Writing clean scientific software for plasma simulation

Nicholas A. Murphy

Center for Astrophysics | Harvard & Smithsonian

61st Annual Meeting of the APS Division of Plasma Physics
October 21–25, 2019

With thanks to the PlasmaPy & Python in Heliophysics Communities, and support from DOE grant DE-SC0016363 & NSF CSSI grant 1931388.

plasmapy

DOI 10.5281/zenodo.3491142

Background

- The **PlasmaPy** project is working toward an open source Python ecosystem for plasma research and education
  - Our code must be **usable**, **readable**, and **maintainable**
- We want our research to be **scientifically reproducible**
  - Code must be understandable in order to be auditable
- This talk with draw upon:
  - My own experiences
  - Lessons from PlasmaPy
  - Advice from software engineers[1]

---

[1] Some of the suggestions are from R. Martin's books on *Clean Code* and *Clean Architecture*, and the article on *Best Practices for Scientific Computing* by G. Wilson et al. (2014)

Common pain points and change preventers

- Difficult installation
  - Setting compiler flags & paths in Makefiles
  - Compiling and linking libraries
  - Fine-tuning fragile installation scripts
- Hard to read code
  - Function names like `dtpttf`
  - Magic numbers
  - Monolithic functions and classes
  - High-level code intermixed with low-level details
  - Obsolete or misleading comments
- Cryptic error messages
  - Obscures cause of problem

Common pain points and change preventers

- Lack of tests
  - Much more difficult to track down bugs
  - Fear that changes will introduce hidden bugs
  - Less confidence that code gives correct results
- Inadequate or obsolete documentation
  - Hard for newcomers to get started
  - Increases impact of other pain points
- Lack of interoperability[2]
  - Difficult to switch to a substantially different numerical method
  - Unnecessarily difficult to benchmark different codes

[2] Projects like OMFIT, openPMD, PICMI, and PlasmaPy are addressing this.

Why do these pain points exist?

- Programming **not covered in physics coursework**
- We tend to be **self-taught** as programmers
- Code is often **written in a rush** to get a paper out
- **Time pressure** prevents us from taking time to learn
- Most common measure of worth is **number of publications**
- Software is **not valued** as a research product

Moving beyond the legacy code era of plasma simulation

- We highly value:
  - Performance
  - Scalability
  - Verification & validation
- We should equally value:
  - User-friendliness
  - Readability
  - Maintainability
  - Auditability
  - Community
- **Code is communication!**

My definition of clean code

- Readable and modifiable
- Communicates intent
- Well-tested
- Sufficient, up-to-date documentation
- Succinct
- High-level code separated from implementation details
- No duplication
- Changing behavior means editing the code in **one** place
- Makes research fun!

Which is more readable?

```
>>> omega_ce = 1.76e7*(B/u.G)*u.rad/u.s

>>> electron_gyrofrequency = e * B / m_e
```

How do we choose good variable names?

- Names should reveal intention and meaning
- Use meaningful distinctions
  - Avoid ambiguity
- Choose clarity over brevity
  - Prefer long variable names over unclear abbreviations
  - Use pronounceable and searchable names
- Be consistent
  - Pick one word for each concept

Writing clean functions

- Functions should be **short**
- Functions should **do exactly one thing**
- Minimize the number of arguments
  - Define classes or types instead
- Separate high-level code from low-level details
  - Low-level code obscures intent of nearby high-level code
- High-level code should not depend on implementation details
  - Descend one level of abstraction per function

Write code as a top-down narrative

To **perform a numerical simulation**, we (1) read the input parameters, (2) make a grid, (3) set initial conditions, and (4) do the time advances.

Write code as a top-down narrative

To **perform a numerical simulation**, we (1) read the input parameters, (2) make a grid, (3) set initial conditions, and (4) do the time advances.

- To **read the input parameters**, we (1.1) open the input file, (1.2) read in each individual parameter, and (1.3) close the input file.

Write code as a top-down narrative

To **perform a numerical simulation**, we (1) read the input parameters, (2) make a grid, (3) set initial conditions, and (4) do the time advances.

- To **read the input parameters**, we (1.1) open the input file, (1.2) read in each individual parameter, and (1.3) close the input file.
  - To **read in each individual parameter**, we (1.2.1) read in a line of text, (1.2.2) parse the text, and (1.2.3) store the variable in a dictionary

“Program to an interface, not an implementation”

- Suppose we have a program that accesses atomic data
- We’re using the Chianti database, but want to use AtomDB
- If our high-level code repeatedly calls Chianti, then...
  - Changing to AtomDB will be a *pain*!
- If our high-level code calls functions that call Chianti
  - We will only need to modify these interface functions
  - Changes will be isolated to one place
  - High-level abstractions can remain the same!

Avoid mixing high-level code with low-level implementation details

Comments are not inherently good!

- As code evolves, comments often:
  - Become out-of-date
  - Contain misleading information
- "A comment is a lie waiting to happen."

Unhelpful comments

- Commented out code
  - Becomes less relevant quickly
  - Use version control instead
- Definitions of variables
  - Encode definitions in variable names instead
- Redundant comments

```
i += 1  # increment i
```

- Description of the *implementation*
  - Becomes obsolete quickly
  - Communicate the implementation in the code

Good commenting practices

- **Refactor code instead of explaining how it works**
- Explain the *intent* but not the implementation
- Amplify important points
- Explain why a possible approach *was not* used
- Provide context and references
- Update comments when updating code

Well-written tests make code *more* flexible

- Without tests:
  - Changes might introduce hidden bugs
  - Less likely to make changes for fear of breaking something
- With clean tests:
  - We can tell if our change broke something
  - Bugs can be tracked down quickly
- Testing best practices
  - Turn every bug into a new test
  - Write useful error messages

Julia combines the best features of Fortran, C, Python, Lisp, and MATLAB for scientific computing

- Julia uses a **just-in-time compiler** to achieve performance comparable to Fortran and C
- Uses **multiple dispatch** with **type inference**
  - Compile different versions for different input types
  - Select appropriate compiled version at runtime
- **Parallelism** is built into the language
- Julia is **interactive** $\Rightarrow$ much faster code development
- Only **dynamically-typed** language to reach **petascale**
- Can prototype in the same language used for high performance!

Suggestion: let's write the next generation of plasma simulation software in Julia!

More suggestions!

- Have a **code of conduct**
- Make code **open source**
- Upload code to **Zenodo** to make it citable
- Use **version control**
- Learn about **software architecture** and the SOLID principles
- Read about **design patterns**
- Engage in **friendly & supportive code review**
- Look up **automatic differentiation** if you have to calculate large Jacobians

Final thoughts

- **Code is communication!**
- We should take time to learn better programming skills
- Writing clean code is an iterative process
  - Constructive code review helps
- No single way to write clean code
- Please let's talk if you're interested in:
  - An open source Python ecosystem for plasma physics
  - Plasma simulation software in Julia
- These slides are in the PlasmaPy Community on Zenodo:

DOI 10.5281/zenodo.3491142