Week 3, Lecture 6 - Reproducible Computational Workflows

Aaron Meyer

Outline

- Administrative Issues
- Testing
  - Unit testing
  - Integration testing
- Example

Bugs and Testing

- **software reliability**: Probability that a software system will not cause failure under specified conditions.
  - Measured by uptime, MTTF (mean time till failure), crash data.
- Bugs are inevitable in any complex software system.
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.
- testing: A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation).

Cost-benefit tradeoff

There is no one answer to testing

- You're putting together a quick script to calculate the amount of a chemical you need
  - You probably don't need to test this
- You're assembling a machine learning model to identify heart attacks
  - Not testing this could be considered professional malpractice

Difficulties of testing

- Testing is seen as a novice's job.
  - Assigned to the least experienced team members.
  - Done as an afterthought (if at all).
  - "My code is good; it won't have bugs. I don't need to test it."
  - "I'll just find the bugs by running the program."
- Limitations of what testing can show you:
  - It is impossible to completely test a system.
  - Testing does not always directly reveal the actual bugs in the code.
  - Testing does not prove the absence of errors in software.

Unit Testing

unit testing: Looking for errors in a subsystem in isolation. - Generally a "subsystem" means a particular class or object.

The basic idea:

- For a given class Foo, create another class FooTest to test it, containing various "test case" methods to run.
- Each method looks for particular results and passes / fails.
- Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

Python's unittest

```
import unittest

class TestStoneMethods(unittest.TestCase):
    def test_nchoosek(self):
        self.assertTrue(nchoosek(5)[3] == 10)
        self.assertTrue(nchoosek(6)[3] == 20)
        self.assertTrue(nchoosek(7)[3] == 35)
        self.assertTrue(nchoosek(8)[3] == 56)
        self.assertTrue(nchoosek(9)[3] == 84)

if __name__ == '__main__':
    unittest.main()
```

Assertions

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Assertions

Method	Checks that	New in
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*	3.2
`assertLogs(logger, level)`	The `with` block logs on *logger* with minimum *level*	3.4

Setup and Teardown

```
import unittest
from ..StoneModel import StoneModel

class TestStoneMethods(unittest.TestCase):
    def setUp(self):
        self.M = StoneModel()

    def test_dataImport_mfiAdjMean(self):
        self.assertTrue(self.M.mfiAdjMean.shape == (24, 8))
```

Tips for Testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe add usually works, but fails after you call remove
  - make multiple calls; maybe size fails the second time only

Trustworthy tests

- Test one thing at a time per test method
  - 10 small tests are much better than 1 test 10x as large
- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails stops the test
  - You won't know whether a later assertion would have failed
- Tests should avoid logic
  - Minimize if/else, loops, switch, etc.
  - Avoid try/catch
  - If it's supposed to throw, use assert for that

Torture tests are okay, but only *in addition to* simple tests.

Regression testing

- **regression**: When a feature that used to work, no longer works.
  - Likely to happen when code changes and grows over time.
  - A new feature/fix can cause a new bug or reintroduce an old bug.
- **regression testing**: Re-executing prior unit tests after a change.
  - Often done by scripts during automated testing.
  - Used to ensure that old fixed bugs are still fixed.

Many products have a set of mandatory check-in tests that must pass before code can be added to a source code repository.

Test-driven development

- Unit tests can be written after, during, or even before coding.
  - **test-driven development**: Write tests, then write code to pass them.
- Imagine that we'd like to add a method `subtractWeeks` to a `Date` class, that shifts this Date backward in time by the given number of weeks.
- Write code to test this method before it has been written.
- Then once we do implement the method, we'll know if it works.

Things to avoid in unit tests

- Tests should be self-contained and not care about each other.

Avoid:

- Constrained test order
  - Test A must run before Test B.
  - Usually a misguided attempt to test order/flow.
- Tests call each other
  - Test A calls Test B's method
  - Calling a shared helper is OK, though
- Mutable shared state
  - Tests A/B both use a shared object.
  - If A breaks it, what happens to B?

