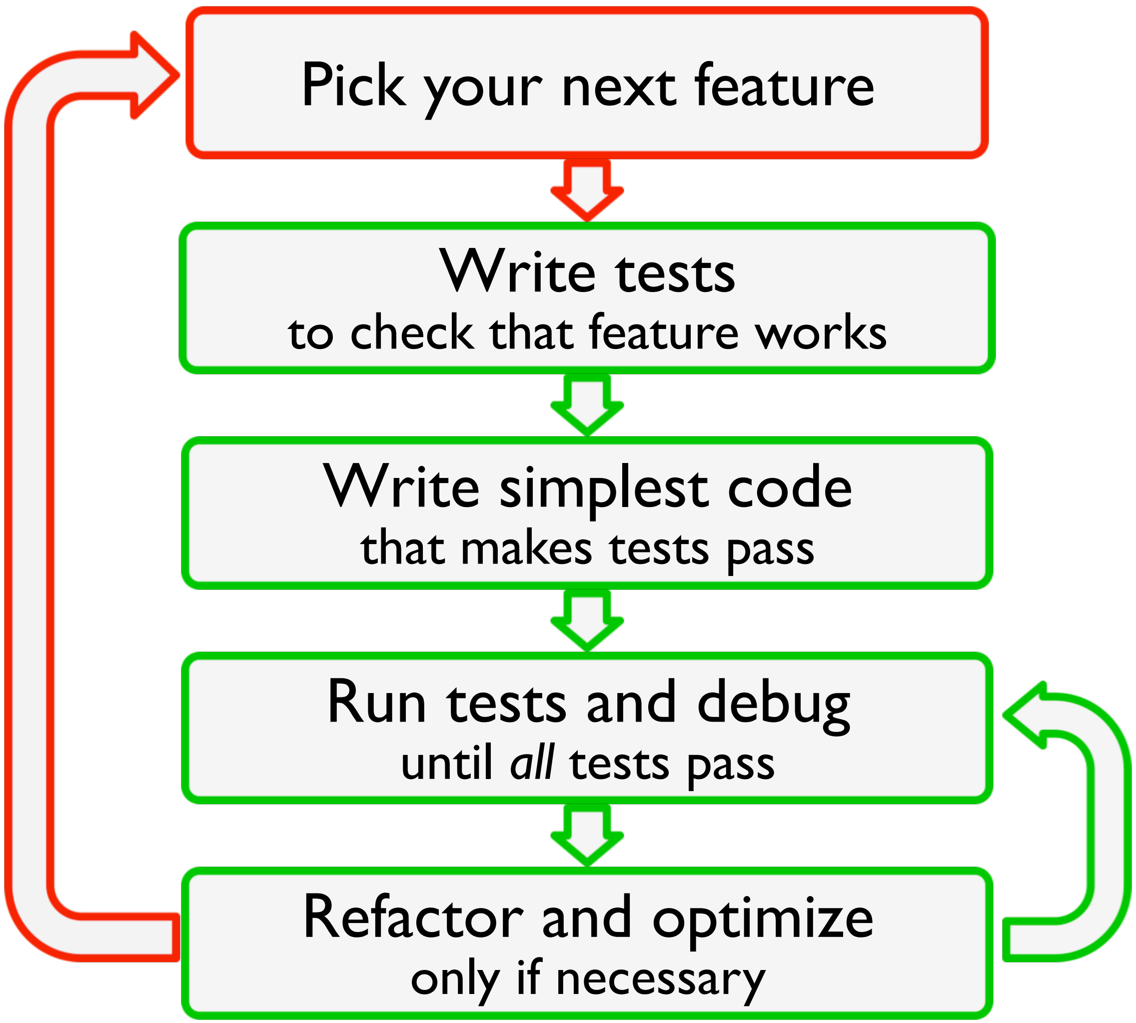
Python tools for writing scientific code

Pietro Berkes, Twitter Cx

Python tools for agile development



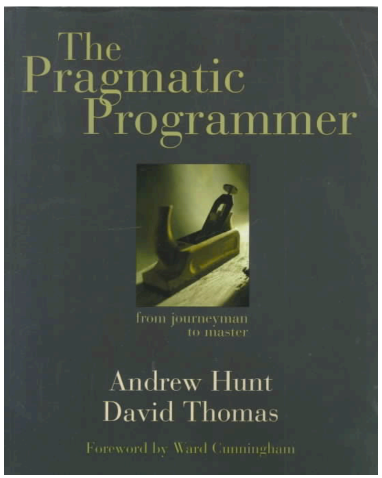
`py.test`

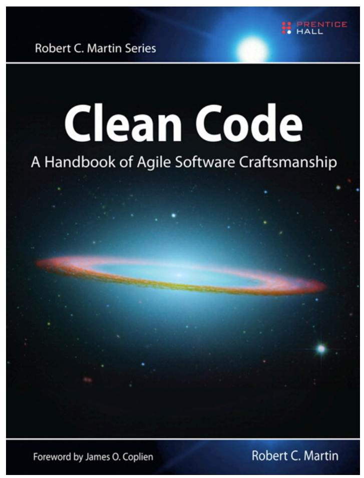
`timeit`
`cProfile`
`callgrind`
`line_profiler`

