Ten simple rules for collaborative lesson development

Gabriel A. Devenyi[1‡], Rémi Emonet[2‡], Rayna M. Harris[3‡], Kate Hertweck[4‡], Damien Irving[5‡], Ian Milligan[6‡], Greg Wilson[7‡*]

1 Douglas Mental Health University Institute, McGill University / gdevenyi@gmail.com
2 Université Jean Monnet / remi.emonet@univ-st-etienne.fr
3 The University of Texas at Austin / rayna.harris@utexas.edu
4 The University of Texas at Tyler / khertweck@uttyler.edu
5 CSIRO Oceans and Atmosphere / irving.damien@gmail.com
6 University of Waterloo / i2millig@uwaterloo.ca
7 Rangle.io / gvwilson@third-bit.com

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

Abstract

Lessons take significant effort to build, and even more effort to maintain. The collaborative code development methods pioneered by the open source community offer a way forward, allowing us to create lessons which are open, accessible, and continually updated and improved by a community of contributors. This paper provides ten simple rules that can help others create sustainable lessons.

Author summary

The model of collaborative code development presents an alternative model to traditional scientific lesson development. By leveraging a community approach, educational resources can be more sustainable, robust, and responsive. These ten simple rules outline best practices for this model of collaborative resource development.

Introduction

Lessons take significant time and energy to build and even more effort to maintain. Collaborative lesson development refers to the combined efforts of teachers to build and maintain lessons. By leveraging a community approach, educational resources can be more sustainable, robust, and responsive. When lessons are created in an open source community, all lessons are open and accessible, and they can be continually updated and improved by a community of contibutors.

Despite the proven success of open-source and collaboratively developed lesson plans, it is surprisingly uncommon in the traditional accademic setting. Each year, thousands of university lecturers teach subjects ranging from first year biology, to graduate-level courses in Indian film. Some might use a common textbook written by a single author or two, but besides that lecturers develop and improve their course materials in isolation. It is staggering to think how many wheels are being re-invented, how much valuable time has been wasted, and ultimately how the progress of tertiary education has been held back by the lack of community collaboration and sharing in this process.

This problem obviously extends beyond the university sector, but it is curious that it is endemic in tertiary education, given that research depends so critically on collaboration and sharing, and that most researchers complain about how much time teaching takes away from research.

The authors collectively have experience with community-developed lesson plans in the context of research computing in the sciences and humanities. They primarily draw on their experiences with pedagogical organizations such as Software Carpentry and Programming Historian. Software Carpentry was founded in 1998 to teach scientists basic computing skills, and has since spawned two sibling organizations called Data Carpentry and Library Carpentry. Programming Historian was founded in 2008, and has evolved into a collaboratively-edited site providing lessons to humanists. These organizations have been successful in adapting open source development methods to lesson development and maintenance. Their guiding principles are that lessons should be:

1. open and easily accessible, and
2. continually maintained, refined, and improved by a community of contributors.

Open education projects satisfy the first criterion by definition, but very few projects satisfy the second. While their lessons might occasionally be updated by a small author group, as happens when a new edition of a book is edited and published, this is not the same as continuous improvement by a large community of contributors. The ten simple rules that follow summarize what we have learned about doing that as maintainers, editors, and reviewers of lessons used by tens of thousands of people.

