Tools to efficiently build scientific code

Mostly testing, some profiling, a little debugging

Pietro Berkes, Twitter Cortex

@masterbaboon

#aspp2016

You as the Master of Research

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

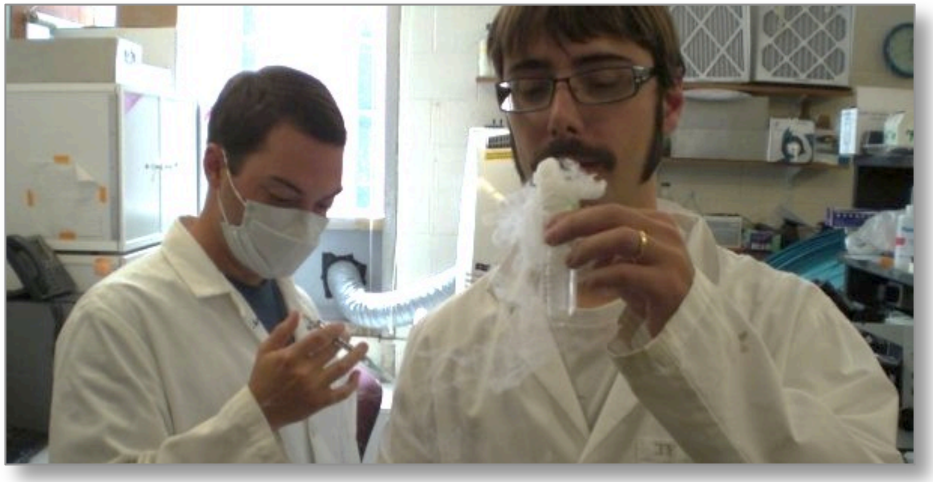
You **quickly develop a prototype** of the most promising ones; once a prototype is finished, you can **confidently decide** whether that lead is a dead end, or worth pursuing.

Once you find an idea that is worth spending energy on, you take the prototype and **easily re-organize it 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.

Reaching 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



Warm-up project

- Write a function 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])

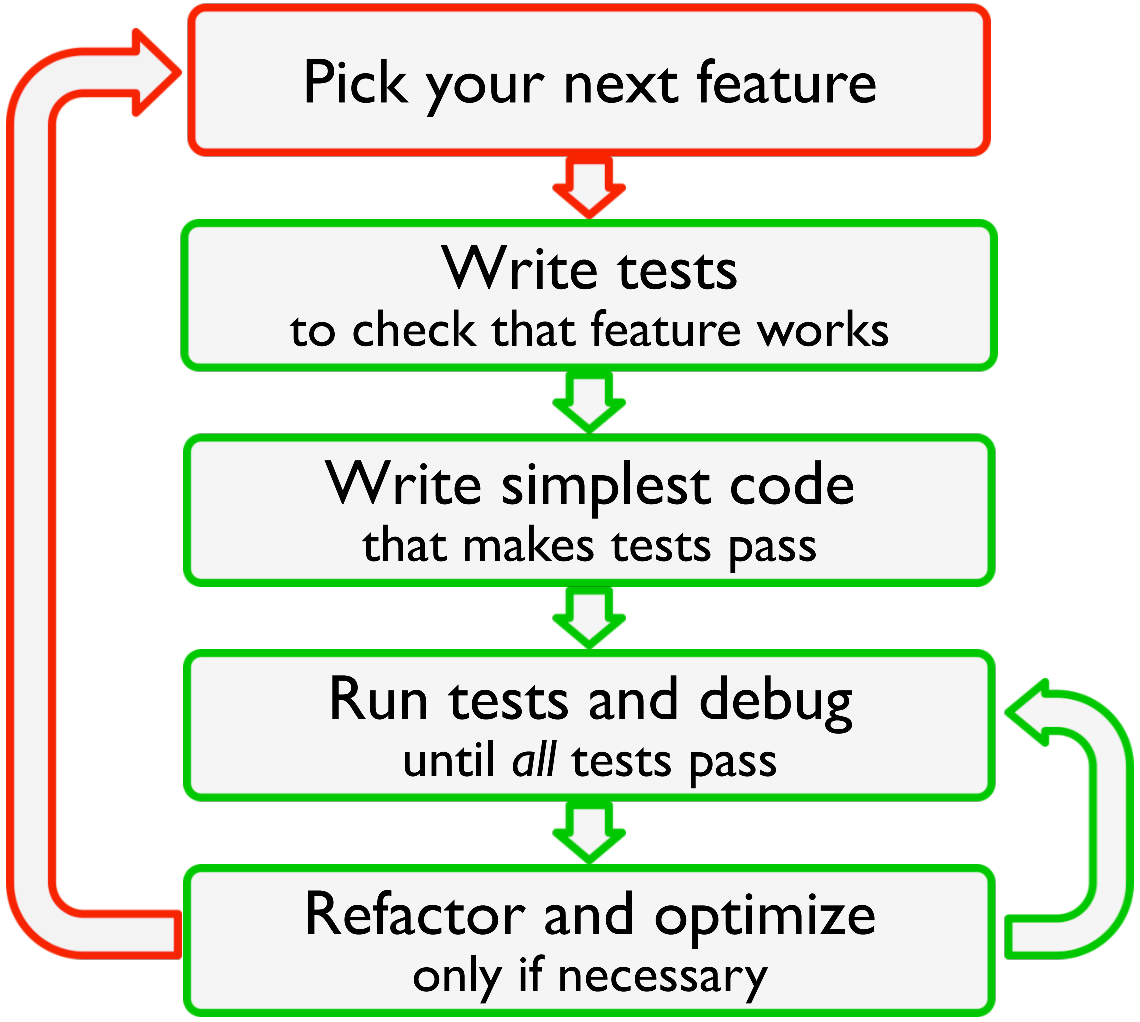
Outline

- The agile programming cycle
- Testing scientific code
- Profiling and optimization
- Debugging

