plasmapy

Writing Clean Scientific Software

Nick Murphy

Center for Astrophysics | Harvard & Smithsonian

With thanks to: the PlasmaPy, SunPy, and Astropy communities; the Python in Heliophysics Community; Sumana Harihareswara; Leonard Richardson; Sterling Smith; Janeway Granche; and many others.

Many of these suggestions are from: *Clean Code* & *Clean Architecture* by R. C. Martin, *Best Practices for Scientific Computing* by Wilson et al., *Code Complete* by S. McConnell, *Design Patterns* by Gamma et al, *Software Engineering for Science* edited by Carver et al., and the *Copyright Guide for Scientific Software* by Albert et al.

Where I’m coming from...

- This talk does not come from:
  - Years of experience writing clean code
- Rather, this talk comes from:
  - Years of experience writing messy code
  - And then living with the consequences...

Common pain points with scientific software

- Often not openly available
- Difficult installation
- Inadequate documentation
- Lack of user-friendliness
- Cryptic error messages
- Missing tests
- Unreadable code

Why do these pain points exist?

- Programming **not covered in physics courses**
- We tend to be **self-taught** programmers
- Worth often measured by **number of publications**
- Code is often **written in a rush**
- **Time pressure** prevents us from taking time to learn
- Software **not valued** as a research product

Consequences of these pain points

- Beginning research is hard
- Collaboration is difficult
- Duplication of functionality
- Research is less reproducible
- Research can be frustrating

How do we address these pain points?

- Make our software open source
- Use a high-level language
- Prioritize documentation
- Create automated test suites
- Develop code as a community
- **Write readable, reusable, & maintainable code**

My definition of clean code

- Readable and modifiable
- Communicates intent
- Well-tested
- Good documentation
- Succinct
- Lets us understand the big picture
- Makes research fun!

“Code is communication!”

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?

- **Reveal intention and meaning**
- **Choose clarity over brevity**
  - Longer names are better than unclear abbreviations
- **Avoid ambiguity**
  - Is `electron_gyrofrequency` an *angular* frequency?
  - Is volume in m^3 or in barn-megaparsecs?
- **Be consistent**
  - Use one word for each concept
- **Use searchable names**

Change numerical values to named constants

- In this expression:

```
velocity = -9.81 * time
```

  - Where does `-9.81` come from?
  - Are we sure it's correct?
  - What if we go to a different planet?

- Clarify intent by using *named constants* instead:

```
velocity = gravitational_acceleration * time
```

Decompose large programs into functions

- Huge chunks of code are hard to:
  - Read
  - Test
  - Keep track of in our mind
- Breaking code into functions helps us:
  - Re-use code
  - Improve readability
  - Isolate bugs

Don’t repeat yourself

- Copying and pasting code is fraught with peril
  - Bugs would need to be fixed *for every copy*
- Create functions instead of copying code
  - Simplifies fixing bugs
  - Reduces code duplication
- To change *one thing* in the code, we should only need to change it in *one place*

How do we write clean functions?

- Functions should:
  - Be **short**
  - Do **one thing**
  - Have **no side effects**
- Write explanatory note at top of function
- Avoid having too many required arguments
  - Use keywords or optional arguments
  - Define classes or data structures

High-level vs. low-level code

- High-level code:
  - Describes the big picture
  - “Abstracts away” implementation details
- Low-level code:
  - Describes implementation details
  - Contains concrete instructions for a computer

High-level vs. low-level cooking instructions

- High-level: describe goal of recipe
  - `Bake a cake`
- Low-level: a line in a recipe
  - `Add 1 barn-Mpc of baking powder to flour`

Avoid mixing low-level & high-level code

- Mixing low-level & high-level code makes it harder to:
  - Understand what the program is doing
  - Change how code is implemented
- Separate high-level, big picture code from low-level implementation details

Write code as a top-down narrative[1]

To **perform a numerical simulation**, we:

1. Read in the inputs
2. Set initial conditions
3. Perform the time advances
4. Output the results

[1] This is called the “Stepdown Rule” in *Clean Code* by R. Martin.

