Testing scientific code

Because you're worth it

Pietro Berkes and Lisa Schwetlick

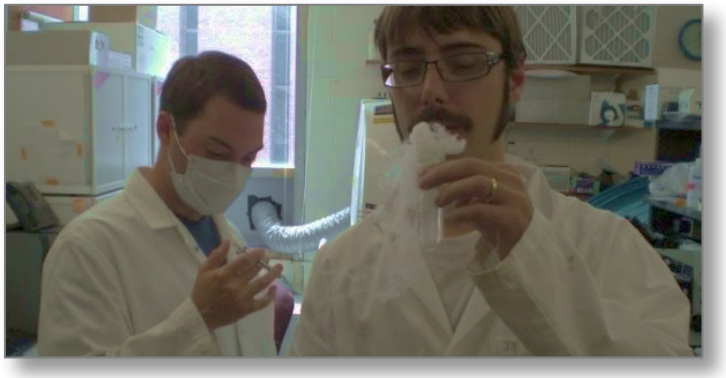
You, as the Master of Research

You start a new project and identify a number of possible leads.

You **quickly develop a prototype** of the most promising ones; once a prototype is finished, you can **confidently decide** whether it is is a dead end, or worth pursuing.

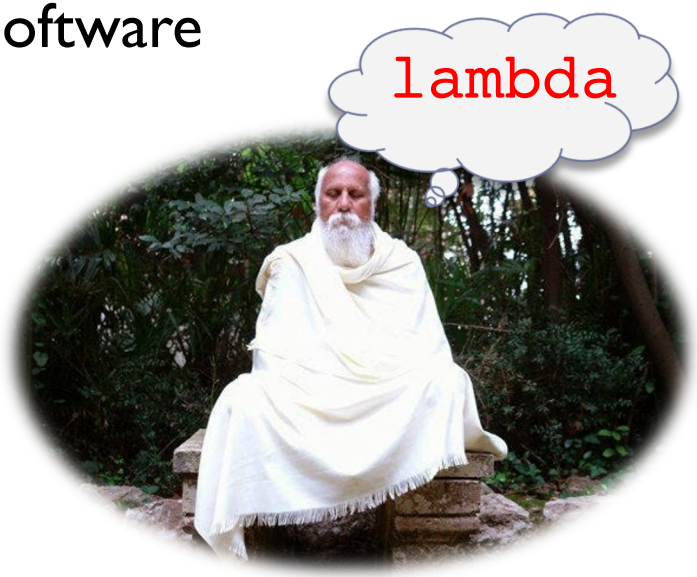
Once you find an idea that is worth spending energy on, you take the prototype and **easily re-organize and optimize it** so that it scales up to the full size of your problem.

As expected, the scaled up experiment delivers good results and your next paper is under way.



How to reach enlightenment

- How do we get to the blessed state of **confidence** and **efficiency**?
- Being a Python expert is not sufficient, good programming practices make a big difference
- We can learn a lot from the development methods developed for commercial and open source software



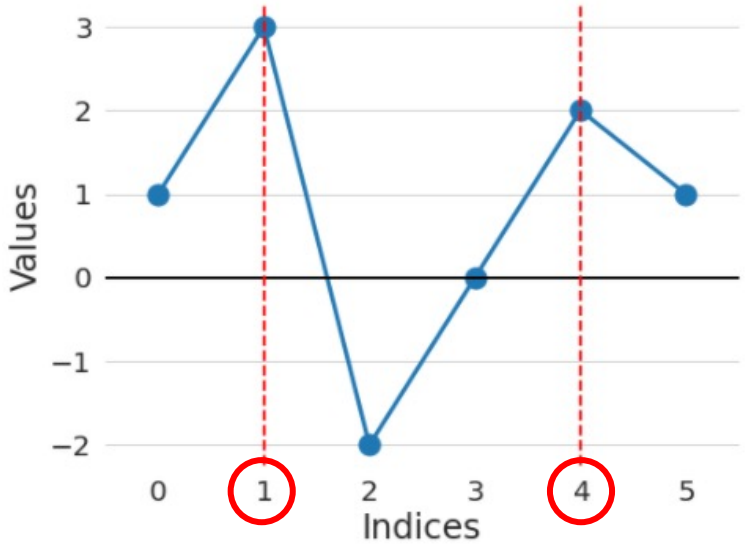
Outline

- The agile programming cycle
- Testing scientific code basics
- Testing patterns for scientific code
- Continuous Integration

