Testing, debugging, profiling

Python tools for building software

Pietro Berkes, Enthought UK

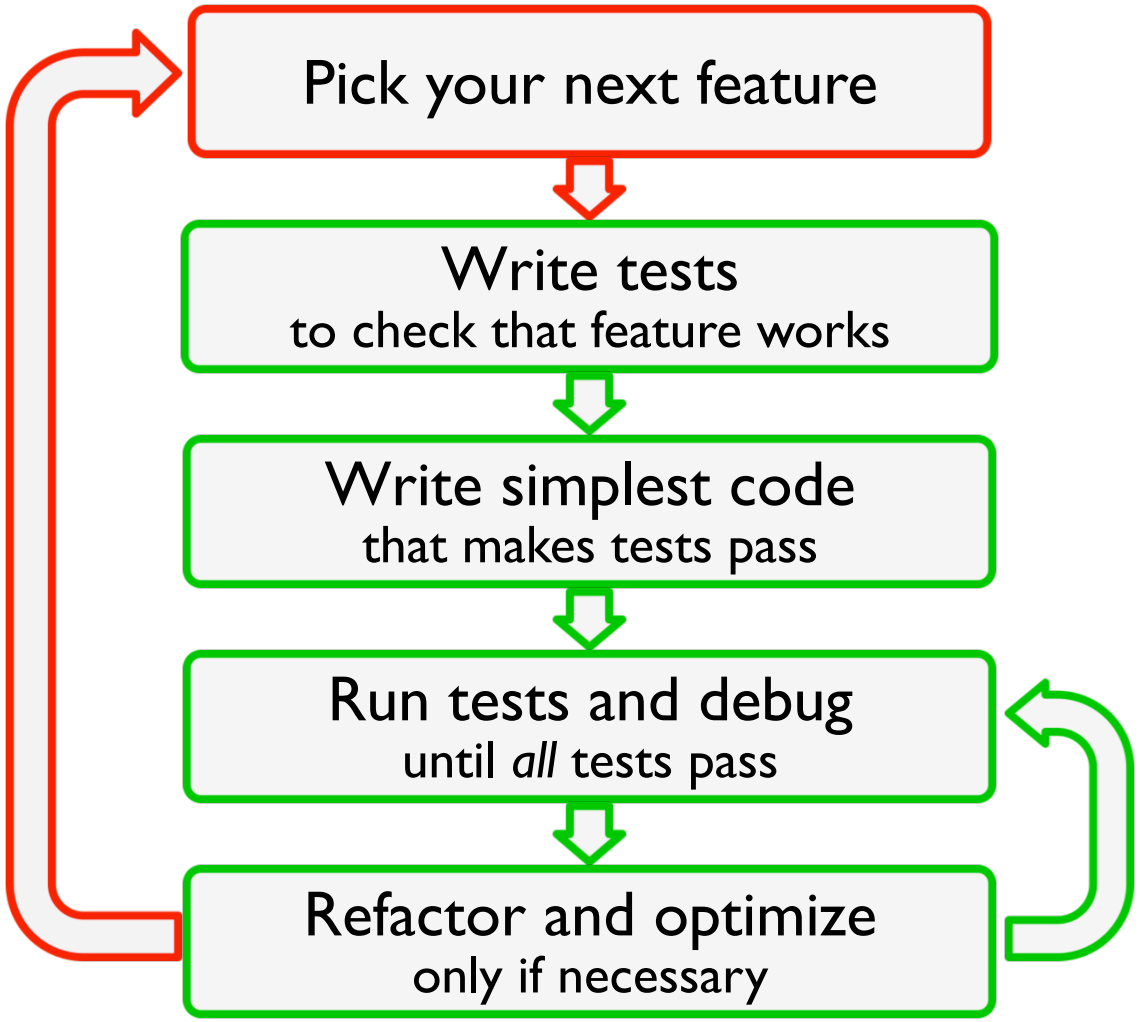
Outline

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

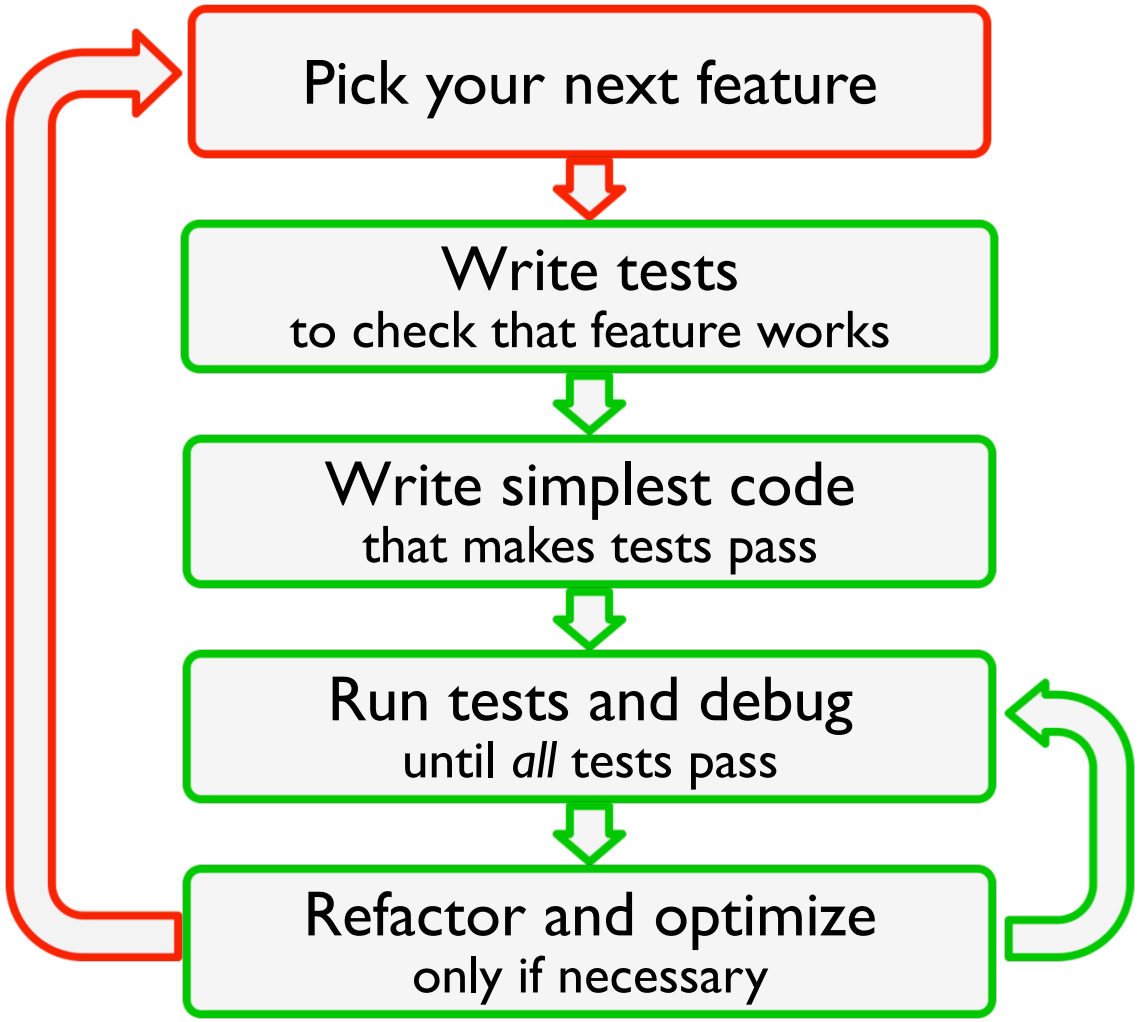
Before we start

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

The agile development cycle



Python tools for agile development



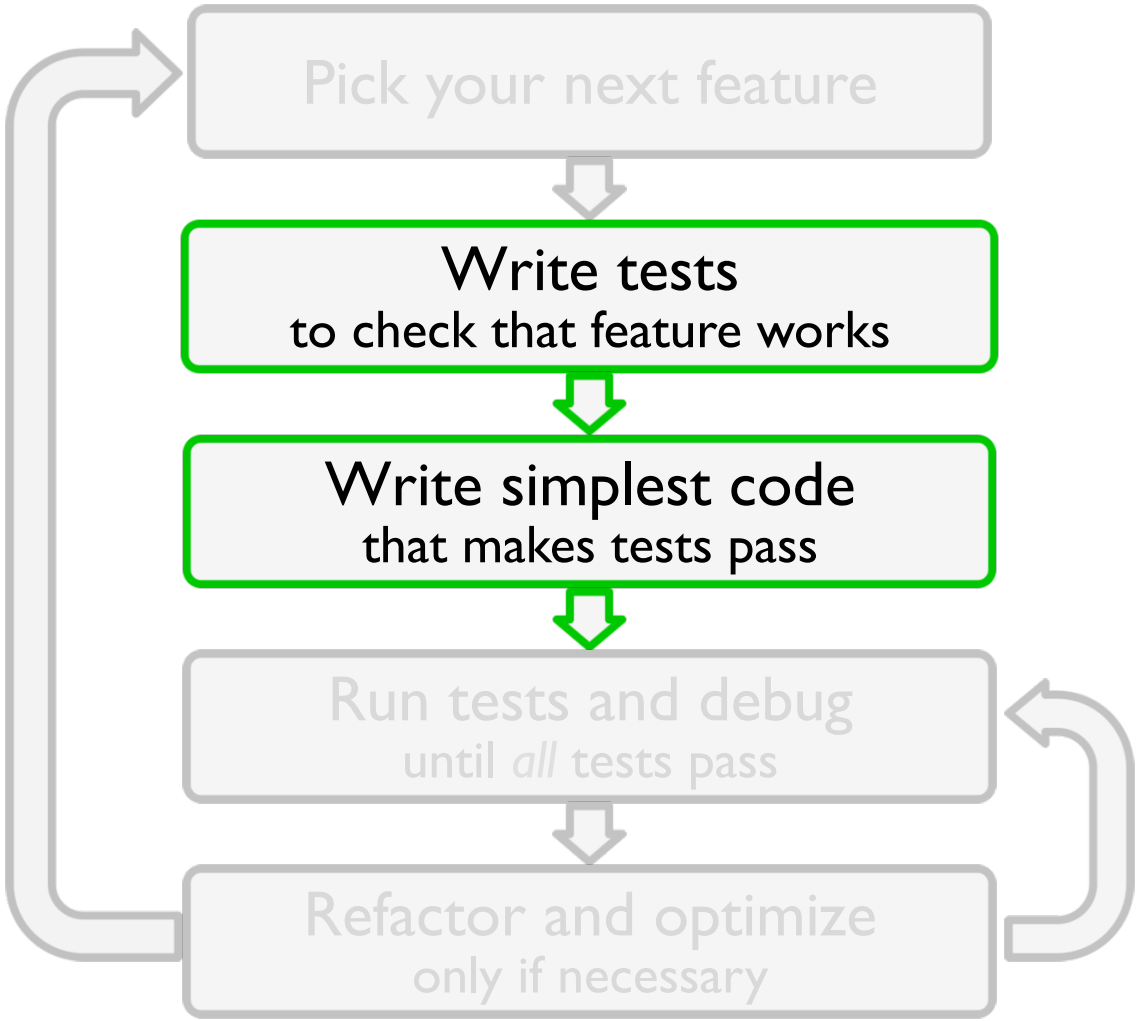
`unittest`

`pdb`

`timeit`
`cProfile`
`line_profiler`

Testing scientific code

The agile development cycle



`unittest`

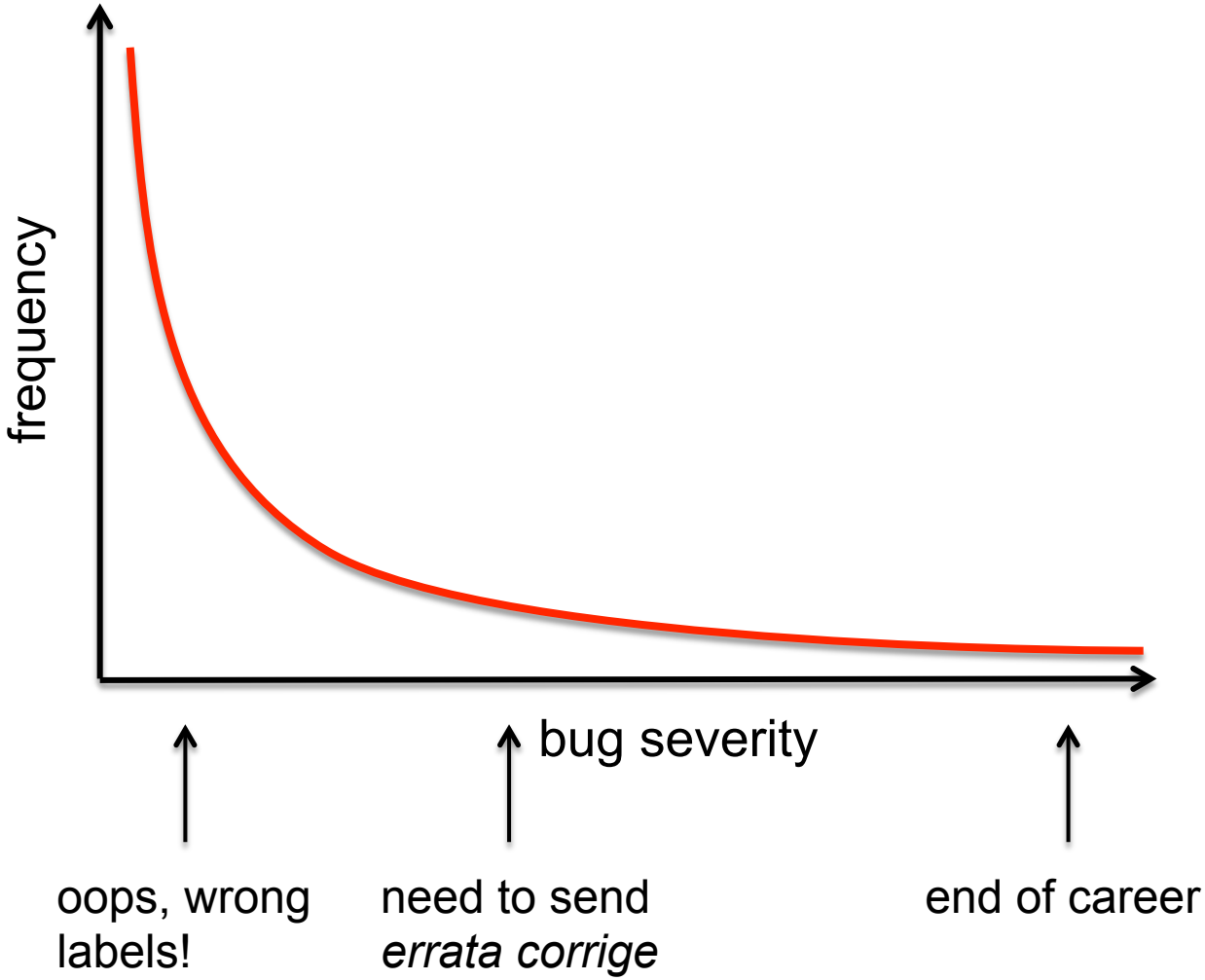
`pdb`

`timeit`
`cProfile`
`line_profiler`

Why write tests?

- Correctness:
  - Main requirement for scientific code
  - You must have a strategy to ensure correctness
  - Tests are the best way to trust your code

Effect of software bugs in science



Software bugs in research are a serious business

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.

PLoS Comp Bio, July 2007: retraction because "As a result of a bug in the Perl script used to compare estimated trees with true trees, the clade confidence measures were sometimes associated with the incorrect clades."

Retraction: Measures of Clade Confidence Do Not Correlate with Accuracy of Phylogenetic Trees

Barry G. Hall, Stephen J. Salipante

In *PLoS Computational Biology*, volume 3, issue 3, doi:10.1371/journal.pcbi.0030051:

As a result of a bug in the Perl script used to compare estimated trees with true trees, the clade confidence measures were sometimes associated with the incorrect clades. The error was detected by the sharp eye of Professor Sarah P. Otto of the University of British Columbia. She noticed a discrepancy between the example tree in Figure 1B and the results reported for the gene *nuoK* in Table 1, and requested that she be sent all ten *nuoK* Bayesian trees. She painstakingly did a manual comparison of those trees with the true trees, concluded that for that dataset there was a strong correlation between clade confidence and the probability of a clade being true, and suggested the possibility of a bug in the Perl script. Dr. Otto put in considerable effort, and we want to acknowledge the generosity of that effort.

Not only in academia…

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.