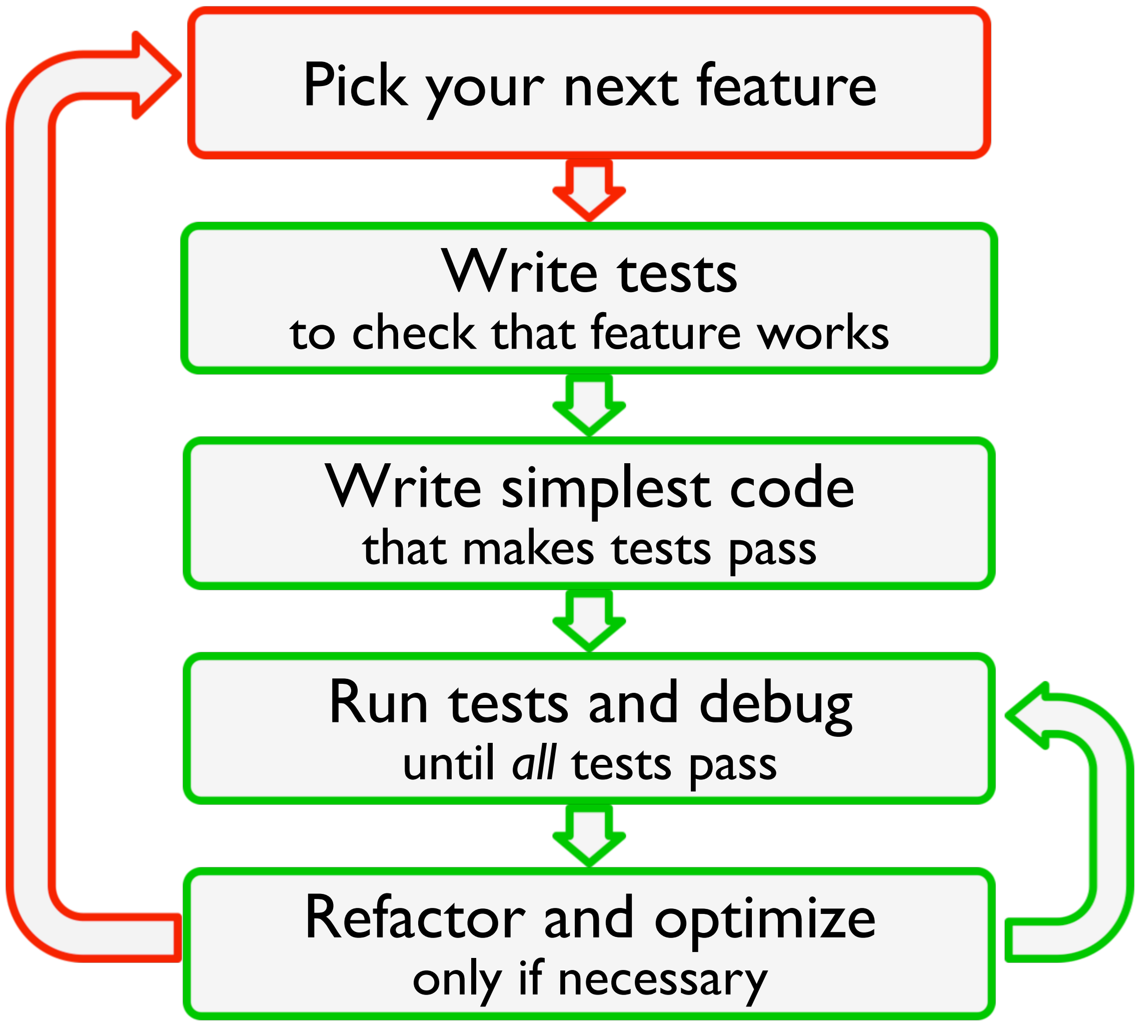
Before we start

- Clone the repository with the material for this class:
  `https://github.com/ASPP/testing_debugging_profiling.git`

The agile development cycle



Python tools for agile development



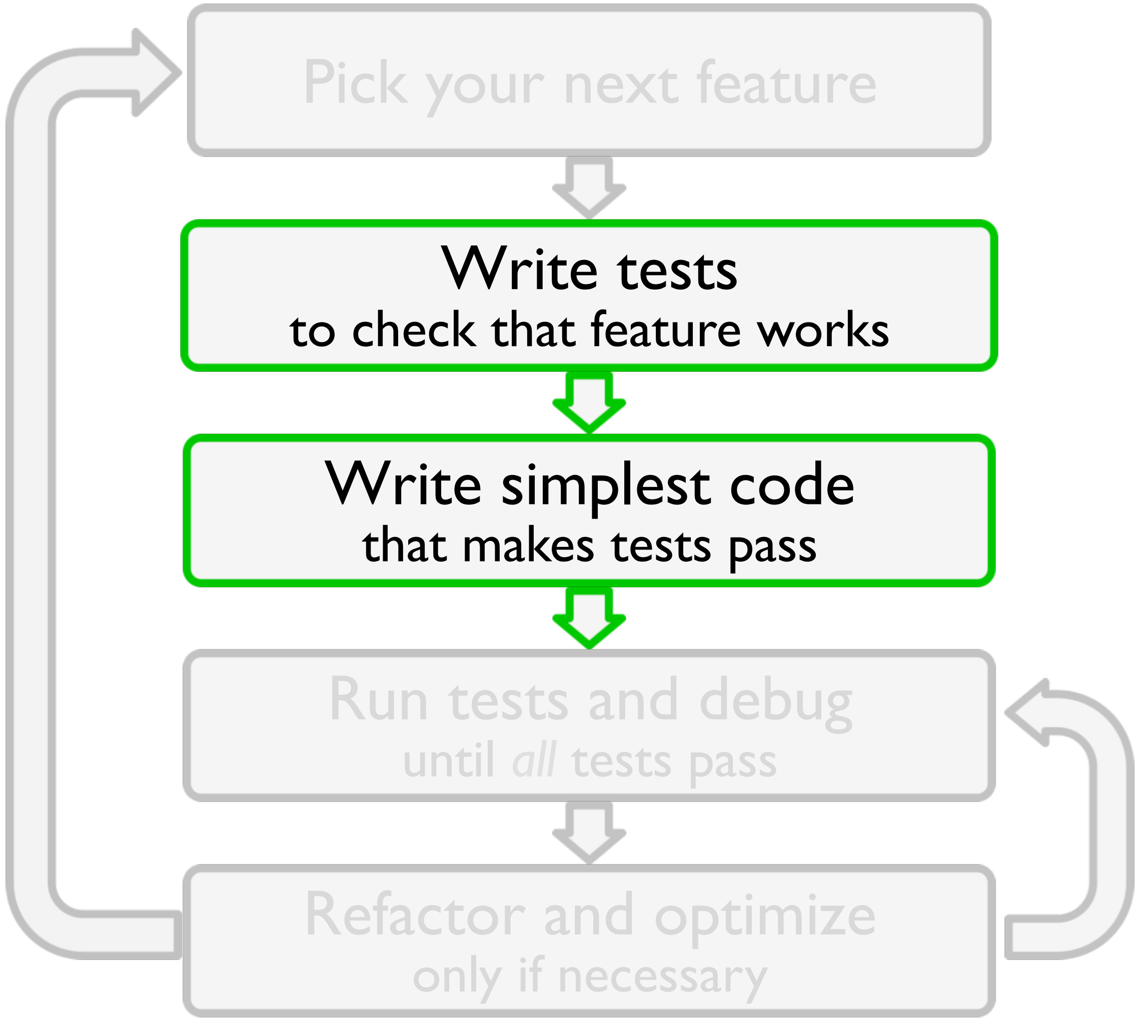
```
py.test

pdb

timeit
cProfile
line_profiler
```

