Testing

What Kinds of Testing Exist?

- This is a big topic:
  - https://en.wikipedia.org/wiki/Software_testing
- We most commonly hear about:
  - Unit testing
    - Tests that a single function does what it was designed to do
  - Integration testing
    - Tests whether the individual pieces work together as intended
    - Sometimes done one piece at a time (iteratively)
  - Regression testing
    - Checks whether changes have changed answers
  - Verification & Validation (from the science perspective)
    - Verification: are we solving the equations correctly?
    - Validation: are we using the right equations in the first place?

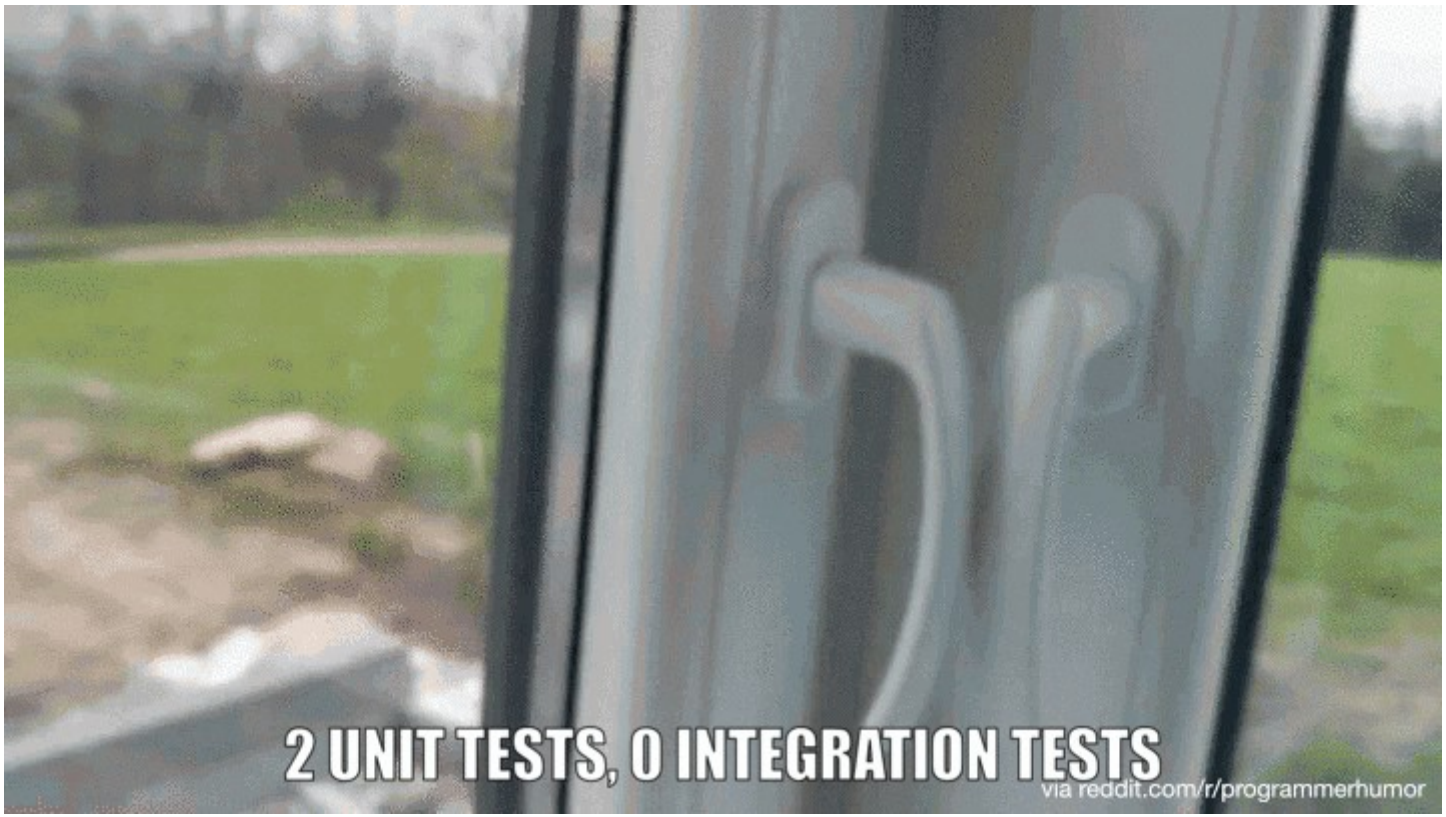
Automation

- The best testing is automated—you don't need to remember to run it
  - Developers can forget to manually run tests
- Github / bitbucket have *continuous integration* service that can be run on pull requests
  - This is commonly used on large development projects

Unit Testing

- We want to test the smallest piece of functionality alone
  - E.g., a simple test that ensures that a function does what it is designed to do
- Unit testing alone is not enough
  - Although it gives you confidence that each piece works alone as designed, it doesn't test what happens when you put it all together
  - *Integration testing* is a much harder problem