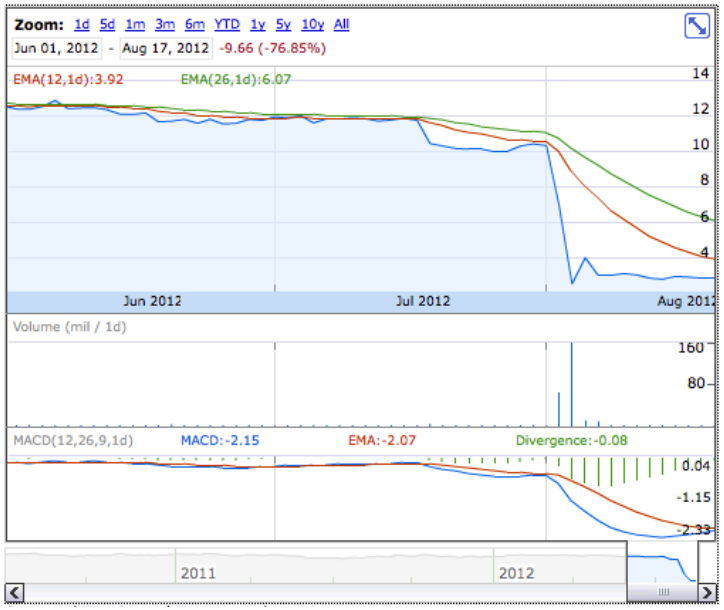
4:01 p.m. | Updated

$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

Not only in academia…

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

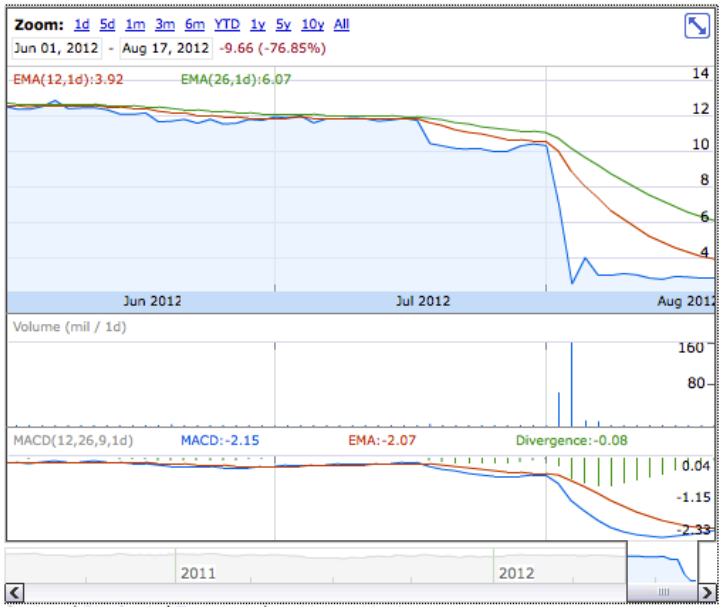
$10 million a minute.

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

Most recent software failure

Software upgrade grounds hundreds of flights over US east coast

FAA says 'technical issues' with an air traffic control computer undergoing a software update caused 492 flight delays and 476 cancellations over weekend

Thousands of passengers were stuck at airports with delayed or cancelled flights after technical issues with an air traffic control system. Photograph: Liu Shuai/Xinhua Press/Corbis

The Guardian, 17 August 2015

Most recent software failure

Software upgrade grounds hundreds of flights over US east coast

FAA says 'technical issues' with an air traffic control computer undergoing a software update caused 492 flight delays and 476 cancellations over weekend

Thousands of passengers were st… …ssues with an air traffic control system. Photograph: Liu Shua…

The Guardian, 17 August 2015

Why write tests?

- Correctness:
  - Main requirement for scientific code
  - You must have a strategy to ensure correctness
  - Tests are the best way to trust your code

- Flexibility:
  - Code can change, and correctness is assured by tests
  - Leads to better and faster code

Save your future self some trouble

- **Practical example:** `mdp.utils.routine.permute`

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_choice (__main__.TestSequenceFunctions) ... ok
test_sample (__main__.TestSequenceFunctions) ... ok
test_shuffle (__main__.TestSequenceFunctions) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.110s
OK
```

Hands-on!

- Go to `hands_on_exercises/pyanno_voting`
- Execute the tests:
  `python -m unittest -v discover`

How to run tests

- Option 1: Discover all tests in all subdirectories
  ```
  python -m unittest [-v] discover
  ```
- Option 2: Execute all tests in one module
  ```
  python -m unittest [-v] test.module
  ```
- Option 3: Add this snippet at the end of a test file, and execute the test file.
  ```
  if __name__ == '__main__':
      unittest.main()
  ```
  unittest.main() will execute all tests in all TestCase classes in a file.

Test suites in Python: `unittest`

- Writing tests with unittest is simple enough:
  - Each test case is a subclass of `unittest.TestCase`
  - Each test unit is a method of the class, whose name starts with '`test`'
  - Each unit tests **one** aspect of your code, and checks that it behaves correctly using "assertions". An exception is raised if it does not work as expected.

Anatomy of a TestCase

Create new file, `test_something.py`:

```
import unittest

class FirstTestCase(unittest.TestCase):

    def test_truisms(self):
        """All methods beginning with 'test' are executed"""
        self.assertTrue(True)
        self.assertFalse(False)

    def test_equality(self):
        """Docstrings are printed during executions
        of the tests in some test runners"""
        self.assertEqual(1, 1)

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

`TestCase.assertSomething`

- `TestCase` defines utility methods to check that some conditions are met, and raise an exception otherwise
- Check that statement is true/false:

```
assertTrue('Hi'.islower())          => fail
assertFalse('Hi'.islower())         => pass
```

- Check that two objects are equal:

```
assertEqual(2+1, 3)                 => pass
assertEqual([2]+[1], [2, 1])        => pass
assertNotEqual([2]+[1], [2, 1])     => fail
```

`assertEqual` can be used to compare all sorts of objects: numbers, lists, tuples, dicts, sets, frozensets, and unicode

Hands-on!