Warm-up project

- Create a directory called `local_maxima` in the directory `hands_on`
- In a file called `local_maxima.py`, write a function `find_maxima` that finds the indices of local maxima in a list of numbers

- For example,
  `find_maxima([1, 3, -2, 0, 2, 1])`
  should return
  `[1, 4]`
  the indices of the two local maxima

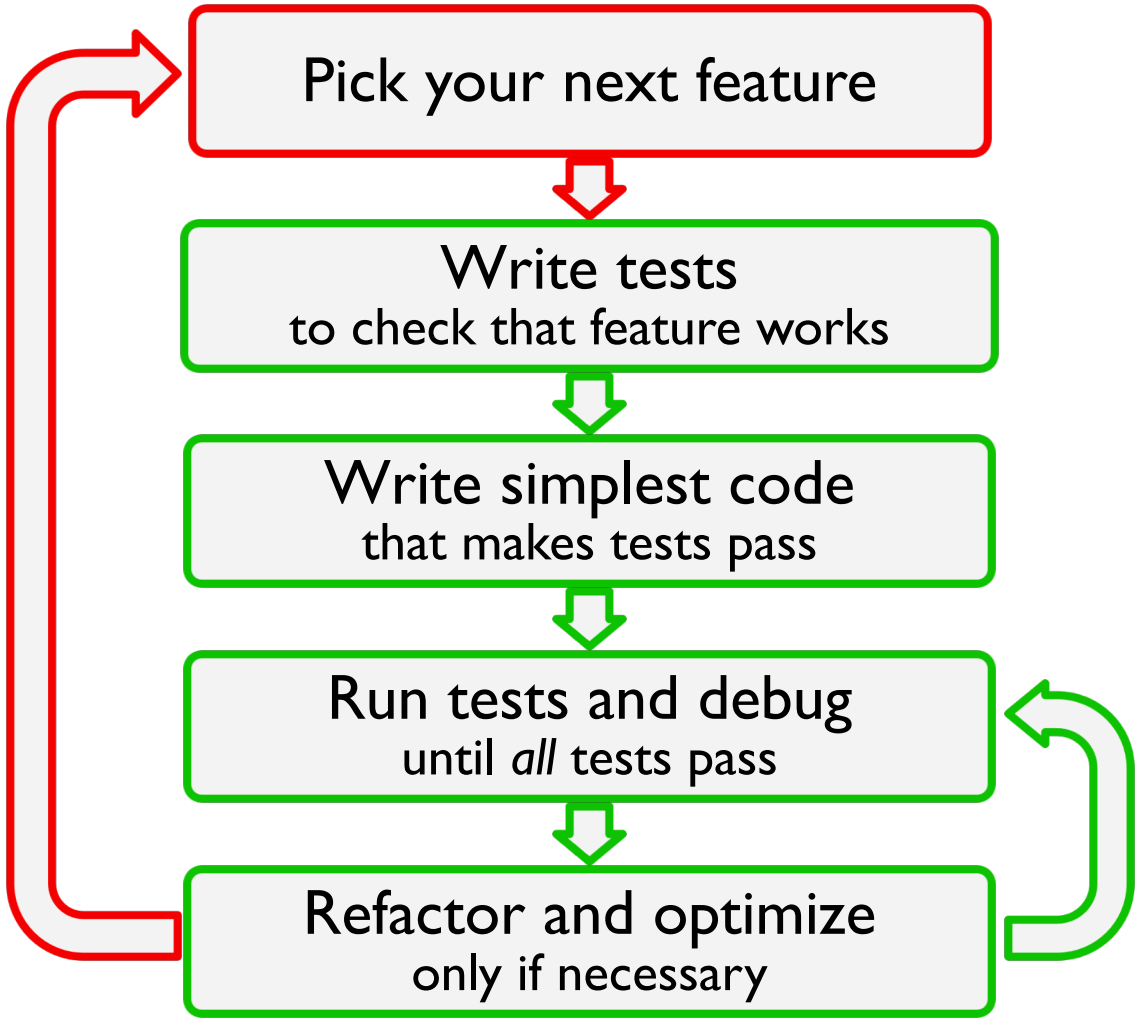


Warm-up project