Unit vs. Integration Testing



Unit vs. Integration Testing

Unit Testing

- When to write tests?
  - Some people advocate writing a unit test for a specification before you write the functions they will test
    - This is called Test-driven development (TDD): https://en.wikipedia.org/wiki/Test-driven_development
    - This helps you understand the interface, return values, side-effects, etc. of what you intend to write
  - Often we already have code, so we can start by writing tests to cover some core functionality
    - Add new tests when you encounter a bug, precisely to ensure that this bug doesn't arise again
- Tests should be short
  - You want to be able to run them frequently

Unit Testing

- There are several frameworks for unit testing in python—we'll focus on `pytest`
  - Note: like packaging, unit testing frameworks in python appear in a state of flux
  - For a long time, `nose` was one of the most popular frameworks, but its development has ceased
    - `nose2` is a successor to nose that may become standard soon
  - `unittest` is built into python, and may interact with nose2 in the future
  - `pytest` (sometimes `py.test`) seems to be the most popular now—we'll explore that
- Basic elements:
  - Discoverability: it will find the tests
  - Automation
  - Fixtures (setup and teardown)

pytest Unit Testing

- Install (single-user) via: `pip3 install -U pytest –user`
  - `pytest` should now be in your path
    - Note: older name was `py.test`
  - For user-install, it may be that `~/.local/bin/` comes later in your PATH than a system-wide directory with an older `pytest`
    - You may need to explicitly alias it
- For coverage reports: `pip3 install -U pytest-cov --user`

pytest Unit Testing

- Test discovery makes unit testing easy—adhere to these conventions and your tests will be found:
  - File names should start or end with "`test`":
    - `test_example.py`
    - `example_test.py`
  - For tests in a class, the class name should begin with "`Test`"
    - e.g., `TestExample`
    - There should be no `__init__()`
  - Test method / function names should start with "`test_`"
    - e.g., `test_example()`

https://pytest.readthedocs.io/en/reorganize-docs/new-docs/user/naming_conventions.html

pytest Unit Testing

- Tests use assertions (via python's `assert` statement) to check behavior at runtime
  - https://docs.python.org/3/reference/simple_stmts.html#assert
  - Basic usage: `assert expression`
    - Raises `AssertionError` if `expression` is not true

Simple pytest Example

- Here's a simple example (`examples/testing/pytest/simple/` in our git repo):

```
def multiply(a, b):
    return a*b

def test_multiply():
    assert multiply(4, 6) == 24

def test_multiply2():
    assert multiply(5, 6) == 24
```

  - There are 2 tests here
    - First will pass, second will fail
  - Run the tests as: `pytest -v .`

pytest Fixtures

- Unit tests sometimes require some setup to be done before the test
  - Fixtures provide this capability
  - We'll look at setup and teardown functions/methods for tests
    - These are sometimes referred to the standard xUnit fixtures
    - pytest supports a more flexible system for fixtures in addition to these, but we won't look at it here
    - http://pytest.org/dev/fixture.html
- By default, `pytest` will capture stdout, and only show it on failures
  - See, e.g., https://docs.pytest.org/en/latest/capture.html
  - This can be changed with the `-s` flag
- Example of function-level setup/teardown in `examples/testing/pytest/function_setup/`

pytest Unit Testing

- Class example
  - Note, a class used for testing is not a full-fledged class—it simply helps to organize data used for a bunch of tests with common needs
  - In particular, it does not have a constructor (`__init__()`)
    - See, e.g., https://stackoverflow.com/questions/21430900/py-test-skips-test-class-if-constructor-is-defined
- We'll look at an example with a NumPy array
  - `examples/testing/pytest/class/` in our class git
  - We always want the array to exist for our tests, so we'll use fixtures (in particular `setup_method()`) to create the array
  - Using a class means that we can access the array created in setup from our class
- NumPy has its own assertion functions
  - https://docs.scipy.org/doc/numpy/reference/routines.testing.html
  - Note in particular, the approximately equal tests

pytest Coverage

- We can determine the coverage of our testing
  - Note just because a line / function is covered in our suite, doesn't mean we'll capture every error—there can always be corner-cases, things you don't anticipate
  - Adding unit tests is an ongoing process
- Basic running:
  - `pytest --cov .`
  - Here, '`.`' is the path we are testing
- Detailed report (lines missed):
  - `pytest --cov-report term-missing --cov .`

pytest Coverage

- Let’s look at a larger example:
  - pyro is my tutorial hydrodynamics code: https://github.com/zingale/pyro2
- pyro has both unit tests (via pytest) and regression testing (more on that later)

Regression Testing

- Basic idea:
  - Pick some problems / workflows that exercise your codebase
    - Might need many tests to get good coverage / explore all options
  - Store a benchmark containing the “right” answer
  - Each time you change your code, run the regression tests
  - Compare the new answer to the stored benchmark
- If a regression test fails, then either:
  - You’ve introduced a bug—look at what changed and fix it
  - You fixed a bug—update the benchmarks

Regression Testing

- Automating the testing
  - You need a tool to do the comparisons
  - Store benchmarks in a separate directory (so they are not overwritten when you run)
  - Run your code, at the end of the run, compare the new output to the stored benchmark and report
- Our example, pyro, has regression testing built in with the `--compare_benchmark` option
- Here's another example from my research codes: http://bender.astro.sunysb.edu/Castro/test-suite/test-suite-gfortran/