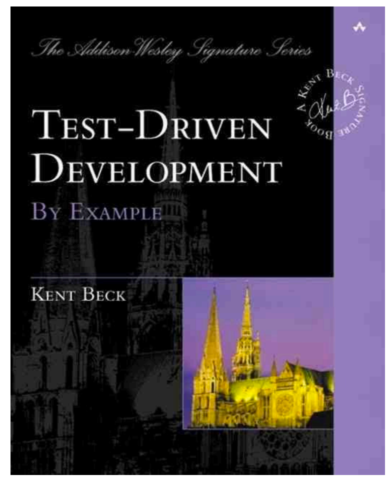
Outline

- Testing scientific code
- Mocking
- Profiling and optimization
- Packaging

Recommended readings





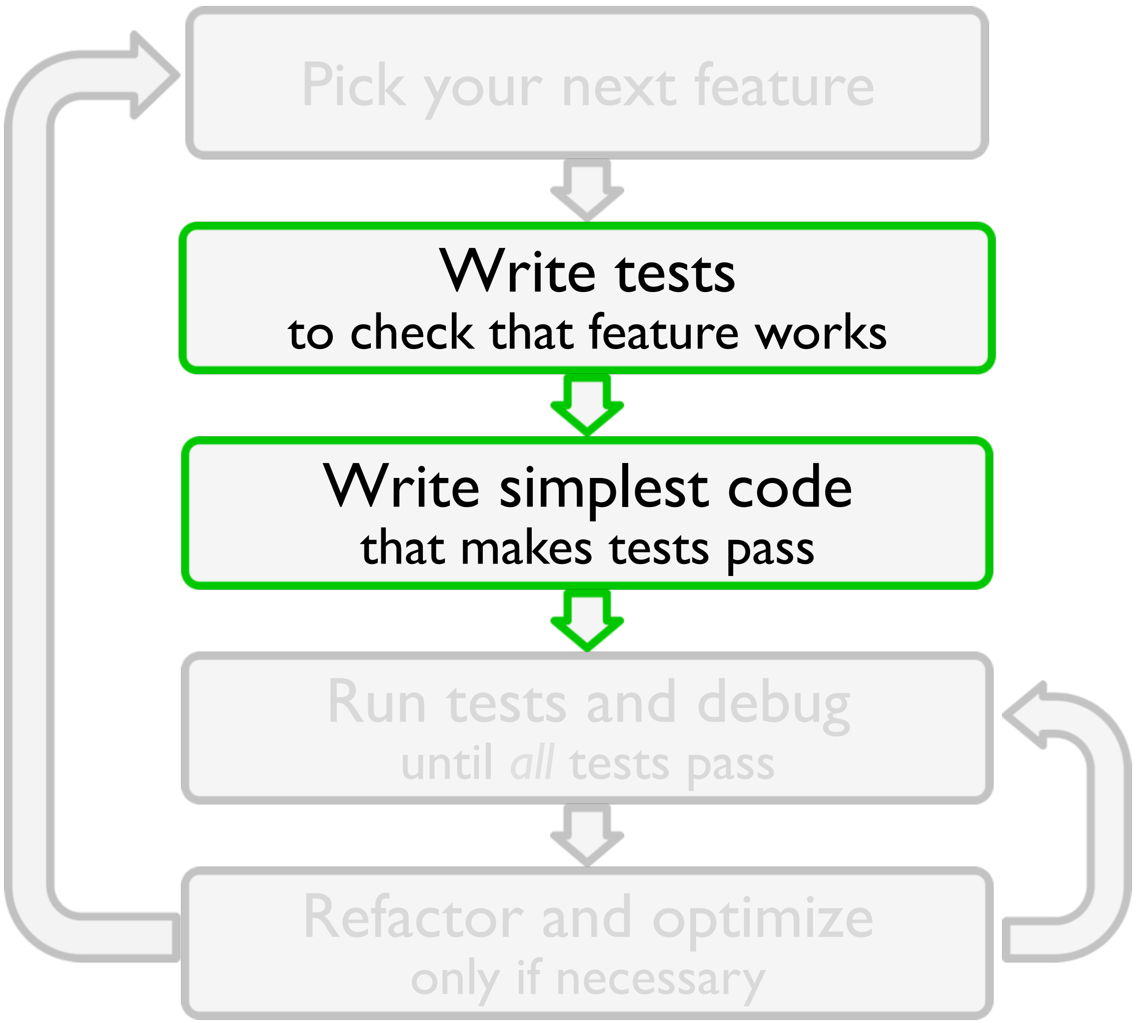


Before we start

- Unzip the materials in your favorite directory

Testing scientific code

The agile development cycle



`py.test`

`pdb`

`timeit`
`cProfile`
`line_profiler`

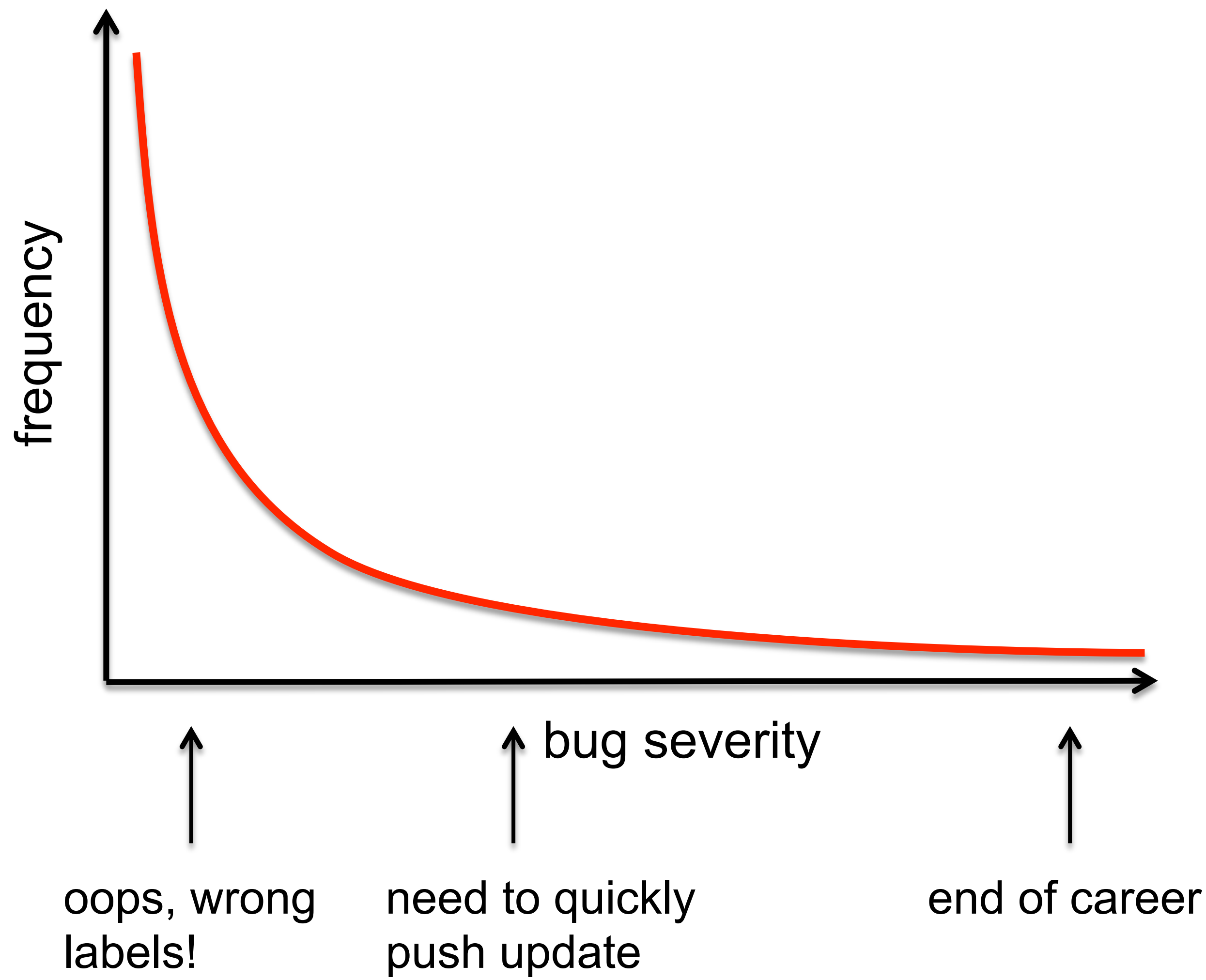
Testing is good for you

- Confidence:
  - Write the code once and use it confidently everywhere else (the negative result effect)