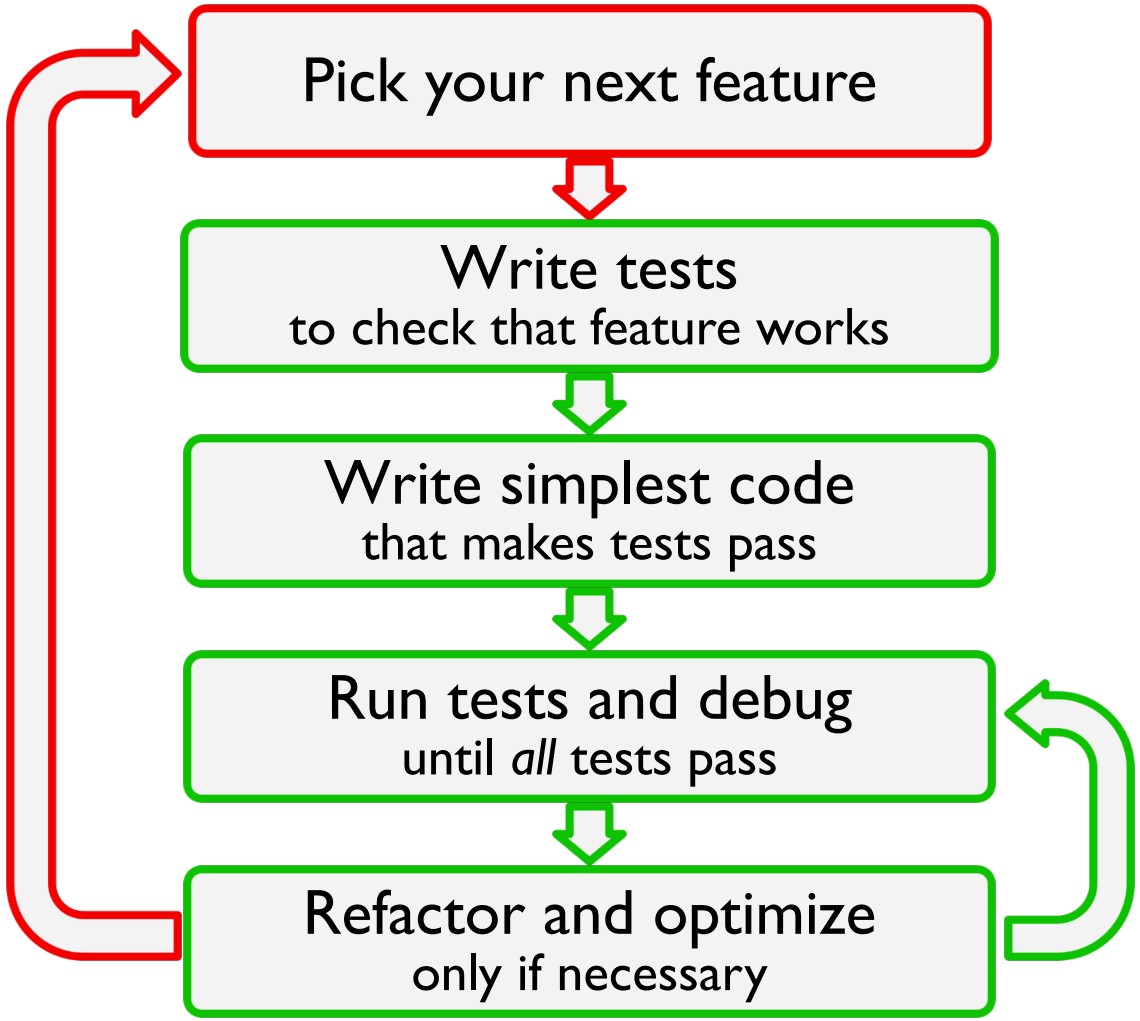
- Write a function that finds the position of local maxima in a list of numbers
- Check your solution with these inputs:
  - Input: [1, 3, -2, 0, 2, 1] Expected result: [1, 4]
  - Input: [-1, -1, 0, -1] Expected result: [2]
  - Input: [4, 2, 1, 3, 1, 5] Expected result: [0, 3, 5]
  - Input: [1, 2, 2, 1] Expected result: [1] (or [2], or [1, 2])

