Before we start, clone the git repository

```
https://github.com/ASPP/2019-camerino-testing-debugging
```

Testing scientific code

Because you're worth it

Pietro Berkes, NAGRA Insight

You, as the Master of Research

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

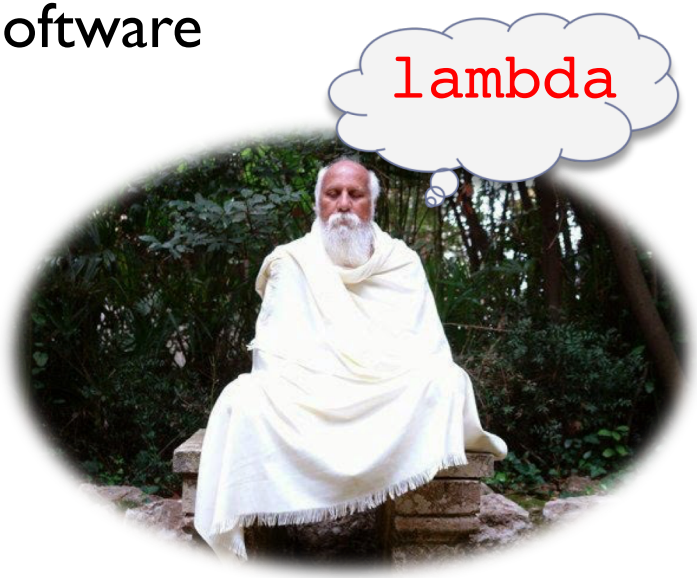
You **quickly develop a prototype** of the most promising ones; once a prototype is finished, you can **confidently decide** whether it is is a dead end, or worth pursuing.

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

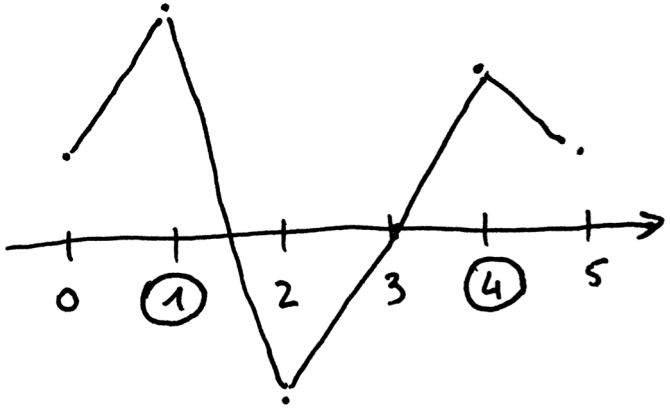
- The agile programming cycle
- Testing scientific code basics
- Testing patterns for scientific code

