Building Reliable Code:
Notes on Unit Testing

Andy Boughton

Abecasis Group Meeting

July 30, 2019

Motivation

- Many research findings depend on software
  - ...but most software has bugs
  - ...and needs maintenance
- A project may pass through multiple authors over time
  - ...with no clear specification
  - ...data formats may not be well documented
- Computational science depends on reproducibility and reliability

Is your method reusable?

(tl;dr: Probably not)

"We selected 41 open access papers... and executed the R code...

- *46 out of 97 reproduced figures deviated from the original figures on a deeper level, e.g. graphs had different curves, and **key numbers were missing or different**.*
- *The code of two papers ran without any issues, 33 had resolvable issues, and two were partially executable, i.e. the code produced output but also had issues that we could not resolve. We classified four papers as being irreproducible, as we could not solve all issues.... **In total, we encountered 173 issues in 39 papers** (mean μ=4.4 issues per paper). ..."*

Konkol, M., Kray, C. & Pfeiffer, M. Computational reproducibility in geoscientific papers: Insights from a series of studies with geoscientists and a reproduction study. *International Journal of Geographical Information Science* **33**, 408-429 (2019).

Case study: RAREMETAL

- 2017 bug: burden tests used incorrect parameters and gave wrong results. Released for >1 yr before problem was found
- 2018a: A one-letter typo caused all calculations to load the wrong data and give wrong results (*caught by system tests before release)
- 2018b: Did not actually support RVTESTS file format, and did not fail gracefully. (*tests added*)

Verify as you go

How to build things with unit testing and automation in mind

Unit Testing

*"**UNIT TESTING** is a level of software **testing** where individual units/ components of a software are tested... A **unit** is the smallest testable part of any software. It usually has one or a few inputs and usually a single output."*

This should be an automated process.

