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; Sterling Smith; Sumana Harihareswara; Leonard Richardson; and many others.

About me

- Grew up in Michigan
  - Side effect: I really like suspension bridges
- Undergrad at the University of Michigan
  - Side effect: I know about rabbit dietary preferences
- Grad school in astronomy at the University of Wisconsin
  - Thesis: simulating magnetic reconnection
  - Side effect: I started reading sci-fi poetry
- Have been at Center for Astrophysics for a decade
  - Side effect: getting to drive ~1800 km with a very grumpy cat
- Helped begin the PlasmaPy project

What is PlasmaPy?

Mission

To grow an ***open source software ecosystem*** *for plasma research & education*

Anatomy of a software ecosystem

- **PlasmaPy core package**
  - Most frequently needed functionality
  - Under active development (clean coding is important)
- **Affiliated packages**
  - Will contain specialized functionality
- **Educational resources**
  - Introduce plasma concepts using PlasmaPy
- **Community**
  - You’re welcome to say hi on our Riot chat room!
  - Or join in our weekly community meetings!

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... 😿
    - Lessons from PlasmaPy
- Many of these suggestions come from:
    - *Clean Code* and *Clean Architecture* by R. Martin
    - *Best Practices for Scientific Computing* by G. Wilson et al.
    - Pseudo-random blog posts on the `World Wide Web`™
    - Pseudo-random friends

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 an automated test suite
- 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
- Can 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
  - Can re-use code
- If we want 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

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” 🙀

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
  - 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

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

Testing best practices

- Write assertions into code
  - Raise error if `positive_number` is negative
- Turn every bug into a new test
  - Tells us when that bug is fixed
  - Prevents bug from happening in future
- Error messages should
  - Describe what went wrong
  - Provide information needed to fix problem
- Run tests often!!!!
  - To find bugs as soon as we introduce them

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?

- We write tests to:
  - Provide confidence that code gives correct results
  - Help us find and track down bugs
- 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 `1.00001`
- Test that code *fails* correctly
  - If a function requires a value ≥ `1`, test at `0.999`

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. Construct the grid
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. Construct the grid
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. Construct the grid
3. Perform the time advances
4. Output the results

When is it worth taking time to write clean code?

- Writing clean code requires time and effort 😿
  - Sometimes it's worth it 😸
  - Sometimes it's not 🙀
- Code to be used once needn't be (very) clean 😹
- Taking time to write clean code is worth it when:
  - You'll re-use the code 😺
  - The code will be shared with others 🐈 🐈 🐈 🐈 🐈 🐈
- Avoid perfectionism 😼
  - Best to *mostly* (but not completely) follow this advice

Announcing APS DPP open source mini-conference

- Mini-conference on: *Growing an open source software ecosystem for plasma science*
  - To be held at virtual APS DPP meeting in November
  - Abstracts due by June 29 at 5 pm EDT

Final thoughts

- **Code is communication!**
- Helpful to practice reading code
- Important to take time to learn
- Break up complicated code into manageable chunks
- Writing clean code is an iterative process
- No single way to write clean code
- I’m happy to talk more later about PlasmaPy!

Extra slides

“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