COMMENTARY

Software Carpentry: lessons learned [version 1; peer review: 3 approved]

Greg Wilson

Mozilla Science Lab, Mountain View CA, 94041-2021, USA

V1 First published: 19 Feb 2014, 3:62
<https://doi.org/10.12688/f1000research.3-62.v1>
Latest published: 28 Jan 2016, 3:62
<https://doi.org/10.12688/f1000research.3-62.v2>

Abstract

Over the last 15 years, Software Carpentry has evolved from a week-long training course at the US National Laboratories into a worldwide volunteer effort to raise standards in scientific computing. This article explains what we have learned along the way, the challenges we now face, and our plans for the future.

This article is included in the **Bioinformatics Education and Training Collection** collection.

Open Peer Review

Approval Status

	1	2	3
version 2 (revision) 28 Jan 2016			
version 1 19 Feb 2014	 view	 view	 view

1. **Juha Sorva**, Aalto University, Espoo, Finland
2. **Philip Guo**, University of Rochester, Rochester, NY, USA
3. **Hilmar Lapp** , National Evolutionary Synthesis Center (NESCent), Durham, NC, USA

Any reports and responses or comments on the article can be found at the end of the article.

Corresponding author: Greg Wilson (gvwilson@software-carpentry.org)

Competing interests: The author is an employee of the Mozilla Foundation. Over the years, Software Carpentry has received support from: The Sloan Foundation, Microsoft, NumFOCUS, Continuum Analytics, Enthought, The Python Software Foundation, Indiana University, Michigan State University, MITACS, The Mozilla Foundation, Queen Mary University London, Scimatic Inc., SciNET, SHARCNET, The UK Met Office, The MathWorks, Los Alamos National Laboratory, Lawrence Berkeley National Laboratory.

Grant information: Software Carpentry is currently supported by a grant from the Sloan Foundation.

The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Copyright: © 2014 Wilson G. This is an open access article distributed under the terms of the [Creative Commons Attribution License](#), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. Data associated with the article are available under the terms of the [Creative Commons Zero "No rights reserved" data waiver](#) (CC0 1.0 Public domain dedication).

How to cite this article: Wilson G. **Software Carpentry: lessons learned [version 1; peer review: 3 approved]** F1000Research 2014, 3:62 <https://doi.org/10.12688/f1000research.3-62.v1>

First published: 19 Feb 2014, 3:62 <https://doi.org/10.12688/f1000research.3-62.v1>

Introduction

In January 2012, John Cook posted this to his widely-read blog¹:

In a review of linear programming solvers from 1987 to 2002, Bob Bixby says that solvers benefited as much from algorithm improvements as from Moore's law: "Three orders of magnitude in machine speed and three orders of magnitude in algorithmic speed add up to six orders of magnitude in solving power. A model that might have taken a year to solve 10 years ago can now solve in less than 30 seconds".

A million-fold speed-up is impressive, but hardware and algorithms are only two sides of the iron triangle of programming. The third is programming itself, and while improvements to languages, tools, and practices have undoubtedly made software developers more productive since 1987, the speed-up is percentages rather than orders of magnitude. Setting aside the minority who do high-performance computing (HPC), the time it takes the "desktop majority" of scientists to produce a new computational result is increasingly dominated by how long it takes to write, test, debug, install, and maintain software.

The problem is that most scientists are never taught how to do this. While their undergraduate programs may include a generic introduction to programming or a statistics or numerical methods course (in which they are often expected to pick up programming on their own), they are almost never told that version control exists, and rarely if ever shown how to design a maintainable program in a systematic way, or how to turn the last twenty commands they typed into a re-usable script. As a result, they routinely spend hours doing things that could be done in minutes, or don't do things at all because they don't know where to start^{2,3}.

This is where Software Carpentry comes in. We ran 91 workshops for over 3500 scientists in 2013. In them, more than 100 volunteer instructors helped attendees learn about program design, task automation, version control, testing, and other unglamorous but time-tested skills⁴. Two independent assessments in 2012 showed that attendees are actually learning, and applying at least some of what we taught⁵:

The program increases participants' computational understanding, as measured by more than a two-fold (130%) improvement in test scores after the workshop. The program also enhances their habits and routines, and leads them to adopt tools and techniques that are considered standard practice in the software industry. As a result, participants express extremely high levels of satisfaction with their involvement in Software Carpentry (85% learned what they hoped to learn; 95% would recommend the workshop to others).

Despite these generally positive results, many researchers still find it hard to apply what we teach to their own work, and several of our experiments, most notably our attempts to teach online, have been failures.

From red to green

Some historical context will help explain where and why we have succeeded and failed.

Version 1: Red light

In 1995–96, the author organized a series of articles in *IEEE Computational Science & Engineering* titled, "What Should Computer Scientists Teach to Physical Scientists and Engineers?"⁶. The articles grew out of the frustration he had working with scientists who wanted to run before they could walk, i.e., to parallelize complex programs that were not broken down into self-contained functions, that did not have any automated tests, and that were not under version control⁷.

In response, John Reynders (then director of the Advanced Computing Laboratory at Los Alamos National Laboratory) invited the author and Brent Gorda (now at Intel) to teach a week-long course on these topics to LANL staff. The course ran for the first time in July 1998, and was repeated nine times over the next four years. It eventually wound down as the principals moved on to other projects, but two valuable lessons were learned:

1. Intensive week-long courses are easy to schedule (particularly if instructors are travelling) but by the last two days, attendees' brains are full and learning drops off significantly.
2. Textbook software engineering is not the right thing to teach most scientists. In particular, careful documentation of requirements and lots of up-front design are not appropriate for people who (almost by definition) do not yet know what they are trying to do. Agile development methods, which rose to prominence during this period, are a less bad fit to researchers' needs, but even they are not well suited to the "solo grad student" model of working so common in science.

Versions 2 and 3: Another red light

The Software Carpentry course materials were updated and released in 2004–05 under a Creative Commons license thanks to support from the Python Software Foundation⁸. They were used twice in a conventional term-long graduate course at the University of Toronto aimed at a mix of students from Computer Science and the physical and life sciences.

The materials attracted 1000–2000 unique visitors a month, with occasional spikes correlated to courses and mentions in other sites. But while grad students (and the occasional faculty member) found the course at Toronto useful, it never found an institutional home. Most Computer Science faculty believe this basic material is too easy to deserve a graduate credit (even though a significant minority of their students, particularly those coming from non-CS backgrounds, have no more experience of practical software development than the average physicist). However, other departments believe that courses like this ought to be offered by Computer Science, in the same way that Mathematics and Statistics departments routinely offer service courses. In the absence of an institutional mechanism to offer credit courses at some inter-departmental level,

this course, like many other interdisciplinary courses, fell between two stools.

It works too well to be interesting

We have also found that what we teach simply isn't interesting to most computer scientists. They are interested in doing research to advance our understanding of the science of computing; things like command-line history, tab completion, and "select * from table" have been around too long, and work too well, to be publishable any longer. As long as universities reward research first, and supply teaching last, it is simply not in most computer scientists' own best interests to offer this kind of course.