Testing summary

- Tests need failure atomicity (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
  - Each test should have roughly just a couple assertions at its end.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always assertTrue.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, setup functions to reduce redundancy between tests.

Reproducible builds

- One element that unit testing highlights is the contribution of randomness
  - For example, if we start optimization at a random point, something might just break 1% of the time
  - This makes it challenging to identify when something breaks
  - Often, to avoid this, random number generators are set to a constant seed
  - But, keep in mind this then only tests one outcome of this value

Reproducible builds

- Other source of variation: the code surrounding your analysis
  - E.g. `seaborn` package changes the way it orders colors on a graph
  - A solution to this is `virtualenv`
    - This is a package that creates a virtual environment to run your python code
    - Can avoid anything someone else might have installed / messed with
  - Often projects will have a `requirements.txt` file
    - Stores the names *and versions* of all the packages used
    - Then can install specifically these before doing anything

Containerization

- What about Linux vs. Windows vs. MacOS differences? Time differences? "Gremlins"?
- Way to get around this variability is to make sure **everything** else in the environment stays constant
  - One way to do this is with a container
  - E.g. docker, which downloads and runs a virtual machine
- Essentially you start up a "virtual computer", then install everything you need

Integration testing

- **integration**: Combining 2 or more software units
  - often a subset of the overall project (!= system testing)
- Why do software engineers care about integration?
  - new problems will inevitably surface
    - many systems now together that have never been before
  - if done poorly, all problems present themselves at once
    - hard to diagnose, debug, fix
  - cascade of interdependencies
    - cannot find and solve problems one-at-a-time

Two Approaches To Integration

- **phased ("big-bang") integration**:
  - design, code, test, debug each class/unit/subsystem separately
  - combine them all
  - hope it all comes together
- **incremental integration**:
  - develop a functional "skeleton" system
  - design, code, test, debug a small new piece
  - integrate this piece with the skeleton
  - test/debug it before adding any other pieces

Benefits Of Incremental

- Benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
- Drawbacks:
  - May need to create "stub" versions of some features that have not yet been integrated
  - Not always possible

When To Build

- daily build: Compile working executable on a daily basis
  - allows you to test the quality of your integration so far
  - helps morale; product "works every day"; visible progress
  - best done *automated* or through an easy script
  - quickly catches/exposes any bug that breaks the build
- smoke test: A quick set of tests run on the daily build.
  - NOT exhaustive; just sees whether code "smokes" (breaks)
  - used (along with compilation) to make sure daily build runs
- continuous integration:
  - Adding new units immediately as they are written.

Example - Genomics Annotations

Reproducibility of computational workflows is automated using continuous analysis

Brett K Beaulieu-Jones[1] & Casey S Greene[2]

Replication, validation and extension of experiments are crucial for scientific progress. Computational experiments are scriptable and should be easy to reproduce. However, computational analyses are designed and run in a specific computing environment, which may be difficult or impossible to match using written instructions. We report the development of continuous analysis, a workflow that enables reproducible computational analyses. Continuous analysis combines Docker, a container technology akin to virtual machines, with continuous integration, a software development technique, to automatically rerun a computational analysis whenever updates or improvements are made to source code or data. This enables researchers to reproduce results without contacting the study authors. Continuous analysis allows reviewers, editors or readers to verify reproducibility without manually downloading and rerunning code and can provide an audit trail for analyses of data that cannot be shared.

Leading scientific journals have highlighted a need for improved reproducibility to increase confidence in results and reduce the number of retractions[1–5]. In a recent survey, 90% of researchers acknowledged that there 'is a reproducibility crisis'[6]. Computational reproducibility is the ability to exactly reproduce results given the same data, as opposed to replication, which requires an independent experiment. Computational protocols used for research should be readily reproducible, because all of the steps are scripted into a only with discrepancies. Additionally, Hothorn and Leisch found that more than 80% of manuscripts did not report software versions[10].

