A Scientist-Friendly Approach to Software Testing

Jeff Soules and Brian Ward
Center for Computational Mathematics, Flatiron Institute

Introduction

What are we talking about today?

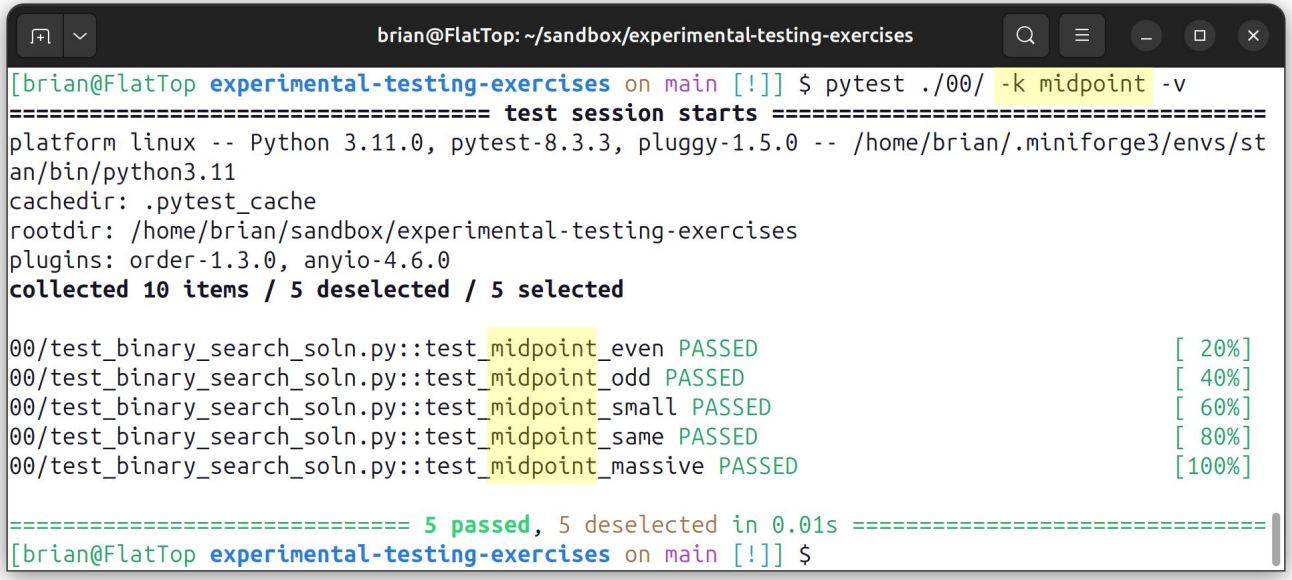
- Automated software tests
  - Regularly re-run (by machine)
  - Evaluated objectively (by machine)
  - Expected to work on all user deployments

    → ANYONE can run & interpret
- Reproducible, deterministic results
  - We do not want a reproducibility crisis in scientific software

STATUS	RUN	COMMIT	BRANCH
✓	441	–	master
✓	1	–	PR-1555
✓	440	cc00c23	master
✓	3	–	PR-1542
✗	2	1e43ad3	PR-1542
✓	439	4156b8e	master
✓	1	–	PR-1553

Automated testing frameworks

- Discovers and runs test functions
- Can help ensure **test isolation**
- Let you **select subsets** of tests to run
- Reporting features (coverage, xml, …)



