Writing robust scientific code with testing (and Python)

Pietro Berkes, Enthought UK
Modern programming practices and science

- Researchers and scientific software developers write software daily, but few have been trained to do so
- Good programming practices make a BIG difference
- We can learn a lot from the development methods developed for commercial and open source software in the past 10 years
Requirements for scientific programming

- Main requirement: scientific code must be **error free**
- Scientist time, not computer time is the bottleneck
  - Being able to explore many different models and statistical analyses is more important than a very fast single approach
- Reproducibility and re-usability:
  - Every scientific result should be independently reproduced at least internally before publication (DFG, 1999)
  - Increasing pressure for making the source code used in publications available online (especially for theoretical papers)
  - No need for somebody else to re-implement your algorithm
Effect of software errors in science

frequency

severity

oops, wrong labels!

need to send *errata corringe*

end of career

Testing scientific code
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 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

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. Sullivan

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 these 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.
This includes the industry

This includes the industry

Knight Capital Says Trading Glitch Cost It $440 Million

BY NATHANIEL POPPER

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
Outline

- Overview / reminder: The agile programming cycle
- Basic testing
  - Testing with Python: unittest
  - What to test, and why
  - Testing scientific code
- Advanced testing
  - Mock objects in testing
  - Coverage
  - unittest, advanced features
- Hands-on test-driven-development session [time permitting]
Agile development

- Generic name for set of more specific software development processes: XP, Scrum, Kanban
- At the core: set of good programming practices
- Particularly suited for small teams facing unpredictable or rapidly changing requirements
Best practices:
The “agile development” cycle

1. Write tests to check your code.
2. Write *simplest* version of the code.
3. Run tests and debug until all tests pass.
4. Optimize only at this point.

Testing scientific code
Testing in agile development

- Formal software testing has become one of the most important parts of modern software development.
- Tests become part of the programming cycle and 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
Testing benefits

- Tests are the only way to trust your code
- Encourages better code and optimization: code can change, and consistency is assured by tests
- Faster development:
  - Bugs are always pinpointed
  - Avoids starting all over again when fixing one part of the code causes a bug somewhere else
- It might take you a while to get used to writing them, but it will pay off quite rapidly

Write tests to check your code
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 the interface to your code before writing it

- The result is code whose features can be tested individually, leading to maintainable, decoupled code

Write tests to check your code
Start simple

- Write small, testable chunks of code
  - Write intention-revealing code
  - Unnecessary features are not used but need to be tested and maintained
  - Re-use external libraries (if well-tested)
- Do not try to write complex, efficient code at this point

Write simplest version of the code
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

- Don’t start littering your code with *print* statements
How to handle bugs with testing

1. Isolate the bug
   • Test cases should already eliminate most possible causes
   • Use a debugger, not print statements

2. Add a test that reproduces the bug to your test suite (make sure it fails)

3. Solve the bug

4. Run all tests and check that they pass
Python code optimization

- Python is slower than C, but not prohibitively so
- In scientific applications, this difference is even less noticeable, as *numpy*, *scipy*, … do costly operations in C or Fortran
- Don’t rush into writing optimizations
How to optimize

- Usually, a small percentage of your code takes up most of the time
- Stop optimizing as soon as possible

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

Optimize only at this point
Optimization methods hierarchy

- **Warning: controversial content**

- In order of preference:
  - Vectorize code using numpy
  - Spend some money on better hardware, optimized libraries (e.g., Intel’s MKL)
  - Use a “magic optimization” tool, like numexpr, or scipy.weave
  - Use Cython
  - Parallelize your code
  - Use GPU acceleration

**Optimize only at this point**
The “agile development” cycle - review

1. Write tests to check your code
2. Write *simplest* version of the code
3. Run tests and debug until all tests pass
4. Optimize only at this point
Outline

- Overview / reminder: The agile programming cycle
- **Basic testing**
  - Testing with Python: unittest
  - What to test, and why
  - Testing scientific code
- Advanced testing
  - coverage
  - mock objects in testing
  - unittest advanced features
Testing with Python

- **unittest:**
  - Has been part of the Python standard library since v. 2.1
  - Interface a bit awkward (camelCase methods…), very basic functionality until...
  - Major improvement with 2.7, now at the level of other modern testing tools
  - Backward compatible, unittest2 back-port for earlier versions of Python