Secondly, despite repeated invitations, other people did not contribute updates or new material beyond an occasional bug report. Piecemeal improvement may be normal in open source development, but Wikipedia aside, it is still rare in other fields. In particular, people often use one another's slide decks as starting points for their own courses, but rarely offer their changes back to the original author in order to improve them. This is partly because educators' preferred file formats (Word, PowerPoint, and PDF) can't be handled gracefully by existing version control systems, but more importantly, there simply isn't a "culture of contribution" in education for projects like Software Carpentry to build on.

The most important lesson learned in this period was that while many faculty in science, engineering, and medicine agree that their students should learn more about computing, they *won't* agree on what to take out of the current curriculum to make room for it. A typical undergraduate science degree in the US or Canada has roughly 1800 hours of class and laboratory time; anyone who wants to add more programming, statistics, writing, or anything else must either lengthen the program (which is financially and institutionally infeasible) or take something out. However, everything in the program is there because it has a passionate defender who thinks it's vitally important, and who is likely senior to those faculty advocating the change.

It adds up

Saying, "We'll just add a little computing to every other course," is a cheat: five minutes per hour equals four entire courses in a four-year program, which is unlikely to ever be implemented. Pushing computing down to the high school level is also a non-starter, since that curriculum is also full.

The sweet spot for this kind of training is therefore the first two or three years of graduate school. At that point, students have time (at least, more time than they'll have once they're faculty) and real problems of their own that they want to solve.

Version 4: orange light

The author rebooted Software Carpentry in May 2010 with support from Indiana University, Michigan State University, Microsoft, MITACS, Queen Mary University of London, Scimatic, SciNet,

SHARCNet, and the UK Met Office. More than 120 short video lessons were recorded during the subsequent 12 months, and six more week-long classes were run for the backers. We also offered an online class three times (a MOOC *avant la lettre*).

This was our most successful version to date, in part because the scientific landscape itself had changed. Open access publishing, crowd sourcing, and dozens of other innovations had convinced scientists that knowing how to program was now as important to doing science as knowing how to do statistics. Despite this, though, most still regarded it as a tax they had to pay in order to get their science done. Those of us who teach programming may find it interesting in its own right, but as one course participant said, "If I wanted to be a programmer instead of a chemist, I would have chosen computer science as my major instead of chemistry".

Despite this round's overall success, there were several disappointments:

1. Once again, we discovered that five eight-hour days are more wearying than enlightening.
2. And once again, only a handful of other people contributed material, not least because creating videos is significantly more challenging than creating slides. Editing or modifying them is harder still: while a typo in a slide can be fixed by opening PowerPoint, making the change, saving, and re-exporting the PDF, inserting new slides into a video and updating the soundtrack seems to take at least half an hour regardless of how small the change is.
3. Most importantly, the MOOC format didn't work: only 5–10% of those who started with us completed the course, and the majority were people who already knew most of the material. Both figures are in line with completion rates and learner demographics for other MOOCs⁹, but are no less disappointing because of that.

The biggest take-away from this round was the need to come up with a scalable, sustainable model. One instructor simply can't reach enough people, and cobbling together funding from half a dozen different sources every twelve to eighteen months is a high-risk approach.

Version 5: green light

Software Carpentry restarted once again in January 2012 with a new grant from the Sloan Foundation, and backing from the Mozilla Foundation. This time, the model was two-day intensive workshops like those pioneered by The Hacker Within, a grassroots group of grad students helping grad students at the University of Wisconsin - Madison.

Shortening the workshops made it possible for more people to attend, and increased the proportion of the material they retained. It also forced us to think much harder about what skills scientists really needed. Out went object-oriented programming, XML, Make, GUI construction, design patterns, and software development life-cycles. Instead, we focused on a handful of tools (discussed in the

next section) that let us introduce higher-level concepts without learners really noticing.

Reaching more people also allowed us to recruit more instructors from workshop participants, which was essential for scaling. Switching to a “host site covers costs” model was equally important: funding is still needed for the coordinator positions (the author and two part-time administrative assistants at Mozilla, and part of one staff member’s time at the Software Sustainability Institute in the UK), but our other costs now take care of themselves.

Our two-day workshops have been an unqualified success. Both the number of workshops, and the number of people attending, have grown steadily (Figure 1 and Figure 2).

Figure 1. Cumulative number of workshops.

Figure 2. Cumulative enrolment.

**Attendance and number of Software Carpentry workshops
January 2012–January 2014**

2 Data Files

<http://dx.doi.org/10.6084/m9.figshare.928547>

More importantly, feedback from participants is strongly positive. While there are continuing problems with software set-up and the

speed of instruction (discussed below), 80–90% of attendees typically report that they were glad they attended and would recommend the workshops to colleagues.

What we do

So what does a typical workshop look like?

- *Day 1 a.m.*: The Unix shell. We only show participants a dozen basic commands; the real aim is to introduce them to the idea of combining single-purpose tools (via pipes and filters) to achieve desired effects, and to getting the computer to repeat things (via command completion, history, and loops) so that people don’t have to.
- *Day 1 p.m.*: Programming in Python (or sometimes R). The real goal is to show them when, why, and how to grow programs step-by-step as a set of comprehensible, reusable, and testable functions.
- *Day 2 a.m.*: Version control. We begin by emphasizing how this is a better way to back up files than creating directories with names like “final”, “really_final”, “really_final_revised”, and so on, then show them that it’s also a better way to collaborate than FTP or Dropbox.
- *Day 2 p.m.*: Using databases and SQL. The real goal is to show them what structured data actually is (in particular, why atomic values and keys are important) so that they will understand why it’s important to store information this way.

As the comments on the bullets above suggest, our real aim isn’t to teach Python, Git, or any other specific tool: it’s to teach *computational competence*. We can’t do this in the abstract: people won’t show up for a hand-waving talk, and even if they do, they won’t understand. If we show them how to solve a specific problem with a specific tool, though, we can then lead into a larger discussion of how scientists ought to develop, use, and curate software.

We also try to show people how the pieces fit together: how to write a Python script that fits into a Unix pipeline, how to automate unit tests, etc. Doing this gives us a chance to reinforce ideas, and also increases the odds of them being able to apply what they’ve learned once the workshop is over.

Of course, there are a lot of local variations around the template outlined above. Some instructors still use the command-line Python interpreter, but a growing number have adopted the **IPython Notebook**, which has proven to be an excellent teaching and learning environment.

We have also now run several workshops using R instead of Python, and expect this number to grow. While some people feel that using R instead of Python is like using feet and pounds instead of the metric system, it is the *lingua franca* of statistical computing, particularly in the life sciences. A handful of workshops also cover tools such as LaTeX, or domain-specific topics such as audio file processing. We hope to do more of the latter going forward now that we have enough instructors to be able to specialize.

We aim for no more than 40 people per room at a workshop, so that every learner can receive personal attention when needed. Where

possible, we now run two or more rooms side by side, and use a pre-assessment questionnaire as a sorting hat to stream learners by prior experience, which simplifies teaching and improves their experience. We do *not* shuffle people from one room to another between the first and second day: with the best inter-instructor coordination in the world, doing so would still result in duplication, missed topics, and jokes that make no sense.