- Add a dummy test to test_voting: test that
  one + two == three
- Execute the tests

Hands-on!

- Add a dummy test to test_voting: test that
  one + two == three
- Execute the tests
- Now test that 1.1 + 2.2 == 3.3

Floating point equality

- Floating point numbers are rarely equal. When developing numerical code, we have to allow for machine precision errors.
- Check that two numbers are equal up to a given precision:
  `assertAlmostEqual(x, y, places=7)`
- `places` is the number of decimal places to use:

```
assertAlmostEqual(1.121, 1.12, 2) => pass
assertAlmostEqual(1.121, 1.12, 3) => 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

```
class TestNumpyEquality(unittest.TestCase):
    def test_equality(self):
        x = numpy.array([1, 1])
        y = numpy.array([2, 2])
        z = numpy.array([3, 3])
        self.assertEqual(x + y, z)
```

```
E
======================================================================
ERROR: test_equality (test_numpy_equality.TestNumpyEquality)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/pberkes/o/pyschool/testing_debugging_profiling/hands_on/numpy_equality/
test_numpy_equality.py", line 11, in test_equality
    self.assertEqual(x + y, z)
  File "/envs/gnode/lib/python3.4/unittest/case.py", line 797, in assertEqual
    assertion_func(first, second, msg=msg)
  File "//envs/gnode/lib/python3.4/unittest/case.py", line 787, in _baseAssertEqual
    if not first == second:
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any()
or a.all()

----------------------------------------------------------------------
Ran 1 test in 0.006s

FAILED (errors=1)
```

Testing with numpy arrays

- `numpy.testing` defines appropriate function:

```
numpy.testing.assert_array_equal(x, y)
numpy.testing.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; rtol is relative difference, atol is absolute difference
  - combine with `logical_and, logical_or, logical_not`:

```
# test that all elements of x are between 0 and 1
assertTrue(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:

```
with self.assertRaises(SomeException):
    do_something()
    do_something_else()
```

- For example:

```
with self.assertRaises(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:

```
with self.assertRaises(IOError):
    file(1, 'r')                          => fail
```

as expected, but

```
with self.assertRaises(Exception):
    file(1, 'r')                          => pass
```

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

TestCase.assertSomething

- Many more "assert" methods:
  (complete list at http://docs.python.org/library/unittest.html)

```
assertGreater(a, b) / assertLess(a, b)
```

```
assertRegexpMatches(text, regexp)
```
verifies that regexp search matches text

```
assertIn(value, sequence)
```
assert membership in a container

```
assertIsNone(value)
```
verifies that value is None

```
assertIsInstance(obj, cls)
```
verifies that an object is an instance of a class

```
assertItemsEqual(actual, expected)
```
verifies equality of members, ignores order

```
assertDictContainsSubset(subset, full)
```
tests whether the entries in dictionary full are a superset of those in subset

Hands-on!

- Somebody sent us a new set of annotations… they indicate missing values with -999 :-(
- Modify the existing code so that the function accept an optional missing value, which by default is -1
  - Add a test for `labels_count` and `majority_vote` to exercise the new argument.
  - Keep running *all* tests to make sure you don't break any existing functionality.

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) It's obvious that this code works
  2) The tests are longer than the code
  3) The test code is a duplicate of the real code

Basic structure of test

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

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

```
class LowerTestCase(unittest.TestCase):

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

        # When
        output = string.lower()

        # Then
        self.assertEqual(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(self):
    # Given
    string = ''
    expected = ''

    # When
    output = string.lower()

    # Then
    self.assertEqual(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(self):
    # 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
        self.assertEqual(output, expected)
```

Common testing pattern

- Better still:

```
def test_lower(self):
    # 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:
        with self.subTest(i=string):
            # When
            output = string.lower()
            # Then
            self.assertEqual(output, expected)
```

- See `hands_on/sub_test`

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

```
class TestVar(unittest.TestCase):


        x = numpy.array([-2.0, 2.0])
        expected = 4.0
        self.assertAlmostEqual(numpy.var(x), expected)

    def test_var_fuzzing(self):


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

        # Test multiple times with random data
        for _ in range(20):
            # Generate random, D-dimensional data
            x = rand_state.randn(N, D) * numpy.sqrt(expected)
            variance = numpy.var(x, axis=0)
            numpy.testing.assert_array_almost_equal(variance, expected, 1)
```