The agile programming cycle

The agile development cycle



Python tools for agile development



`py.test`

`pdb`

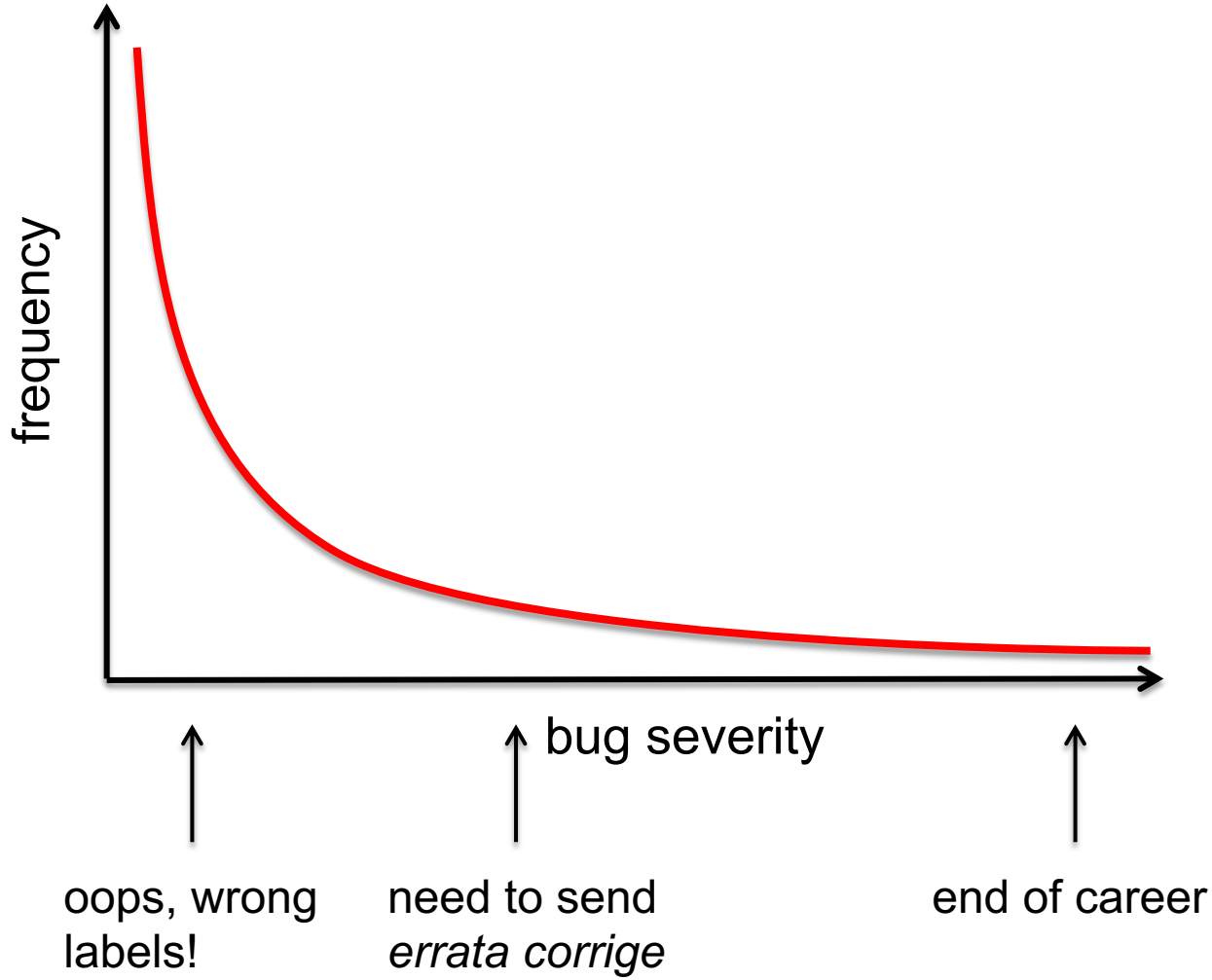
`timeit`
`cProfile`
`line_profiler`

Testing scientific code

Why write tests? Confidence and correctness

- Confidence:
  - Write the code once and use it confidently everywhere else: avoid the *negative result* effect!
  - **Correctness** is main requirement for scientific code
  - You must have a strategy to ensure correctness

Effect of software bugs in science



The unfortunate story of Geoffrey Chang

Science, Dec 2006: 5 high-profile retractions (3x Science, PNAS, J. Mol. Biol.) because "an in-house data reduction program introduced a change in sign for anomalous differences"

SCIENTIFIC PUBLISHING

A Scientist's Nightmare: Software Problem Leads to Five Retractions

Until recently, Geoffrey Chang's career was on a trajectory most young scientists only dream about. In 1999, at the age of 28, the protein crystallographer landed a faculty position at the prestigious Scripps Research Institute in San Diego, California. The next year, in a cer-

2001 *Science* paper, which described the structure of a protein called MsbA, isolated from the bacterium *Escherichia coli*. MsbA belongs to a huge and ancient family of molecules that use energy from adenosine triphosphate to transport molecules across cell membranes. These

LETTERS

edited by Etta Kavanagh

Retraction