Our workshops are often free, but many now charge a small registration fee (typically \$20–40), primarily because it reduces the no-show rate from a third to roughly 5%. When this is done, we must be careful not to trip over institutional rules about commercial use of their space: some universities will charge hundreds or thousands of dollars per day for use of their classrooms if any money changes hands. As this is usually several times more than a small registration fee would bring in, we usually choose the higher no-show rate as the lesser evil.

We have also experimented with refundable deposits, but the administrative overheads were unsustainable. It also does not help get around the rules mentioned in the previous paragraph, since money still appears to be changing hands in the university's eyes.

Commercial offerings

Our material^{10,11} is all covered by the Creative Commons Attribution license, so anyone who wants to use it for corporate training can do so without explicit permission from us. We encourage this: it would be great if graduate students could help pay their bills by sharing what they know, in the way that many programmers earn part or all of their living from working on open source software.

What *does* require permission is use of our name and logo, both of which are trademarked. We are happy to give such permission if we have certified the instructor and have a chance to double-check the content, but we do want a chance to check: we have had instances of people calling something “Software Carpentry” when it had nothing to do with what we usually teach. We’ve worked hard to create material that actually helps scientists, and to build some name recognition around it, and we’d like to make sure our name continues to mean something.

As well as instructors, we rely on local helpers to wander the room and answer questions during practical sessions. These helpers may be alumni of previous workshops who are interested in becoming instructors, grad students who have picked up some or all of this on their own, or members of the local open source community; where possible, we aim to have at least one helper for every eight learners.

We find workshops go a lot better if people come in groups (e.g., 4–5 people from one lab) or have other pre-existing ties (e.g., the same disciplinary background). They are less inhibited about asking questions, and can support each other (morally and technically) when the time comes to put what they’ve learned into practice after the workshop is over. Group sign-ups also yield much higher turnout from groups that are otherwise often under-represented, such as

women and minority students, since they know in advance that they will be in a supportive environment.

Small things add up

As in chess, success in teaching often comes from the accumulation of seemingly small advantages. Here are a few of the less significant things we do that we believe have contributed to our success.

Live coding

We use live coding rather than slides: it’s more convincing, it enables instructors to be more responsive to “what if?” questions, and it facilitates lateral knowledge transfer (i.e., people learn more than we realized we were teaching them by watching us work). This does put more of a burden on instructors than a pre-packaged slide deck, but most find it more fun.

Open everything

Our grant proposals, mailing lists, feedback from workshops, and everything else that isn’t personally sensitive are out in the open (see¹⁰ for links). While we cannot prove it, we believe that the fact that people can see us actively succeeding, failing, and learning earns us some credibility and respect.

Open lessons

This is an important special case of the previous point. Anyone who wants to use our lessons can take what we have, make changes, and offer those back by sending us a pull request on GitHub. As mentioned earlier, this workflow is still foreign to most educators, but it is allowing us to scale and adapt more quickly and more cheaply than the centralized approaches being taken by many high-profile online education ventures.

Use what we teach

We also make a point of eating our own cooking, e.g., we use GitHub for our web site and to plan workshops. Again, this buys us credibility, and gives instructors a chance to do some hands-on practice with the things they’re going to teach. The (considerable) downside is that it can be quite difficult for newcomers to contribute material; we are therefore working to streamline that process.

Meet the learners on their own ground

Learners tell us that it is important to them to leave the workshop with their own working environment set up. We therefore continue to teach on all three major platforms (Linux, Mac OS X, and Windows), even though it would be simpler to require learners to use just one. 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. We have also tried using virtual machines (VMs) in the cloud, but this makes us dependent on university-quality WiFi.

Collaborative note-taking

We often use [Etherpad](#) for collaborative note-taking and to share snippets of code and small data files with learners. (If nothing else, it saves us from having to ask students to copy long URLs from the presenter’s screen to their computers.) It is almost always

mentioned positively in post-workshop feedback, and several workshop participants have started using it in their own teaching.

We are still trying to come up with an equally good way to share larger files dynamically as the lessons progress. Version control does *not* work, both because our learners are new to it (and therefore likely to make mistakes that affect classmates) and because classroom WiFi frequently can't handle a flurry of multi-megabyte downloads.

Sticky notes and minute cards

Giving each learner two sticky notes of different colors allows instructors to do quick true/false questions as they're teaching. It also allows real-time feedback during hands-on work: learners can put a green sticky note on their laptop when they have something completed, or a red one when they need help. We also use them as minute cards: before each break, learners take a minute to write one thing they've learned on the green sticky note, and one thing they found confusing (or too fast or too slow) on the red. It only takes a couple of minutes to collate these, and allows the instructors to adjust to learners' interests and speed.

Pair programming

Pairing is a good practice in real life, and an even better way to teach: 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. To facilitate this, we strongly prefer flat (dinner-style) seating to banked (theater-style) seating; this also makes it easier for helpers to reach learners who need assistance.

Keep experimenting

We are constantly trying out new ideas (though not always on purpose). Among our current experiments are:

Partner and adapt: We have built a very fruitful partnership with the Software Sustainability Institute (SSI), which now manages our activities in the UK, and are adapting our general approach to meet particular local needs.

A driver's license for HPC: As another example of this collaboration, we are developing a "driver's license" for researchers who wish to use the DiRAC HPC facility. During several rounds of beta testing, we have refined an hour-long exam to assess people's proficiency with the Unix shell, testing, Makefiles, and other skills. This exam was deployed in late 2013, and we hope to be able to report on it by mid-2014.

New channels: On June 24–25, 2013, we ran our first workshop for women in science, engineering, and medicine. This event attracted 120 learners, 9 instructors, a dozen helpers, and direct sponsorship from several companies, universities, and non-profit organizations. Our second such workshop will run in March 2014, and we are exploring ways to reach other groups that are underrepresented in computing.

Smuggling it into the curriculum: Many of our instructors also teach regular university courses, and several of them are now using part or all of our material as the first few lectures in them. We

strongly encourage this, and would welcome a chance to work with anyone who wishes to explore this themselves.

Instructor training

To help people teach, we now run an online training course for would-be instructors¹². It takes 2–4 hours/week of their time for 12–14 weeks (depending on scheduling interruptions), and introduces them to the basics of educational psychology, instructional design, and how these things apply to teaching programming. It is necessarily very shallow, but most participants report that they find the material interesting as well as useful.

Why do people volunteer as instructors?

To make the world a better place. The two things we need to get through the next hundred years are more science and more courage; by helping scientists do more in less time, we are helping with the former.

To make their own lives better. Our instructors are often asked by their colleagues to help with computing problems. The more those colleagues know, the more interesting those requests are.

To build a reputation. Showing up to run a workshop is a great way for people to introduce themselves to colleagues, and to make contact with potential collaborators. This is probably the most important reason from Software Carpentry's point of view, since it's what makes our model sustainable.

To practice teaching. This is also important to people contemplating academic careers.

To help diversify the pipeline. Computing is 12–15% female, and that figure has been *dropping* since its high point in the 1980s¹³. Some of our instructors are involved in part because they want to help break that cycle by participating in activities like our workshops for women in science and engineering.

