Teaching Software Carpentry: Better Science through Science

W. Trevor King
Department of Physics, Drexel University

Most scientists use software to help with their research, and many scientists write their own software to control experiments, analyze data, or simulate theoretical behavior. During their education, scientists may be trained in experiment design and numerical analysis, but training in software construction is usually limited to a single introduction to C or numerical methods course. While languages and algorithms are certainly fundamental, the lack of formal training in higher level software design skills leads to difficulty in generating robust, reproducible scientific software. The Software Carpentry organization has been leading workshops and teaching courses at institutions around the world introducing scientists to the basics of software development: version control, testing, data management, modular construction, and more. Current workshops are two-day events with subject matter experts lecturing with students (helped by knowledgeable assistants) following along on their personal computer. With limited time and resources, maximizing the demonstrable efficiency of instruction is important. I will discuss factor analysis and related methods for designing and analyzing pre- and post-workshop surveys to discuss the effect of the workshop on student understanding.

Software Carpentry

Software Carpentry training tries to condense the best practices of software engineering into a set of guidelines for scientists. Following these guidelines will generally reduce the effort needed to produce, maintain, and validate scientific software.

1. Write programs for people, not computers
   1.1. A program should not require its readers to hold more than a handful of facts in memory at once
   1.2. Names should be consistent, distinctive, and meaningful
   1.3. Code style and formatting should be consistent
   1.4. All aspects of software development should be broken down into tasks roughly an hour long

2. Automate repetitive tasks
   2.1. Use an issue tracking tool
   2.2. Save recent commands in a file for re-use
   2.3. Use a build tool to automate scientific workflows

3. Use the computer to record history
   3.1. Software tools should be used to track computational work automatically

4. Make incremental changes
   4.1. Work in small steps with frequent feedback and course correction

5. Use version control
   5.1. Use a version control system
   5.2. Everything that has been created manually should be put in version control

6. Don’t repeat yourself (or others) (DRY)
   6.1. Every piece of data must have a single authoritative representation in the system
   6.2. Code should be modularized rather than copied and pasted
   6.3. Re-use code instead of rewriting it

7. Plan for mistakes
   7.1. Add assertions to programs to check their operation
   7.2. Use an off-the-shelf unit testing library
   7.3. Use all available resources when testing programs
   7.4. Turn bugs into test cases
   7.5. Use a symbolic debugger

8. Optimize software only after it works correctly
   8.1. Use a profiler to identify bottlenecks
   8.2. Write code in the highest-level language possible

9. Document design and purpose, not mechanics
   9.1. Document interfaces and reasons, not implementations
   9.2. Use re-actor code instead of explaining how it works
   9.3. Embed the documentation for a piece of software in that software

10. Collaborate
    10.1. Use pre-merge code reviews
    10.2. Pair programming when bringing someone new up to speed and when tackling particularly tricky problems
    10.3. Use a issue tracking tool

Workshops introduce these practices in the context of a range of tools and languages, depending on the needs of the students and the capability of the instructors.

Abstract

Factor analysis involves modeling measurable attributes as manifestations of underlying hidden factors. Given a set of surface attribute measurements \( x_i \) (test scores, survey results, . . . ), it is easy to extract the mean for each attribute. The other parameters \( A, y, \) and \( \mu \) can be found using a variety of techniques, including an iterative expectation maximization approach. The common factors \( y \) are uncorrelated Gaussian random variables with zero mean and unit variance. Because of this, \( y \) is distributed isotropically in an n-dimensional Gaussian sphere, and the model is independent of rotations on \( A \). To make the common factors easier to interpret, they are often rotated to maximize some criterion. One popular method is varimax rotation, which maximizes the varimax criterion:

\[
\text{Varimax criterion } V(A) = \sum_{i=1}^{d} \sum_{j=1}^{n} \left( \frac{A_{ij}^2}{\sum_{j=1}^{n} A_{ij}^2} \right)^2
\]

(2)

The varimax criterion \( V(A) \) is maximized by iteratively rotating \( A \) to make it an identity matrix with two dimensional rotation subspace. The optimal rotation angle \( \phi \) is:

\[
\phi = \frac{1}{2} \arctan \left( \frac{1}{\sqrt{d(d-1)}} \left( \sum_{p} \sum_{q} (A_{pq} - A_{qp}) \right)^2 \right)
\]

(4)

where \( i = \sqrt{-1} \) and \( d \) is the number of rows in \( A \) (also the number of surface attributes).