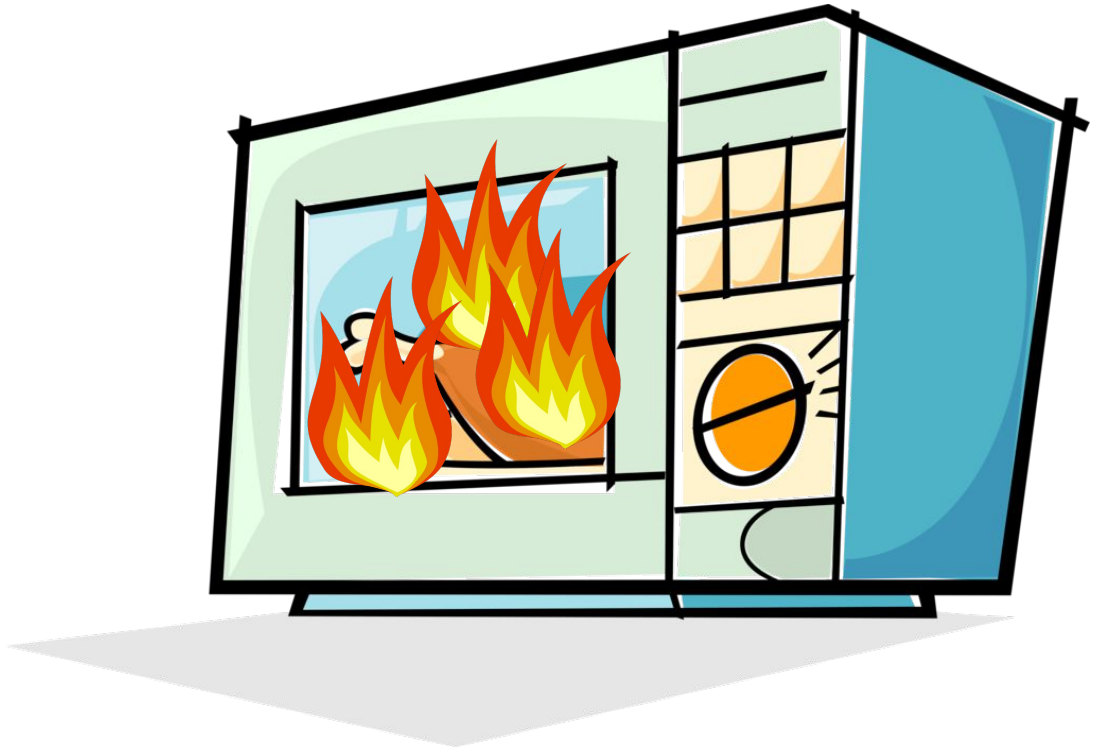
FLATIRON INSTITUTE

Scientific testing means treating tests as . . .

- **controlled experiments** that
- each **confirm one specific property** or **invariant** of
- the **code artifact** you are publishing

Test result must depend on **your code – a small part of your code – and nothing but your code**

Test your oven with a thermometer, not a turkey.

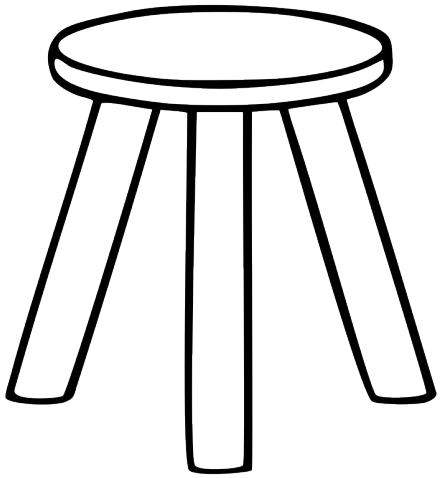


Scientific testing can

- Show **correctness** of our implementation
- **Localize** sources of error
- Increase confidence during **future code changes**
- **Document** code expectations/design choices

Three pillars of test suites

- Fast
- Trustworthy
- Specific



Honorable mention: maintainable
Implicit requirement: independent

What you need to follow along today

- https://github.com/flatironinstitute/sciware/tree/main/42_ScientificTesting
- If you're using Python, we recommend installing pytest
- If you want to use C++, we will provide links to the Compiler Explorer

Anatomy of a test

FLATIRON INSTITUTE

What makes a test?

To do a test, you must:

- Set up the preconditions for the behavior you're testing
- Call the code whose behavior you're testing
- Compare the actual result with the expected result

Mnemonic is "arrange, act, assert"

…but the greatest of these is **assert**

Assertion libraries

```
1  #include <gtest/gtest.h>
2
3  TEST(fp, equality) {
4      EXPECT_EQ(0.1 + 0.2, 0.3);
5  }
6
7  TEST(fp, better_equality){
8      EXPECT_FLOAT_EQ(0.1 + 0.2, 0.3);
9  }
```

- You've only tested what you've **asserted**
- Good tools help you **zero in on discrepancies**
  - Expressive comparisons, error handlers, etc. let you make more precise assertions
- **Hypothesis testing tools** can help turn invariants into assertions automatically
- Gold/expect/`cram` testing can help ensure reliable output (but may be brittle)

```
[----------] 2 tests from fp
[ RUN      ] fp.equality
/app/example.cpp:4: Failure
Expected equality of these values:
  0.1 + 0.2
    Which is: 0.30000000000000004
  0.3

[  FAILED  ] fp.equality (0 ms)
[ RUN      ] fp.better_equality
[       OK ] fp.better_equality (0 ms)
```

FLATIRON INSTITUTE

Let’s look at some tests

Follow along at
https://github.com/flatironinstitute/sciware/tree/main/42_ScientificTesting

Folder ‘`binary_search`’

Goals for tests

One test, one property

- Every test should test a specific, explicitly stated property of the code
- It should pass if, and only if, that property holds

Yes, this will lead to lots of tests!

This is not a problem so long as you keep them all fast.

FLATIRON INSTITUTE

Controlled experiments

- Success or failure should depend **only** on the property under test
- Any **other** factors should be set to known values
- Systems not under test should be **omitted**
  - by use of fakes/stubs/mocks, monkey-patching, etc.

FLATIRON INSTITUTE

Only test your code

- A useful test must **call the implementation**
  - It's common to write a slow/naive version of an algorithm for testing, but make sure you're comparing that to the *real* one at some point!
- There's little point to testing library code

FLATIRON INSTITUTE

Three varieties of bad tests

Those that…

- Never fail (tautological, or fail to assert anything)
- Never pass (or: fail if you breathe on them)
- Might fail or pass on the same inputs

Also fishy: tests that take a huge amount of work to achieve control

FLATIRON INSTITUTE

Assumptions and invariants

What to assert?