It has been proposed that open science could aid reproducibility[3,11]. In open science, the data and source code are shared. Intermediate results and project planning are sometimes also shared (as, for example, with Thinklab, https://thinklab.com/). Sharing data and source code is necessary, but not sufficient, to make research reproducible. Even when code and data are shared, variability in computing environments, operating systems and the software versions used during the original analysis make it difficult to reproduce results. It is common to use one or more software libraries during a project. Using these libraries creates a dependency on a particular version of the library; research code often works only with old versions of these libraries[12]. Developers of newer versions may rename functions, resulting in broken code, or change the way a function works to yield a slightly different result without returning an error. For example, Python 2 would perform integer division by default, so 5/2 would return 2. Python 3 performs floating-point division by default, so the same 5/2 command now returns 2.5. In addition, old or broken dependencies can mean that it is not possible for readers or reviewers to recreate the computational environment used by the authors of a study. In this case it becomes impossible to validate or extend results.

We first illustrate, using a practical example, the problem of reproducibility of computational studies. Then we describe the development and validation of a method named continuous analysis that can address this problem.

Example - Genomics Annotations

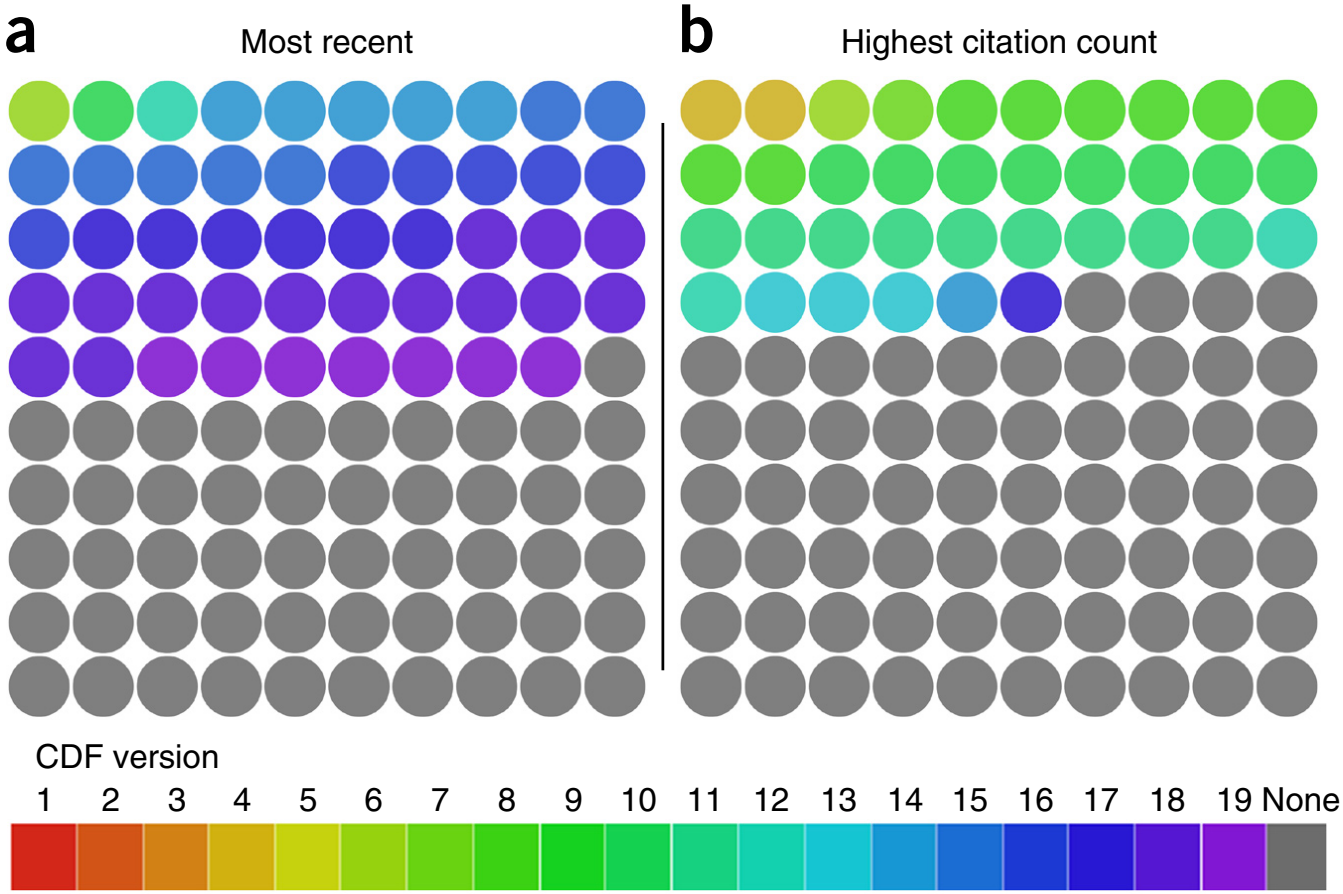

Figure 1 Reporting of Custom CDF file descriptors in published papers. (**a**,**b**) CDF version reporting in the 100 most recent (**a**) and the 100 most cited (**b**) papers citing Dai *et al.*[13] that use Custom CDF. Each circle represents one manuscript; color coding indicates the Custom CDF version used.

Example - Genomics Annotations

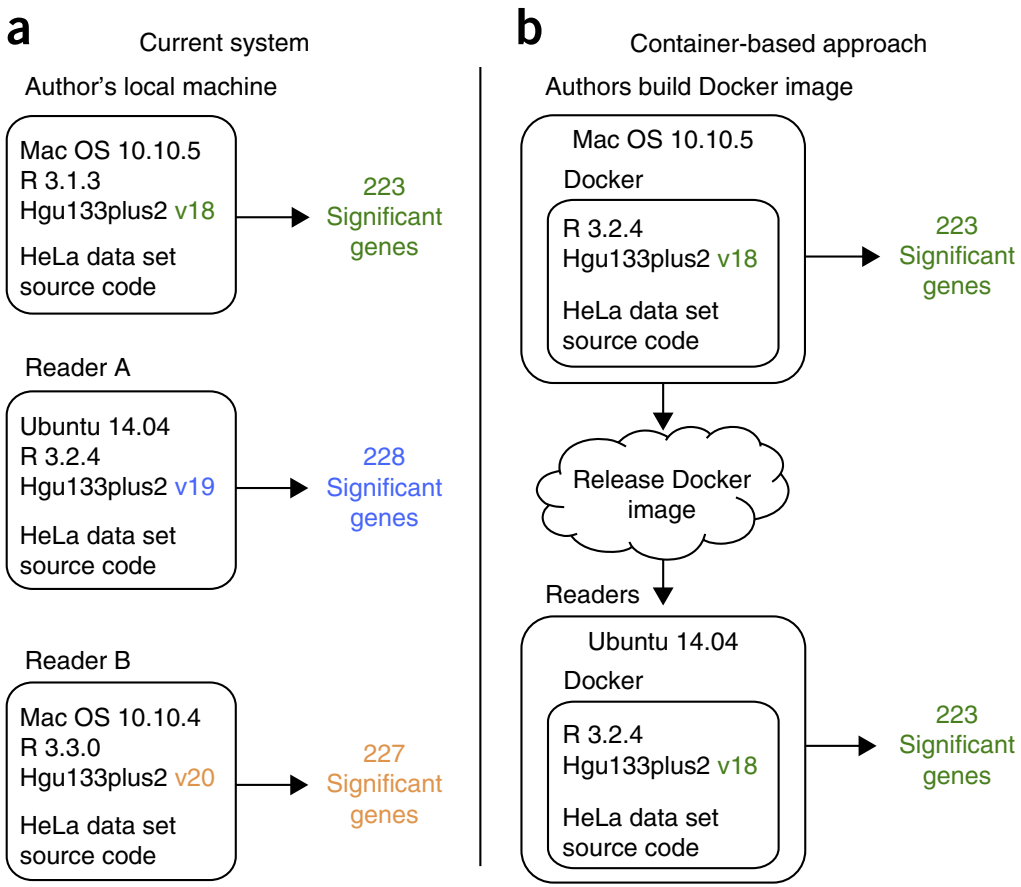

Figure 2 Research computing versus container-based approaches for differential gene expression analysis of HeLa cells. (**a**,**b**) Numbers of significantly differentially expressed genes identified using different versions of software packages (**a**) and a container-based approach with a defined computing environment (**b**). $n = 3$ biological replicates per group (wild-type or double-knockdown HeLa cells).