  - **Correctness** is main requirement for scientific code
  - Save your future self some trouble

Testing is good for your team

- Keep your code open for scalability and optimization
  - Team members, new hires, and intern can start contributing immediately
  - Example: Porting a codebase to Python 3 is a matter of half a day, if the code is well exercised by tests

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 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

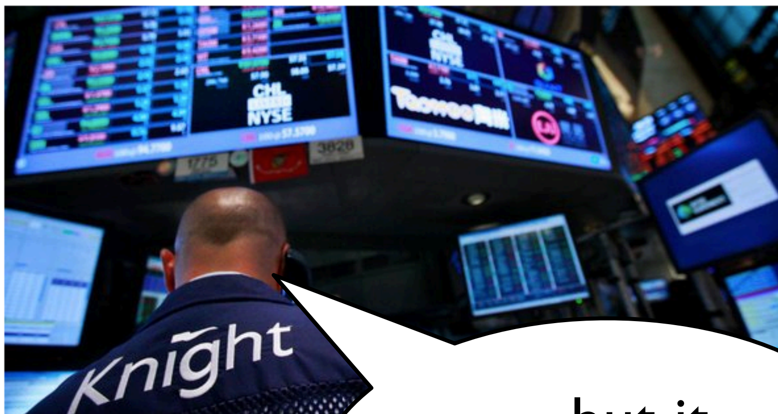
Source: Google Finance

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.