To learn new things, or learn old things in more detail. Working alongside an instructor with more experience is a great way to learn more about the tools, as well as about teaching.

It's fun. Our instructors get to work with smart people who actually want to be in the room, and don't have to mark anything afterwards. It's a refreshing change from teaching undergraduate calculus. . .

TODO

We've learned a lot, and we're doing a much better job of reaching and teaching people than we did eighteen months ago, but there are still many things we need to improve.

Too slow and too fast

The biggest challenge we face is the diversity of our learners' backgrounds and skill levels. No matter what we teach, and how fast or how slow we go, 20% or more of the room will be lost, and there's a good chance that a different 20% will be bored.

The obvious solution is to split people by level, but if we ask them how much they know about particular things, they regularly under- or over-estimate their knowledge. We have therefore developed a

short pre-assessment questionnaire (listed in the [Supplementary materials](#)) that asks them whether they could accomplish specific tasks. While far from perfect, it seems to work well enough for our purposes.

Finances

Our second-biggest problem is financial sustainability. The “host site covers costs” model allows us to offer more workshops, but does not cover the two full-time equivalent coordinating positions at the center of it all. We do ask host sites to donate toward these costs, but are still looking for a long-term solution.

Long-term assessment

Third, while we believe we’re helping scientists, we have not yet done the long-term follow-up needed to prove this. This is partly because of a lack of resources, but it is also a genuinely hard problem: no one knows how to measure the productivity of programmers, or the productivity of scientists, and putting the two together doesn’t make the unknowns cancel out.

What we’ve done so far is collect verbal feedback at the end of every workshop (mostly by asking attendees what went well and what didn’t) and to administer surveys immediately before and afterwards. Neither has been done systematically, though, which limits the insight we can actually glean. We are taking steps to address this, but the larger question of what impact we’re having on scientists’ productivity still needs to be addressed.

Meeting our own standards

One of the reasons we need to do long-term follow-up is to find out for our own benefit whether we’re teaching the right things the right way. As just one example, some of us believe that Subversion is significantly easier for novices to understand than Git because there are fewer places data can reside and fewer steps in its normal workflow. Others believe just as strongly that there is no difference, or that Git is actually easier to learn. While the large social network centered around GitHub is a factor in our choice as well, we would obviously be able to make better decisions if we had more quantitative data to base them on.

“Is it supposed to hurt this much?”

Fourth, getting software installed is often harder than using it. This is a hard enough problem for experienced users, but almost by definition our audience is *inexperienced*, and our learners don’t (yet) know about system paths, environment variables, the half-dozen places configuration files can lurk on a modern system, and so on. Combine that with two versions of Mac OS X, three of Windows, and two oddball Linux distributions, and it’s almost inevitable that every time we introduce a new tool, it won’t work as expected (or at all) for at least one person in the room. Detailed documentation has not proven effective: some learners won’t read it (despite repeated prompting), and no matter how detailed it is, it will be incomprehensible to some, and lacking for others.

Edit this

And while it may seem like a trivial thing, editing text is always harder than we expect. We don’t want to encourage people to use naive editors like Notepad, and the two most popular legacy editors on Unix (Vi and Emacs) are both usability nightmares. We now recommend a handful of GUI editors, but it remains a stumbling block.

Teaching on the web

Challenge number five is to move more of our teaching and follow-up online. We have tried several approaches, from MOOC-style online-only offerings to webcast tutorials and one-to-one online office hours via internet phone calls and desktop sharing. In all cases, turnout has been mediocre at the start and dropped off rapidly. The fact that this is also true of most high-profile MOOCs is little comfort.

What vs. how

Sixth on our list is the tension between teaching the “what” and the “how” of programming. When we teach a scripting language like Python, we have to spend time up front on syntax, which leaves us only limited time for the development practices that we really want to focus on, but which are hard to grasp in the abstract. By comparison, version control and databases are straightforward: what you see is what you do is what you get.

We also don’t as good a job as we would like teaching testing. The mechanics of unit testing with an xUnit-style framework are straightforward, and it’s easy to come up with representative test cases for things like reformatting data files, but what should we tell scientists about testing the numerical parts of their applications? Once we’ve covered floating-point roundoff and the need to use “almost equal” instead of “exactly equal”, our learners quite reasonably ask, “What should I use as a tolerance for my computation?” for which nobody has a good answer.

Standardization vs. customization

What we *actually* teach varies more widely than the content of most university courses with prescribed curricula. We think this is a strength, and one of the reasons we recruit instructors from among scientists is that they are best placed to customize content and delivery for local needs. However, we do need to be more systematic about varying our content on purpose rather than by accident.

Watching vs. doing

Finally, we try to make our teaching as interactive as possible, but we still don’t give learners hands-on exercises as frequently as we should. We also don’t give them as diverse a range of exercises as we should, and those that we do give are often at the wrong level. This is partly due to a lack of time, but disorganization is also a factor.

There is also a constant tension between having students do realistic exercises drawn from actual scientific workflows, and giving

them tasks that are small and decoupled, so that failures are less likely and don't have knockon effects when they occur. This is exacerbated by the diversity of learners in the typical workshop, though we hope that will diminish as we organize and recruit along disciplinary lines instead of geographically.

Better teaching practices

Computing education researchers have learned a lot in the past two decades about why people find it hard to learn how to program, and how to teach them more effectively^{14–18}. We do our best to cover these ideas in our instructor training program, but are less good about actually applying them in our workshops.

Conclusions

To paraphrase William Gibson, the future is already here: it's just that the skills needed to implement it aren't evenly distributed. A small number of scientists can easily build an application that scours the web for recently-published data, launch a cloud computing node to compare it to home-grown data sets, and push the result to a GitHub account; others are still struggling to free their data from Excel and figure out which of the nine backup versions of their paper is the one they sent for publication.

The fact is, it's hard for scientists to do the cool things their colleagues are excited about without basic computing skills, and impossible for them to know what other new things are possible. Our ambition is to change that: not just to make scientists more productive today, but to allow them to be part of the changes that are transforming science in front of our eyes. If you would like to help, we'd like to hear from you: please mail us at admin@software-carpentry.org.

Data availability

figshare: Attendance and number of Software Carpentry workshops January 2012–January 2014, doi: [10.6084/m9.figshare.92854719](https://doi.org/10.6084/m9.figshare.92854719)

Competing interests

The author is an employee of the Mozilla Foundation. Over the years, Software Carpentry has received support from: The Sloan Foundation, Microsoft, NumFOCUS, Continuum Analytics, Enthought, The Python Software Foundation, Indiana University, Michigan

State University, MITACS, The Mozilla Foundation, Queen Mary University London, Scimatic Inc., SciNET, SHARCNET, The UK Met Office, The MathWorks, Los Alamos National Laboratory, Lawrence Berkeley National Laboratory.

Grant information

Software Carpentry is currently supported by a grant from the Sloan Foundation.