Warm-up project

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

Warm-up project

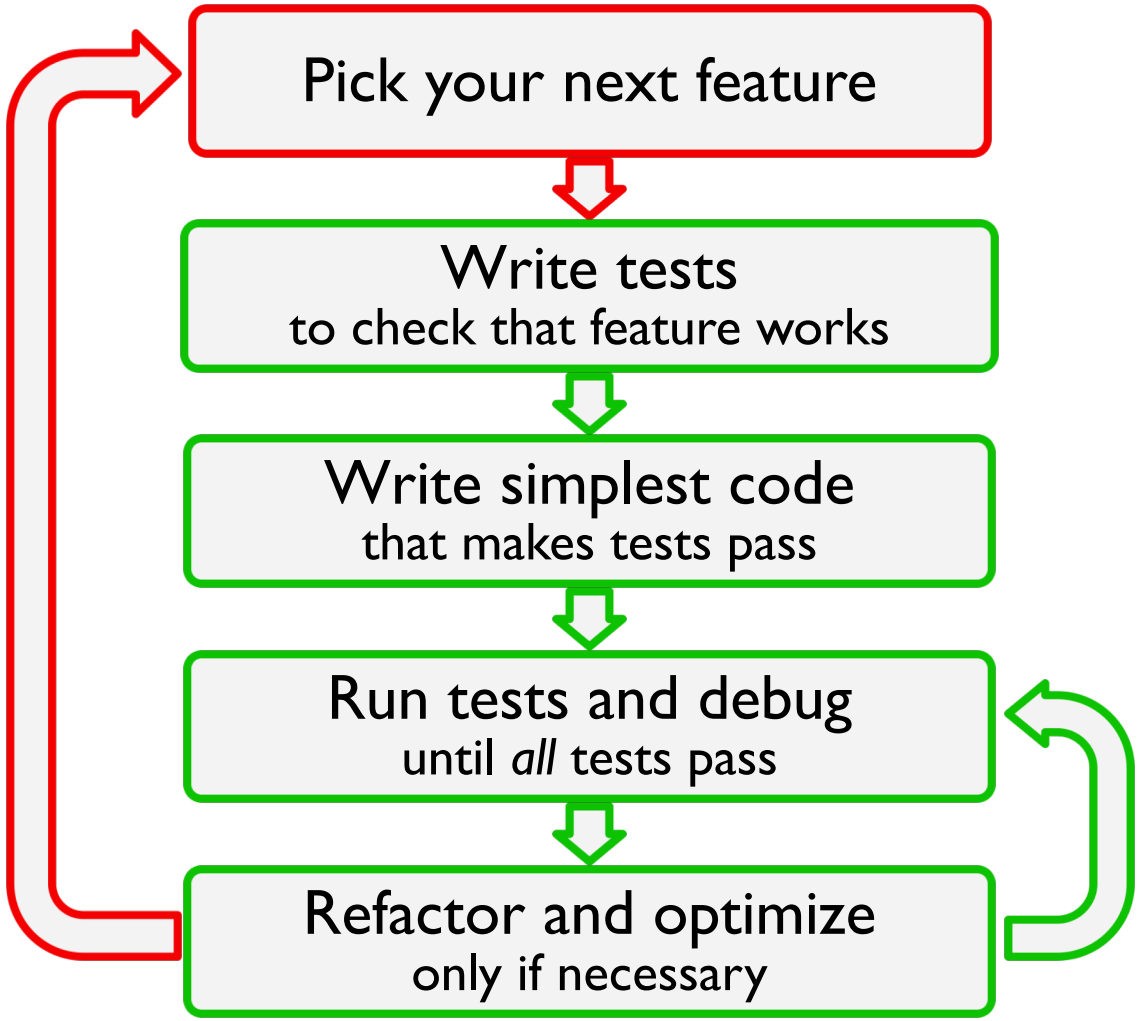
- Write a function that finds the position of local maxima in a list of numbers