- **Alternatives:**
  - nosetests
  - py.test
Test suites in Python: unittest

- unittest: standard Python testing library
- Each test case is a subclass of unittest.TestCase
- Each test unit is a method of the class, whose name starts with ‘test’
- Each test unit checks one aspect of your code, and raises an exception if it does not work as expected
Anatomy of a TestCase

Create new file, `test_something.py`:

```python
import unittest

class FirstTestCase(unittest.TestCase):
    def test_truisms(self):
        """All methods beginning with 'test' are executed""
        selfassertTrue(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()
```

Testing scientific code
Multiple TestCases

```python
import unittest

class FirstTestCase(unittest.TestCase):
    def test_truisms(self):
        self.assertTrue(True)
        selfassertFalse(False)

class SecondTestCase(unittest.TestCase):
    def test_approximation(self):
        self.assertAlmostEqual(1.1, 1.15, 1)

if __name__ == '__main__':
    # execute all TestCases in the module
    unittest.main()
```

Testing scientific code
**TestCase** defines utility methods to check that some conditions are met, and raise an exception otherwise

- **Check that statement is true/false:**
  ```python
  assertTrue('Hi'.islower()) => fail
  assertFalse('Hi'.islower()) => pass
  ```

- **Check that two objects are equal:**
  ```python
  assertEqual(2+1, 3) => pass
  assertEqual([2]+[1], [2, 1]) => pass
  assertNotEqual([2]+[1], [2, 1]) => fail
  ```

**assertEqual** can be used to compare numbers, lists, tuples, dicts, sets, frozensets, and unicode objects
**TestCase assertsomething**

- Check that two numbers are equal up to a given precision:
  ```python
testCase.assertEqual(x, y, places=7)
```  
- `places` is the number of decimal places to use:
  ```python
testCase.assertAlmostEqual(1.121, 1.12, 2)  # pass
testCase.assertAlmostEqual(1.121, 1.12, 3)  # fail
```

Formula for almost-equality is

```python
round(x - y, places) == 0.
```

And so

```python
testCase.assertAlmostEqual(1.126, 1.12, 2)  # fail
```
One can also specify a maximum difference:

```
assertAlmostEqual(x, y, delta=0.)
```

e.g.:

```
assertAlmostEqual(1.125, 1.12, 0.06)  =>  pass
assertAlmostEqual(1.125, 1.12, 0.04)  =>  fail
```

Can be used to compare any object that supports subtraction and comparison:

```
import datetime
delta = datetime.timedelta(seconds=10)
second_timestamp = datetime.datetime.now()

self.assertAlmostEqual(first_timestamp, second_timestamp, delta=delta)
```
**TestCase.assertRaises**

- Check that an exception is raised:
  
  ```python
  assertRaises(exception_class, function,
  arg1, arg2, kwarg1=None, kwarg2=None)
  ```

  executes
  
  ```python
  function(arg1, arg2, kwarg1=None, kwarg2=None)
  ```

  and passes if an exception of the appropriate class is raised.

- For example:
  
  ```python
  assertRaises(IOError, file, 'inexistent', 'r') => pass
  ```

  Use the most specific exception class, or the test may pass because of collateral damage:
  
  ```python
  tc.assertRaises(IOError, file, 1, 'r') => fail
  tc.assertRaises(Exception, file, 1, 'r') => pass
  ```
The most convenient way to use `assertRaises` is as a context manager:

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

For example:

```python
with self.assertRaises(ValueError):
    int('XYZ')
```

passes, because

```python
int('XYZ')
ValueError: invalid literal for int() with base 10: 'XYZ'
```
**TestCase.assertSomething**

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

  ```python
  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
  ```
**TestCase.asserSomething**

- Most of the `assert` methods accept an optional `msg` argument that overwrites the default one:

  ```python
  assert_true('Hi'.islower(),
              'One of the letters is not lowercase')
  ```

- Most of the `assert` methods have a negated equivalent, e.g.:

  ```python
  assert_is_none
  assert_is_not_none
  ```
Testing with numpy arrays