The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Acknowledgements

The author wishes to thank Brent Gorda, who helped create Software Carpentry sixteen years ago; the hundreds of people who have helped organize and teach workshops over the years; and the thousands of people who have taken a few days to learn how to get more science done in less time, with less pain. Particular thanks go to the following for their comments, corrections, and inspiration:

- Azalee Bostroem (Space Telescope Science Institute)
- Chris Cannam (Queen Mary, University of London)
- Stephen Crouch (Software Sustainability Institute)
- Matt Davis (Datapad, Inc.)
- Luis Figueira (King's College London)
- Richard "Tommy" Guy (Microsoft)
- Edmund Hart (University of British Columbia)
- Neil Chue Hong (Software Sustainability Institute)
- Katy Huff (University of Wisconsin)
- Michael Jackson (Edinburgh Parallel Computing Centre)
- W. Trevor King (freelance software developer)
- Justin Kitzes (University of California, Berkeley)
- Stephen McGough (University of Newcastle)
- Lex Nederbragt (University of Oslo)
- Tracy Teal (Michigan State University)
- Ben Waugh (University College London)
- Lynne J. Williams (Rotman Research Institute)
- Ethan White (Utah State University)

Supplementary materials

Pre-assessment questionnaire

These are a few of the questions in our standard pre-assessment questionnaire.

- What is your career stage?
 - Undergraduate
 - Graduate
 - Post-doc
 - Faculty
 - Industry
 - Support Staff
 - Other:
- What is your discipline?
 - Space sciences
 - Physics
 - Chemistry
 - Earth sciences (geology, oceanography, meteorology)
 - Life science (ecology, zoology, botany)
 - Life science (biology, genetics)
 - Brain and neurosciences
 - Medicine
 - Engineering (civil, mechanical, chemical)
 - Computer science and electrical engineering
 - Economics
 - Humanities and social sciences
 - Tech support, lab tech, or support programmer
 - Administration
 - Other:
- In three sentences or less, please describe your current field of work or your research question.
- With which programming languages, if any, could you write a program from scratch which imports some data and calculates mean and standard deviation of that data?
 - C
 - C++
 - Perl
 - MATLAB
 - Python
 - R
 - Java
 - Other:
- What best describes how often you currently program?
 - I have never programmed.
 - I program less than one a year.
 - I program several times a year.
 - I program once a month.
 - I program once a week or more.
- What best describes the complexity of your programming? (Choose all that apply.)
 - I have never programmed.
 - I write scripts to analyze data.
 - I write tools to use and that others can use.
 - I am part of a team which develops software.
- A tab-delimited file has two columns showing the date and the highest temperature on that day. Write a program to produce a graph showing the average highest temperature for each month.
 - Could not complete.
 - Could complete with documentation or search engine help.
 - Could complete with little or no documentation or search engine help.
- Consider this task: given the URL for a project on GitHub, check out a working copy of that project, add a file called notes.txt, and commit the change.
 - Could not complete.
 - Could complete with documentation or search engine help.
 - Could complete with little or no documentation or search engine help.
- Consider this task: a database has two tables: Scientist and Lab. Scientist's columns are the scientist's user ID, name, and email address; Lab's columns are lab IDs, lab names, and scientist IDs. Write an SQL statement that outputs the number of scientists in each lab.
 - Could not complete.
 - Could complete with documentation or search engine help.
 - Could complete with little or no documentation or search engine help.
- How would you solve this problem: A directory contains 1000 text files. Create a list of all files that contain the word "Drosophila" and save the result to a file called results.txt.
 - Could not create this list.
 - Would create this list using "Find in Files" and "copy and paste".
 - Would create this list using basic command line programs.
 - Would create this list using a pipeline of command line programs.

References

1. Cook JD: **Moore's Law Squared**, 2012.
[Reference Source](#)
2. Hannay JE, Langtangen HP, MacLeod C, *et al.*: **How do scientists develop and use scientific software?** In *Second International Workshop on Software Engineering for Computational Science and Engineering (SECSE09)*. 2009; 1–8.
[Publisher Full Text](#)
3. Prabhu P, Jablin TB, Raman A, *et al.*: **A survey of the practice of computational science**. In *Proceedings of the 24th ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*, 2011.
[Publisher Full Text](#)
4. Wilson G, Aruliah DA, Brown CT, *et al.*: **Best practices for scientific computing**. *PLoS Biol.* 2014; **12**(1): e1001745.
[PubMed Abstract](#) | [Publisher Full Text](#) | [Free Full Text](#)
5. Aranda J: **Software Carpentry Assessment Report**, 2012.
[Reference Source](#)
6. Wilson GV: **What should computer scientists teach to physical scientists and engineers?** *IEEE Computational Science and Engineering*. Summer and Fall 1996; **3**(2): 46–65.
[Publisher Full Text](#)
7. Wilson G: **Where's the Real Bottleneck in Scientific Computing?** *Am. Sci.* 2006; **94**(1): 5.
[Publisher Full Text](#)
8. Wilson G: **Software carpentry: getting scientists to write better code by making them more productive**. *Comput Sci Eng.* 2006; **8**(6): 66–69.
[Publisher Full Text](#)
9. Jordan K: **MOOC completion rates: The data**, 2013.
[Reference Source](#)
10. **Software Carpentry web site**.
[Reference Source](#)
11. **Software Carpentry github learning materials repository**.
[Reference Source](#)
12. **Software Carpentry instructor training**.
[Reference Source](#)
13. **Women in computing: the gender gap**.
[Reference Source](#)
14. Guzdial M: **Why is it so hard to learn to program?** In Andy Oram and Greg Wilson, editors, *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, 2010; 111–124.
[Reference Source](#)
15. Guzdial M: **Exploring hypotheses about media computation**. In *Proc. Ninth Annual International ACM Conference on International Computing Education Research*. ICER'13, 2013; 19–26.
[Publisher Full Text](#)
16. Hazzan O, Lapidot T, Ragonis N: **Guide to Teaching Computer Science: An Activity-Based Approach**. Springer, 2011.
[Publisher Full Text](#)
17. Porter L, Guzdial M, McDowell C, *et al.*: **Success in introductory programming: What works?** *Communications of the ACM.* 2013; **56**(8): 34–36.
[Publisher Full Text](#)
18. Sorva J: **Visual Program Simulation in Introductory Programming Education**. PhD thesis, Aalto University, 2012.
[Reference Source](#)
19. Wilson G: **Attendance and number of Software Carpentry workshops January 2012 - January 2014**. *figshare*. 2014.
[Data Source](#)

Open Peer Review

Current Peer Review Status:

Version 1

Reviewer Report 24 March 2014

<https://doi.org/10.5256/f1000research.3787.r3816>

© 2014 Lapp H. This is an open access peer review report distributed under the terms of the [Creative Commons Attribution License](#), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Hilmar Lapp

National Evolutionary Synthesis Center (NESCent), Durham, NC, USA

The article describes some of the origins, driving motivations and lessons learned over the more than 15 years of iterative improvements and reboots of Software Carpentry, a brand of (meanwhile) travelling workshops teaching fundamental best practices in software engineering to programming scientists.