WE WISH TO RETRACT OUR RESEARCH ARTICLE "STRUCTURE OF MsbA from *E. coli*: A homolog of the multidrug resistance ATP binding cassette (ABC) transporters" and both of our Reports "Structure of the ABC transporter MsbA in complex with ADP·vanadate and lipopolysaccharide" and "X-ray structure of the EmrE multidrug transporter in complex with a substrate" (*1*–*3*).

The recently reported structure of Sav1866 (*4*) indicated that our MsbA structures (*1*, *2*, *5*) were incorrect in both the hand of the structure and the topology. Thus, our biological interpretations based on these inverted models for MsbA are invalid.

An in-house data reduction program introduced a change in sign for anomalous differences. This program, which was not part of a conventional data processing package, converted the anomalous pairs (I+ and I–) to (F– and F+), thereby introducing a sign change. As the diffraction data collected for each set of MsbA crystals and for the EmrE crystals were processed with the same program, the structures reported in (*1*–*3*, *5*, *6*) had the wrong hand.

Catastrophic software errors doomed Boeing's airplanes and nearly destroyed its NASA spaceship. Experts blame the leadership's 'lack of engineering culture.'

Morgan McFall-Johnsen | Publié le 29/02/2020 à 14h07

Just 31 minutes after Boeing's CST-100 Starliner spaceship launched into space, Mission Control knew something was wrong.

In the early stages of that crucial test flight on December 20, the Starliner's engines were supposed to fire automatically, setting the ship on a course toward the International Space Station - but they never did.