Testing scientific code

The agile development cycle



`py.test`

`pdb`

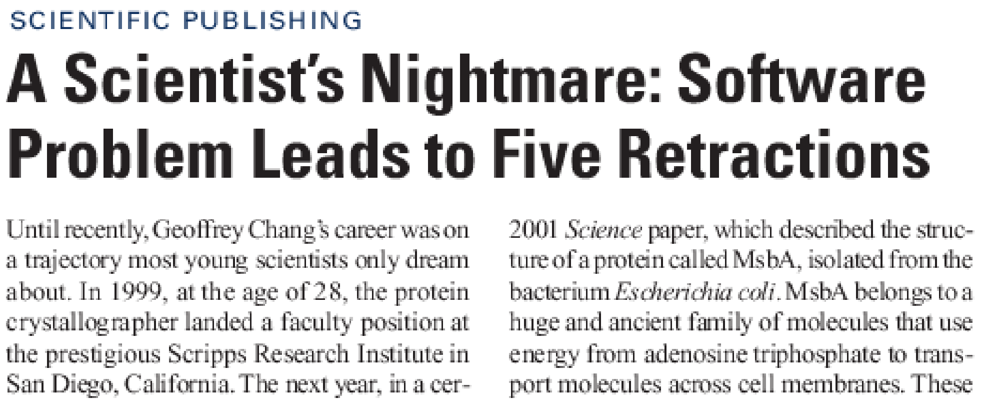
`timeit`
`cProfile`
`line_profiler`

Why write tests?

- Confidence:
  - Tests make you **trust your code**
    - You will know when a result is negative because the approach is wrong, and when there is a bug
  - **Correctness** is main requirement for scientific code
  - You must have a strategy to ensure correctness

The unfortunate story of Geoffrey Chang

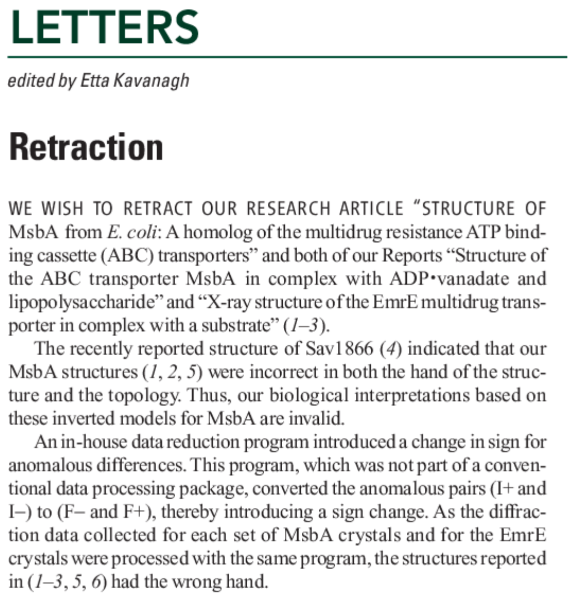
Science, Dec 2006: 5 high-profile retractions (3x Science, PNAS, J. Mol. Biol.) because ”an in-house data reduction program introduced a change in sign for anomalous differences”



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 certain 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 on Wall Street…

LEGAL/REGULATORY | AUGUST 2, 2012, 9:07 AM | 357 Comments

Knight Capital Says Trading Glitch Cost It $440 Million

BY NATHANIEL POPPER

Brendan McDermid/Reuters

◀ 1 2 3 4 ▶

Errant trades from the Knight Capital Group began hitting the New York Stock Exchange almost as soon as the opening bell rang on Wednesday.

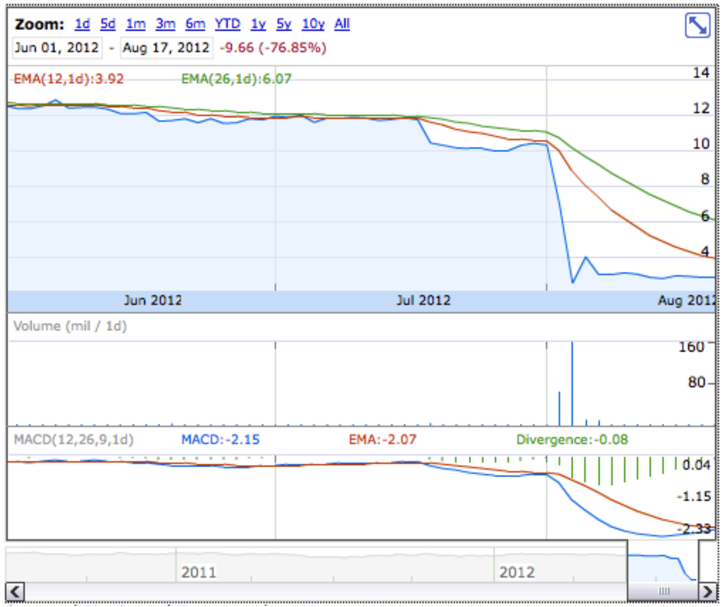
4:01 p.m. | Updated

$10 million a minute.

That's about how much the trading problem that set off turmoil on the stock market on Wednesday morning is already costing the trading firm.