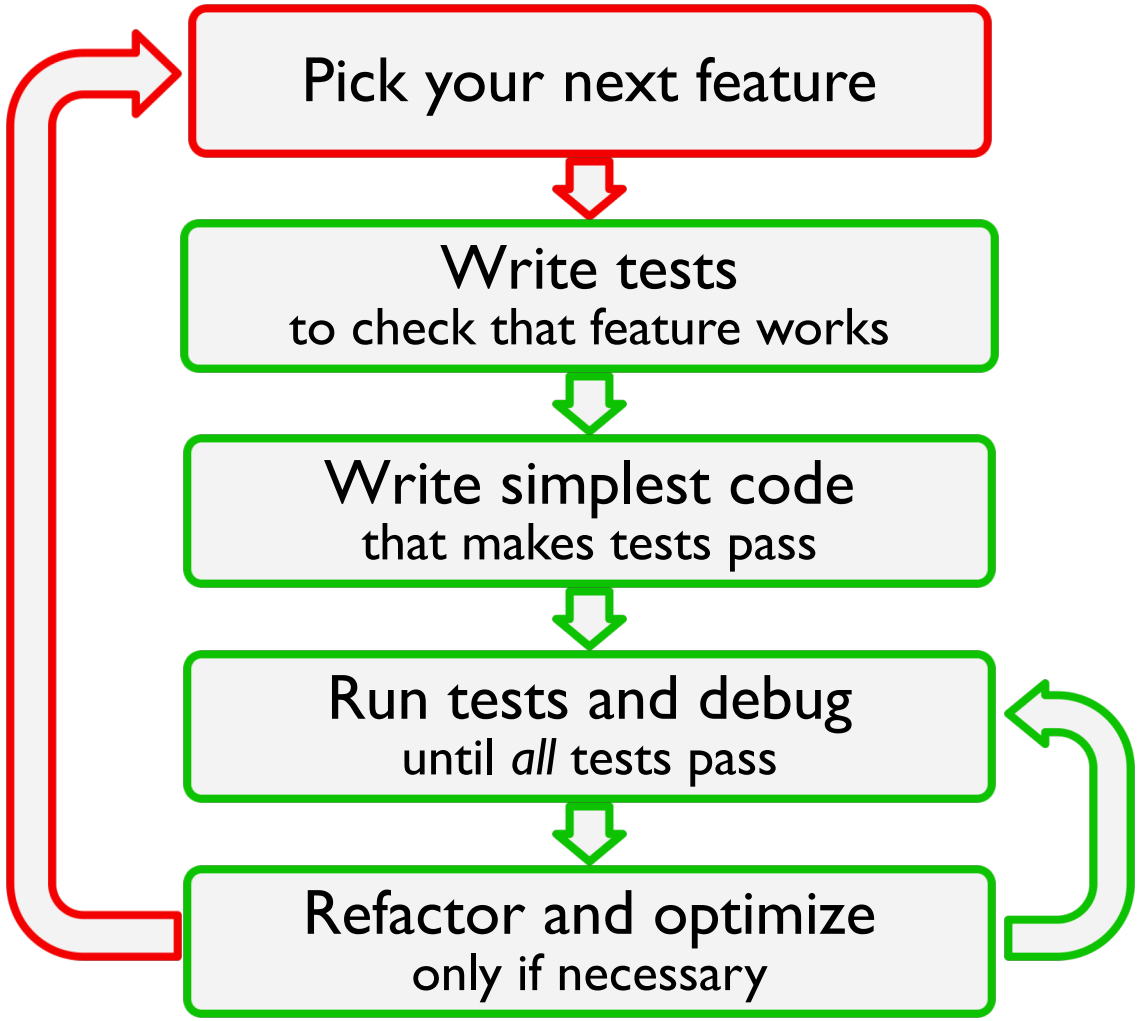
- Check your solution with these inputs:
  - Input: [1, 4, -5, 0, 2, 1] Expected result: [1, 4]
  - Input: [-1, -1, 0, -1] Expected result: [2]
  - Input: [4, 2, 1, 3, 1, 5] Expected result: [0, 3, 5]
  - Input: [1, 2, 2, 1] Expected result: [1] (or [2], or [1, 2])