Software Carpentry has received wide acclaim, and helps fill critical gaps in a time when creating and using computational tools is becoming indispensable to increasingly many scientific fields. As such, the topic is of broad interest without question. The text is well-written, and in most places well argued. My only two overarching critiques are (1) that the author in some places seems to conflate cause and effect; and (2) that in some places I feel the reader is left hanging with a too little information. However, none of these rise to the level of calling into question the validity of the overall conclusions, and thus don't exceed what one might call "minor revisions".

Since this is an open review, I have chosen to record my detailed comments as public text annotations, using the Hypothes.is (<http://hypothes.is>) platform, with a transcription also provided below. A [PDF version](#) of the comments is also available.

The Hypothes.is version of these comments can be accessed at this URL:

<https://hypothes.is/stream/?user=hlapp&uri=http:%2F%2Ff1000research.com%2Farticles%2F3-62%2Fv1>

Unfortunately, the ordering of the comments on Hypothes.is appears to be in reverse chronological order (most recent first), and the comments should therefore be read last to first to align with reading the text start to end.

Any comments or replies to these comments should be made using the *F1000Research* 'Add yours' option but could also be added to Hypothes.is directly if desired.

Introduction

1. Paragraph 1: *"hardware and algorithms are only two sides of the iron triangle of programming"*

Is there a reference for this form of the Iron Triangle? Googling the phrase only turns up the well-known project management Iron Triangle, and its adaptation to software projects. The latter has Resources, Scope, and Time at its corners, not hardware, algorithms, and programming.

2. Paragraph 1: *"desktop majority"*

Do you mean the complement to those doing HPC? The phrase strikes me as needlessly cryptic. And are you sure that scientists developing HPC software are exempt from the trend you describe?

3. Paragraph 2: *"rarely if ever shown how to design a maintainable program in a systematic way"*

Are they not in fact taught, even if only indirectly, that programs are typically not revisited again once passed on (to the course instructor, for example), and hence thinking about maintainability is wasted effort?

4. Paragraph 3: *"learning, and applying"*

Assuming that learning binds to at least some of what we taught as well, the comma is extraneous. Or add a comma after "and applying".

5. Paragraph 4: *"many researchers still find it hard to apply what we teach"*

Researchers at-large, or researchers who participated in a Software Carpentry workshop?

From red to green

Version 1: Red light

1. Paragraph 1: *"i.e., to parallelize complex programs"*

This seems more an example to me than a restatement of run before they could walk. Thus, this should be "e.g." (or spelled out "for example").

2. Paragraph 2: *"(then director of the Advanced Computing Laboratory at Los Alamos National Laboratory)"*

Change parentheses to comma. The parenthetical phrase is important to make sense of the sentence. (And if similar contextual information can be given about Brent Gorda, i.e., information that helps to understand why he was invited, I suggest that be added too, as his current affiliation fails to explain that.)

3. Paragraph 2: *"In response, John Reynders (then director of the Advanced Computing Laboratory at Los Alamos National Laboratory) invited the author and Brent Gorda (now at Intel) to teach a week-long course on these topics to LANL staff. The course ran for the first time in July 1998, and was repeated nine times over the next four years."*

I suggest that the author highlights the major ways in which these courses differ from the SwC courses run today. As written now, deducing that from the two lessons learned is left as an exercise to the reader, and only those already familiar with SwC will know that indeed today's SwC workshops do differ in these ways.

Versions 2 and 3: Another red light

1. Paragraph 2: *"(even though a significant minority of their students, particularly those coming from non-CS backgrounds, have no more experience of practical software development than the average physicist"*

Remove parentheses.

2. Paragraph 2: *"In the absence of an institutional mechanism to offer credit courses at some inter-departmental level, this course, like many other interdisciplinary courses, fell between two stools."*

Perhaps this would be beyond the scope of the paper as a commentary, but it would be interesting to see whether this is then different at decidedly interdisciplinary programs, for example programs interfacing computational biology / computer science / math.

3. Paragraph 3: *"It works too well to be interesting"*

Based on context, "it" would be the SwC workshop or material. I suggest to reword so it is clear that it actually refers to the practices and tools being taught by SwC.

4. Paragraph 3: *"As long as universities reward research first, and supply teaching last, it is simply not in most computer scientists own best interests to offer this kind of course."*

If this is the main driver behind this kind of course not finding interest at university CS programs, is the situation then different at teaching-focused schools, such as small liberal arts colleges? There are small liberal arts colleges with strong CS programs; have they indeed been more welcoming to adopting SwC into their curricula?

5. Paragraph 4: *"This is partly because educators' preferred file formats (Word, PowerPoint, and PDF) can't be handled gracefully by existing version control systems, but more importantly, there simply isn't a "culture of contribution" in education for projects like Software Carpentry to build on"*

I'm not convinced that one isn't mostly or entirely a consequence of the other. Open source and collaborative development also was far less widespread in scientific software development before many of the barriers to that were significantly reduced by distributed version control such as Git, and usability and social coding focused resources such as Github. If the tools and file formats that are most widely used are simply refractory to collaboration, it's not a surprise if then a culture of collaboration is rare.

6. Paragraph 7: *"The sweet spot for this kind of training is therefore the first two or three years of graduate school. At that point, students have time (at least, more time than they'll have once*

they're faculty) and real problems of their own that they want to solve."

Perhaps it's primarily the "real problems of their own" that provide the motivation for having the time (to learn about addressing them). I.e., percentage-wise, how many students does SwC get today who take the course primarily because they have time, and who do not yet have real problems of their own for which they hope to learn solutions?

More importantly perhaps, does this not also point out a path for justifying the inclusion of SwC-inspired teaching units into undergraduate CS curricula? While for some (or most?) academic research career paths the relevance of version control mastery is perhaps less obvious, it's a qualification nearly all of industry ask of CS graduates applying for a software engineer position.

Version 4: orange light

1. Paragraph 1: *"The author rebooted Software Carpentry in May 2010 with support from Indiana University, Michigan State University, Microsoft, MITACS, Queen Mary University of London, Scimatic, SciNet, SHARCNet, and the UK Met Office."*

The backstory to what motivated (or necessitated?) the large consortium of funders is missing here. However, given the last paragraph in this section, it seems there would be interesting aspects of it that would help make setting up the argument. Does the large consortium reflect primarily wide buy-in to SwC's utility, or primarily the difficulty of obtaining enough funding from any one institution or partner? The last paragraph suggests it's the latter, but it's not clear.

2. Paragraph 1: *"MOOC"*

Spell out at first use.

3. Paragraph 2: *"Open access publishing, crowd sourcing, and dozens of other innovations had convinced scientists that knowing how to program was now as important to doing science as knowing how to do statistics."*

Is there evidence or references for the factors the author enumerates constituting the major driving causes? More specifically, the list is conspicuously missing the explosion of data that had swept, and has continued to sweep into almost every scientific discipline. Data richness is enormously powerful for science, yet wrestling insight from it at this scale invariably and pervasively requires computational processing. Maybe this is part of the "dozens of other innovations", but I would still argue that the data deluge has constituted a primary rather than a marginal driver of this landscape change.