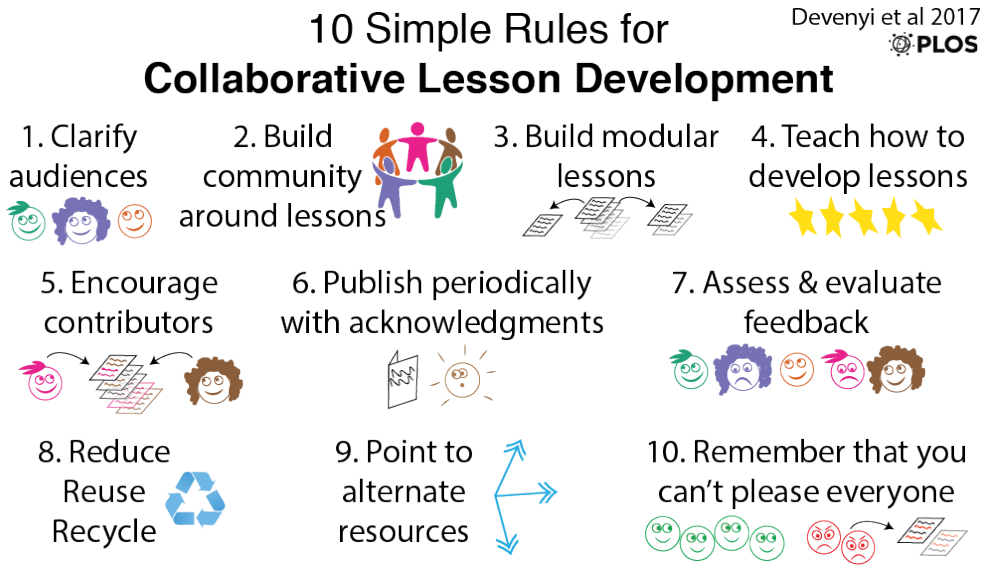

Fig 1. Graphical abstract of 10 simple rules for collaborative lesson development

1 Clarify your audience

The first requirement for building lessons together is to know who they are being built for. "Archaeology students" is far too vague: are you and your collaborators thinking of first-year students who need an introduction to the field, graduate students who intend to specialize in the sub-discipline which is the lesson's focus, or someone in between?

Prerequisite knowledge, equipment or software required, how much time learners will actually have: if different contributors believe different things about these, they will find it difficult or impossible to work together.

Thinking systematically about difficulty levels can help manage expectations. For example, Programming Historian project labels lessons "beginner"", "intermediate", and "advanced" to help authors write at appropriate levels. This method works, however, only if there is prior agreement on what those terms mean.

Rather than itemizing prior knowledge and learning objectives, it can be helpful to write *learner profiles* to clarify the learner's general background, what they already know, what *they* think they want to do, how the material will help them, and any special needs they might have.

2 Build community around lessons

Lessons don't maintain themselves. For technical lessons software versions and dependencies are constantly changing, while for more traditional academic lessons the literature is advancing at an ever increasing pace. In either case, what was cutting edge in 2017 may be dated and less useful in 2018. This needs to be clear to contributors, so they understand that this lesson is just the starting point. Sustainability needs to be front of mind.

The focus accordingly needs to be on creating a community. Authors cannot be expected to maintain continual vigilance on a lesson, but this is necessary if one expects continual use of a lesson! By working online, creating opportunities for collaboration and contribution, many eyes can keep the lesson usable. As we note below, this only works if you make space for little contributions just as you do for larger ones.

3 Build modular lessons that can be re-purposed

Every instructor's needs are different, so build small chunks that can be re-purposed in many ways. A university lecturer in meteorology, for instance, might construct a course by bringing together lessons on differential equations, fluid mechanics and absorption spectroscopy. This task is greatly simplified if existing courses on mathematics, physics and chemistry consist of numerous small, discrete lessons, as opposed to a few large, monolithic lessons.

Doing this shifts the instructor's burden from writing to finding and synthesizing, both of which are easier if lessons have been designed by people with a shared world-view (Rule 4), and if lessons clearly signal what they cover (Rule 1). In particular, if lessons reference specific points in the model curriculum guidelines promulgated by many professional societies, it can be much easier for people to find them and integrate them.

Note also that individual lesson topics hinge on the learners having the proper prerequisite background, so this rule further emphasizes the importance of Rule 1.

4 Teach best practices for lesson development

Decades of pedagogical research has yielded many insights into how best to build and deliver lessons [1]. Unfortunately, since most college and university faculty have little or no training in education, this knowledge and expertise is rarely transferred into classroom practice.

Experience shows that even a brief introduction to a handful of key practices can help collaborative lesson development in at least three ways. If people have a shared

understanding of how lessons should be developed, it is easier for them to work together. Less obviously, if people have a shared model of how lessons are going to be *used*, they are more likely to try to build the same kind of material. Finally, teaching people how to teach is a great way to introduce them to each other and build community.

An example of a particular lesson development practice is *reverse instructional design* [2]. When this is used, lessons are built by identifying learning objectives, creating *summative assessments* to determine whether those objectives have been met, designing *formative assessments* to gauge learners' progress and give them a chance to practice key skills, putting those formative assessments in order, and only then writing lessons to connect each to the next. This method is effective in its own right, but its greatest benefit is that it gives everyone a common framework within which to collaborate.

An example of how to teach these practices is Software Carpentry's instructor training program. First offered in 2012, is now a two-day course delivered both in-person and online [3–5]. Not only does it teach good pedagogical practices, it serves and an onboarding process to get everyone on the same page regarding who lessons are for, how they are delivered, and how they are maintained.