… but it worked on *my* machine!

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

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_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
```

Warm up your fingers!

- **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

```
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!

- Fix the test before: 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])
```

Hands-on! (time permitting)

- 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!

- **Update** `test_image_sample_point` **in** `mpt/face_tracking/test_dataset.py` **to use** `raises`

Testing patterns

What a good test looks like

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

Basic structure of a 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
```

Even better with py.test

- This is better as it shows which test case fails (if any):

```
import pytest

@pytest.mark.parametrize(
    "string, expected",
    [('HeLlO wOrld', 'hello world'),
     ('hi', 'hi'),
     ('123 ([?', '123 ([?'),
     ('', '')])
def test_lower(string, expected):
    output = string.lower()
    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

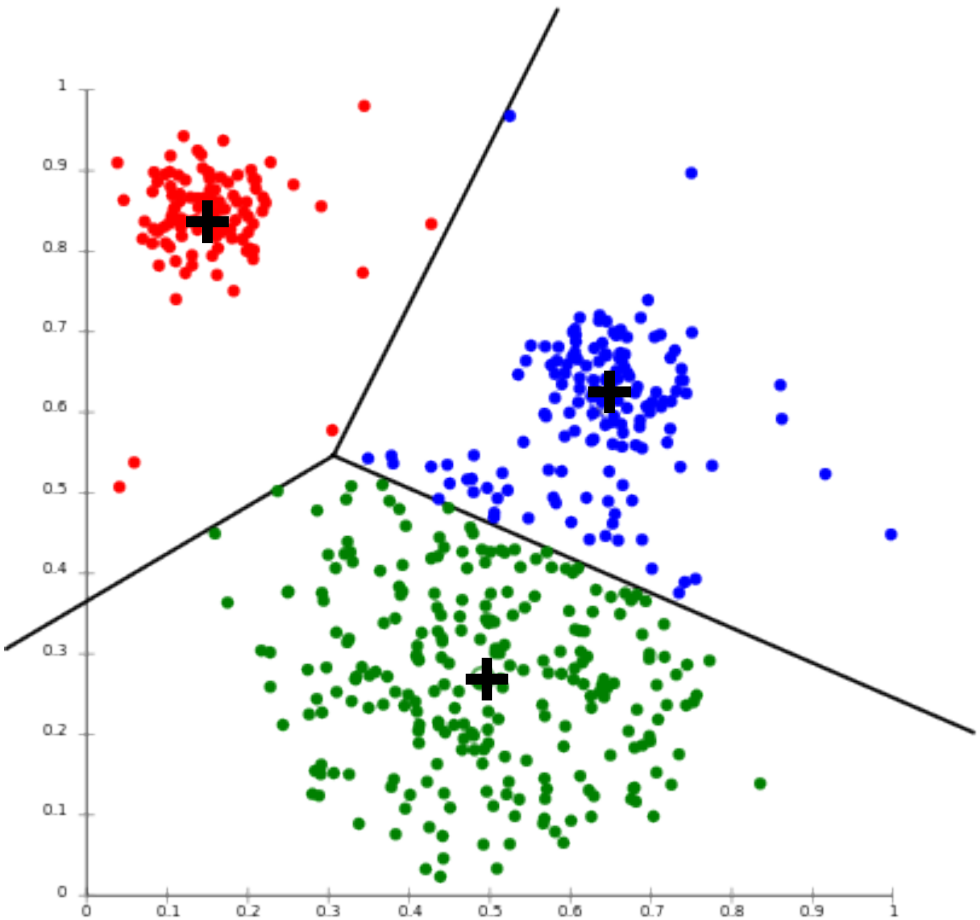
- Test general routines with specific ones
  - Example: test `polynomial_expansion(data, degree)` with `quadratic_expansion(data)`
- Test optimized routines with brute-force approaches
  - Example: test function computing analytical derivative with numerical derivative

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?

Code Kata

- Write k-means implementation using TDD



Mocking

Mock objects for testing

- *Mock object*: object that mimics the behavior of a real object, but doesn't actually do much
- Main reasons to use mocking:
  - Code would have undesired side effect
    - Commit to central database
    - Post to Twitter
    - Take a very long time to complete
  - Results depends on things we don't control
    - E.g. current time, or temperature
- Python3 ships with a `unittest.mock` package, on Python2 you need to `pip install mock`

Example: how do we test this function?

