

20. Hanks B, Fitzgerald S, McCauley R, Murphy L, Zander C. Pair Programming in Education: a Literature Review. Computer Science Education. 2011;21(2):135–173. doi:10.1080/08993408.2011.579808.

21. Celepkolu M, Boyer KE. Thematic Analysis of Students' Reflections on Pair Programming in CS1. In: 2018 Technical Symposium on Computer Science Education (SIGCSE'18). Association for Computing Machinery (ACM); 2018.

22. Crouch CH, Mazur E. Peer Instruction: Ten Years of Experience and Results. American Journal of Physics. 2001;69(9):970–977. doi:10.1119/1.1374249.

23. Smith MK, Wood WB, Adams WK, Wieman CE, Knight JK, Guild N, et al. Why Peer Discussion Improves Student Performance on In-class Concept Questions. Science. 2009;323(5910):122–124. doi:10.1126/science.1165919.

24. Porter L, Bouvier D, Cutts Q, Grissom S, Lee CB, McCartney R, et al. A Multi-Institutional Study of Peer Instruction in Introductory Computing. In: 2016 Technical Symposium on Computer Science Education (SIGCSE'16). Association for Computing Machinery (ACM); 2016.

25. Mayer RE, Moreno R. Nine Ways to Reduce Cognitive Load in Multimedia Learning. Educational Psychologist. 2003;38(1):43–52. doi:10.1207/s15326985ep3801_6.

26. Mayer RE. Multimedia Learning. 2nd ed. Cambridge University Press; 2009.

27. Sung E, Mayer RE. When Graphics Improve Liking but not Learning from Online Lessons. Computers in Human Behavior. 2012;28(5):1618–1625. doi:10.1016/j.chb.2012.03.026.

28. Friend M, Cook L. Interactions: Collaboration Skills for School Professionals. Eighth ed. Pearson; 2016.

29. Aurora V. Frame Shift Consulting Workshop on Ally Skills; 2017. `https://frameshiftconsulting.com/ally-skills-workshop/`.

30. Burgstahler SE. Universal Design in Higher Education: From Principles to Practice. 2nd ed. Harvard Education Press; 2015.

31. Lee CB. What Can I Do Today to Create a More Inclusive Community in CS?; 2017. `http://bit.ly/2oynmSH`.