http://softwaretestingfundamentals.com/unit-testing/

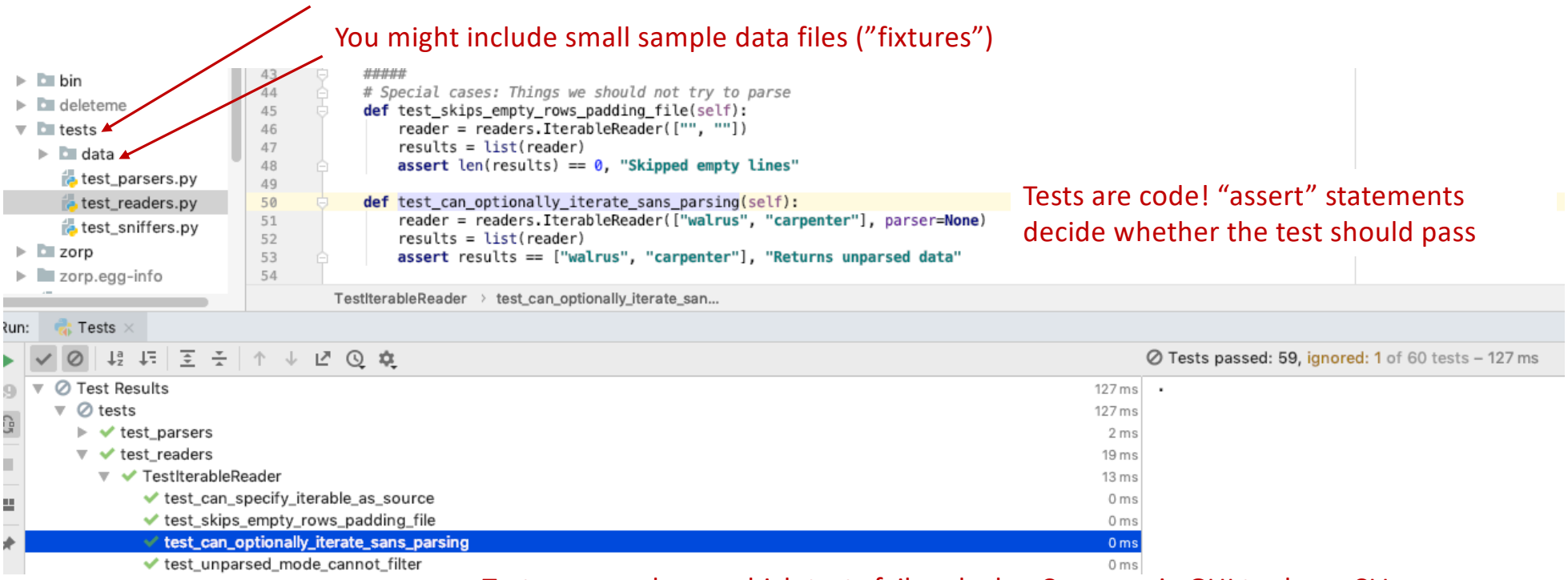
Benefits of unit testing

- Each segment becomes easier to understand
  - Smaller pieces = better defined behavior for each piece
- Able to add new features without breaking old ones
- They become a "source of truth"
  - Makes it obvious what the requirements are
  - Provide sample input/output/usage showing how to function
- Easier to keep up to date
  - Documentation is just text
  - ...but unit tests can be run as code

A sample unit test

Many tools automatically find folder/file/method(s) with the word `Test` in their names

You might include small sample data files (”fixtures”)



Tests are code! “assert” statements decide whether the test should pass

Test runner: shows which tests fail and why. Can run via GUI tools, or CLI:
$ pytest ./tests/

Is the “unit” reachable?

BAD

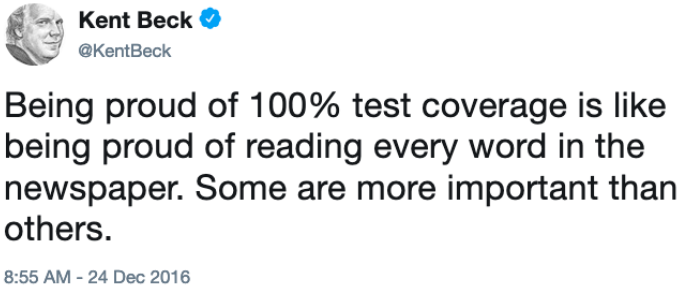
```
calc.exe --do-all giant_file.inp

conf = { “value”: 1 }
def depends_on_global(val):
    conf[‘value’] += val
    return conf[‘value’]

def test_definition_of_insanity():
    assert depends_on_global(1) == 2
    assert depends_on_global(1) == 3
```

GOOD

```
def does_one_thing(val):
    return val + 1

def test_positive_addition():
    assert does_one_thing(1) == 2

def test_negative_addition():
    assert does_one_thing(-1) == 0
```

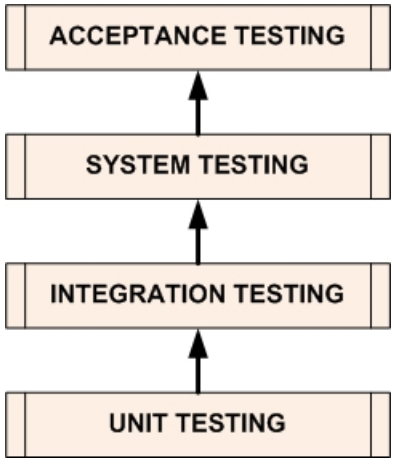
What to test for research software

- A good test gives timely, actionable feedback
  - Quick to run
  - Helps you find the bug easily
- Bugs in data loading
  - Bugs can occur in data loading, filtering, or calculation
  - No one will manually test all the file formats you claim to support
- Calculations
  - Does it produce the expected result?
  - Does it handle edge cases?



Levels of testing

- There is more than one way to test a system
- Each offers different benefits
- Some (acceptance testing) may involve other people
- Choose the pieces that are right for you