Universal invariants: confirming expectations

- Results of a (deterministic) computation
- Correct logical branch was taken
- Error states are handled appropriately
- Does your code ‘do nothing’ correctly?

By making assumptions **explicit,** tests act as a kind of documentation

FLATIRON INSTITUTE

What to assert?

System invariants: interactions with external systems

- Did the file actually get written?
- Database call was correct?
- Handled unexpected response from system?

Requires **experimental control** (e.g. through fakes) and **test isolation**

What to assert?

Scientific invariants: domain-specific results

- Is energy conserved?
- Are results properly normalized?
- Do lists have the right length/shape, loops have right iterations, etc

Consider using **unrealistic inputs**

Test implementation, not methods

Implementation correctness is orthogonal to scientific correctness.

- Right result ≠ correct code
  - evaluated by summary statistic?
  - error tolerance too loose?
- Wrong result ≠ incorrect code
  - Newtonian simulator, relativistic problem

This is exactly why you need controls!

More code examples

https://github.com/flatironinstitute/sciware/tree/main/42_ScientificTesting
Folder ‘`file_writer`’

Gaining control

What is a control?

Removing influence of variables not under investigation

- In software tests, avoiding calls to code not under test
- Interpreted languages (Python): *mocking libraries*
- Compiled languages (C++): need to use *dependency injection*

Replace calls to outside libraries/expensive functions with observable substitutes

Why use controls?

- External resources (e.g. data downloads)
  - Unreliable (in uninteresting ways)
  - Costly to use for repeated testing
- Other parts of your code (e.g. file writing, computations)
  - Introduce stochasticity
  - Require cleanup
  - Slow
- Generating errors/rare states reliably

Consider: should it actually be user-facing?

Mocking in Python

`unittest.mock`

- `Mock` object
  - Replaces any functions or objects with minimal config
  - Can set function return values, side effects, errors thrown etc.
  - Tracks any calls made & what values were passed
- `patch` function
  - Replaces arbitrary code with `Mock` object
  - works as function decorator or context manager
  - syntax is idiosyncratic

Fakes in C++

Compiled: can’t replace code arbitrarily

- Dependency injection
  - Replace hard-coded calls with function parameters or templates
  - Use a controlled implementation for testing
  - Exposes flexibility for actual users of code too
- This can get messy if taken too far, it’s common to provide an overload with defaults for the user

These techniques can also be applied in interpreted languages!

FLATIRON INSTITUTE

Experimenting on a *world you shape*

FLATIRON INSTITUTE

The envy of every experimentalist

- You create the very world you experiment upon!
- Reshape interfaces to something more convenient
- Or keep the interface but update the implementation
- Use testability as a guide to writing better code

FLATIRON INSTITUTE

Tests: beyond pass and fail

Code that's difficult to test is often also difficult to…

- Understand and reason about
- Explain and document
- Maintain & extend
- Actually use

Listen to your pain.

FLATIRON INSTITUTE

Signs that things need to change

- Lots of setup/fakes/mocks needed to achieve experimental control
- Cleanup required after calling ordinary functions
- Tests need a lot of logic to match the right case
- Difficult to state the expected result of an operation
- Things work with fakes but not with real values

Responding to common issues

- *Control is hard* → the units are too big/too ambitious
- *Frequent cleanup* → isolate code with external effects
- *Test contains logic* → separate control-flow and operational code
- *Hard to describe results* → code unit is too ambitious
- *Test passes with fakes only* → Poor interface documentation; ought to have many more tests

Final code examples

https://github.com/flatironinstitute/sciware/tree/main/42_ScientificTesting
Folders ‘`histogram`’ and ‘`obfuscated`’

Conclusion

Tests are experiments

They should be:

- Controlled
- Focused
- Repeatable

FLATIRON INSTITUTE

You have an advantage

- You usually control the system you're experimenting on
- Testability is highly correlated with other measures of code quality

Thank you

FLATIRON INSTITUTE

FLATIRON INSTITUTE

SURVEY

Appendix: Helpful tools

Tools to help with testing

Coverage reporting

- Visual tool to see what definitely hasn't been tested
- Cannot say what you **meaningfully asserted**, only what you called
- **Not** a progress bar (despite appearances)
- Particularly useful for checking conditional branches

```
102
103 const loadSamplingOptsFromString = (
104   data: ProjectDataModel,
105   json: string,
106   clearExisting: boolean = false,
107 ): ProjectDataModel => {
108   const newSampling = parseSamplingOpts(json);
109   const newSamplingOptsMember = clearExisting
110     ? { ...newSampling }
111     : { ...data.samplingOpts, ...newSampling };
112   return { ...data, samplingOpts: newSamplingOptsMember };
113 };
```