4. Paragraph 4: *"Most importantly, the MOOC format didn't work"*

I think it's worth to qualify this statement in respect to the goals. As the paragraph goes on, it could be said that In some definition the MOOC format has worked (for example, compared to retention and completion rates of other MOOCs); the failure that the author reports presumably means chiefly that the goals laid out for a SwC course weren't met by

the MOOC format.

5. Paragraph 5: *"The biggest take-away from this round was the need come up with a scalable, sustainable model. One instructor simply can't reach enough people, and cobbling together funding from half a dozen different sources every twelve to eighteen months is a high-risk approach."*

For readers who aren't already fully on board with this, It would help to better set up the argument. Why is scaling up the model desirable or necessary? What is enough people? Couldn't funding also come from a single or few sources? Many courses are sustained by student tuition; how would this likely not work for SwC?

Version 5: green light

1. Paragraph 1 *"and backing from the Mozilla Foundation"*

The difference in wording suggests that the Mozilla Foundation's backing didn't come in the form of a grant. Can it be spelled out (at least broadly) what that support consisted of?

2. Paragraph 1: *"This time, the model was two-day intensive workshops"*

I'm curious as to why 2 days. The lessons learned stated earlier seem to say that attention drops after 3 days, not 2 days. Why was the decision made to shorten to 2 days, not 3 days?

3. Paragraph 1: *"The Hacker Within"*

Is there no link or other reference available?

4. Paragraph 3: *"Switching to a "host site covers costs" model was equally important: funding is still needed for the coordinator positions (the author and two part-time administrative assistants at Mozilla, and part of one staff member's time at the Software Sustainability Institute in the UK), but our other costs now take care of themselves."*

I'd find it really useful to spell this out a little more. What are "our other costs"? Instructor travel and expenses, room rental? What tasks do the coordinators perform, how does this scale? Or in other words, presumably there is a division between costs of operating that benefit from economies of scale, and those that do not. More insight into this division would be quite helpful as a lesson learned.

5. Paragraph 4: *"have grown steadily (Figure 1 and Figure 2)."*

The figures suggest a tapering off in the recent past. Is this more likely a fluke due to limited or censored data, or is there a trend showing?

6. Figure 2 : *"Enrolment"*

Typo (one instead of two 'l')

7. Description of Figshare Data: *"Hopefully these two effects more or less cancel out and should not detract from the overall trend displayed."*

Hope is nice but not a good basis on which to base scientific conclusions. Do you have evidence that suggests that neither fraction of people is significant with respect to those enrolled and attending both days? Evidence that both fractions of people has stayed relatively constant over time, and not changed more recently?

8. Paragraph 5: *"80-90% of attendees typically report"*

What does typically mean? 80-90% of all SwC enrolled students, or on average 80-90% of those enrolled in a workshop? I.e., how much variance is there between workshops?

What we do

1. Paragraph 5: *"While some people feel that using R instead of Python is like using feet and pounds instead of the metric system"*

I have heard concerns and objections some people have with R's syntax and way of doing things. But every language (including Python) has its detractors, and I don't think the particular concerns with R are necessarily widely known let alone understood. So I would suggest to either delete this clause (is it really needed for the argument?), or if chosen to be left in place, to substantiate it, at least by giving a reference to a fuller discussion of R's problems.

2. Paragraph 5: *"now that we have enough instructors to be able to specialize"*

It's probably not just a question of having instructors, but also of having demand for (and thus acceptance of) the SwC curriculum as useful in increasingly many disciplinary areas.

3. Paragraph 6: *"with the best"*
Insert "even" before "with".

4. Paragraph 7: *"As this is usually several times more than a small registration fee would bring in, we usually choose the higher no-show rate as the lesser evil."*

The biggest problem of a significant rate of no-shows is probably the fact that due to the space limitations other students who would have and benefitted from the course had to be denied because of the no-shows taking the space away. Have other possibilities to deter no-shows been explored (and if so, how effective have they been found)?

If the no-show rate is somewhat predictable (and it sounds like it is), then wait-listed students could be told to show up anyway on the day of the course, because there would likely be enough no-shows to make room for them. Has this been tried, and to what extent does it work?

5. Paragraph 9: *"What does require permission is use of our name and logo, both of which are trademarked. We are happy to give such permission if we have certified the instructor and have a*

chance to double-check the content, but we do want a chance to check: we have had instances of people calling something "Software Carpentry" when it had nothing to do with what we usually teach. We've worked hard to create material that actually helps scientists, and to build some name recognition around it, and we'd like to make sure our name continues to mean something."

This whole paragraph doesn't mention the words "brand", "brand recognition", and "brand reputation"; yet it is essentially about those concepts, isn't it? Why not say it directly?

Small things add up

Use what we teach

1. Paragraph 1: *"The (considerable) downside is that it can be quite difficult for newcomers to contribute material; we are therefore working to streamline that process."*

This needs some qualification to fully make sense as following from the preceding sentence. If the tools and approaches SwC teaches are good ones that "work", and SwC uses those tools and approaches itself, how can this be a downside, presuming that those able to contribute material are in fact familiar with those tools and approaches. I can imagine some ways in which this can still be a downside, but for clarity this should be spelled out better.

Keep experimenting

1. Paragraph 3: *"DiRAC"*

Spell out. Also, how about a URL?

2. Paragraph 5: *"Many of our instructors also teach regular university courses, and several of them are now using part or all of our material as the first few lectures in them."*

Isn't this somewhat contradicting some lessons learned stated earlier, which seemed to say that for several reasons the SwC curriculum faces impossibly high barriers for integration into university curricula, at least in the current environment. If contrary to expectation this has now become possible, can something be learned from the cases where it has been successfully integrated?

TODO

Long-term assessment

1. Paragraph 1: *"no one knows how to measure the productivity of programmers, or the productivity of scientists"*

I think this assertion needs better qualification to be really justified. Obviously, several ways to assess programmer productivity, and also scientist productivity, exist. Hiring and tenure committees regularly assess productivity of scientists. Arguably, the ways this is usually done suffers from various problems such as failing to encompass the full spectrum of products resulting from a scientist's work. Perhaps the author means that it is some of these shortcomings of current productivity assessment methods that effectively prevent

measuring the productivity impact of SwC's teachings, but that needs to be spelled out better.

"Is it supposed to hurt this much?"

1. Paragraph 2: *"naive"*

Is this meant to be "native"?

Teaching on the web

1. Paragraph 1: *"The fact that this is also true of most high-profile MOOCs is little comfort."*

If your goal is a high rate of retention and completion, that is. However, widening reach could also be a worthwhile goal. If a single MOOC reaches 10,000 students instead of 800 students reached by 20 physical SwC workshops, even a completion rate of only 10% will still have taught more students with the single MOOC than with the 20 physical workshops. MOOCs clearly aren't a panacea, and they may indeed be ill-suited to the learning objectives of SwC, but that and why this is so needs a little more depth to be convincing.

What vs. how

1. Paragraph 2: *"don't as good a job"*

Insert "do" after "don't".

2. Paragraph 2: *"xUnit-style framework"*

I'm embarrassed to ask what's an xUnit style framework. Spell out what that is, and/or add a reference or URL?

