Writing Clean Scientific Software

Date: July 12 14, 2023

Presented by: Nick Murphy (Center for Astrophysics, Harvard & Smithsonian)

(The slides are available under “Materials from the Webinar” in the above link.)

Q. How did you justify investing in your personal growth in writing software given the realities of PDD?

A. Thank you for asking such an important question. I try to think about how helpful investing in our personal growth is for the long term. This doesn’t help much when we’re facing immediate pressure to get a job or a postdoc, but it’s something advisors can encourage their advisees to do.

It’s worth thinking about the big picture too. We really need to change the culture in research communities, so that we value people more than productivity. We also need to change embedded institutional practices that contribute to the publication-driven development paradigm and burnout.

Sometimes there are things you can do in your spare time too. During the pandemic, I got into the habit of listening to audiobooks and podcasts on this topic during my daily walks, and still do that on my walks to and from work.

Q. Can you comment on self-documenting code?

A. We want the code itself to communicate what the code is doing. In that sense, code should be self-documenting. It is often preferable to rewrite code to make it readable without comments than to add a comment describing how it works. There are times, though, when we really need to add comments.

Q. The benefits of longer names are well explained in the slides. However, how does one deal with the problem of long variable names in the code when they are part of long and/or complex equations? If one is trying to verify that the equation in the code matches the equation as presented in the theory described in a paper, this can be challenging. An error might find its way into the code because the code no longer looks like the equation in the paper. What about using unicode for variable names, so that they can use the same symbols as used in the associated theory manual/paper?

A. This is a fantastic question! For short equations, I usually prefer to spell out the full names of variables. This works less well for complicated equations. We want to make sure that code is easy to understand and change. If we are transcribing an equation from a paper and want to make sure it’s correct, then using the symbols would indeed make it easier to find and fix bugs.

One possibility that was mentioned was having a table in the code or documentation that lists standard symbols for variables. This can really help — so long as that table is maintained and kept up-to-date, which doesn't always happen.

When working in Python, I usually prefer to use Unicode symbols for variables rather than spelling them out (i.e. `μ` instead of `mu` and `ω` instead of `omega`), but I only use ASCII characters for the names of arguments to functions that others will use. Using Unicode symbols might make code a little more difficult to write.

C. Unicode causes some brutal cross-platform compatibility issues, too - all of the sudden it makes things like locales matter in unexpected ways. UTF-8/UTF-16/UTF-32 and byte order marks matter a lot to make unicode symbols work, which has caused serious problems for some of our work.

C. These are important points. Python defaults to UTF-8, so I've found Python/Unicode to work pretty well together in my experience. As you point out, this may not be the case for other languages/platforms/situations. If anyone is thinking about trying out Unicode characters in their code, I'd suggest trying out adding a single Unicode character first to see if it causes any problems.

Fun fact: if you type a LaTeX character like `\alpha` in a Jupyter notebook and press tab, it'll usually convert it to the Unicode character! Doesn't seem to work on all platforms, though.

Q. Do short, unsearchable variable names (e.g., loop indices like i, j, k, etc.) have a place in scientific software?

A. There are occasionally times where short, unsearchable variable names are appropriate. For example, in plasma physics, B is universally recognized as a symbol for the magnetic field. It might be reasonable to use single letter loop indices, but it's worth thinking about whether there might be a more descriptive name that can communicate what the variable means. It's a lot easier to search for a variable name with modern integrated development environments than it used to be, but there might be people editing the code using plain text editors, or doing a web search on the code on GitHub.

Q. Wouldn't the requirements for readable code be fulfilled in proper documentation of any given code base?

A. It is really important to have proper documentation and a contributor guide for shared projects, but it's not a replacement for making the code readable. For example, if we have to go back and forth between code and documentation to understand what different variables mean, then that would add an extra layer of difficulty to understanding and changing the code.

Q. How does units checking in a dynamic language like Python impact performance? (NOTE: In static languages like C++, the type checking can be done at compile time with no runtime performance penalty. But it can massively increase compile-time cost and exacerbate binary bloat.)

A. Unit operations do have a performance overhead, so using a units package would not be the best choice for high performance computing. I find units packages like `astropy.units` most helpful for interactive data analysis and cooking. When using them interactively, I haven't personally noticed any issues with performance compared to using NumPy arrays. Even if there's a bit of a performance penalty, using units packages can end up saving us time when they help us find errors.

C. For Python, wouldn't adding type hints help with this?

A. That's a good point. Type hints can be used to help with checking units. The `astropy.units.quantity_input` decorator can be used to check that the units of inputs and outputs match what is expected. I don't know of any static analysis tools like `mypy` that can check for this ahead of time, so it presently has to be done at runtime.

C. More of a remark, your first version of the is_electron function has no return statement if charge is correct but not the mass.

A. Great catch! I was worried that this was too simple of an example, but it really does show that even "simple" nested if statements can hide errors! I'm wondering if I should purposefully keep that error in the slide, just to show how bug-prone these statements can be. 😅

Q. How do you define "isclose" so that it is universally useful?

C. I don't think you can… There are tolerance settings: `numpy.isclose`

A. Agreed, it would be hard to define `isclose` universally. When working with astropy.units, for example, we could use `astropy.units.isclose` in order to handle units.

