Ten simple rules for teaching programming

Neil C.C. Brown[1‡*], Greg Wilson[2‡*]

1 King's College London / neil.c.c.brown@kcl.ac.uk
2 Rangle.io / gvwilson@third-bit.com

‡ These authors contributed equally to this work.
* corresponding author

Introduction

Research from educational psychology suggests that teaching and learning are subject-specific activities [1]: 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 doesn't. This paper presents ten simple rules that should be the foundation of any teaching of programming, whether formal or informal.

1 There is no geek gene

[2] 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 rule 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 [3]. 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 some students are innately predisposed to do well in computer science.

Beliefs such as this are known to have powerful effects on education outcomes [4–6]. If instructors believe that "some 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.

2 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 customise 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 [7], it has been studied extensively in a wide variety of contexts, including programming [8, 9]. In simplified form, peer instruction proceeds in three 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 in Figure 1 which relates to integer comparison and loops. There is no point asking a trivial question that all students will get right. The ideal questions are those where students are most likely to vote equally for each of the options.
3. Learners then vote on the answer to the question.
    - If they all have the right answer, move on.
    - If they all have the same wrong answer, the instructor addresses that specific misconception.
    - If they have a mix of right and wrong answers, they are given several minutes to discuss those answers with one another in small groups (typically 2-4 students) and then reconvene and vote again.

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 (Rule 5) can be used to mitigate this.

3 Use live coding

Rather than using slides, instructors should create programs in front of their learners [10]. 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.
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., using keyboard 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.
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 alright to make mistakes of their own [11]. 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), can distract learners from the intended thrust of a lesson. As [12] 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.)

4 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. 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 [13], and in fact many learners misremember the outcome of demonstrations afterwards [14]. 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 neighbour. 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 criticise students who predicted wrongly, but rather to use those incorrect predictions as a spur to further exploration.

5 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 [15], and also a good way to teach [16]. 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.

6 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 them, even though more advanced programmers would recognise them as isomorphic.

[17–19] 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 in Figure 2. Subgoal labels provide a structure which students can apply to future tasks that they attempt themselves, and allow them to communicate with their peers and instructors more efficiently.

7 Stick to one language

A principle that applies across all areas of education is that transference only comes with mastery [20]. 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.

8 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 [21]. 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). [22] 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 primary.

One caution 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” [23].

9 Remember that novices are not experts

This principle is tautological, but it is easily forgotten. Novices program differently than experts, 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 memorised 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.

10 Don't just code

Our final rule for teaching programming is that you don't have to program to do it. Between syntax, semantics, algorithms, and design, examples that seem small to instructors can 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 [18, 24]. 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.

Conclusion

Like anything 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. Venues like SIGCSE (`http://sigcse.org/`), ITiCSE (`http://iticse.acm.org/`) and ICER (`https://icer.hosting.acm.org`) present a growing number of rigorous, insightful studies with immediate practical application. Future work may overturn or qualify some of our ten simple rules, but they form a solid basis for any educational effort.

We offer one final observation: do not forget the human element. Programmers have a reputation for pouring scorn on certain programming languages (e.g., PHP), or for gatekeeping (e.g., stating that you can't learn programming if you didn't 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.

```
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?
a) 10
b) 5
c) 4
d) 3

Fig 1. An example multiple choice question probing learners' understanding of loops and integer comparisons.

Conventional Materials

```
1. Click on "My Blocks" to see the blocks for components you created.
2. Click on "clap"
3. Drag out a when clap.Touched block
4. Click on \clapSound"
5. Drag out call clapSound.Play
6. Connect it after when clap.Touched
```

Subgoal Labeled Materials

```
Handle Events from My Blocks

1. Click on "My Blocks" to see the blocks for components you created.
2. Click on "clap"
3. Drag out a when clap.Touched block

