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; US-RSE; the Carpentries; Sumana Harihareswara; Leonard Richardson; Erik Everson; Dominik Stańczak-Marikin; Sterling Smith; Janeway Granche; Royce James; and the National Science Foundation.

Many of these tips come from resources such as: *Clean Code* and *Clean Architecture* by R. Martin, *The Pragmatic Programmer* by Thomas & Hunt, *Design Patterns* by Gamma et al., *Best Practices for Scientific Computing* by G. Wilson et al., and *Unit Testing Principles, Practices, and Patterns* by V. Khorikov

Where I’m coming from...

- These suggestions do not come from:
    - Years of experience writing clean code
- Rather, these suggestions come from:
    - Years of experience writing messy code 😅
    - And then living with the consequences… 🙀

Common pain points with scientific software

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

Why do these pain points exist?

- Programming **not covered in science courses**
- Scientists 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

Publication-driven development (PDD)

- Measure worth of researchers by number of publications
- Write code in a rush to get articles published
- Deprioritize user-friendliness
- Prioritize journal articles over documentation & tests
- Fund research projects, not infrastructure & maintenance
- Avoid training and hiring research software engineers
- Build up technical debt over time

PDD gives us legacy code!

Consequences of these pain points

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

Alternative: Sustainability-driven development

- Cover research software engineering skills in coursework
- Grow open source software ecosystems
- Invest in long-term health of research software
- Regularly refactor code to reduce technical debt
- Prioritize documentation & continuous integration testing
- Shift towards executable research articles
- Develop code as a community
- **Write readable, reusable, & maintainable code**

My definition of clean code

- Readable
- Easy to change
- Communicates intent
- Well-tested
- Well-documented
- Succinct
- Navigable
- Lets us understand the big picture and little details
- Makes research fun!

“Code is communication!”

Which is more readable?

```
>>> omega_ce=1.76e7*B
>>> electron_gyrofrequency = e * B / m_e
```

How do we choose good variable names?

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

Clean Code, Ch. 2

Measure the length of a variable name not by the number of characters, but by the time needed to understand its meaning!

When can we use mathematical symbols as variables?

- Spelling out variable names (e.g., `angular_frequency`):
    - Better understandability
    - Improved searchability
    - Occasionally end up with long lines of code
- Mathematical symbols as variable names (e.g., `omega`)
    - Compact/mathematical notation
    - Easier to compare code to equations from a book or article
    - ⚠️ Risk of confusing people who are unfamiliar with notation
- How do we mitigate the risk of confusion?

Using mathematical symbols as variable names

- Use standard symbols when possible
- Define symbols near where they are used
- Include equations in documentation of functions (with references to book or articles)
- Specify units! Is `time` in seconds or gigayears?
- Can keep track of mathematical symbols in a table
  - As long as the table is updated!
- Can use UTF-8 symbols in variable names in languages like Python and Julia (i.e. `θ` instead of `theta`)

Change numbers 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? 🪐
- **Use named constants** to clarify intent:

```
velocity = gravitational_acceleration * time
```

Clean Code, Ch. 17

Use quantities with units instead of numbers

- In this expression:

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

  - What units does `-9.81` have?
- **Use a units package** to prevent [$328M mistakes]() 🛰️

```
from astropy import units
acceleration = -9.81 * units.meter / units.second**2
time = 15 * units.second
velocity = 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:
    - Reuse code
    - Improve readability
    - Improve testability
    - Isolate bugs 🪲

Don’t repeat yourself (DRY)

- 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*

The Pragmatic Programmer

How do we write clean functions?

- Functions should:
  - Be **short**
  - Do **one thing**
  - Have **no side effects**
- Use **pure functions**

Clean Code, Ch. 3

Complex control flow makes code hard to read

```
def is_electron(charge, mass):
    if isclose(charge, -1.67e-19):
        if isclose(mass, 9.11e-31):
             return True
        else:
            return False
    else:
        return False
```

- Nested `if`/`else` statements and `for` loops make code:
    - Harder to understand
    - Harder to modify
    - More bug-prone

Use guard clauses instead of nested conditionals

```
def is_electron(charge, mass):

    if not isclose(charge, -1.67e-19):
        return False

    if not isclose(mass, 9.11e-31):
        return False

    return True
```

- Take care of edge cases first to simplify subsequent code

Document each function

- State what the function does
- Describe arguments provided to the function
- Describe the value returned by the function
- Include usage examples
- Include additional notes & references as necessary

Adapted from numpydoc style guide

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 the implementation
- **Separate high-level, big picture code from low-level implementation details**

Clean Code, Ch. 3

Write code as a top-down narrative*

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

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

---

*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

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

How do we apply this stepdown rule?