```
report_template = """
Report
======

The experiment was a {judgment}!
Let's do this again, with a bigger budget.
"""

def send_report(result, smtp):
    if result > 0.5:
        judgment = 'big success'
    else:
        judgment = 'total failure'
    report = report_template.format(judgment=judgment)

    smtp.send_message(
        report,
        from_addr='pony@magicpony.com',
        to_addrs=['ferenc@magicpony.com'],
    )
```

The Mock object

- The superstar of the library, `Mock`, absorbs everything you throw at it:

```
>>> from unittest.mock import Mock
>>> mock = Mock()

>>> print mock.x
<Mock name='mock.x' id='24379952'>
>>> mock.x = 3
>>> print mock.x
3

>>> mock.whatever(3, key=2)
<Mock name='mock.whatever()' id='24470128'>
```

Interactions are recorded

```
>>> mock=Mock()
>>> mock.foo(2,3)
>>> mock.foo('a')

>>> mock.foo.called
True
>>> mock.baz.called
False

>>> mock.f.call_args
call('a')
>>> mock.f.call_count
2
>>> mock.f.call_args_list
[call(2, 3), call('a')]
```

Support for testing

```
>>> mock=Mock()
>>> mock.foo(2,3)
>>> mock.foo('a')

>>> mock.foo.assert_called_with('a')

>>> mock.foo.assert_called_once_with('a')
Traceback (most recent call last):
AssertionError: Expected to be called once. Called 2 times.

>>> mock.foo.assert_called_with(2, 3)
Traceback (most recent call last):
AssertionError: Expected call: f(2, 3)
Actual call: f('a')

>>> mock.foo.assert_any_call(2, 3)
```

Mimicking an existing class

- Use `spec` to inherit interface from class:

```
>>> from smtplib import SMTP
>>> mock_smtp = Mock(spec=SMTP)

>>> isinstance(mock_smtp, SMTP)
True

>>> mock_smtp.<TAB>
                    mock_smtp.assert_any_call          mock_smtp.attach_mock
                    mock_smtp.assert_called_once_with mock_smtp.auth
                    mock_smtp.assert_called_with       mock_smtp.auth_cram_md5
                    mock_smtp.assert_has_calls         mock_smtp.auth_login
                    mock_smtp.assert_not_called        mock_smtp.auth_plain
                    ...
```

Mimicking an existing class

- Use `spec` to inherit interface from class:

```
>>> mock_smtp.bogus
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-4856e93b6e10> in <module>()
----> 1 mock_smtp.bogus

/Users/pberkes/miniconda3/envs/gnode/lib/python3.5/unittest/mock.py in
__getattr__(self, name)
    576         elif self._mock_methods is not None:
    577             if name not in self._mock_methods or name in
_all_magics:
--> 578                 raise AttributeError("Mock object has no attribute
%r" % name)
    579         elif _is_magic(name):
    580             raise AttributeError(name)

AttributeError: Mock object has no attribute 'bogus'
```

Returning values

- Use `return_value` for a single return value:

```
>>> mock=Mock()
>>> mock.bar.return_value = 7
>>> mock.bar(32)
7
>>> mock.bar(one=2, two=4)
7
```

- Use `side_effect` for a list of return values, one per call:

```
>>> mock.bar.side_effect = [1, 4, 5]
>>> mock.bar()
1
>>> mock.bar()
4
>>> mock.bar()
5
>>> mock.bar()
Traceback (most recent call last):
StopIteration
```

Side effects

- Mock calls with side effects:

```
>>> mock.baz.side_effect = lambda x: x.append(2)
>>> a=[1]
>>> mock.baz(a)
>>> a
[1, 2]
```

- Raising exceptions:

```
>>> mock.baz.side_effect = Exception('Noooo')
>>> mock.baz(2)
Traceback (most recent call last):
Exception: Noooo
```

Example: how do we test this function?

```
report_template = """
Report
======

The experiment was a {judgment}!
Let's do this again, with a bigger budget.
"""

def send_report(result, smtp):
    if result > 0.5:
        judgment = 'big success'
    else:
        judgment = 'total failure'
    report = report_template.format(judgment=judgment)

    smtp.send_message(
        report,
        from_addr='pony@magicpony.com',
        to_addrs=['ferenc@magicpony.com'],
    )
```

Example: use mock!