- When testing numerical algorithms, numpy arrays have to be compared elementwise:

```python
class NumpyTestCase(unittest.TestCase):
    def test_equality(self):
        a = numpy.array([1, 2])
        b = numpy.array([1, 2])
        self.assertEqual(a, b)
```

```
E
======================================================================
ERROR: test_equality (__main__.NumpyTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "numpy_testing.py", line 8, in test_equality
    self.assertEqual(a, b)
  File "/Library/Frameworks/Python.framework/Versions/6.1/lib/python2.6/unittest.py", line 348, in failUnlessEqual
    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 0.000s

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 if 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)))`
How to run tests

- Option 1: `unittest.main()` will execute all tests in all `TestCase` classes in a file

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

- Option 2: Execute all tests in one file
  ```bash
  python -m unittest [-v] test_module
  ```

- Option 3: Discover all tests in all subdirectories
  ```bash
  python -m unittest discover
  ```
Basics of testing

- What to test, and how?

- At the beginning, testing feels weird:
  1) It’s obvious that this code works (not TDD…)
  2) The tests are longer than the code
  3) The test code is a duplicate of the real code

- What does a good test looks like?

- What should I test?

- Anything specific to scientific 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**: The key actions of the test
  - Typically one or two lines of code

- **Then**: Compare outcomes of the key actions 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

```python
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?

```python
def test_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:

```python
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)
```
Fixtures

- Tests require an initial state or test context in which they are executed (the “Given” part), which needs to be initialized and possibly cleaned up.

- If multiple tests require the same context, this fixed context is known as a fixture.

- Examples of fixtures:
  - Creation of a data set at runtime
  - Loading data from a file or database
  - Creation of mock objects to simulate the interaction with complex objects
import unittest

class FirstTestCase(unittest.TestCase):

    def setUp(self):
        """setUp is called before every test""
        pass

    def tearDown(self):
        """tearDown is called at the end of every test""
        pass

    @classmethod
def setUpClass(cls):
        """Called once before all tests in this class.""
        pass

    @classmethod
def tearDownClass(cls):
        """Called once after all tests in this class.""
        pass

    # ... all tests here ...
Numerical fuzzing

- Use deterministic test cases when possible
- In most numerical algorithm, this will cover only oversimplified 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 algorithms, it is 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

Testing scientific code
Numerical fuzzing – example

class VarianceTestCase(unittest.TestCase):
    def setUp(self):
        self.seed = int(numpy.random.randint(2**31-1))
        numpy.random.seed(self.seed)
        print 'Random seed for the tests:', self.seed
    def test_var(self):
        N, D = 100000, 5

        # goal variances: [0.1, 0.45, 0.8, 1.15, 1.5]
        desired = numpy.linspace(0.1, 1.5, D)

        # test multiple times with random data
        for _ in range(20):
            # generate random, D-dimensional data
            x = numpy.random.randn(N, D) * numpy.sqrt(desired)
            variance = numpy.var(x, axis=0)
            numpy.testing.assert_array_almost_equal(variance, desired, 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 $y = ax + b + \text{noise}$ 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 `z = outer(x, y)` with

    ```python
    M, N = x.shape[0], y.shape[0]
    z = numpy.zeros((M, N))
    for i in range(M):
        for j in range(N):
            z[i, j] = x[i] * y[j]
    ```
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 corner 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?
DEMO

Testing scientific code
Outline

- Overview / reminder: The agile programming cycle
- Basic testing
  - Testing with Python: unittest
  - What to test, and why
  - Testing scientific code
- Advanced testing
  - mock objects in testing
  - coverage
  - unittest advanced features
Mock objects

- What is a “mock object”?
  
  *Mock object*: object that mimics the behavior of a “real” object, without some unwanted behavior or side effect

- Sometimes people distinguish between
  
  *Fake object*: anything that is not a real, production object
  
  *Mock object*: a fake object, plus recording and assertions

- Testing is the main application for mock objects
Mock objects in testing

- Blank out parts of real objects
  - Difficult to set up
    - parameters to constructor require setting up a large number of other objects
    - desired state requires many intermediate steps
  - Too slow
    - connect to remote server
    - initialize hardware
  - Have undesired side effect
    - commit to central database
    - post to Twitter
  - Is non-deterministic
    - depends current time or temperature
Mock objects in testing

- Record access to methods and attributes
  - In event-based systems: trigger event in test, verify that listeners are called in right order

- Decouple test modules from external dependencies
  - Mock one or two object
  - Patch an entire module
The `mock` library

- In Python, you can easily create mock objects by monkey-patching existing ones or by creating new fake objects and using duck-typing.
- `mock` does it for you in a much more convenient way.
The Mock object

- The superstar of this library, Mock, absorbs everything:

```python
>>> from 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'>
```
The Mock object

- Use ‘spec’ to inherit interface from class:

```python
>>> from chaco.api import Plot
>>> mock_plot = Mock(spec=Plot)

>>> isinstance(mock_plot, Plot)
True

>>> mock_plot.<TAB>
mock_plot.add          mock_plot.insert
mock_plot.add_class_trait   mock_plot.invalidate_and_redraw
mock_plot.add_trait   mock_plot.invalidate_draw
mock_plot.add_trait_category   mock_plot.is_in
mock_plot.add_trait_listener   mock_plot.lower_component
[...]
```
The Mock object

- Use ‘spec’ to inherit interface from class:

```python
>>> mock_plot.bogus
```

Traceback (most recent call last):
  File "<ipython-input-35-4edba1c6647d>", line 1, in <module>
    mock_plot.bogus
  File ".../python2.7/site-packages/mock.py", line 700, in __getattr__
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'bogus'

```python
>>> mock_plot.add(1, 2, "I'm making stuff up", None, ['a', 'b'])
<Mock name='mock.add()' id='285179312'>
```
Recording interaction with Mock objects

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

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

```python
>>> mock.foo.call_args
call('a')
>>> mock.foo.call_count
2
>>> mock.foo.call_args_list
[call(2, 3), call('a')]```
Testing interaction with Mock objects

```python
>>> 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: foo('a')

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

- Mock calls with side effects:

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

- Raising exceptions:

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

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

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

class TelescopeModel(object):

    # Minimum safe elevation angle (see handbook)
    MIN_ANGLE = 0.0

    # Maximum safe elevation angle (see handbook)
    MAX_ANGLE = 80.0

    def __init__(self, address):
        self.address = address
        # connect to telescope
        self.connection = telescope_driver.connect(address)
        # get initial state of telescope
        self.current_angle = telescope_driver.get_angle(self.connection)

    def set_elevation_angle(self, angle):
        """ Set the elevation angle in radians of the telescope.

        If the angle is outside the range allowed by the manufacturer,
        raise a ValueError.
        """

        if angle < self.MIN_ANGLE or angle > self.MAX_ANGLE:
            raise ValueError('Unsafe elevation angle: {}).format(angle))

        telescope_driver.set_angle(self.connection, angle)
        self.current_angle = angle
Mocking example: expensive telescope test

```python
import numpy as np
import unittest

class TelescopeTestCase(unittest.TestCase):
    def test_unsafe_elevation_angle(self):
        telescope = TelescopeModel(address='10.2.1.1')
        elevation_angle = np.deg2rad(90)

        with self.assertRaises(ValueError):
            telescope.set_elevation_angle(elevation_angle)

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

Problems:
1. We need to wait for the communication with the telescope:
   Ran 1 test in 12.030s
2. We just broke the telescope
Mocking example: expensive telescope test

```python
>>> python telescope_example.py
E
```

```
ERROR: test_unsafe_elevation_angle (___main___.TelescopeTestCase)
```

```
Traceback (most recent call last):
  File "telescope_example.py", line 43, in test_unsafe_elevation_angle
telescope.set_elevation_angle(elevation_angle)
  File "telescope_example.py", line 28, in set_elevation_angle
    set_telescope_angle(self.connection, angle)
  File "telescope_driver.py", line 11, in set_telescope_angle
...
  IOError: Telescope jammed -- please call technical support
```

```
Ran 1 test in 12.000s
```

```
FAILED (errors=1)
```

- We gave the maximum angle in degrees… ooops!
Mocking example: expensive telescope test

```python
import numpy as np
from mock import Mock
import unittest

class TelescopeTestCase(unittest.TestCase):
    def test_unsafe_elevation_angle(self):
        telescope_driver.connect = Mock(return_value='bogus_connection')
        telescope_driver.get_angle = Mock(return_value=0.0)
        telescope_driver.set_angle = Mock()

        telescope = TelescopeModel(address='10.2.1.1')
        elevation_angle = np.deg2rad(90)

        with self.assertRaises(ValueError):
            telescope.set_elevation_angle(elevation_angle)

>>> python telescope_example.py
F
======================================================================
FAIL: test_unsafe_elevation_angle (__main__.TelescopeTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "telescope_example.py", line 60, in test_unsafe_elevation_angle
    telescope.set_elevation_angle(elevation_angle)
AssertionError: ValueError not raised
----------------------------------------------------------------------
Ran 1 test in 0.001s
```

Testing scientific code
Patches

- The patch context manager are used for patching objects only within a block of code:

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

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

- Also available as function or class decorators
from my_db import DataBase

RESULTS_DATABASE = 'db_filename'

class SimulationManager(object):
    """Schedule simulations and store results in database."""

    def __init__(self):
        self.db = DataBase(RESULTS_DATABASE)

    def run_simulation(self, name, parameters):
        result = big_simulation(parameters)
        self.db.append(key=name, parameters=parameters, result=result)
        return result

    def big_simulation(parameters):
        """This is what your paper is about."""
        return sum(parameters)
Patches example: write to database

- How to test run_simulation?
- This is the test we'd like to write, but it messes up the database:

```python
class SimulationManagerTestCase(unittest.TestCase):
    def test_simulation_manager(self):
        name = 'TEST'
        parameters = [1, 2, 3]
        expected_result = 1 + 2 + 3

        sim_manager = SimulationManager()
        result = sim_manager.run_simulation(name, parameters)

        assert result == expected_result
        # TODO: check db.append has been called correctly
```
Patches example: write to database

- With mock.patch:

```python
class SimulationManagerTestCase(unittest.TestCase):

    def test_simulation_manager(self):
        name = 'TEST'
        parameters = [1, 2, 3]
        expected_result = 1 + 2 + 3

        with mock.patch('my_db.DataBase'):
            sim_manager = SimulationManager()
            result = sim_manager.run_simulation(name, parameters)

            assert result == expected_result

            # check db.append has been called correctly
            sim_manager.db.append.assert_called_once_with(
                key=name, parameters=parameters, result=result
            )
```
Sentinels

- The sentinel object provides a convenient way of providing unique objects for your tests.
- Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object.
- Sentinel objects have a sensible string representation, so that test failure messages are readable.
Decouple infrastructure based on generic interfaces from concrete instances defined in other packages

```python
from mock import Mock, sentinel
def test_add_data_from_file(self):

    # Given
    mock_reader_writer = Mock(spec=ReaderWriter)
    mock_reader_writer.can_read.return_value = True
    mock_reader_writer.read.return_value = sentinel.data

    data_manager = DataManager(
        reader_writers=[mock_reader_writer]
    )

    # When
    data = data_manager.add_data_from_file('bogus.dat')

    # Then
    self.assertIs(data, sentinel.data)
    self.assertEqual(1, len(data_manager.data))
```
mock.py: we just scratched the surface

- There are a lot of others features and parameters to tweak to get the desired behavior from your mock objects
- Mocking is often underappreciated, fragile or ad-hoc solutions are often preferred (e.g. relying on “example” data being present, depending on external libraries, …)
- Be careful you are not mocking away the problem!
Code coverage

- It’s easy to leave part of the code untested
  - Classics:
    - Feature activated by keyword argument
    - Exception raised for invalid input
- Coverage tools mark the lines visited during execution
- Use together with test framework to make sure all your code is covered
coverage.py

- Python script to perform code coverage
- Produces text and HTML reports
- Allows branch coverage analysis
- Not included in standard library, but quite standard
DEMO
Advanced unittest: skipping tests

```python
import mdp

class CachingTestCase(unittest.TestCase):

    @skipIf(mdp.config.has_joblib,
            "This test requires the 'joblib' module.")
    def test_class_caching(self):
        """Test that we can cache individual classes."""

        cached = mdp.nodes.PCANode()
        not_cached = mdp.nodes.SFANode()

        with mdp.caching.cache(cache_classes=[mdp.nodes.PCANode]):
            assert cached.is_cached()
            assert not not_cached.is_cached()
```
Skipping tests

- **When joblib is present:**

  ```
  ==> python -m unittest discover -b -v
  test_class_caching (test_skipping.CachingTestCase)
  Test that we can cache individual classes. ... ok
  
  Ran 1 test in 0.002s
  OK
  ```

- **Not present:**

  ```
  ==> python -m unittest discover -b -v
  test_class_caching (test_skipping.CachingTestCase)
  Test that we can cache individual classes. ... skipped "This test requires the 'joblib' module."
  
  Ran 1 test in 0.000s
  OK (skipped=1)
  ```
Advanced unittest: TypeEqualityFunc

- `assertEqual` can be used to compare numbers, lists, tuples, dicts, sets, frozensets, and unicode objects.
- This is done by dispatching to type-specific methods when the types of the arguments match:

<table>
<thead>
<tr>
<th>Method</th>
<th>Used to compare</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertMultLineEqual(a, b)</code></td>
<td>strings</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertSequenceEqual(a, b)</code></td>
<td>sequences</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertListEqual(a, b)</code></td>
<td>lists</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertTupleEqual(a, b)</code></td>
<td>tuples</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertSetEqual(a, b)</code></td>
<td>sets or frozensets</td>
<td>2.7</td>
</tr>
<tr>
<td><code>assertDictEqual(a, b)</code></td>
<td>dicts</td>
<td>2.7</td>
</tr>
</tbody>
</table>
It is possible to register new type-specific function through the `addTypeEqualityFunc` method.