http://softwaretestingfundamentals.com/software-testing-levels/

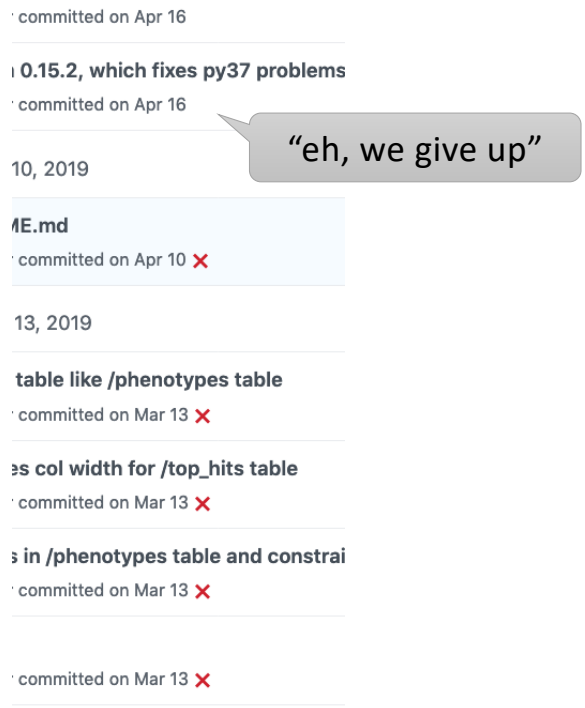
What kind of tests should you write?

- Unit tests are good for localizing the problem to one piece
- Sometimes, the problem occurs at the boundary between two pieces
- Think about how your software is designed

Image credit: https://devrant.com/rants/519301/2-unit-tests-0-integration-tests-source-https-twitter-com-thepracticaldev-status

Advice on making it work

- Write tests as you go, not at the end
- Don’t ignore failures: error messages should not feel “normal”
- Run tests in a way that reflects the real system
  - Including non-code artifacts: current sample data, dependencies, DB schema, etc



When not to test

- Trivial output from third party code (numpy already has tests)
- Exploratory or one-off code (validate, but maybe don't automate)
- When **effort >> value** (eg it's hard to auto-test pixel layout in a UI)
- Team culture: Shared practices require shared values

Samples of <code> testing at CSG

Designed at the start

- Raremetal.js (JS, numerical): https://github.com/statgen/raremetal.js/tree/master/test
- Credible Sets (JS, numerical): https://github.com/statgen/gwas-credible-sets/tree/develop/test
- LD Server (C++, web server): https://github.com/statgen/LDServer/blob/master/core/tests/LDServerTest.cpp
- GWAS Parser (Python, Data handling): https://github.com/abought/zorp/tree/develop/tests
- LocusZoom (JS, browser): https://github.com/statgen/locuszoom/tree/develop/test

Added later

- RAREMETAL (C++): https://github.com/statgen/raremetal/tree/master/tests

Appendices

CSG: Examples by type

- Numerical edge and corner cases:
  - https://github.com/statgen/gwas-credible-sets/blob/develop/test/unit/stats.js#L16
- Integration tests for complex methods: https://github.com/statgen/raremetal.js/blob/master/test/unit/stats.js#L94
- Validation of complicated rules
  - https://github.com/abought/zorp/blob/develop/tests/test_parsers.py#L47
- ”Outside in” tests of existing program:
  - https://github.com/statgen/raremetal/blob/master/tests/raremetal/test_tut_rm/script.cmake
- Mocks + stubs + browser features:
  - https://github.com/statgen/locuszoom/blob/c3007e1cb4b2fce4e888241396562512a8aeb90c/test/unit/ext_dynamic-urls.js#L75

Mock, patch, stub, and spy

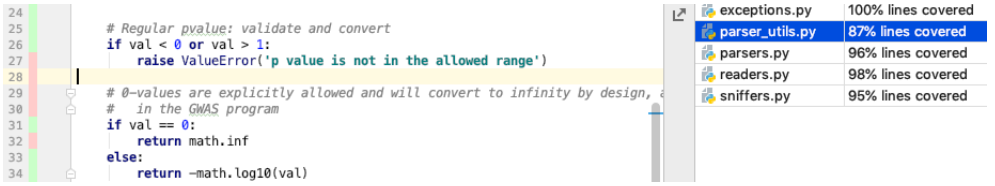
```
@patch('config.threshold', 42)
def test():
    assert config.threshold == 42
```

```
it('changes the URL with new information from plot state with specified
mapping', function() {
  this.historySpy = this.sandbox.stub(history, 'replaceState');

  var stateUrlMapping = { chr: 'chrom' };
  this.extension.plotUpdatesUrl(this.plot, stateUrlMapping);

  this.plot.applyState({ chr : 7 });

  assert.ok(this.historySpy.called, 'replaceState was called');
  assert.ok(this.historySpy.calledWith({}, '', '?chrom=7'));
});
```