The agile programming cycle

The agile development cycle



Python tools for agile development



`py.test`

`pdb`

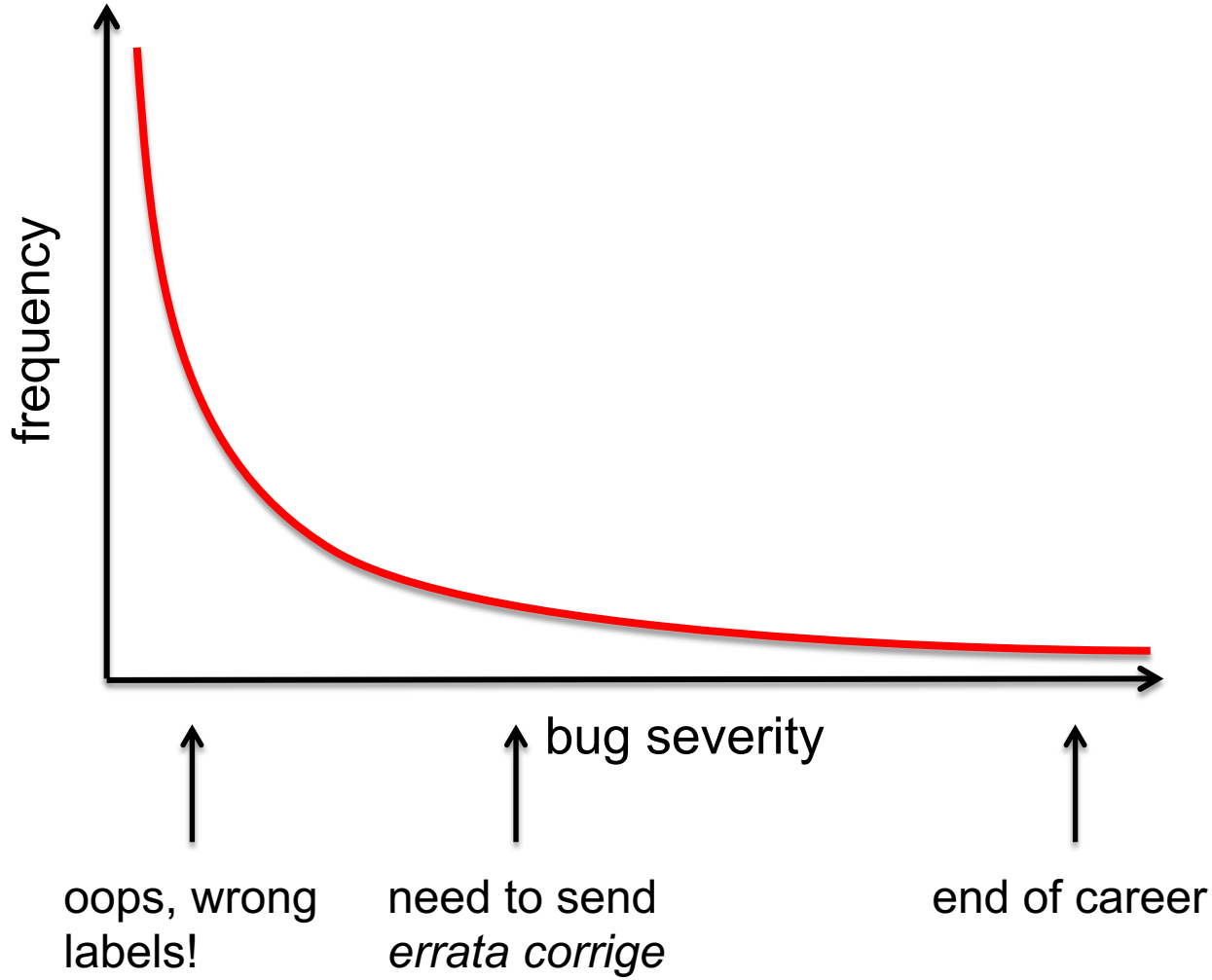
`timeit`
`cProfile`
`line_profiler`

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.

Meanwhile, in Hawaii…

Hawaii Sends Out a State-Wide False Alarm About a Missile Strike

On January 13, the citizens of Hawaii were notified to take immediate cover in the face of an inbound ballistic missile strike. It turned out to be a false alarm, although it took over 30 minutes (and, presumably, several thousand heart attacks) before the alert was retracted. Investigations found that while the problem was largely due to human error, there were "troubling" design flaws in the Hawaii Emergency Management Agency's alert origination software. *Read more*.

Why it's interesting: The Hawaii Emergency Management Agency was chastised for having no discernible differences between their testing and live alert environments, making it extremely easy to mistake which environment they were using at the moment. While software *bugs* (defined as a software failing to perform as designed) are the most common types of fails in the Software Fail Watch, it is dangerous to underestimate the damage poorly *designed* software can incur.