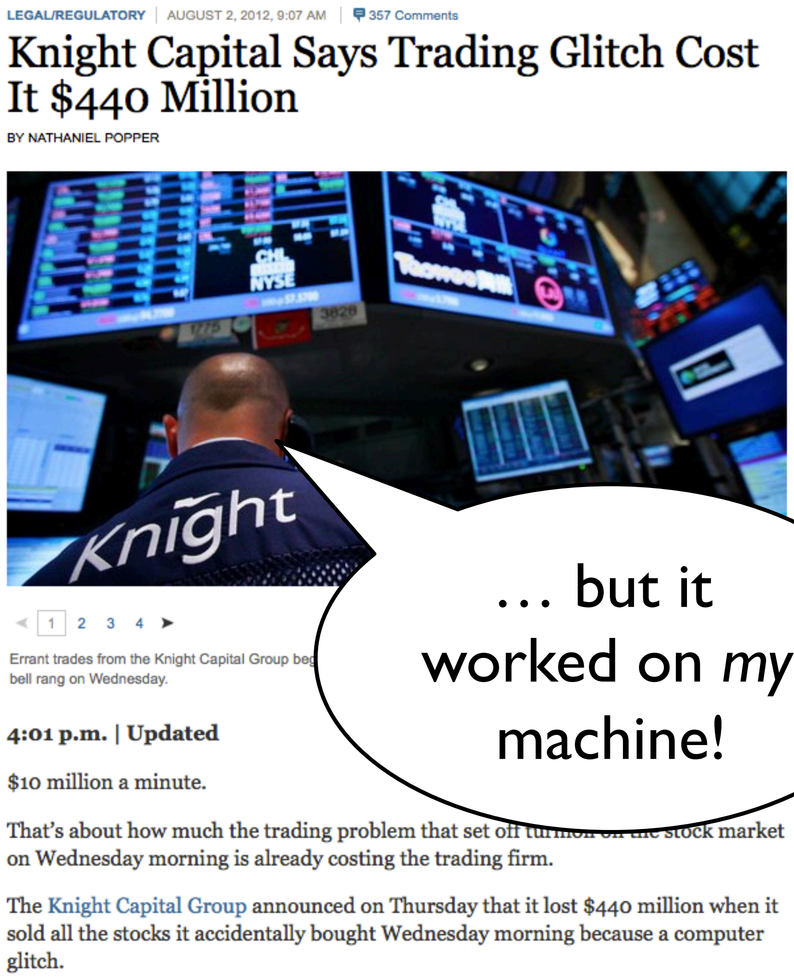
The Knight Capital Group announced on Thursday that it lost $440 million when it sold all the stocks it accidentally bought Wednesday morning because a computer glitch.

NYT, 2 August 2012



Source: Google Finance

Meanwhile on Wall Street…



LEGAL/REGULATORY | AUGUST 2, 2012, 9:07 AM | 357 Comments

Knight Capital Says Trading Glitch Cost It $440 Million

BY NATHANIEL POPPER

◄ 1 2 3 4 ►

Errant trades from the Knight Capital Group beg
bell rang on Wednesday.

4:01 p.m. | Updated

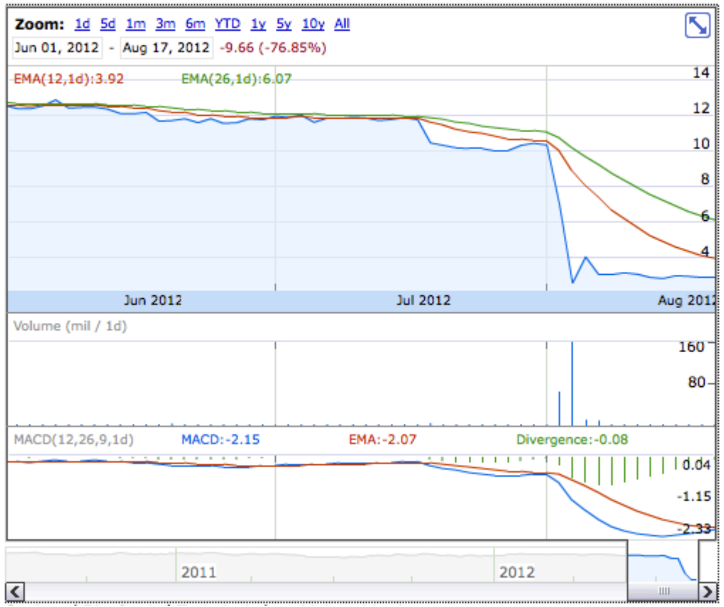
$10 million a minute.

That's about how much the trading problem that set off turmoil on the stock market on Wednesday morning is already costing the trading firm.

The Knight Capital Group announced on Thursday that it lost $440 million when it sold all the stocks it accidentally bought Wednesday morning because a computer glitch.

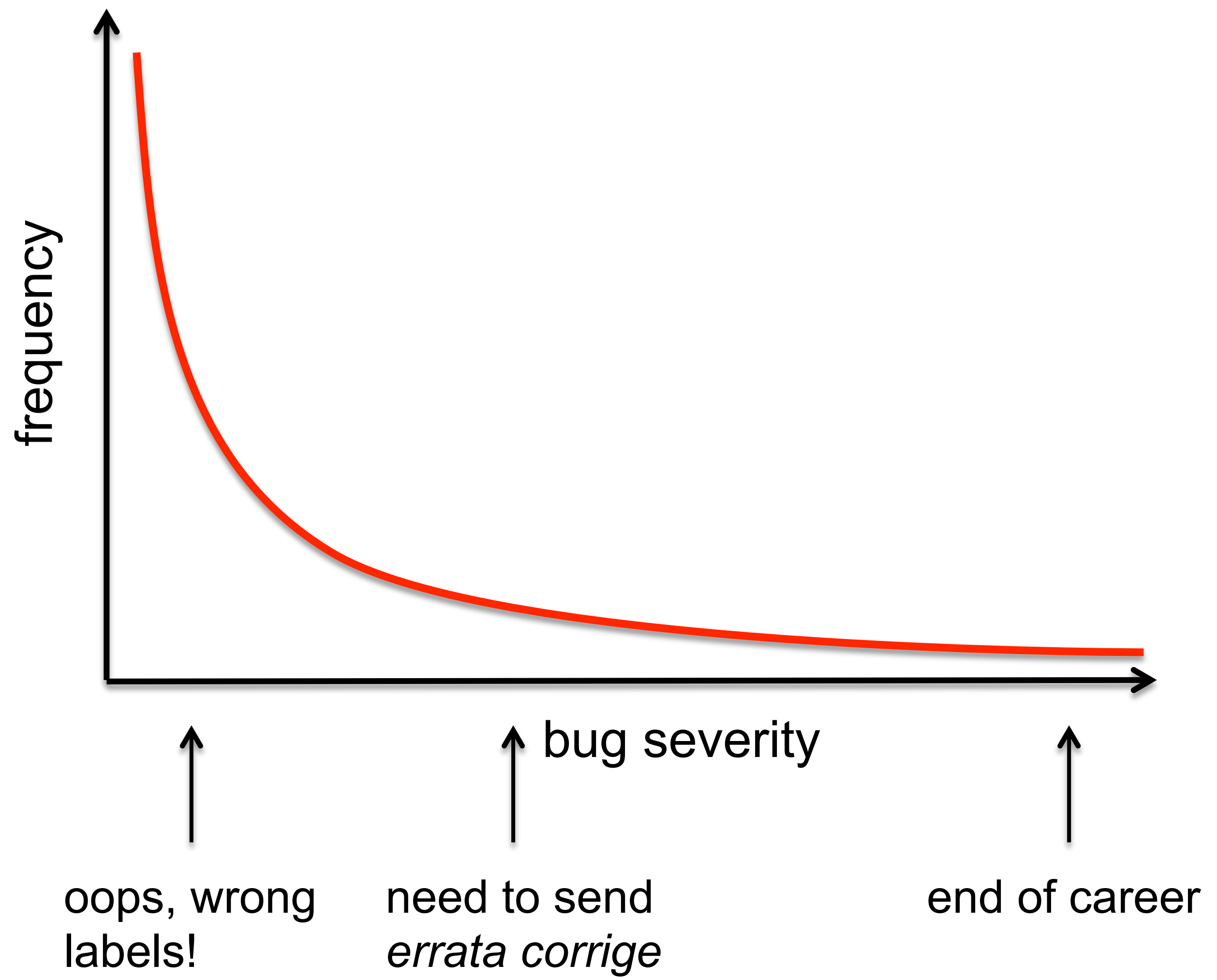
… but it worked on *my* machine!