Write code as a top-down narrative

To **perform a numerical simulation**, we:

1. To **read in the inputs**, we:
    1.1. Open the input file
    1.2. Read in each individual parameter
    1.3. Close the input file
2. Set initial conditions
3. Perform the time advances
4. Output the results

- Each of these lines can be a function

Write code as a top-down narrative

To **perform a numerical simulation**, we:

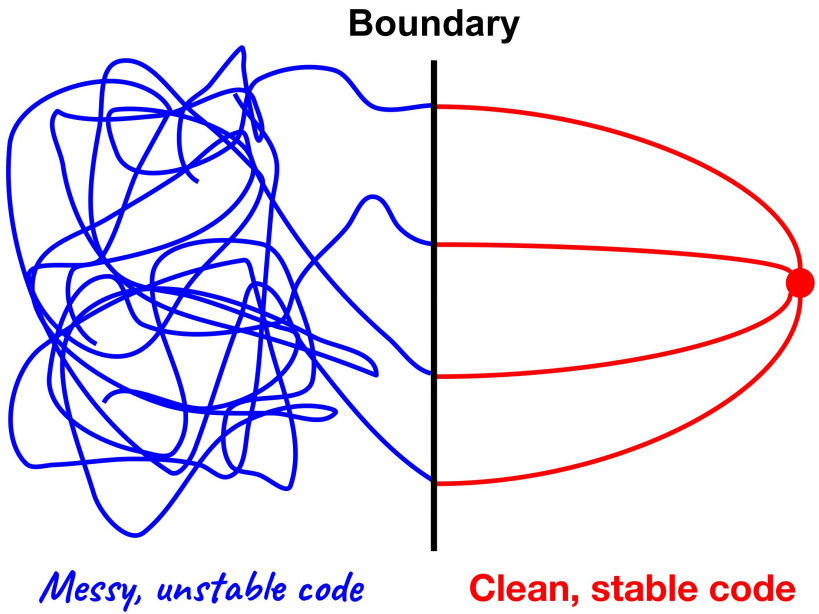
1. To **read in the inputs**, we:
   1.1. Open the input file
   1.2. To **read in each individual parameter**, we:
      1.2.1. Read in a line of text
      1.2.2. Parse the text
      1.2.3. Store the variable
   1.3. Close the input file
2. Set initial conditions
3. Perform the time advances
4. Output the results

“Program to an interface, not an implementation”

- Suppose our program uses atomic data
- We’re using the Chianti database, but want to use AtomDB
- If our high-level code repeatedly calls Chianti, then…
  - Switching to AtomDB will be a pain!
- If our high-level code calls *functions that call Chianti*…
  - We need only make these *interface functions* call AtomDB instead
  - The high-level code can remain unchanged!

Separate stable & unstable code with boundaries

- These *interface functions* represent a **boundary**
- The **clean, stable code** depends directly on the **boundary**, not the *messy unstable code*
- The **boundary** should be stable



Strive for high cohesion & low coupling

- **Cohesion** is the degree to which the contents of a module belong together
- **Coupling** is the degree to which the contents of a module depend on other modules
- Code elements that change together at the same time for the same reasons belong together
- Separate code elements that do not change with each other

Comments are not inherently good!

- As code evolves, comments often:
  - Become out-of-date
  - Contain misleading information
  - Get displaced from the corresponding code
- “A comment is a lie waiting to happen” 🙀

Not so helpful comments

- Commented out code
  - Quickly becomes irrelevant
  - Use version control instead
- Definitions of variables
  - Encode definitions in variables names instead
- Redundant comments

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

- Description of the implementation (usually)
  - Becomes obsolete quickly
  - Communicate the implementation in the code itself

Helpful commenting practices

- Explain the *intent* but not the implementation
  - Refactor code instead of explaining how it works
- Amplify important points
- Explain why an 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 change code for fear of breaking something
- With clean tests:
  - We know if a change broke something
  - We can track down bugs more quickly
- “Legacy code is code without tests.”
  — from *Working Effectively with Legacy Code* by M. Feathers

Why do we write tests?