```
def calibrate_observation(raw_image):
    # Subtract bias
    (~20 lines of code)
    # Remove dark current
    (~20 lines of code)
    # Flag cosmic rays
    (~20 lines of code)
```

- This function does more than one thing!
- What if we want to do only one of these steps?
- How do we test each individual step?

The extract function refactoring pattern

Convert each section of code into its own function:

```
def subtract_bias(image): ...

def remove_dark_current(image): ...

def flag_cosmic_rays(image): ...

def calibrate_observation(raw_image):
    image_level1 = subtract_bias(raw_image)
    image_level2 = remove_dark_current(image_level1)
    image_level3 = flag_cosmic_rays(image_level2)
    return image_level3
```

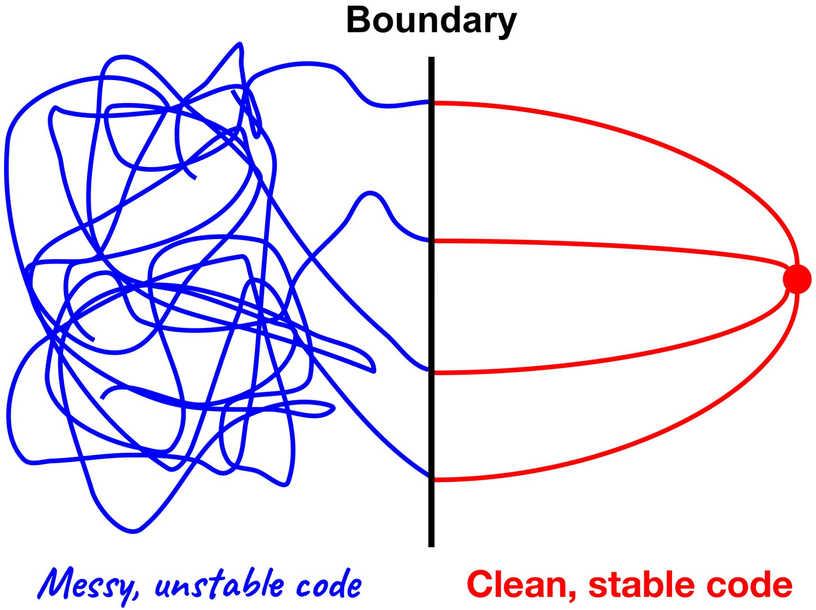
“Program to an interface, not an implementation”

- 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! 😸

Quote from *Design Patterns*, Ch. 1

These interface functions represent a boundary

- Put a **boundary** between stable & unstable code
- 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

Clean Architecture, Ch. 13

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” 🙀

Clean Code, Ch. 4

Potentially unhelpful comments

- **Commented out code**
  - Quickly becomes irrelevant
  - Keep track of old code using version control instead
- **Definitions of variables**
  - Encode definitions in variable names instead

```
# torque
tau = ...
```

← definition in comment

```
torque = ...
```

← definition given in variable name

- **Redundant comments**

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

Clean Code, Ch. 4

Helpful commenting practices

- Prefer refactoring code over explaining how it works
- Explain the intent and interface
- Amplify important points
- Explain why an approach was *not* used
- Provide context and references
- Explain concepts unfamiliar to readers
- **Update comments when updating code**

Wilson et al. (2014) & *Clean Code*, Ch. 4

Helpful commenting practices

- Write comments for the broadest probable audience
- Write what you wish you knew an hour ago
- Use an issue tracker instead of long-term “to do” comments
- Avoid referring to something by a mutable characteristic
    - Variable names that are likely to change
    - Position of an item in a numbered list that could be re-ordered

Clean Code, Ch. 4

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*

Why do we write tests?

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

Unit tests

- A **unit test:**
  - Verifies a **single unit of behavior**,
  - Does it **quickly**, and
  - Does it in **isolation from other tests**.
- Well-written unit tests
  - Increase code reliability
  - Simplify finding & fixing bugs
  - Make code easier to change

Definition from *Unit Testing Principles, Practices, and Patterns*

A minimal software test