NYT, 2 August 2012



Source: Google Finance

Effect of software bugs in science



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_solution, 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
```

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.

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=1e-1)    => pass
assert isclose(1.121, 1.2, abs_tol=1e-2)    => fail
```

- `rel_tol` **controls the relative tolerance:**

```
assert isclose(120.1, 121.4, rel_tol=1e-1) => pass
assert isclose(120.4, 121.4, rel_tol=1e-2) => 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

```python
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, 55 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!

- In `voting` , there is an empty function, `labels_frequency`. Write a test for it, then an implementation.

```
def labels_frequency(annotations, nclasses):
    """Compute the total frequency of labels in observed annotations.

    Example:
    >>> labels_frequency([[1, 1, 2], [-1, 1, 2]], 4)
    array([ 0. ,  0.6,  0.4,  0. ])

    Arguments
    ---------
    annotations : array-like object, shape = (n_items, n_annotators)
        annotations[i,j] is the annotation made by annotator j on item i
    nclasses : int
        Number of label classes in `annotations`

    Returns
    -------
    freq : ndarray, shape = (n_classes, )
        freq[k] is the frequency of elements of class k in `annotations`, i.e.
        their count over the number of total of observed (non-missing) elements
    """
```

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!

- Have a look at the docstring of `labels_count`:
  It raises an error if there are no valid observations, but that's not tested!

- Add a test checking that the function raises an error if:

  1) We pass a list of invalid annotations (all missing values)
  2) We pass an empty list of annotations

How to test like a pro

- What does a good test looks like?
- What should I test?
- Anything specific to scientific code?

- At first, testing is awkward:
  1) Where do I begin?
  2) What do I write in the test?
  3) It’s too much effort, it’s slowing me down!

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
```

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

Hands-on!

- Write two tests for the function numpy.var :
  1) First, a deterministic test
  2) Then, a numerical fuzzing test

Numerical fuzzing – solution

```
def test_var_deterministic():
    x = numpy.array([-2.0, 2.0])
    expected = 4.0
    assert isclose(numpy.var(x), expected)


def test_var_fuzzing():
    rand_state = numpy.random.RandomState(8393)

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

    # Generate random, D-dimensional data
    x = rand_state.randn(N, D) * numpy.sqrt(expected)
    variance = numpy.var(x, axis=0)
    numpy.testing.assert_allclose(variance, expected, rtol=1e-2)
```

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?

No safety net!

- Testing of contributed code

Hands-on!

- Write a test for your `find_maxima` function
- Correct the function if the function was incorrect, or clean it up if it wasn't
- Run the test again and watch it pass

Testing is good for your self-esteem

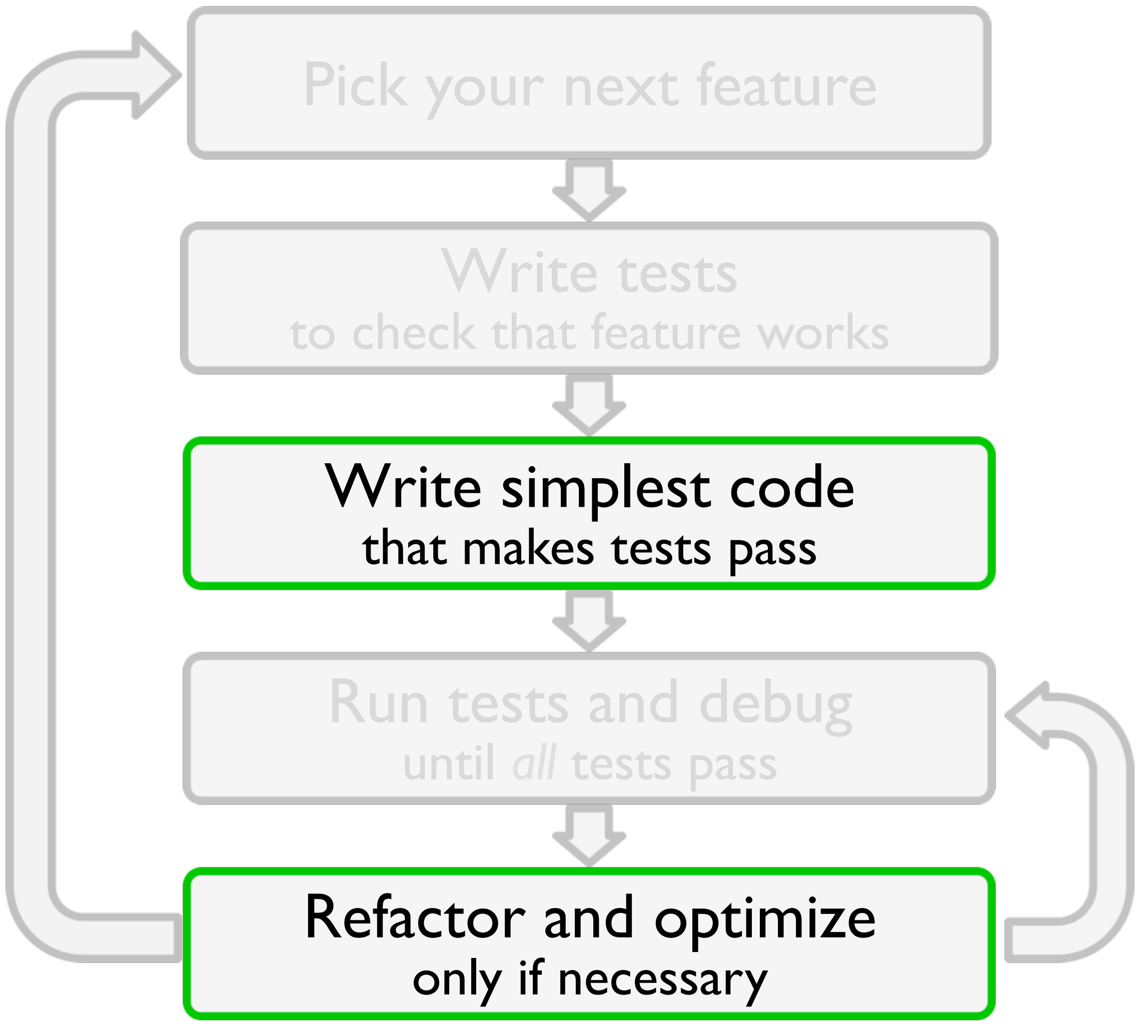
- Immediately: Always be confident that your results are correct, whether your approach works of not
- In the future: save your future self some trouble!
  - Example: `mdp.utils.routine.permute`

Optimization and profiling

Next up: Testing makes you efficient, too!

- Confidence:
  - Tests make you **trust your code**
  - **Correctness** is main requirement for scientific code
  - You must have a strategy to ensure correctness
- Efficiency:
  - An additional big bonus of testing is that your code is ready for improvements
  - Code can change, and correctness is assured by tests
  - **Happily scale your code up!**

The agile development cycle



`unittest`

`pdb`