Set Output from My Blocks

4. Click on "clapSound" and
5. Drag out call clapSound.Play
6. Connect it after when clap.Touched
```

Fig 2. An example of subgoal labeling (taken from [19])

References

1. Mayer RE. Teaching of Subject Matter. Annual Review of Psychology. 2004;55(1):715–744. doi:10.1146/annurev.psych.55.082602.133124.

2. Guzdial M. Top 10 Myths About Teaching Computer Science; 2015. `https://cacm.acm.org/blogs/blog-cacm/189498-top-10-myths-about-teaching-co`

3. Patitsas E, Berlin J, Craig M, Easterbrook S. Evidence That Computer Science Grades Are Not Bimodal. In: Proceedings of the 2016 ACM Conference on International Computing Education Research. ICER '16. New York, NY, USA: ACM; 2016. p. 113–121. Available from: `http://doi.acm.org/10.1145/2960310.2960312`.

4. Alvidrez J, Weinstein RS. Early teacher perceptions and later student academic achievement. Journal of Educational Psychology. 1999;91(4):731–746.

5. Brophy JE. Research on the self-fulfilling prophecy and teacher expectations. Journal of Educational Psychology. 1983;75(5):631–661.

6. Jussim L, Eccles J. Social perception, social stereotypes, and teacher expectations: Accuracy and the quest for the powerful self-fulfilling prophecy. Advances in Experimental Social Psychology. 1996;28:281–388.

7. Mazur E. Peer Instruction: A User's Manual. Prentice Hall; 1996.

8. Porter L, Bailey Lee C, Simon B, Cutts Q, Zingaro D. Experience Report: A Multi-classroom Report on the Value of Peer Instruction. In: Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education; 2011. p. 138–142.

9. Porter L, Guzdial M, McDowell C, Simon B. Success in introductory programming: What works? Communications of the ACM. 2013;56(8).

10. Rubin MJ. The Effectiveness of Live-coding to Teach Introductory Programming. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education. SIGCSE '13. New York, NY, USA: ACM; 2013. p. 651–656. Available from: `http://doi.acm.org/10.1145/2445196.2445388`.

11. Barker LJ, Garvin-Doxas K, Roberts E. What Can Computer Science Learn from a Fine Arts Approach to Teaching? In: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education. SIGCSE '05. New York, NY, USA: ACM; 2005. p. 421–425. Available from: `http://doi.acm.org/10.1145/1047344.1047482`.

12. Willingham DT. Why don't students like school?: A cognitive scientist answers questions about how the mind works and what it means for the classroom. John Wiley & Sons; 2009.

13. Crouch C, Fagen AP, Callan JP, Mazur E. Classroom demonstrations: Learning tools or entertainment? American Journal of Physics. 2004;72(6):835–838. doi:10.1119/1.1707018.

14. Miller K, Lasry N, Chu K, Mazur E. Role of physics lecture demonstrations in conceptual learning. Phys Rev ST Phys Educ Res. 2013;9:020113. doi:10.1103/PhysRevSTPER.9.020113.

15. Hannay JE, Dybå T, Arisholm E, , Sjøberg DIK. The effectiveness of pair programming: a meta-analysis. Information and Software Technology. 2009;51(7).

16. 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.

17. Morrison BB, Margulieux LE, Guzdial M. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In: Proceedings of the Eleventh Annual International Conference on International Computing Education Research. ICER '15. New York, NY, USA: ACM; 2015. p. 21–29. Available from: http://doi.acm.org/10.1145/2787622.2787733.

18. Morrison BB, Margulieux LE, Ericson B, Guzdial M. Subgoals Help Students Solve Parsons Problems. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education. SIGCSE '16. New York, NY, USA: ACM; 2016. p. 42–47. Available from: http://doi.acm.org/10.1145/2839509.2844617.

19. Margulieux LE, Guzdial M, Catrambone R. Subgoal-labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications. In: Proceedings of the Ninth Annual International Conference on International Computing Education Research; 2012. p. 71–78.

20. Gick ML, Holyoak KJ. The cognitive basis of knowledge transfer. In: Cormier SM, Hagman JD, editors. Transfer of learning: Contemporary research and applications. Academic Press; 1987.

21. Guzdial M. Exploring hypotheses about media computation. In: Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research. ICER '13; 2013. p. 19–26.

22. Bouvier D, Lovellette E, Matta J, Alshaigy B, Becker BA, Craig M, et al. Novice Programmers and the Problem Description Effect. In: Proceedings of the 2016 ITiCSE Working Group Reports. ITiCSE '16. New York, NY, USA: ACM; 2016. p. 103–118. Available from: http://doi.acm.org/10.1145/3024906.3024912.

23. Repenning A. Moving beyond syntax: lessons from 20 years of blocks programing in AgentSheets. Journal of Visual Languages and Sentient Systems. 2017;3. doi:10.18293/VLSS2017-010.

24. Parsons D, Haden P. Parsons Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In: Proceedings of the 8th Australasian Conference on Computing Education - Volume 52. ACE '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc.; 2006. p. 157–163. Available from: http://dl.acm.org/citation.cfm?id=1151869.1151890.