Survey implementation

Pre- and post-workshop surveys were distributed on paper during workshops, with demographic questions as well as subject-specific questions for a range of topics. Most questions were self-assessments such as “I can write a Python program that computes the average of three numbers,” with answers given on a five-point Likert scale. There were also open ended questions, but these are harder to analyze numerically. Collected answers were analyzed using factor analysis with a single hidden factor for each subject.

Likert-scale survey questions were split into three subjects:

1. Perception of Computational Ability (rank your Python coding ability. Unix scripting ability. . . )
2. Computational Understanding (I understand what cp does. I know what SOL is . . . )
3. Python Coding Ability (I can write a small Python program to solve a problem that is familiar to me. I can design a Python program in a modular manner. . . )

Factor analysis yielded pre- and post-workshop factor loadings for each question. Libarkin’s aggregate results for 56 participants in workshops at Michigan State University and University of Texas-Austin in May 2012 were:

<table>
<thead>
<tr>
<th>Question</th>
<th>Mean Pre</th>
<th>Mean Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perception of Computational Ability</td>
<td>1.73 ± 0.49</td>
<td>1.47 ± 0.47</td>
</tr>
<tr>
<td>Computational Understanding</td>
<td>2.06 ± 0.21</td>
<td>1.91 ± 0.21</td>
</tr>
<tr>
<td>Python Coding Ability</td>
<td>2.48 ± 0.63</td>
<td>3.00 ± 0.45</td>
</tr>
</tbody>
</table>

Some of her post-workshop Python Coding Ability factor loadings were:

<table>
<thead>
<tr>
<th>Question</th>
<th>Loading Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>I can write syntactically correct Python statements</td>
<td>0.81 ± 0.44</td>
</tr>
<tr>
<td>I understand the language structure of Python</td>
<td>0.78 ± 0.37</td>
</tr>
<tr>
<td>I can write logically correct code using Python</td>
<td>0.84 ± 0.46</td>
</tr>
<tr>
<td>I cannot complete a programming project unless someone else helps me get started.</td>
<td>-0.80 ± 0.44</td>
</tr>
</tbody>
</table>

Libarkin’s post-workshop results for 33 participants in a workshop at the University of Chicago in January 2013 were:

<table>
<thead>
<tr>
<th>Question</th>
<th>Loading Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>I can write syntactically correct Python statements</td>
<td>0.74 ± 0.26 ± 0.76</td>
</tr>
<tr>
<td>I understand the language structure of Python</td>
<td>0.79 ± 0.27 ± 0.60</td>
</tr>
<tr>
<td>I can write logically correct code using Python</td>
<td>0.85 ± 0.31 ± 0.60</td>
</tr>
<tr>
<td>I cannot complete a programming project unless someone else helps me get started.</td>
<td>-0.54 ± 0.27 ± 0.91</td>
</tr>
</tbody>
</table>

Her post-workshop results were:

<table>
<thead>
<tr>
<th>Question</th>
<th>Loading Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>I can write syntactically correct Python statements</td>
<td>0.79 ± 0.27 ± 0.76</td>
</tr>
<tr>
<td>I understand the language structure of Python</td>
<td>0.83 ± 0.31 ± 0.76</td>
</tr>
<tr>
<td>I can write logically correct code using Python</td>
<td>0.91 ± 0.37 ± 0.76</td>
</tr>
<tr>
<td>I cannot complete a programming project unless someone else helps me get started.</td>
<td>-0.97 ± 0.37 ± 0.76</td>
</tr>
</tbody>
</table>

Future work can build on these results by refining the questions (with low loading don’t give much information about a student’s overall understanding, either the question is bad, or the material behind it was poorly taught) and also by refining the workshop content to teach to the questions that are deemed important.

References