```
timeit
cProfile
line_profiler
```

Be careful with optimization

- Python is slower than C, but not prohibitively so
- In scientific applications, this difference is often not noticeable: the costly parts of `numpy`, `scipy`, … are written in C or Fortran
- In many cases, scientist time, not computer time is the bottleneck
  - Researchers need to be able to explore many different ideas
  - Always weight the time you spend on a task vs benefits
  - Keep this diagram around: https://xkcd.com/1205/

Optimization methods hierarchy

- (This is mildly controversial)
- In order of preference:
  - Don't do anything
  - Vectorize your code using numpy
  - Use a "magic optimization" tool, like numexpr, or numba
  - Spend some money on better hardware (faster machine, SSD), optimized libraries (e.g., Intel's MKL)
  - Use Cython
  - Use GPU acceleration ⬆
  - Parallelize your code ⬇

How to optimize

- Usually, a small percentage of your code takes up most of the time

1. Identify time-consuming parts of the code
   Where's the bottleneck? Computations? Disk I/O?
   Memory I/O? (see also Francesc's class later this week)
   Use a profiler!
2. Only optimize those parts of the code
3. Keep running the tests to make sure that code is not broken

- Stop optimizing as soon as possible

Measuring time: `timeit`

- IPython magic command: `%timeit`
- Precise timing of a function/expression
- Test different versions of a small amount of code, often used in interactive Python shell

```
In [6]: %timeit cube(123)
10000000 loops, best of 3: 185 ns per loop
```

Hands-on!

- Write a dot product function in pure Python and time it in IPython using `%timeit`:

  dot_product(x, y) is
  x[1] * y[1] + x[2] * y[2] + … + x[N] * y[N]

- Write a version using numpy (vectorized), time it again
- Time numpy.dot
- Try with large (1000 elements) and small vectors (5 elements)

factorial

Follow with me while we profile the file
`hands_on/factorial/factorial.py`

Measuring time: `time`

- On *nix systems, the command `time` gives a quick way of measuring time:

```
$ time python your_script.py

real    0m0.135s
user    0m0.125s
sys     0m0.009s
```

- "real" is wall clock time
- "user" is CPU time executing the script
- "sys" is CPU time spent in system calls

`cProfile`

- standard Python module to profile an entire application (`profile` is an old, slow profiling module)
- Running the profiler from command line:

```
python -m cProfile -s cumulative myscript.py
```

- Sorting options:

`tottime`	:	time spent in function only
`cumtime`	:	time spent in function and sub-calls
`calls`	:	number of calls

cProfile

- Or save results to disk for later inspection:

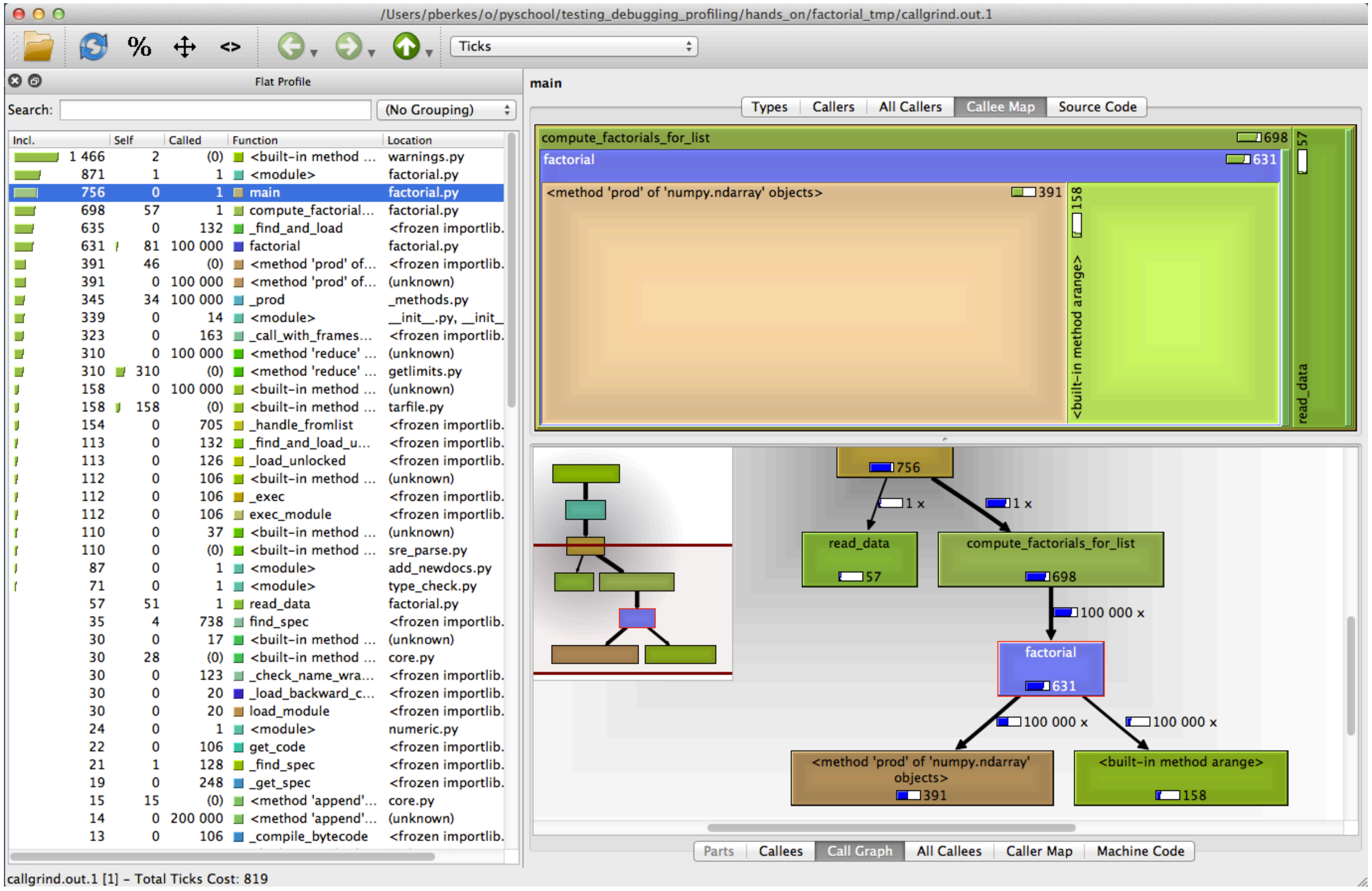
```
python -m cProfile -o filename.prof myscript.py
```

- Explore with