Financial Damage	Brand Damage	Bodily Damage

January 13, 2018

https://www.tricentis.com/blog/2018/04/12/software-fail-watch-q1-2018/

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

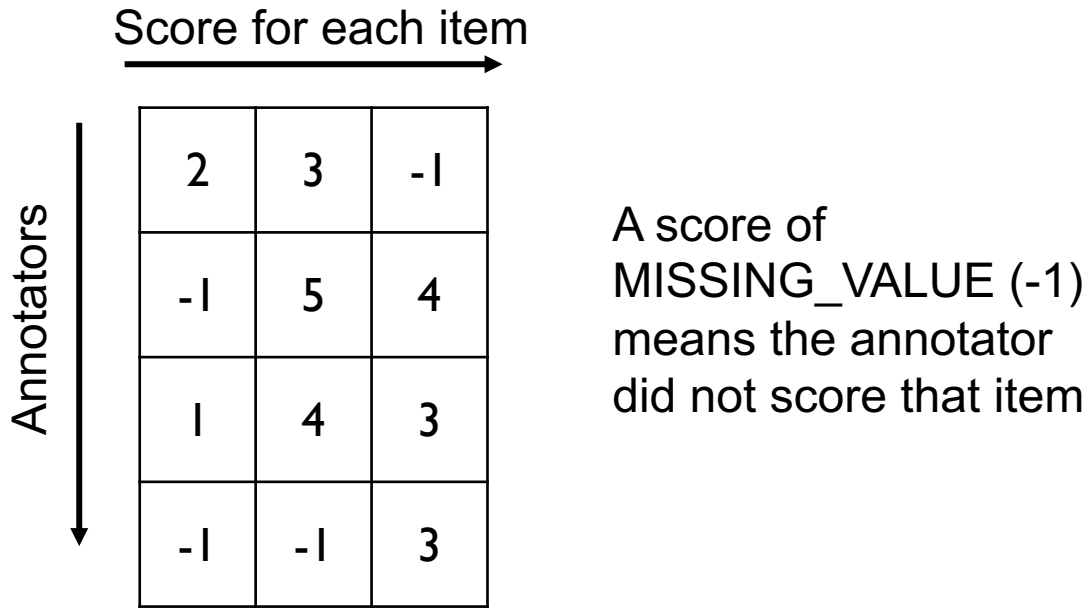
- Tests are automated:
  - Write test suite in parallel with your code
  - External software runs the tests and provides reports and statistics

```
============================= 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:
  `py.test`



How to run tests

- 1) Discover all tests in all subdirectories

```
py.test -v
```

- 2) Execute all tests in one module

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

- 3) Execute one single test

```
py.test -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 #2 on GitHub
  https://github.com/ASPP/2019-camerino-testing-debugging

  - 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

Short digression on Continuous Integration

- Some PRs have a green check, other a red cross: I connected the repository to Travis CI
- You should always run the tests locally before submitting the PR, as going through Travis CI requires a few minutes and iterating that way can be looong
- The Travis CI script is a first quality check when reviewing PRs:
  - Check that tests have been added for the new functionality
  - Check all tests pass on Travis CI, for all desired Python versions
  - You can add flake8 to the Travis CI script to enforce coding conventions in your project

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 #4 on GitHub
  https://github.com/ASPP/2019-camerino-testing-debugging

  - 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

Testing patterns

What a good test looks like

- What does a good test looks like? What should I test?
- Good:
  - Short and quick to execute
  - Easy to read
  - Exercise *one* thing
- Bad:
  - Relies on data files
  - Messes with "real-life" files, servers, databases

Basic structure of test

- A good test is divided in three parts:
  - **Given**: Put your system in the right state for testing
    - Create data, initialize parameters, define constants…
  - **When**: Execute the feature that you are testing
    - Typically one or two lines of code
  - **Then**: Compare outcomes with the expected ones
    - Define the expected result of the test
    - Set of *assertions* that check that the new state of your system matches your expectations

Test simple but general cases

- Start with simple, general case
  - Take a realistic scenario for your code, try to reduce it to a simple example
- Tests for 'lower' method of strings

```
def test_lower():
    # Given
    string = 'HeLlO wOrld'
    expected = 'hello world'

    # When
    output = string.lower()

    # Then
    assert output == expected
```