```
from unittest.mock import Mock


def test_send_report_success():
    smtp = Mock()

    send_report(0.6, smtp)
    assert smtp.send_message.call_count == 1
    pos_args, kw_args = smtp.send_message.call_args
    message = pos_args[0]
    assert 'success' in message

    smtp.reset_mock()

    send_report(0.4, smtp)
    assert smtp.send_message.call_count == 1
    args, kwargs = smtp.send_message.call_args
    message = args[0]
    assert 'failure' in message
```

Patches

- Most of the time, Mock objects are not created from scratch. Rather, one momentarily substitute an object with a mock.
- The `patch` context manager is used for patching objects only within a block of code:

```
with mock.patch('my_module.MyObject'):
    # here MyObject is patched with a Mock object
# here MyObject is restored to normal
```

- It automatically handle the un-patching for you, even if exceptions are raised

Demo

- Expensive telescope

Hands-on!

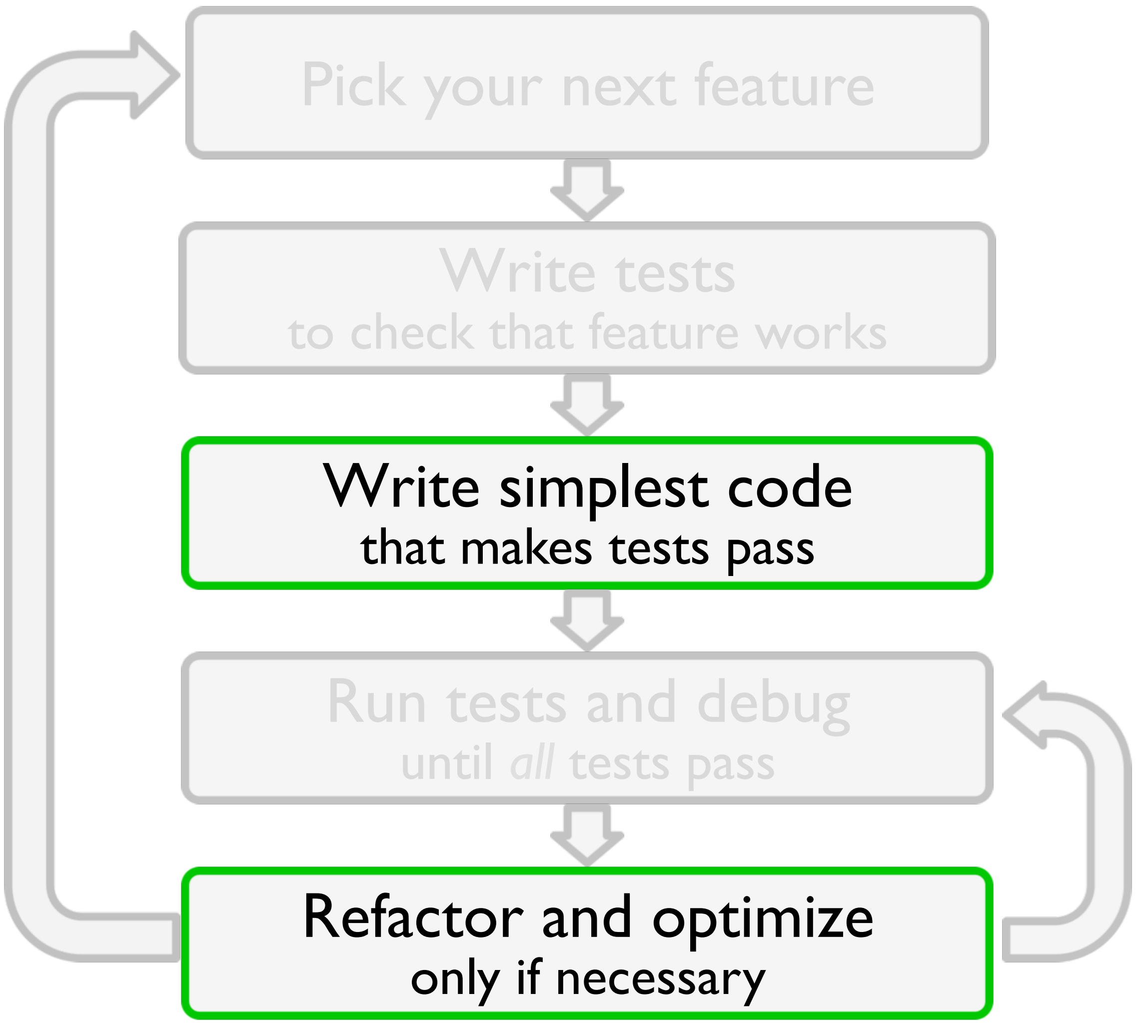
- Timed experiment

Optimization and profiling

Testing makes you efficient, too!

- 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
callgrind
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 practical 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 optimizing code 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? Use a profiler!
   (see also Francesc Alted's videos)
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:

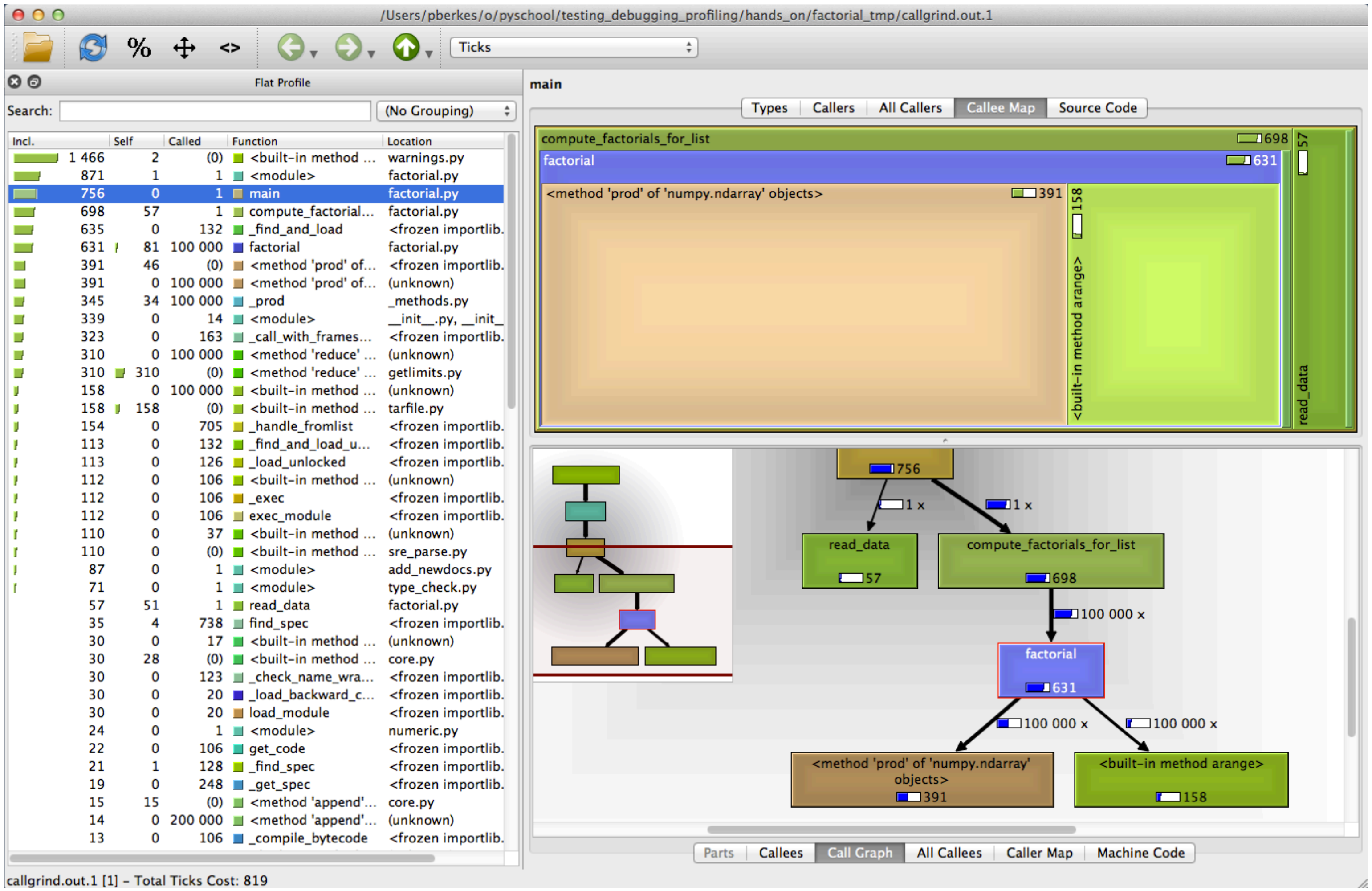
```
python -m cProfile -o filename.prof myscript.py
```

- 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.
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
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: line_profiler

- 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
```