5 Encourage and empower contributors

Making the process for contributing to a lesson explicit is the key to actually getting contributions. New contributors require a straightforward and transparent introduction to understand the process of tweaking and adapting lessons. Licensing, code of conduct, governance, and contribution procedure all need to be explicit rather than implicit to lower the social barriers to contribution.

Tools can help, especially if they allow proposed changes to be viewed and discussed prior to their incorporation/merger into the lessons (in open source software development this is known as "pre-merge review"). Providing a gentle on-ramp for new contributors is essential, but some tools come with a considerable up-front learning curve. GitHub with pull requests is ideal for pre-merge review, for instance, but it requires contributors to know how to use Git, which is a famously user-hostile tool. Allowing people to edit a Google Doc or Wiki-based editing does not offer pre-merge review, but the low barrier to entry can help get conversations started.

The best way to choose tools for managing lessons is to look for those that provide a gentle on-ramp for new contributors. Ask potential contributors what they are comfortable with rather than requiring *them* to come to *you*. Remember also that contributing to a lesson is probably not their top priority, and look for ways to reduce their cognitive load. For example, threaded discussion on particular topics (e.g., using GitHub issues) can help increase the signal-to-noise ratio by reducing long reply-all email excahnges.

Finally, working in the open can be great, but can also unintentionally suppress voices. At Programming Historian, for example, an ombudsperson is available for private chats and facilitation.

6 Publish periodically and recognize contributions

Like software, lessons should have releases of fixed content so that learners or instructors who may wish to use the material have a stable version to refer to for the duration of their use. These releases should be periodic so that improvements and adjustments are made available for new learners and instructors just starting their use. Periodic releases are also essential for enabling recognition of the contribution of authors and maintainers.

Academia has only a few ways of recognizing contributions. Until these are expanded, it is important to publish lessons in ways that traditional academic systems can digest. One is to give particular releases of lessons DOIs supplied by providers such as Zenodo (`https://zenodo.org/`) or DataCite (`https://www.datacite.org/`). Contributors can be listed as authors and the maintainers of the lesson as editors to differentiate recognition of their contributions. Each time the lesson is published, names and identifiers (e.g. ORCIDs (`https://orcid.org`)) should be gathered for all contributors.

A lesson release is a good opportunity to bring the material into a stable shape by fixing outstanding issues and merging contributions. Version control helps a lot in continuously maintaining a list of contributors but also in remembering which version is used for release (e.g., using branches or tags). Lesson releases should use a consistent naming scheme, such as the full year and month of the release, e.g. "2017.05". This is the scheme Software Carpentry has used in its releases of its lessons [6,7].

If lessons are being released regularly, automate the process with a shell script or something similar. Old versions of lessons should be archived in a discoverable location for reference.

7 Evaluate lessons at several scales

The purpose of feedback is to guide lesson development so that authors aren't designing and arguing in a vacuum. What people immersed in the lessons think needs fixing can often differ from what learners think.

Micro-scale feedback can be gathered by an instructor while teaching a particular lesson. Learners might provide feedback on things like typographical errors, clarity/ease of quiz questions and/or the order in which topics are presented, which the instructor can enter into a work-tracking system (e.g., GitHub issues) at the end of class. As well as encouraging direct verbal feedback, it's a good idea to provide learners with a means to provide feedback anonymously during class (e.g., on small pieces of paper like sticky notes).

Pre- and post-class surveys can be used to discover larger macro-scale issues. These issues often relate to the fact that it can be difficult for lesson developers to fully understand the frame of reference of their audience. For instance, a lesson might inadvertently assume prior knowledge that many learners don't have, which is information that can be collected in a survey. If possible, it's a good idea to conduct the post-class survey 30–60 days after the fact. This allows people time to reflect, meaning they are more likely to give accurate feedback on what they learned rather than how entertained they were.

Pre- and post-class surveys also are essential in focusing in on the audience for a given lesson. Referring back to Rule 1, it can sometimes be difficult for lesson developers to understand the frame of a given learner, so surveys (particularly post-class surveys) can reveal hidden assumed knowledge that can be expanded on or acknowledged to refine the target audience.

8 Reduce, re-use, recycle

Don't create a new lesson if there is an existing one that you can use or contribute to. Just as a scholar would not write a paper without a literature review, the same holds for lessons. Before writing that introduction to the Bash command line, for example, do a search: has anybody else written it? Is it complementary to your goals? Could it be tweaked or modified to meet your own goals? Could your planned lesson be tweaked to compliment the existing lessons so that topics aren't duplicated?