Test special cases and boundary conditions

- Code often breaks in corner cases: empty lists, None, NaN, 0.0, lists with repeated elements, non-existing file, …
- This often involves making design decision: respond to corner case with special behavior, or raise meaningful exception?

```
def test_lower_empty_string():
    # Given
    string = ''
    expected = ''

    # When
    output = string.lower()

    # Then
    assert output == expected
```

- Other good corner cases for string.lower():
  - 'do-nothing case': `string = 'hi'`
  - symbols: `string = '123 (!'`

Common testing pattern

- Often these cases are collected in a single test:

```
def test_lower():
    # Given
    # Each test case is a tuple of (input, expected_result)
    test_cases = [('HeLlO wOrld', 'hello world'),
                  ('hi', 'hi'),
                  ('123 ([?', '123 ([?'),
                  ('', '')]

    for string, expected in test_cases:
        # When
        output = string.lower()
        # Then
        assert output == expected
```

Hands-on!

- Submit a Pull Request for Issue #5 on GitHub
  https://github.com/ASPP/2019-camerino-testing-debugging

  - Check out and update the master branch
  - Create a branch with a new, unique name
  - Solve and create a PR as you did before

Numerical fuzzing

- Use deterministic test cases when possible
- In most numerical algorithm, this will cover only over-simplified situations; in some, it is impossible
- Fuzz testing: generate random input
  - Outside scientific programming it is mostly used to stress-test error handling, memory leaks, safety
  - For numerical algorithm, it is often used to make sure one covers general, realistic cases
  - The input may be random, but you still need to know what to expect
  - Make failures reproducible by saving or printing the random seed

Numerical fuzzing example

```
def test_mean_deterministic():
    x = numpy.array([-2.0, 2.0, 6.0])
    expected = 2.0
    assert isclose(numpy.mean(x), expected)


def test_mean_fuzzing():
    rand_state = numpy.random.RandomState(1333)

    N, D = 100000, 5
    # Goal means: [0.1 ,  0.45,  0.8 ,  1.15,  1.5]
    expected = numpy.linspace(0.1, 1.5, D)

    # Generate random, D-dimensional data with the desired mean
    x = rand_state.randn(N, D) + expected
    means = numpy.mean(x, axis=0)
    numpy.testing.assert_allclose(means, expected, rtol=1e-2)
```

Hands-on!

- Submit a Pull Request for Issue #6 on GitHub
  https://github.com/ASPP/2019-camerino-testing-debugging

  - Check out and update the master branch
  - Create a branch with a new, unique name
  - Solve and create a PR as you did before

Strategies for testing learning algorithms

- Learning algorithms can get stuck in local maxima, the solution for general cases might not be known (e.g., unsupervised learning)
- Turn your validation cases into tests
- Stability tests:
  - Start from final solution; verify that the algorithm stays there
  - Start from solution and add a small amount of noise to the parameters; verify that the algorithm converges back to the solution
- Generate data from the model with known parameters
  - E.g., linear regression: generate data as $y = a*x + b + noise$ for random a, b, and x, then test that the algorithm is able to recover a and b

Other common cases

- Test general routines with specific ones
  - Example: test `polynomial_expansion(data, degree)` with `quadratic_expansion(data)`
- Test optimized routines with brute-force approaches
  - Example: test function computing analytical derivative with numerical derivative

Example: eigenvector decomposition

- Consider the function `values, vectors = eigen(matrix)`
- Test with simple but general cases:
  - use full matrices for which you know the exact solution (from a table or computed by hand)
- Test general routine with specific ones:
  - use the analytical solution for 2x2 matrices
- Numerical fuzzing:
  - generate random eigenvalues, random eigenvector; construct the matrix; then check that the function returns the correct values
- Test with boundary cases:
  - test with diagonal matrix: is the algorithm stable?
  - test with a singular matrix: is the algorithm robust? Does it raise appropriate error when it fails?