- To provide confidence that our code gives correct results
- So we can define what “correct” behavior actually is
- To catch and fix bugs
  - Preferably as soon as we introduce them
- To keep track of bugs to be fixed later
- To show future developers how code should be used
- So we can change the code with confidence that we are not introducing hidden bugs elsewhere in the program

A minimal software test

```
def test_douglas_adams_number():
    """Test answer to life, the universe, & everything."""
    assert 6 * 9 == 42,"Universe is broken."
```

- Descriptive name
- Descriptive docstring
- A check that a condition is met
- Descriptive error message if condition is not met

Testing best practices

- Write assertions directly into code
  - Raise error if `positive_number` becomes negative
- Turn every bug into a new test
  - Tells us when that bug is fixed
  - Prevents bug from happening in future
- Make tests deterministic
  - Hard to tell if a test that fails intermittently is fixed
  - Use same random seed
- Run tests often!!!!
  - To find bugs as soon as we introduce them

Error messages are vital documentation

- The best error messages help users pinpoint a problem and understand how to fix it within seconds
- Poorly written error messages can cause hours of frustration or cause us to give up

How do we write clean error messages?

- Error messages should:
  - State the problem
  - Describe why it happened
  - Help us fix the problem
- Error messages should be:
  - Helpful!
  - Friendly and supportive
  - Succinct
  - Non-technical when possible
- Provide enough information to solve the problem with minimal extraneous information

Test-driven development

- Most common practice:
  - Write a function
  - Write tests for that function
  - Fix bugs in the function
- Test-driven development
  - Write tests for a function
  - Write and edit the function until tests pass
- Advantages of writing tests first
  - Makes us think about what each function will do
  - Saves us time!

How do we know what tests to write?

- Test some typical cases
- Test special cases
  - If a function acts weird near `0`, test at `0`
- Test near and at the boundaries
  - If a function requires a value ≥ `1`, test at `1` and `1.001`
- Test that code *fails* correctly
  - If a function requires a value ≥ `1`, test at `0.999`

Testing strategies for numerical modeling

- Test against analytical solutions
  - Equilibria, waves, etc.
- Test equilibrium configurations
- Test against conservation properties
  - Conservation of mass, momentum, & energy
- Test convergence properties
  - Example: test that a 4th order accurate algorithm actually is 4th order
- Test limiting cases

Prioritize readability over computational efficiency

- Readability is *usually* more important than speed
  - Computers are fast and getting faster
  - Our time is more valuable than computing time
- A tenfold improvement is irrelevant for code that takes a millisecond to run and is only run occasionally
- We should optimize code:
  - Only when necessary
  - After the code is working correctly
  - After identifying the bottlenecks
- Clean coding requires balancing competing priorities

When is it worth taking time to write clean code?

- Some clean coding habits save time quickly
  - Writing short functions that do one thing
  - Writing tests (instead of interactively testing a function)
- Interactive exploration of a data set does not necessitate particularly clean code
- Investing extra time to write clean code, documentation, and tests is worthwhile if:
  - You'll re-use the code
  - The code will be shared with others
- Avoid perfectionism
  - Better to *mostly* (but not completely) follow this advice

Choosing an open source license

- If software is shared *without* a license:
  - Copyright protections stay with original developers
  - Users not given legal right to use, modify, & share software
- Adopt an *unmodified* Open Source Initiative (OSI) approved license
  - Customizations can have unforeseen consequences
  - Modified licenses might not meet Open Source Definition
- *Permissive* licenses cause fewer headaches than *copyleft* licenses
  - *Permissive* licenses allow code to be shared under compatible licenses
  - *Copyleft* licenses require code to be shared under same license
- **Use the open source license most widely adopted in your field**
  - Physics & astronomy: BSD 2-clause or BSD 3-clause license

Summary

- **Code is communication!**
- Break up complicated code into manageable chunks
  - Write short functions that do one thing
  - Separate big picture code from implementation details
- Refactor code rather than explaining how it works
  - Communicate the implementation in the code itself
- Well-written tests make code *more* flexible
  - Turn every bug into a test case
- Use the open source license most common in your field

Final thoughts

- Worthwhile to take time to learn
  - Helpful to practice reading code
  - Many resources exist on clean coding best practices
- More than one way to write clean code
- Writing clean code is an iterative process
  - Often helpful to get it working first, and then improve it
  - Feedback from constructive code review helps