Example - Genomics Annotations

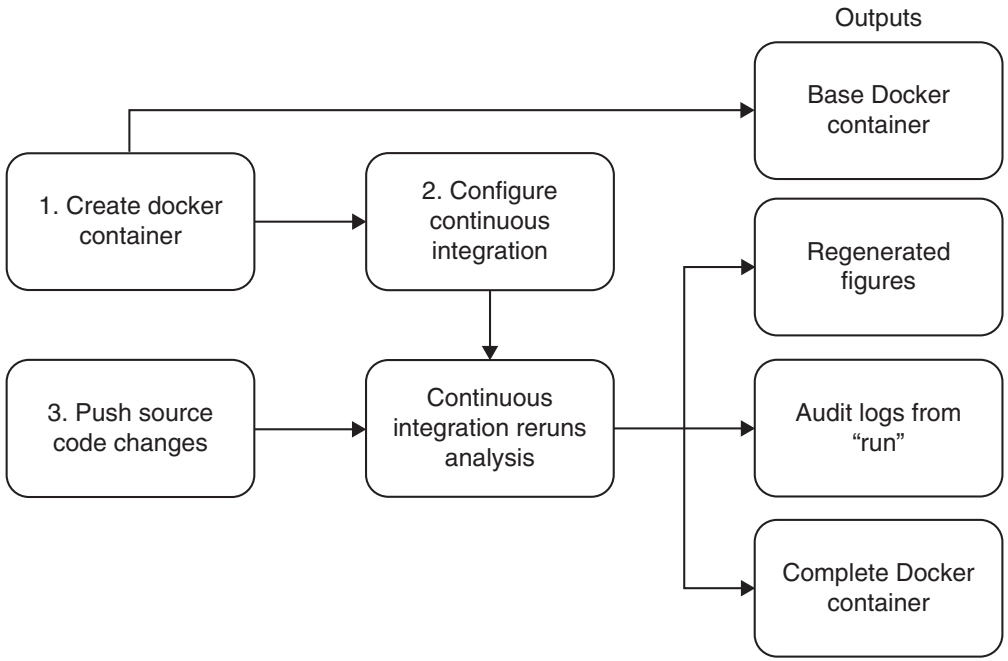

Figure 3 Setting up continuous analysis. Continuous analysis can be set up in three steps. First, the researcher creates a Docker container with the required software (1). The researcher then configures a continuous integration service to use this Docker image (2) then pushes code that includes a script capable of running the analyses from start to finish (3). The continuous integration provider runs the latest version of code in the specified Docker environment without manual intervention. This generates a Docker container with intermediate results that allows anyone to rerun analysis in the same environment, produces updated figures and stores logs describing what occurred. Example configurations are available in Online Methods and at https://github.com/greenelab/continuous_analysis.