Mission controllers soon realized the problem: The Starliner's clock was 11 hours ahead. It was following the steps of a phase of the mission it had not yet reached, firing small thrusters to adjust its position.

[...]

As Boeing workers frantically checked hundreds of thousands of lines of the spaceship's code, they found a *second* error - one that would have caused the wrong thrusters to fire after two modules of the spacecraft separated. That could have led to a disastrous collision in space.

The Orlando Sentinel reported on Wednesday that a critical end-to-end software test ahead of the Starliner launch could have caught the two coding errors, but Boeing didn't conduct that test at all.

https://www.businessinsider.fr/us/boeing-software-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2

Testing basics

Testing frameworks for Python

- unittest
- nosetests
- **py.test**

Test suites in Python with py.test

- Writing tests with py.test is simple:
  - Each test is a function whose name begins by "`test_`"
  - Each test tests **one** feature in your code, and checks that it behaves correctly using "assertions". An exception is raised if it does not work as expected.

Testing with Python

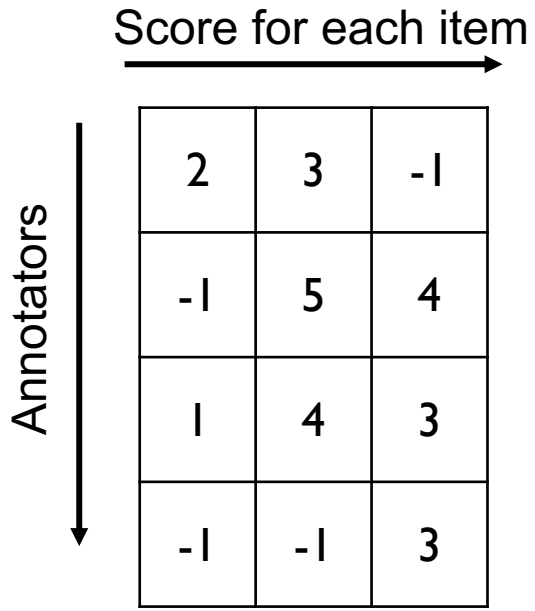
- Tests are automated:
  - Write test suite in parallel with your code
  - External software runs the tests and provides reports and statistics

```
============================= test session starts ==============================
platform darwin -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1 --
/Users/pberkes/miniconda3/envs/gnode/bin/python
cachedir: .cache
rootdir:
/Users/pberkes/o/pyschool/testing_debugging_profiling/hands_on/pyanno_voting_solu
tion, inifile:
collected 4 items

pyanno/tests/test_voting.py::test_labels_count PASSED
pyanno/tests/test_voting.py::test_majority_vote PASSED
pyanno/tests/test_voting.py::test_majority_vote_empty_item PASSED
pyanno/tests/test_voting.py::test_labels_frequency PASSED
========================== 4 passed in 0.23 seconds ============================
```

Hands-on!

- Go to `hands_on/pyanno_voting`
- Execute the tests:
  `pytest`



A score of MISSING_VALUE (-1) means the annotator did not score that item

How to run tests

- 1) Discover all tests in all subdirectories

```
pytest -v
```

- 2) Execute all tests in one module

```
pytest -v pyanno/tests/test_voting.py
```

- 3) Execute one single test

```
pytest -v test_voting.py::test_majority_vote
```

Possibly your first test file

- Create a new file, `test_something.py`:

```
def test_arithmetic():
    assert 1 == 1
    assert 2 * 3 == 6

def test_len_list():
    lst = ['a', 'b', 'c']
    assert len(lst) == 3
```

- Save it, and execute the tests

Assertions

- `assert` statements check that some condition is met, and raise an exception otherwise
- Check that statement is true/false:

```
assert 'Hi'.islower()        => fail
assert not 'Hi'.islower()    => pass
```

- Check that two objects are equal:

```
assert 2 + 1 == 3            => pass
assert [2] + [1] == [2, 1]   => pass
assert 'a' + 'b' != 'ab'     => fail
```

- `assert` can be used to compare all sorts of objects, and py.test will take care of producing an approriate error message

Hands-on!