- Sometimes your code will need to depend on something external
  - Eg configuration value, remote web service, caching layer, etc
- Mocks lets you “pretend” your test is isolated, and even disable dependencies on external programs
- Spies let you monitor nested code to see when and how it is called
- Caveats:
  - “Every mock is a code smell”
  - Terminology varies between languages

Test coverage

- A way to determine which lines/files need a unit test
  - Can visualize the results
- Very useful for branching logic (like data parser that needs to check for edge cases)
- Caveat: does not tell you if the test for that line is a *good* test.
  - Eg, *do_everything()* could hit 75% of your code, but you would still not have any unit tests
  - You can run the tool only on selected files (eg unit but not system tests)



The role of acceptance testing

"...would it then be possible to reduce the overall disagreement by feeding back the presence of obvious errors to the package designers...? The answer turns out emphatically to be yes."

Listen to your users! Give them an obvious way to report bugs.

Hatton, L. & Roberts, A. How accurate is scientific software. *IEEE Transactions on Software Engineering* **20**, 785-797 (1994).

Filing a good bug report

“Components of a complete bug report:

- Environment information
- Minimal and precise steps to reproduce
- Sample data”

“When you find a bug, it is unlikely that you are the only individual affected by it. By reporting a bug... you are performing a... service to the entire... community.”

https://imagej.net/Bug_reporting_best_practices

Bug reports can become tests!

- A good bug fix tries to prevent the problem from ever happening again
- The same test case used to reproduce the bug can become the basis for a new automated test
- Because bugs represent knowledge, make the tracker a shared resource
  - ”oh yeah, I got an email once; remind me to send you a copy” does not scale

Further reading: https://en.wikipedia.org/wiki/Regression_testing
Sample regression test: https://github.com/statgen/raremetal/commit/e734dc1a3ff67379b60fc8cb38ad0310c7afcf01

Tools for unit testing

Many languages have "xUnit" style tools with similar syntax (makes it easier to bring old skills to a new language).

- C++: Google Test Framework: https://github.com/google/googletest
- Python:
  - Unittest (builtin): https://docs.python.org/3/library/unittest.html
  - Pytest (fancy): http://doc.pytest.org/en/latest/getting-started.html
- PHP: PHPUnit
- R: Runit, testthat

https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Further reading: Unit Tests

Lesson Materials

- Python Testing (Lesson Materials): https://katyhuff.github.io/python-testing/
- Best testing practices for data science (PyCon 2017)- "what needs to be tested", 17:00: https://www.youtube.com/watch?v=yACtdj1_IxE
- Testing and SW Eng slides: https://www.southampton.ac.uk/~fangohr/training/Software-Engineering-for-Computational-Science-and-Engineering-Hans-Fangohr.pdf
- Best practices for scientific computing: https://katyhuff.github.io/2018-02-15-pydata/#/10/34

Special topics

- Demystifying the patch function (PyCon 2018): https://www.youtube.com/watch?v=ww1UsGZV8fQ
- Mocking and Patching pitfalls (PyCon 2019): https://www.youtube.com/watch?v=Ldlz4V-UCFw
- NSLS Scientific Python Cookiecutter: https://github.com/NSLS-II/scientific-python-cookiecutter
- Tips for Jupyter notebook + doctest: https://stackoverflow.com/questions/40172281/unit-tests-for-functions-in-a-jupyter-notebook
- Head First Design Patterns (on my bookshelf)
- Hexagonal architecture: https://blog.octo.com/en/hexagonal-architecture-three-principles-and-an-implementation-example/

Further reading: Reproducibility

Tips and best practices

- “Best Practices for Scientific Computing“ https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745
- “Ten Simple Rules for the Care and Feeding of Scientific Data” https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003542

Community

- PLOS initiative for computational reproducibility: https://blogs.plos.org/plos/2019/07/plos-computational-biology-announce-reproducibility-pilot/
- Journal of Open Source Software: https://joss.theoj.org/
- Binder: “Turn a Git repo into interactive notebooks” https://mybinder.org/