Example - Genomics Annotations

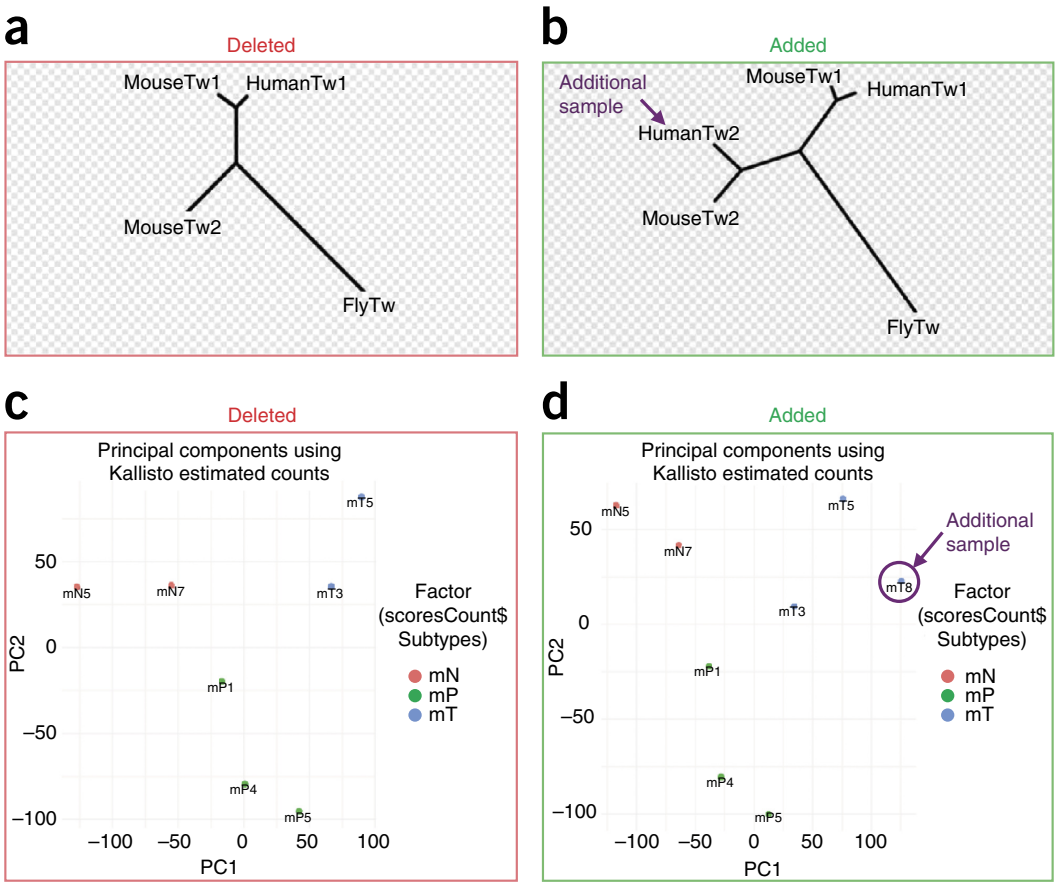

Figure 4 Reproducible workflows with continuous analysis. (**a**,**b**) Phylogenetic tree building with four mRNA samples (MouseTw1, HumanTw1, MouseTw2 and FlyTw) (**a**) and an additional gene (HumanTw2) (**b**). (**c**,**d**) RNA-seq differential expression experiment principal component (PC) analysis before (**c**) and after (**d**) addition of a sample (mT8).

Associated topics - Property-based testing

- Property-based testing is a slight variant of unit testing
  - Rather than specifying an exact program to run, you specify some properties that should hold
  - e.g. if C = f(A, B), C > B and C > A
- Methods exist to test whether this is true, and do it in a way that is more efficient than trying random values
- `hypothesis` is a python package for this

Associated topics - Fuzzing

- Let's say you have a function that reads in some random data from the internet then processes it
  - You want to make sure it won't crash no matter what's sent to it
- Fuzzing is the process of sending all sorts of random data to a program, to make sure it keeps working
  - "Working" can be not crashing, but also that you still get some proper output
- Tools like afl (American Fuzzy Lop) can peer into your program to actively look for random inputs to make it behave differently
- Packages like `datafuzz` can perform the data equivalent—i.e. add noise, etc, that shouldn't influence the analysis

Associated topics - Linting

- It's possible to write a program that is correct and runs, but just isn't written well
  - E.g. Hard to read
  - Fails to follow best practices
  - The interpreter/compiler figures it out, but it's a bad idea
- For example:
  `pythonfunctioniwrotemyselfonasundayafterwaytoomuchcoffe`
  `= False)`
- Linters are programs to check various rules about how code should be written
- Can also use static analysis to determine potential errors
- `pylint` is a linter for python code

Summary

- For complex analysis make sure to use unit testing
- Exactly what testing is necessary or helpful depends on the problem at hand
- More testing isn't as important as better testing
- For multi-part problems, it's important to think of all the factors affecting your outcome

Further Reading

- `unittest`, the built-in python testing framework
- `hypothesis`, a python package for property-based testing
- `datafuzz`, a python package for data fuzzing
- `pylint`, a python linter