C. To understand how to define and use functions like `isclose`, I highly recommend this article: https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/ which explains why both absolute and relative tolerances are required.

Q. In addition to the advice on slide 19 for documentation of functions, I suggest in some cases that the assumptions that are necessary for the function to be applied are also documented.

C. I wonder if this is another example where information should be written in the code itself (e.g., assertions).

C. Oftentimes, preconditions when calling functions can be eliminated through design enhancements. Put another way: can the code be designed such that it is impossible to use in an incorrect way?

A. All really good points! Documentation of functions should contain all of the important information, including preconditions.

Q. Regarding the refactoring code example, you introduce image_level1-3. While this is easier to read, this allocates extra memory (level1 exists at the same time as level3 etc). In my applications we are usually memory limited, so we often can't do this. What are your tips in such a case?

A. This is another great example of competing tradeoffs: readability vs. memory. What I'd probably do is create an `Image` class which has methods that lets us do in-place operations. The high-level code would be:

```
image = Image("andromeda_galaxy.fits")
image.calibrate()
```

where `image.calibrate()` would call `image.subtract_bias()`, `image.remove_dark_current()`, etc. I wanted to avoid object-oriented programming in my examples since a lot of us are unfamiliar with it, but there are times when it can lead to much cleaner code. Julia has a convention of using function names that end with an exclamation point to denote that there are in-place operations on one of the arguments too.

Q. I find that code where high-performance is crucial makes following clean-code guidelines difficult (talking about code that runs for a couple million CPU hours so optimization is important). Do you have any tips for how to find a balance?

A. I'd suggest finding which portions of the code that are most computationally intensive. For the most computationally intensive parts, we can prioritize performance. For the other parts of the code, we can prioritize readability. In either case, one of the most important things about code is that we can change it easily. If code is difficult to change, it may end up unchanged, which has the potential to limit what research questions we are able to answer. It's also helpful to use programming languages that are more amenable to clean coding. I find Julia to be much more readable than Fortran, for example.

Q. When we join a new project that has a huge code base, and it was not developed with best practices. What is your advice on understanding, working on, or using such legacy code and be productive and efficient with it?

A. This is an extremely important question! There are books like *Working Effectively with Legacy Code* and *Refactoring: Improving the Design of Existing Code* which go into this. I believe there are more recent books too. One of the most important things to do is make sure that there are tests. Regression tests, in particular, will tell us if the code is behaving in the same way it used to behave, which will help us find out if any of our changes are having unintended consequences. How to work with legacy code would be a great topic for a future webinar too!

Q. Do you have an opinion on groups like Software Carpentry that teach these sorts of skills?

A. The Carpentries are awesome! I took a Software Carpentry workshop in early 2016 that probably set me on the path to presenting this webinar. Software Carpentry workshop are a great place to get started, but they don't replace the need for dedicated undergraduate and graduate courses on topics like research software engineering.

C. You can try also this: https://missing.csail.mit.edu

Q. In writing code for data analysis, do you have tips for documenting/naming code for linear algebra, e.g. tensor sizes and operations? In my experience, this area tend to be under-documented and under-tested since it's more difficul.t

A. Unfortunately, I don't have any suggestions for this. If anyone else reading has one, please feel free to share!

Q. Can you talk more about the role of testing? How do we balance the time writing tests vs. the time writing the functionality of our code?

A. Software testing is the best thing since sliced ~~bread~~ arrays! I'm of the view that automated tests quickly save us time and reduce frustration. Whatever code we write, we're going to have to check to make sure that it works. If we don't write automated tests, we'll need to check it manually…sometimes several times. Interactive testing is pretty time consuming, and it's easy to forget to do things. Writing clean automated tests is equally as important as writing clean code. The slides from today contain bonus content at the end which covers my thoughts on writing clean tests.

Q. What are your thoughts on literate programming?

A. I didn't have a good understanding of literate programming before today, but had a chance to look it up after. I'd have to learn more, but it seems to have a lot of commonalities with the idea of executable research articles. Thank you for bringing this up!

Q. It is good to practice ‘interface-oriented’ programming, but sometimes, when you have to work with a primitive programming language, like C, it can be difficult to separate the interface with the implementation details, especially when you want to have multiple sets of implementations under the same interface defined in a header file, dynamically switching them programmatically can be very difficult (sometimes even impossible), what to do in this case?

C. Beware of the ravioli code.

A. I hadn’t heard of ravioli code either. It refers to: “isolated bits of code that resemble ravioli. These are easy to understand individually but — taken as a group — add to the app’s call stack and complexity.”

The points you raise are important. I’ve run into similar problems with Fortran. This reminds me of situations where a program is a mixture of easy-to-test functionality and hard-to-test functionality. For example, there could be a graphical user interface (GUI) designed with the purpose of calculating plasma parameters. It would be easy to test the calculations, but hard to test the GUI. In this case, we can separate the easy-to-test code from the hard-to-test code.

Depending on the application, it might also be possible to use Python as “glue” which can call the C code. That way, the interfaces could be at a higher level.

C. For R users - here's a great style guide for documenting functions: https://style.tidyverse.org/documentation.html