- `addTypeEqualityFunc(type, function)`
  - **Calls** `function(a, b, msg=None)` if `a` and `b` are of the same type – not subclasses!
  - `function` **must raise** `self.failureException(msg)` **when** inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message.
import unittest

class StringsTestCase(unittest.TestCase):
    def __init__(self, *args, **kwargs):
        super(StringsTestCase, self).__init__(*args, **kwargs)
        self.addTypeEqualityFunc(str, self.assertStringEqualCaseInsensitive)

    def assertStringEqualCaseInsensitive(self, a, b, msg=None):
        assert a.lower() == b.lower(), msg

class GreetingsTestCase(StringsTestCase):
    def test_passes(self):
        self.assertEqual('Hallo', 'hallo')

    def test_fail(self):
        self.assertEqual('hi', 'hi!')
Skipping tests

- `unittest.skip(msg)`
  Unconditionally skip the decorated test.

- `unittest.skipIf(condition, msg)`
  Skip the decorated test if `condition` is true.

- `unittest.skipUnless(condition, msg)`
  Skip the decorated test unless `condition` is true.

- `unittest.expectedFailure()`
  Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.
Pointers

- Testing documentation
  - doctest

- Continuous integration
  - TravisCI
  - Jenkins

- More information on best practices:
  - Software carpentry course by Greg Wilson
    [http://software-carpentry.org](http://software-carpentry.org)
  - Similar course by Tiziano Zito
    [http://itb.biologie.hu-berlin.de/~zito/teaching/SC](http://itb.biologie.hu-berlin.de/~zito/teaching/SC)
Final thoughts

- Starting point: scientific code has slightly different needs than “regular” code, including the need to ensure correctness.

- Agile programming methods, and testing in particular, go a long way toward this goal.

- Testing in Python is as easy as it gets, there are no excuses not to do it!

Homeworks ;-) 

- If you’re just starting, don’t obsess with tools and techniques: start by putting under test the critical part of your next application.
- If you’re already using testing, give a chance to TDD.
The End

(Hands-on session next)
Hands-on!

- Reproduce the linear regression demo, using a quadratic function and scipy.optimize.fmin instead.
  - Test and implement noiseless generation of data
  - Test and implement noisy generation of data
  - Test quadratic fitting

- New requirement: Given data set, fit multiple times and collect set of final parameters, together with goodness-of-fit.

- Experiment with different design choices!
#TestCase.assertSomething

## TestCase methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assert_(expr[, msg])</code></td>
<td><code>assertTrue(isinstance([1,2], list) =&gt; pass</code></td>
</tr>
<tr>
<td><code>assertTrue(expr[, msg])</code></td>
<td><code>assertTrue('Hi'.islower()) =&gt; fail</code></td>
</tr>
<tr>
<td><code>assertFalse(expr[, msg])</code></td>
<td></td>
</tr>
<tr>
<td><code>assertEqual(first, second[, msg])</code></td>
<td><code>assertEqual([2, 3], [2, 3]) =&gt; pass</code></td>
</tr>
<tr>
<td><code>assertNotEqual(first, second[, msg])</code></td>
<td><code>assertEqual(1.2, 1.3) =&gt; fail</code></td>
</tr>
<tr>
<td><code>assertAlmostEqual(first, second[, places[, msg]])</code></td>
<td><code>assertAlmostEqual(1.125, 1.12, 2) =&gt; pass</code></td>
</tr>
<tr>
<td><code>assertNotAlmostEqual(first, second[, places[, msg]])</code></td>
<td><code>assertAlmostEqual(1.125, 1.12, 3) =&gt; fail</code></td>
</tr>
<tr>
<td><code>assertRaises(exception, callable, ...)</code></td>
<td><code>assertRaises(exceptions.IOError, file, 'inexistent', 'r') =&gt; pass</code></td>
</tr>
<tr>
<td><code>fail([msg])</code></td>
<td><code>fail() =&gt; fail</code></td>
</tr>
</tbody>
</table>
Possible additions