```
def test_addition():
    """Test adding two integers."""
    assert 1 + 1 == 2, "Incorrect value for 1 + 1"
```

- Descriptive name
- Descriptive docstring (if unclear from name)
- An assertion that a condition is met
- Descriptive error message if condition is not met

Common unit test pattern: arrange, act, assert

Testing best practices

- **Write readable and maintainable tests**
    - Low quality tests cause future frustrations
- **Write tests while writing the code being tested**
    - A test delayed is usually a test not written
- **Automate tests**
    - Make sure tests can be run with ≤ 1 command
- **Run tests often!!!!**
    - Change 1 thing & run tests ⇒ easier to isolate location of bugs
    - Change 37 things & run tests ⇒ hard to find location of bugs

Testing best practices

- **Keep tests small**
  - Avoid multiple assertions per test (unless closely related)
  - Avoid conditionals & complex test logic
- **Keep tests fast**
  - If necessary, add an option to skip slow tests
- **Keep tests independent of each other**
  - Interdependent tests are harder to change
- **Make tests deterministic**
  - Hard to tell when a test that fails intermittently is fixed
  - Specify the random seed

Testing best practices

- **Avoid testing implementation details**
  - Tests of implementation details make code harder to refactor
- **Turn every bug into a new test**
  - Helps us fix a bug and prevent it from happening again
  - Bugs happen in clusters — consider adding related tests
- **Use a code coverage tool**
  - Tells us which lines are covered by a test and which are not
  - Helps us write targeted tests and find unused code
- **Consider refactoring code that is difficult to test**
  - Write short functions that do one thing with no side effects

Test-driven development

- More common practice:
    - Write a function
    - Write tests for that function
    - Fix bugs in the function
- Test-driven development
    - Write a failing test
    - Write code to make the test pass
    - Clean up code after tests are passing
- Advantages of writing tests first
    - Makes us think about what each function will do
    - Saves us time
    - Reduces frustration

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 at and near 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`

Test known solutions and properties

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

Error messages are vital documentation

- The best error messages help users pinpoint a problem and understand how to fix it
- Cryptic error messages can cause hours of frustration

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
  - Concise, but complete
  - Understandable to new users & contributors
- Provide enough information to solve the problem with minimal extraneous information

Avoid premature optimization of code

- 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 using a *profiler* to identify bottlenecks
- But plan ahead when writing numerically intensive code!

When should we write clean code?

- Some clean coding habits save time quickly
    - Writing short functions that do one thing
    - Writing tests that can be run automatically
- We don’t need particularly clean code when we’re interactively exploring a data set
- Investing extra time is worthwhile if:
    - You’ll re-use the code
    - The code will be shared with others
- Avoid perfectionism
    - Writing clean code is an iterative process

The nascent field of research software engineering

- Research software engineers (RSEs) include
  - Researchers who spend most of their time programming
  - Software engineers developing scientific software
  - Everyone in between
- Challenges
  - Unclear career paths for RSEs
  - Insufficient training for scientists to become RSEs

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
- Prefer refactoring code over explaining how it works
  - Communicate the implementation in the code itself
- Well-written tests make code *more* flexible

Final thoughts

- Think in terms of trade-offs
  - Need to balance competing priorities (i.e. brevity vs. clarity)
- Software testing is the best thing since sliced arrays! 🍞
  - Run tests often!
- Remember the importance of community
  - A software project is not just code — it’s people too
  - Psychological safety is vital

Psychological safety references: *The Fearless Organization* by A. Edmondson; and Beyond Buzzwords and Bystanders: A Framework for Systematically Developing a Diverse, Mission Ready, and Innovative Coast Guard Work- force by K. Young-McLear et al.