Hands-on!

- Submit a Pull Request for Issue #7 on GitHub
  https://github.com/ASPP/2019-camerino-testing-debugging

  - Check out and update the master branch
  - Create a branch with a new, unique name
  - Solve and create a PR as you did before

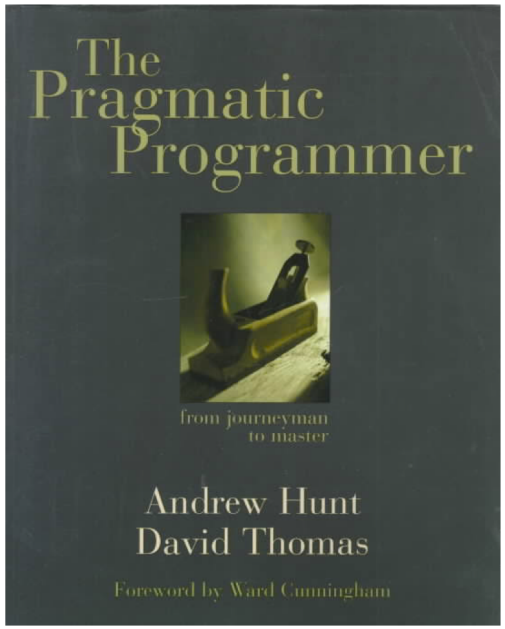
Testing is good for your self-esteem

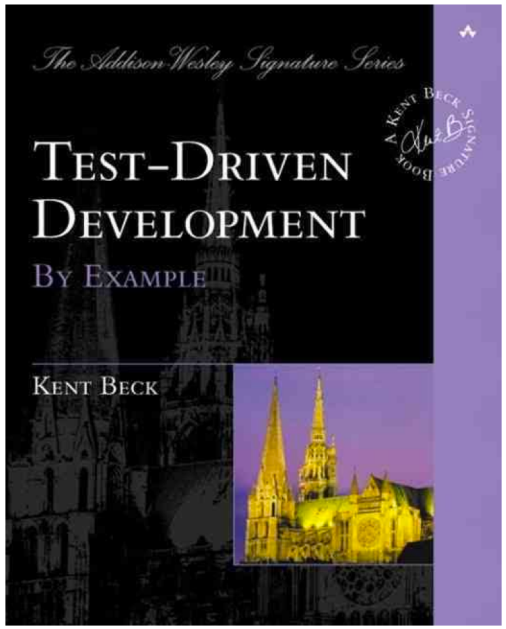
- Immediately: Always be confident that your results are correct, whether your approach works of not
- In the future: save your future self some trouble!
- If you are left thinking "it's cool but I cannot test *my* code because XYZ", talk to me during the week and I'll show you how to do it ;-)

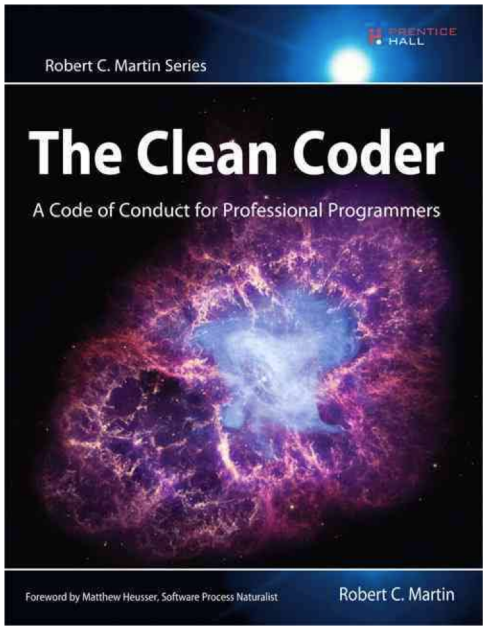
Final thoughts

- Good programming practices, with testing in the front line, make us confident about our results, and efficient at navigating our research projects
- The agile programming cycle gives you intermediate goals to build upon

Recommended reading







Thank you!