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

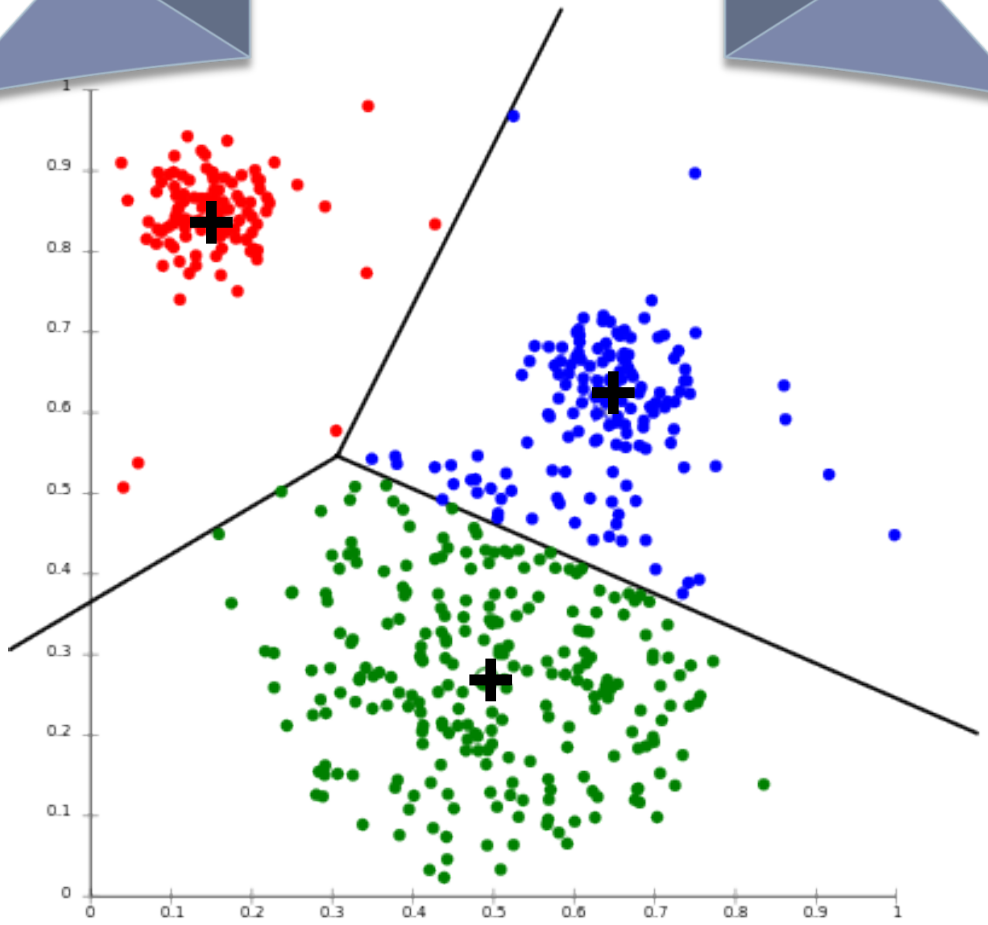
Example: eigenvector decomposition

- 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!

- Anybody offering code for testing and profiling?
- Looking for:
  - Self-contained, relatively small script or library
  - Does not run for too long
  - Does something we can understand
- Make a pull request on the class repository
  - Create branch
  - Commit files in a sub-directory
  - Push to origin
  - Create PR

k-means



Test-driven development (TDD)

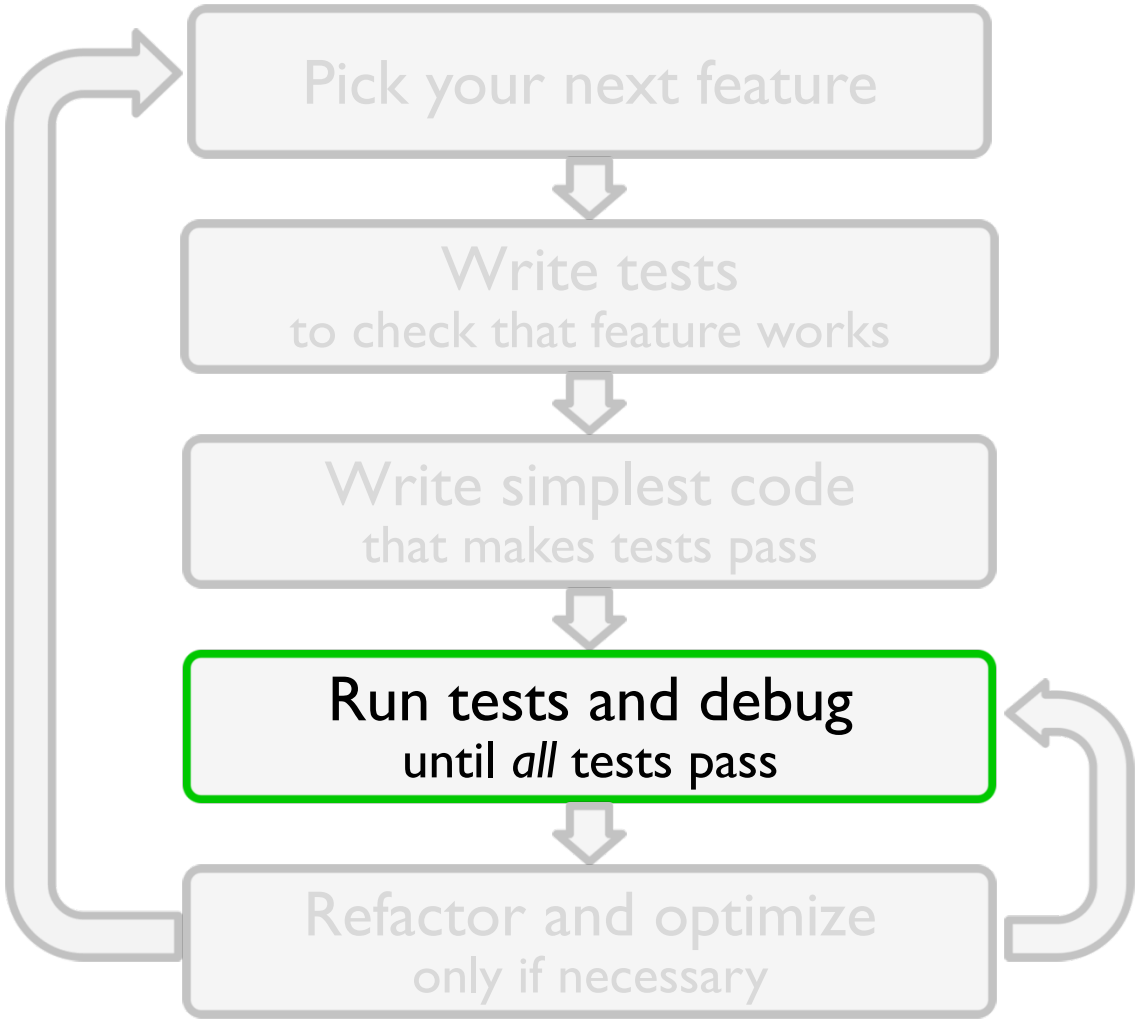
- An influential testing philosophy: write tests *before* writing code
  - Choose what is the next feature you'd like to implement
  - Write a test for that feature
  - Write the simplest code that will make the test pass
- Forces you to think about the design of your code before writing it: how would you like to interact with it?
- The result is code whose features can be tested individually, leading to maintainable, decoupled code
- If the results are bad, then you'll write tests to find a bug. If it works, will you?