```
python -m pstats filename.prof
```

`stats [n | regexp]:` print statistics
`sort [cumulative, time, ...] :` change sort order
`callers [n | regexp]:` show callers of functions
`callees [n | regexp]:` show callees of functions

Callgrind



/Users/pberkes/o/pyschool/testing_debugging_profiling/hands_on/factorial_tmp/callgrind.out.1

Ticks

Flat Profile

Search: (No Grouping)

Incl.	Self	Called	Function	Location
1 466	2	(0)	<built-in method ...	warnings.py
871	1	1	<module>	factorial.py
756	0	1	main	factorial.py
698	57	1	compute_factorial...	factorial.py
635	0	132	_find_and_load	<frozen importlib.
631	81	100 000	factorial	factorial.py
391	46	(0)	<method 'prod' of...	<frozen importlib.
391	0	100 000	<method 'prod' of...	(unknown)
345	34	100 000	_prod	_methods.py
339	0	14	<module>	__init__.py, __init_
323	0	163	_call_with_frames...	<frozen importlib.
310	0	100 000	<method 'reduce' ...	(unknown)
310	310	(0)	<method 'reduce' ...	getlimits.py
158	0	100 000	<built-in method ...	(unknown)
158	158	(0)	<built-in method ...	tarfile.py
154	0	705	_handle_fromlist	<frozen importlib.
113	0	132	_find_and_load_u...	<frozen importlib.
113	0	126	_load_unlocked	<frozen importlib.
112	0	106	<built-in method ...	(unknown)
112	0	106	_exec	<frozen importlib.
112	0	106	exec_module	<frozen importlib.
110	0	37	<built-in method ...	(unknown)
110	0	(0)	<built-in method ...	sre_parse.py
87	0	1	<module>	add_newdocs.py
71	0	1	<module>	type_check.py
57	51	1	read_data	factorial.py
35	4	738	find_spec	<frozen importlib.
30	0	17	<built-in method ...	(unknown)
30	28	(0)	<built-in method ...	core.py
30	0	123	_check_name_wra...	<frozen importlib.
30	0	20	_load_backward_c...	<frozen importlib.
30	0	20	load_module	<frozen importlib.
24	0	1	<module>	numeric.py
22	0	106	get_code	<frozen importlib.
21	1	128	_find_spec	<frozen importlib.
19	0	248	_get_spec	<frozen importlib.
15	15	(0)	<method 'append'...	core.py
14	0	200 000	<method 'append'...	(unknown)
13	0	106	_compile_bytecode	<frozen importlib.

callgrind.out.1 [1] – Total Ticks Cost: 819

Using callgrind

Callgrind gives graphical representation of profiling results:

- Run profiler:

```
python -m cProfile -o factorial.prof factorial.py
```

- Transform results in callgrind format:

```
pyprof2calltree -i factorial.prof -o callgrind.out.1
```

- Run callgrind:

```
qcallgrind callgrind.out.1
```

or

```
kcachegrind callgrind.out.1
```

Hands-on

- Make sure you can profile and run cachegrind
- Optimize the `factorial` funciton
  - Modify the code
  - Run tests to make sure it still works
  - Profile and measure progress

Fine-grained profiling: kernprof

- You can profile a subset of all functions by decorating them with `@profile`

  ```
  kernprof -b -v factorial.py
  ```

- Line-by-line profiling

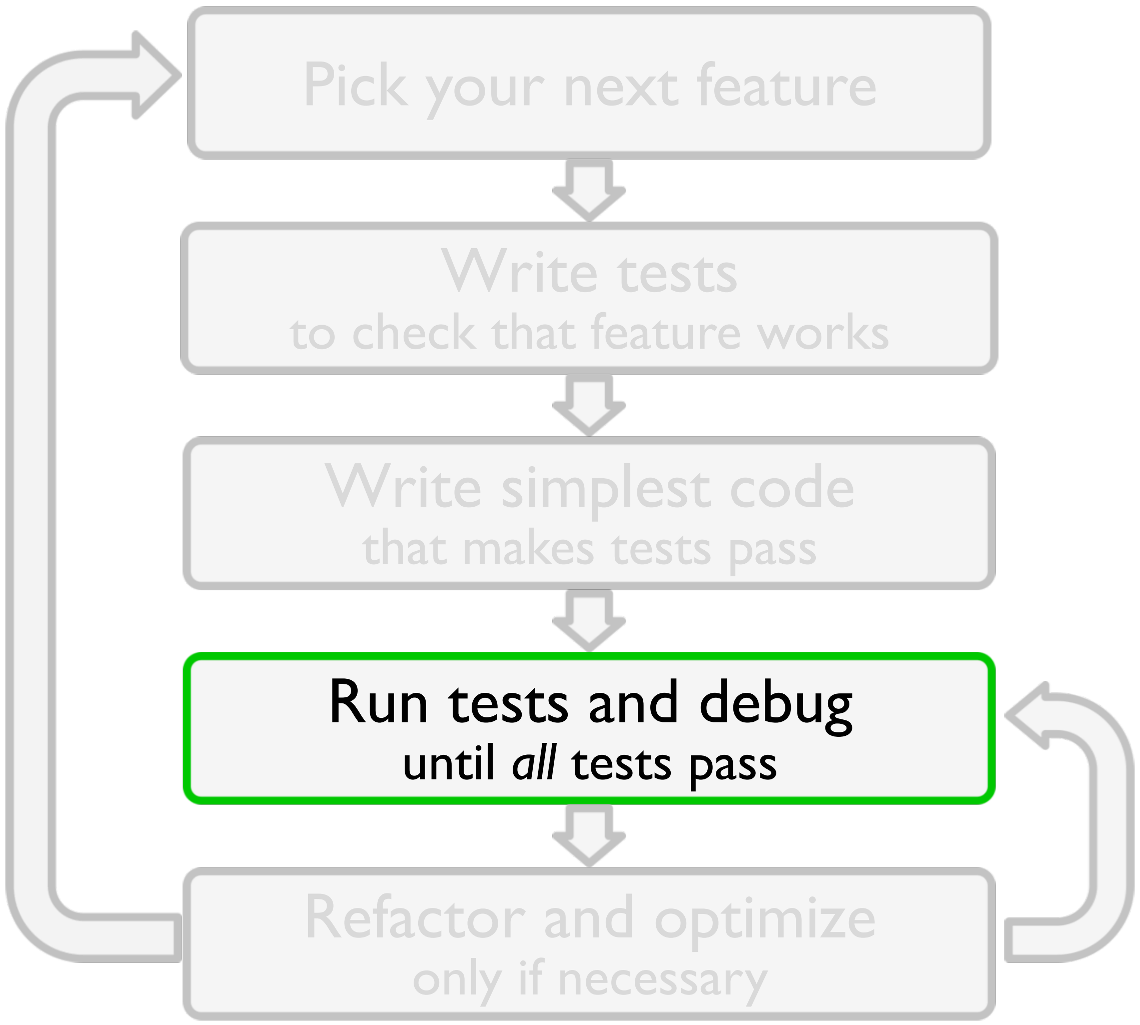
  ```
  kernprof -b -l -v factorial.py
  ```

No safety net!

- Optimization of contributed code

Debugging

The agile development cycle



`unittest`

`pdb`

`timeit`
`cProfile`
`line_profiler`

Debugging

- The best way to debug is to avoid bugs
  - By writing tests, you *anticipate* the bugs
- Your test cases should already exclude a big portion of the possible causes
- Core idea in debugging: you can stop the execution of your application at the bug, look at the state of the variables, and execute the code step by step
- Avoid littering your code with *print* statements

`pdb`, the Python debugger

- Command-line based debugger
- `pdb` opens an interactive shell, in which one can interact with the code
  - examine and change value of variables
  - execute code line by line
  - set up breakpoints
  - examine calls stack

debugger

Entering the debugger

- Enter debugger at the start of a file:

```
python -m pdb myscript.py
```

- Enter at a specific point in the code (alternative to `print`):

```
# some code here
# the debugger starts here
import pdb;
pdb.set_trace()
# rest of the code
```

- If you have it installed, use ipdb instead:

```
import ipdb;
ipdb.set_trace()
```

Entering the debugger from ipython

- From ipython:

  `%pdb` – preventive

  `%debug` – post-mortem

Static checking

One of the problems with debugging in Python is that most bugs only appear when the code executes.

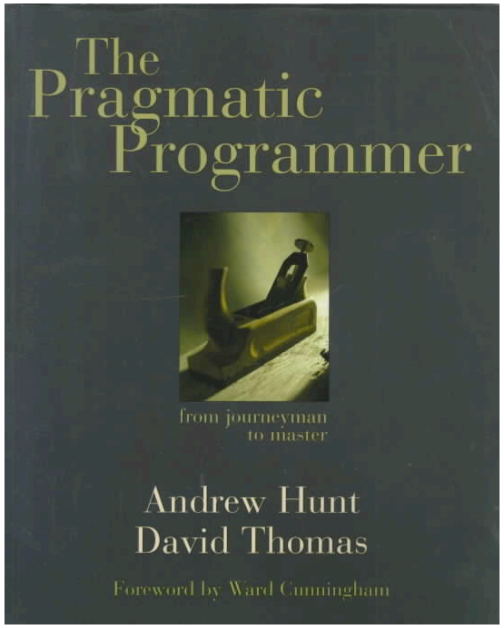
"Static checking" tools analyze the code without executing it.

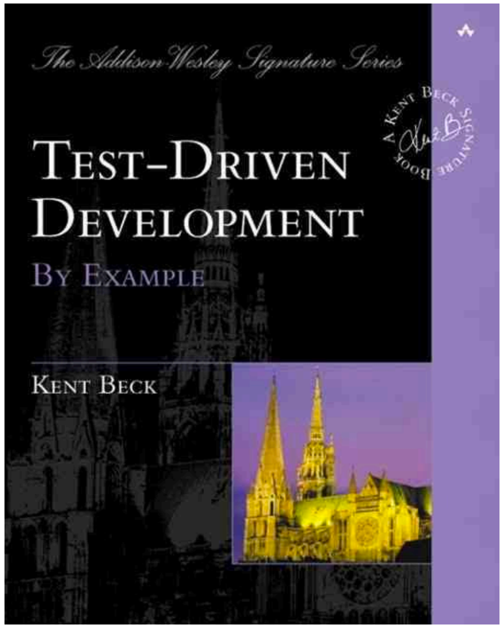
- `pep8`: check that the style of the files is compatible with PEP8
- `pyflakes`: look for errors like defined but unused variables, undefined names, etc.
- `flake8`: pep8 and pyflakes in a single, handy command

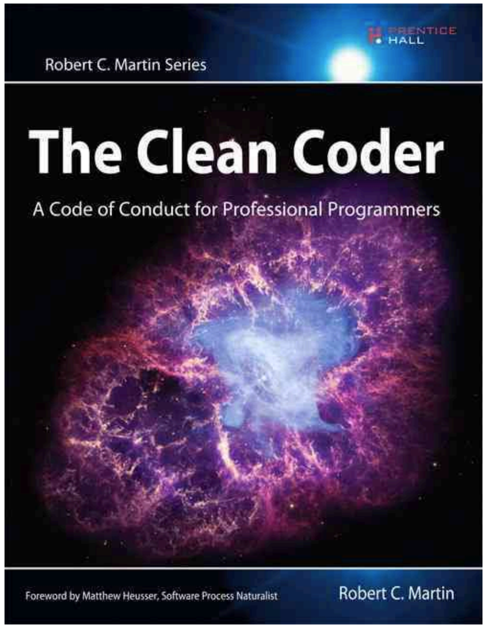
Hands-on!

- Run flake8 on the pyanno package.

Recommended readings




The Addison-Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck


Prentice Hall
Robert C. Martin Series
The Clean Coder
A Code of Conduct for Professional Programmers
Foreword by Matthew Heusser, Software Process Naturalist
Robert C. Martin

Final thoughts

- Good programming practices, with testing in the front line, will help you becoming confident about your results, and efficient at navigating your research project

- For maximum efficiency, check out how these tools can be integrated with your editor / IDE

The End

- Thank you!

Exercises