Before re-using content, make sure to check that the lesson is licensed in an open manner. Both Programming Historian and the Carpentry projects use the Creative Commons - Attribution license (`https://creativecommons.org/licenses/by/4.0/`), which allows people to share and adapt material for any purpose as long as they cite the original source.

The same questions of re-use come when thinking about recycling content of a lesson, such as images, data, figure, or code. Does the license cover that as well? If not, then ask permission, just as you would for any other material.

The converse of this rule is to license your own lessons in a similar open manner, and to make them discoverable. For example, when lessons are published (Rule 6), make sure that have the usual keywords in their bibliographic entries and HTML page headers.

9 Link to other resources

Most learners are unlikely to absorb everything there is to know about a topic from your lesson alone. This is partly a matter of scope—any interesting subject is too large to fit in a single lesson—but also a matter of level and direction. As Caulfield has argued [8], the best resources provide a chorus of explanations that offer many angles and approaches for any given topic, each of which may be the best fit for a different subset of people.

Find these resources and direct learners to them at strategic points. These resources may include textbooks, technical documentation, videos, and web pages; if a community or discussion forum exists for the topic, be sure to include them as well.

Doing this is a lot of work, and maintaining it is even more so, which makes building community around lessons (Rule 2) all the more important. In particular, it is vital to engage the learners as equal participants in that community: they should both be able to propose updates, corrections, and additions to lessons, and know that they are encouraged to do so (Rule 5).

10 You can't please everyone

No single lesson can be right for every learner. Two people with no prior knowledge of a specific subject may still be able to move at very different speeds because of different levels of general background knowledge. Similarly, lessons on ecology for learners in Utah and Florida will probably be more relatable if they use different examples. Equally, no lesson development community can serve all purposes. Some groups may prioritize rapid evolution, while others may prefer a "measure twice, cut once" approach.

If there are several complementary ways to explain something, or several points of views that can cohabit respectfully, it may be possible to present them side by side. There are actually good pedagogical reasons to do this even if contributors *don't* disagree: weighing alternatives fosters higher-order thinking, and as Caulfield has written, the most popular online question and answer sites are successful in part because they present a chorus of explanations geared at different levels and needs rather than a single one [8].

But sometimes choices must be made. The open source software community has wrestled with these issues for three decades, and has evolved some best practices to handle them [9]. As discussed in Rule 5, the first step is to have a clear governance structure and a clear, permissive license. Minor disagreements should be discussed openly and respectfully. If they cannot be resolved—i.e., if they turn out not to be so minor after all—then contributors should split off and evolve the lesson in the way they see best. (This is one of the reasons to have a permissive license.)

Forking rarely happens in practice. When it does, it is important to remember that we all share the same vision of better lessons, built together.

Conclusion

Every day, teachers all over the world spend countless hours duplicating each other's work. These ten rules provide an alternative: adopting the model of collaborative software development to make more robust and sustainable lessons that can be continually improved by those who use them. We hope that our experiences can help others teach more with more impact and less effort.

References

1. Ambrose SA, Bridges MW, DiPietro M, Lovett MC, Norman MK. How Learning Works. Jossey-Bass; 2010.
2. Wiggins G, McTighe J. Understanding by Design. 2nd ed. Association for Supervision and Curriculum Development; 2005.
3. Wilson G. Software Carpentry: Lessons Learned. F1000Research. 2016;3(62). doi:10.12688/f1000research.3-62.v2.
4. Christina Koch and Greg Wilson (eds.) Software Carpentry: Instructor Training; 2016. `https://zenodo.org/record/57571#.WS8huDOZPdQ`.
5. Wilson G. How to Teach Programming (And Other Things). Lulu.com; 2017.
6. Gabriel A. Devenyi and Christina Koch (eds.): Software Carpentry: The Unix Shell; 2015. `https://zenodo.org/record/27355#.WS8lajOZPdQ`.
7. Gabriel A. Devenyi and Ashwin Srinath (eds.): Software Carpentry: The Unix Shell; 2017. `https://zenodo.org/record/278226#.WS74tTOZPdQ`.
8. Caulfield M. Choral Explanations; 2016. https://hapgood.us/2016/05/13/choral-explanations/.
9. Fogel K. Producing Open Source Software. O'Reilly; 2005.