plasmapy

Software testing is awesome!

An introduction to `pytest`

Nick Murphy (CfA)

This work has been supported by:

NSF DOE NASA

Why do we write software tests?

- To catch and fix bugs
  - Preferably as soon as we introduce them!
- To provide confidence that our code gives correct results
- To save time and reduce frustration
- To keep track of bugs to be fixed later
- To provide examples of how code was intended to be used
- So we can change the code with confidence that we are not introducing hidden bugs elsewhere in the program

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

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*

What is pytest?

- The `pytest` framework is “a full featured testing tool that helps you write better programs”
- What `pytest` does:
  - Runs our tests
  - Improves assertion error messages
  - Provides lots of helpful testing tools & capabilities
- `pytest` is highly extensible
  - Finding `pytest` extensions is a good way to procrastinate

Installing `pytest`

`pytest` can be installed from the terminal with:

```
pip install pytest
```

From an activated conda environment, install it with:

```
conda install pytest
```

To test that it is working, run:

```
pytest --version
```

Outline

- Assertions and exceptions
- Floating point comparisons
- Writing our first test with pytest
- Command line options
- Marking and parameterizing tests
- Testing exceptions and warnings
- Testing functions in a Python file
- Next steps and testing practices

```
pip install pytest
```

Now time for the interactive tutorial…

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 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**