Standardization vs. customization

1. Paragraph 1: *"However, we do need to be more systematic about varying our content on purpose rather than by accident."*

As a reader, I feel left hanging by the section ending with this statement. Are there ideas about how this could be done, and to begin with, what were some of the problems encountered with the less systematic approach being practiced now? (The preceding text seems to only cite advantages.)

Watching vs. doing

1. Paragraph 1: *"We also don't give them as diverse a range of exercises as we should, and those that we do give are often at the wrong level."*

How do you know that this is the case? From feedback alone, or are there other kinds of observations or evidence?

2. Paragraph 2: *"though we hope that will diminish as we organize and recruit along disciplinary lines instead of geographically"*

Aren't you arguing above that diversity of backgrounds and starting skills is a constant challenge? It didn't seem from earlier arguments in the text that simply recruiting along a uniform discipline will address this problem.

Better teaching practices

1. Paragraph 1: *"We do our best to cover these ideas in our instructor training program, but are less good about actually applying them in our workshops."*

Is there some insight available into why instructors find it difficult to apply what they have been taught? Is it the imparting of these ideas that needs improvement, or are the ideas not as applicable in SwC as they were thought to be, or is there simply heterogeneity in that some ideas are much easier to apply than others? If the latter, which ones fall into which category?

Conclusions

1. Paragraph 1: *"To paraphrase William Gibson"*

I notice that it's not clear to what exact piece or event to source this. Perhaps still link to the William Gibson Wikiquote page, which includes the quote and its provenance?

Competing Interests: Greg Wilson is one of my collaborators on Data Carpentry, a fledgling offshoot of Software Carpentry aiming to teach best practices for data management.

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Reviewer Report 26 February 2014

<https://doi.org/10.5256/f1000research.3787.r3815>

© 2014 Guo P. This is an open access peer review report distributed under the terms of the [Creative Commons Attribution License](#), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Philip Guo

Department of Computer Science, University of Rochester, Rochester, NY, USA

This article is a retrospective on the past 15 years of the author leading the Software Carpentry effort to educate scientists about the practical value of computational tools. It also describes a set of instructional practices that have worked well (and some that have not worked so well) in this setting. It concludes with ongoing and future work to scale up Software Carpentry in light of large variations in instructor and student backgrounds, and continual changes in modern

computational tools.

This article is a great fit for F1000 due to the topic's relevance to researchers in life sciences (and across a diverse array of science fields), and to the author's cogent firsthand accounts of his experiences and reflections on a subject matter which he is well-suited to discuss.

My main high-level comment is that the language is colloquial in many parts of this article, with lots of asides enclosed (in parentheses). That is probably fine for an opinion-based article, and it makes the writing more personal and approachable. But the author should be aware that this is how the article appears to a first-time reader.

Here are some more detailed comments, none of which are pressing:

- "From red to green" -- it took me a while to understand the "red", "orange", "green" light analogy the author was making in this section. That seems to be culturally specific. (I don't think I've seen an orange traffic light.)
- "Versions 2 and 3: Another red light" - I didn't understand why these were two separate versions. Maybe it's simpler just to call this Version 2 and update the subsequent version numbers?
- "It works too well to be interesting" -- This blurb felt a bit harsh toward CS professors. It makes it sound like they teach only topics that lead to new publishable research. In my experience, teaching and research are fairly decoupled, so professors have no qualms about teaching materials from, say, 30-year-old compilers or databases textbooks, which are obviously not leading to new research. Perhaps a more likely explanation, which the author points out later in the article, is that there simply isn't room in CS curricula to offer these sorts of Software Carpentry-like materials, and nobody vouches strongly enough for them.
- Typo in caption: "Enrolment figures" -> "Enrollment figures"
- "What we do" - "Day 1 a.m.", etc. -- that's hard to parse. I thought the author meant "1am" like they were offering a class at 1 in the morning. Same with "1pm", "2am", "2pm". "Day 1 - morning" would be clearer.
- "during the practical" - I'm not familiar with this phrase. Is that a typo, or a figure of speech?
- "It's a refreshing change from teaching undergraduate calculus." -- would Software Carpentry instructors ordinarily teach calculus? Seems more like they would be teaching physics or programming or something.

Competing Interests: I have served as a volunteer helper in a Software Carpentry course. I was not paid for my participation, nor do I have any financial relationship with Software Carpentry.

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Reviewer Report 24 February 2014

<https://doi.org/10.5256/f1000research.3787.r3814>

© 2014 Sorva J. This is an open access peer review report distributed under the terms of the [Creative Commons Attribution License](#), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Juha Sorva

Department of Computer Science, Aalto University, Espoo, Finland

This is an insightful and well-written commentary on a timely topic. The article documents the past and present practices of Software Carpentry - a project for teaching scientists about computing - and reflects on the project's successes and failures. In doing so, it provides concrete examples of the teaching practices used as well as those discarded. Moreover, the article helps the reader to understand how teaching scientists to about computing is different from teaching computer science majors - a matter that is central to the efforts of Software Carpentry and to the interests of the growing numbers of scientists who need computing skill to work efficiently.

The commentary is well grounded in evidence from the research literature as well as the author's lengthy experience with the project. The achievements and challenges of Software Carpentry are discussed realistically and critically.

Additional comment: There is only so much you can learn in two days (the length of Software Carpentry's current workshops), and whatever you learn in that time is unlikely by itself to change your research practices dramatically. What would be interesting to know in the future is whether and how the workshop participants go about building their computing skills after attending a workshop.

Competing Interests: I have participated in a project (a collaboratively authored book on learning) led by the author of the commentary.

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Comments on this article

Version 1

Reader Comment (F1000Research Advisory Board Member) 29 Dec 2014

Kevin J Black, Department of Psychiatry, Washington University in St Louis, USA

"A small number of scientists can easily build an application that scours the web for recently-published data, launch a cloud computing node to compare it to home-grown data sets, and push

the result to a GitHub account; others are still struggling to free their data from Excel and figure out which of the nine backup versions of their paper is the one they sent for publication."

This quote is not only spot-on, it is brilliantly phrased. I would add that software design knowledge is spotty within as well as across scientists. I'm an example. I do brain imaging research. On the one hand, I took a numerical methods class in college, I have written programs over the years in FORTRAN, Smalltalk, Pascal, and C, and have written and used shell scripts (csh) a good bit; I have tried to implement principles from object-oriented programming and "design by contract," I understand what a relational database is and have made some in Access, I have a Github account, and I've tried to do things at work with R and Python. But on the other hand, the pretest in this article shows that I can't put most of those together usefully.

I tripped across the Software Carpentry web site a month or two ago, and I thought it was the best short introduction ever to show why the concepts they teach are so important for creating useful (reliable, reusable, pragmatic) software. Thank you for this work!

Competing Interests: None.

The benefits of publishing with F1000Research:

- Your article is published within days, with no editorial bias
- You can publish traditional articles, null/negative results, case reports, data notes and more
- The peer review process is transparent and collaborative
- Your article is indexed in PubMed after passing peer review
- Dedicated customer support at every stage

For pre-submission enquiries, contact research@f1000.com

F1000Research