What about Theano?

- Compiled Theano function will show up in the profiling statistics as any other Python function
- If you want to profile the Theano graph:
  1. Set the `profile` and/or `profile_memory` flags to True
  2. Add `profile=True` when creating a Theano `function`
- Theano will print out a profiling report when script exists, or upon request:
  `func.profile.summary()`

Demo

- residuals

Hands on! (time permitting)

- Profile the NLP distance function

Packaging

Standard structure of Python projects

- root
  - package
    - __init__.py
    - module1.py
    - module2.py
    - tests
      - test_module1.py
      - test_module2.py
  - docs
  - LICENSE.txt
  - README.txt
  - setup.py

setup.py

- Describes the project content: packages, data, etc
- Contains project metadata: authors, version numers, etc
- Can be used to install packages, create eggs, upload to PyPI

Minimal setup.py

```
from setuptools import setup, find_packages


setup(
    name='noiser',
    version='1.0',
    packages=find_packages(),
)
```

Cython extensions

```
from Cython.Build import cythonize
import numpy
from setuptools.extension import Extension
from setuptools import setup, find_packages

extensions = [
    Extension(
        "noiser.utils",
        ["noiser/utils.pyx"],
        include_dirs=[numpy.get_include()],
    ),
]

setup(
    name='PyAnnoExample',
    version='1.0',
    packages=find_packages(),
    ext_modules=cythonize(extensions),
)
```