- example testing with HMM: HMM: random data -> 1/M emissions, 1/M transitions; weak, sequence of transitions with p=1, 1->2->3->1-> ...
Three more useful tools

- **pydoc**: creating documentation from your docstrings
  `pydoc [-w] module_name`

- **pylint**: check that your code respects standards
doctests

- **doctest** is a module that recognizes Python code in documentation and tests it
  - docstrings, rst or plain text documents
  - make sure that the documentation is up-to-date

- **From command line:**
  ```
  python -m doctest -v example.py
  ```

- **In a script:**
  ```
  import doctest
  doctest.testfile("example.txt") # test examples in a file
  doctest.testmod([module]) # test docstrings in module
  ```
DEMO

Testing scientific code
The basic agile development cycle

1. Write tests to check your code
2. Write *simplest* version of the code
3. Run tests and debug until all tests pass
4. Optimize only at this point

Tools:
- `unittest` coverage.py
- `pdb`
- `cProfile`
- `timeit`
- `runSnake`
Debugging

- The best way to debug is to avoid bugs
- Your test cases should already exclude a big portion of the possible causes
- Don’t start littering your code with `print` statements
- 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
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
Entering the debugger

- **Enter debugger at the start of a file:**
  
  
  ```python
  python -m pdb myscript.py
  ```

- **Enter in a statement or function:**
  
  ```python
  import pdb
  # your code here
  if __name__ == '__main__':
    pdb.runcall(function[, argument, ...])
    pdb.run(expression)
  ```

- **Enter at a specific point in the code** (alternative to `print`):
  
  ```python
  # some code here
  # the debugger starts here
  import pdb
  pdb.set_trace()
  # rest of the code
  ```
Entering the debugger from ipython

- From ipython:
  - `%pdb` – preventive
  - `%debug` – post-mortem
The basic agile development cycle

1. Write tests to check your code
2. Write *simplest* version of the code
3. Run tests and debug until all tests pass
4. Optimize only at this point

Testing scientific code
Python code optimization

- Python is slower than C, but not prohibitively so.
- In scientific applications, this difference is even less noticeable (numpy, scipy, ...)
  - for basic tasks, as fast as Matlab, sometimes faster.
- Profiler: Tool that measures where the code spends time.
timeit

- Precise timing of a function/expression
- Test different versions of a small amount of code, often used in interactive Python shell

```python
from timeit import Timer

# execute 1 million times, return elapsed time (sec)
Timer("module.function(arg1, arg2)", "import module").timeit()

# more detailed control of timing
T = Timer("module.function(arg1, arg2)", "import module")
# make three measurements of timing, repeat 2 million times
T.repeat(3, 2000000)
```

- In ipython, you can use the `%timeit` magic command
DEMO
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 myscript.py
  ```

- options
  
  `-o output_file`
  `-s sort_mode (calls, cumulative, name, ...)`

- from interactive shell/code:
  
  ```
  import cProfile
  cProfile.run(expression[, "filename.profile"])
  ```

Testing scientific code
cProfile, analyzing profiling results

- From interactive shell/code:
  ```python
  import pstat
  p = pstat.Stats("filename.profile")
  p.sort_stats(sort_order)
  p.print_stats()
  ```

- Simple graphical description with RunSnakeRun
cProfile, analyzing profiling results

- Look for a small number of functions that consume most of the time, those are the *only* parts that you should optimize.

- Way to optimize:
  - use a better algorithm
  - vectorize using numpy
  - if everything fails, consider (in decreasing order of desirability)
    - using Cython
    - parallelizing
    - use graphic cards acceleration
Pointers

- Static code analysis (“lint”-ing)
  - pyflakes -- [https://launchpad.net/pyflakes](https://launchpad.net/pyflakes)
  - pep8 -- [http://www.python.org/dev/peps/pep-0008/](http://www.python.org/dev/peps/pep-0008/)

- Version control
  - git and GitHub
  - SVN

- Continuous integration
  - TravisCI
  - Jenkins
More pointers

- More information on best practices:
  - Software carpentry course by Greg Wilson
    http://software-carpentry.org
  - Similar course by Tiziano Zito
    http://itb.biologie.hu-berlin.de/~zito/teaching/SC