- Add a new test to `test_something.py`: test that 1+2 is 3
- Execute the tests

Hands-on!

- Add a new test to `test_something.py`: test that 1+2 is 3
- Execute the tests
- Now test that 1.1 + 2.2 is 3.3

Floating point equality

- Real numbers are represented approximately as "floating point" numbers. When developing numerical code, we have to allow for approximation errors.
- Check that two numbers are approximately equal:

```
from math import isclose
def test_floating_point_math():
    assert isclose(1.1 + 2.2, 3.3)          => pass
```

- `abs_tol` controls the absolute tolerance:

```
assert isclose(1.121, 1.2, abs_tol=0.1)     => pass
assert isclose(1.121, 1.2, abs_tol=0.01)    => fail
```

- `rel_tol` controls the relative tolerance:

```
assert isclose(120.1, 121.4, rel_tol=0.1)   => pass
assert isclose(120.4, 121.4, rel_tol=0.01)  => fail
```

Hands-on!

- One more equality test: check that the sum of these two NumPy arrays:

```
x = numpy.array([1, 1])
y = numpy.array([2, 2])
```

  is equal to

```
z = numpy.array([3, 3])
```

Testing with NumPy arrays

```
def test_numpy_equality():
    x = numpy.array([1, 1])
    y = numpy.array([2, 2])
    z = numpy.array([3, 3])
    assert x + y == z
```

```
_____________________________________ test_numpy_equality _____________________________________

    def test_numpy_equality():
        x = numpy.array([1, 1])
        y = numpy.array([2, 2])
        z = numpy.array([3, 3])
>       assert x + y == z
E       ValueError: The truth value of an array with more than one element is ambiguous.
Use a.any() or a.all()

code.py:47: ValueError
```

Testing with numpy arrays

- `numpy.testing` defines appropriate functions:

```
assert_array_equal(x, y)
assert_array_almost_equal(x, y, decimal=6)
```

- If you need to check more complex conditions:
  - `numpy.all(x)`: returns True if all elements of x are true

    `numpy.any(x)`: returns True is any of the elements of x is true

    `numpy.allclose(x, y, rtol=1e-05, atol=1e-08)`: returns True if two arrays are element-wise equal within a tolerance

  - combine with `logical_and, logical_or, logical_not`:

```
# test that all elements of x are between 0 and 1
assert all(logical_and(x > 0.0, x < 1.0))
```

Hands-on!

- Submit a Pull Request for Issue #1 on GitHub

  - Create a branch with a unique name (e.g. testing-pb-727)
  - Switch to that branch
  - Solve the issue and commit to the branch (one or more commits)
  - Push the branch to GitHub
  - In GitHub, go to "Pull Requests" and open a pull request.
  - In the PR description write "Fixes #2" somewhere, this is going to create an automatic link to the issue, and close the issue if the PR is merged

Testing error control

- Check that an exception is raised:

```
from py.test import raises
def test_raises():
    with raises(SomeException):
        do_something()
        do_something_else()
```

- For example:

```
with raises(ValueError):
    int('XYZ')
```

passes, because

```
int('XYZ')
ValueError: invalid literal for int() with base 10: 'XYZ'
```

Testing error control

- Use the most specific exception class, or the test may pass because of collateral damage:

```
# Test that file "None" cannot be opened.
with raises(IOError):
    open(None, 'r')                    => fail
```

as expected, but

```
                                       => pass
with raises(Exception):
    open(None, 'r')
```

Hands-on!

- Submit a Pull Request for Issue #2 on GitHub

  - Check out the master branch
  - Update the master branch with the new commits from upstream
  - Create a branch with a new unique name (e.g. testing-pb-007)
  - Solve and create a PR as you did before

Up next:
Testing patterns