Entry points

```python
from Cython.Build import cythonize
import numpy
from setuptools.extension import Extension
from setuptools import setup, find_packages

extensions = [
    Extension(
        "noiser.utils",
        ["noiser/utils.pyx"],
        include_dirs=[numpy.get_include()],
    ),
]

setup(
    name='PyAnnoExample',
    version='1.0',
    packages=find_packages(),
    ext_modules=cythonize(extensions),
    entry_points={
        'console_scripts': [
            'baboon=noiser.main:main',
        ],
    }
)
```

Package data

```
from Cython.Build import cythonize
import numpy
from setuptools.extension import Extension
from setuptools import setup, find_packages

extensions = [
    Extension(
        "noiser.utils",
        ["noiser/utils.pyx"],
        include_dirs=[numpy.get_include()],
    ),
]

setup(
    name='PyAnnoExample',
    version='1.0',
    packages=find_packages(),
    ext_modules=cythonize(extensions),
    entry_points={
        'console_scripts': [
            'baboon=noiser.main:main',
        ],
    },
    package_data={'noiser': ['images/*.png’]}
)
```

Hands-on!

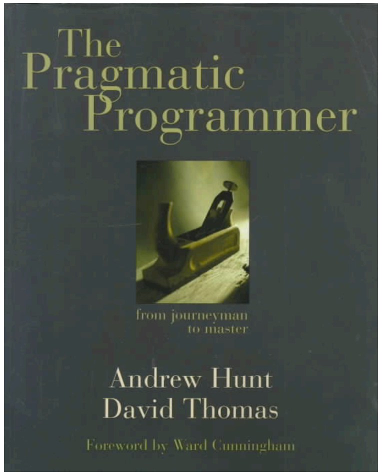
- In `research/research` create a `setup.py` file
  - Include the mpt package
  - Move content of mpt/core/setup.py
  - Create a binary wheel, make sure that it contains the cython code
- Run `python setup.py develop`

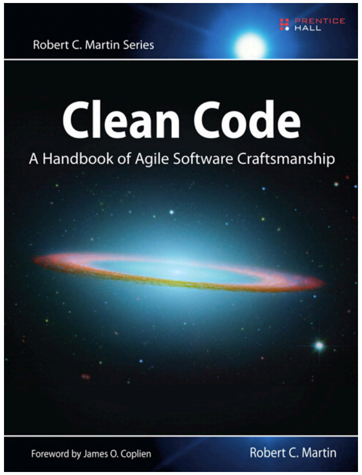
Final thoughts

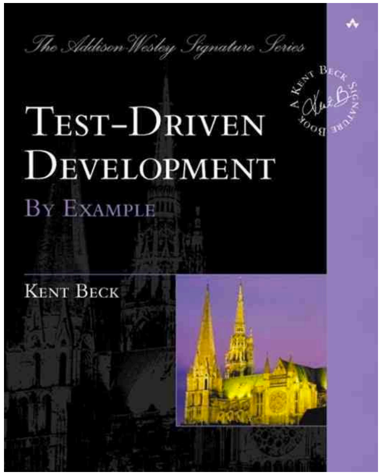
- Testing is your best shot at keeping shared code flexible
- For maximum efficiency, check out how these tools can be integrated with PyCharm

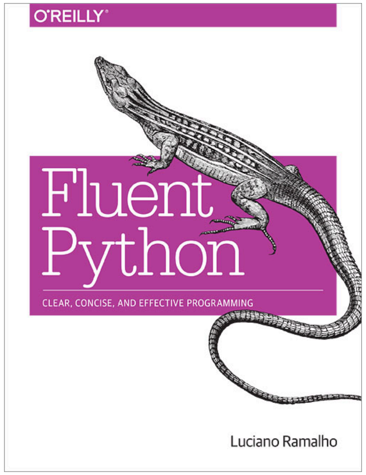
The End

- Thank you!









Code organization

A matter of style

- Style is subjective, but standards are very helpful when working in a team
- What matters: usability, longevity of code
  - Consistency helps people find and read code

What helps in my experience

- Agree on interfaces
  - This is crucial to avoid mistakes: e.g., format of images
- Pick a style, enforce it with flake8
  - flake8 can be customized to match your style
  - Avoids extremes: inconsistency and never ending PR discussions on style
  - Extra step is to have CI fail if flake8 fails
- Take backwards compatibility seriously
  - Do not break backwards compatibility
  - Semantic versioning
  - Delete unused code

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!

- In `research/research` **create a** `tox.ini` **file:**

```
[flake8]
exclude = build,dist,doc,notebooks
max-line-length = 120
```

- Run flake8 on our libraries:

```
flake8
```

- Errors and warnings we don't care about can be silenced by adding them to config file, e.g.:

```
[flake8]
...
ignore = F405,E226
```

Data structures

Performance of data structures

- Demo: complaining brother

Performance of data structure operations

	insert	remove	find	ordered
list	linear	linear	linear	✓
set	constant	constant	constant	✗
dict	constant	constant	constant	✗

Hands-on

- Sentiment analysis