Testing: Money back guarantee

- I guarantee that aggressive testing will improve your code and your research, or you'll get the Python school fee back!
- Just remember, code quality is not just testing:
  - "Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often" (Steve McConnell, *Code Complete*)

Debugging

The agile development cycle



`unittest`

`pdb`

`timeit`
`cProfile`
`line_profiler`

Debugging

- The best way to debug is to avoid bugs
  - In TDD, 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.

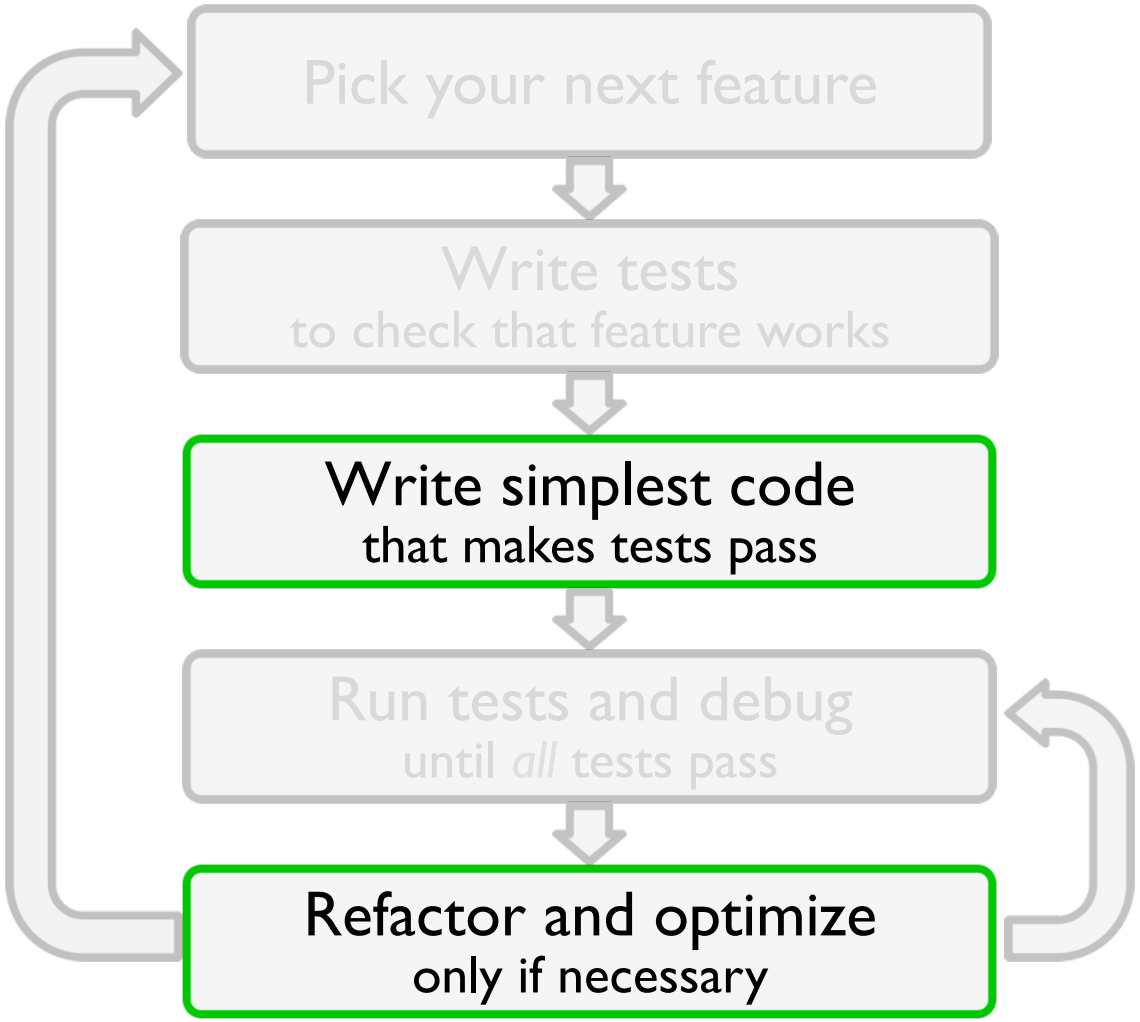
- `pep8`: check that the style of the files is compatible with PEP8
- `pyflakes`: look for errors like defined but unused variables, undefined names, etc.
- `flake8`: pep8 and pyflakes in a single, handy command

Hands-on!

- Run flake8 on the pyanno package.

Optimization and profiling

The agile development cycle



`unittest`

`pdb`

`timeit`
`cProfile`
`line_profiler`

Be careful with optimization

- 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/

Python code 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
- Don't rush into writing optimizations

Optimization methods hierarchy

- (This is mildly controversial)
- In order of preference:
  - Vectorize 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
  - Parallelize your code
  - Use GPU acceleration
- The bottleneck might in memory or disk management, not the CPU (see Francesc Alteds's videos online)

How to optimize

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

1. Identify time-consuming parts of the code
   (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 vectors (1000 elements) and small (5 elements)

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:

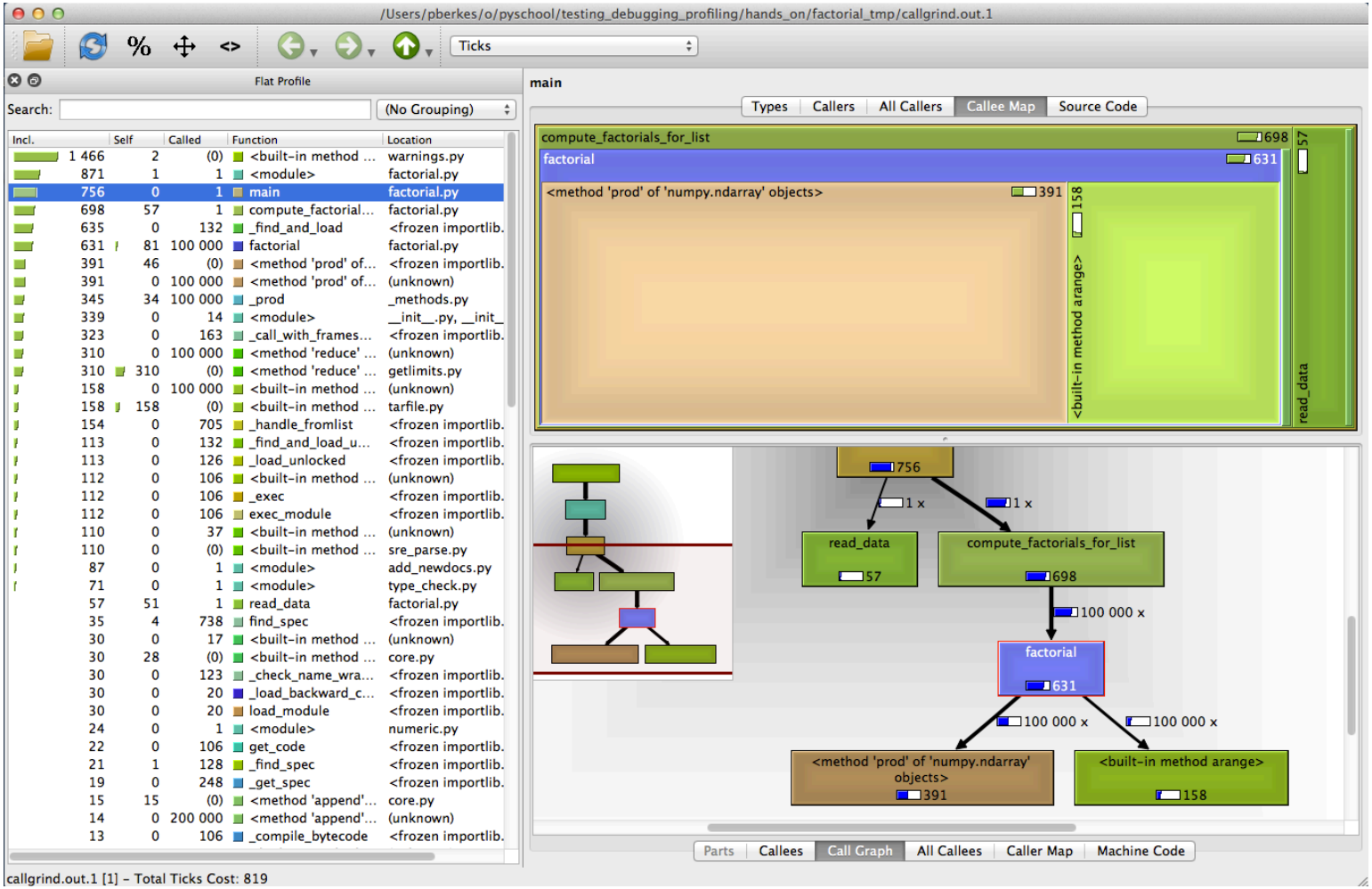
```
python -m cProfile -o filename.prof myscript.py
```

- 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
main
Types Callers All Callers Callee Map Source Code
compute_factorials_for_list 698
factorial 631
<method 'prod' of 'numpy.ndarray' objects> 391
158
<built-in method arange>
57
read_data
756
1 x
1 x
read_data
57
compute_factorials_for_list
698
100 000 x
factorial
631
100 000 x
100 000 x
<method 'prod' of 'numpy.ndarray' objects>
391
<built-in method arange>
158
Parts Callees Call Graph All Callees Caller Map Machine Code

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
  ```

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!

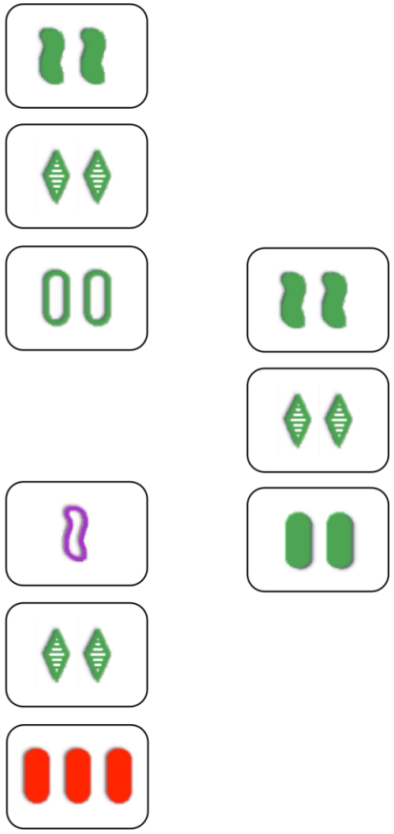
- Optimization of contributed code

Final thoughts

- Scientific code has slightly different needs than "regular" code, most notably the need to ensure correctness
- Agile programming methods, and testing in particular, go a long way toward this goal
- Agile programming in Python is as easy as it gets, there are no excuses not to do it!
- For maximum efficiency, check out how these tools can be integrated with your editor / IDE

The End

- Exercises!



		1						
		2		3				4
			5			6		7
5			1	4				
	7						2	
				7	8			9
8		7			9			
4				6		3		
						5		