Intro to scientific programming
(with Python)

Pietro Berkes, Brandeis University

Outline

- Next 4 lessons:
  - Scientific programming: best practices
  - Classical learning (Hoepfield network)
  - Probabilistic learning
    (Restricted Boltzman Machine)
  - Advanced probabilistic learning
    (Deep Belief Network)

Outline

- Next 4 lessons:
  - Scientific programming: best practices
    - Best practices in scientific programming
    - Introduction to Python and numpy
    - Test driven development in Python
    - Hands-on session
  - Classical learning (Hoepfield network)
  - Probabilistic learning
    (Restricted Boltzman Machine)
  - Advanced probabilistic learning
    (Deep Belief Network)

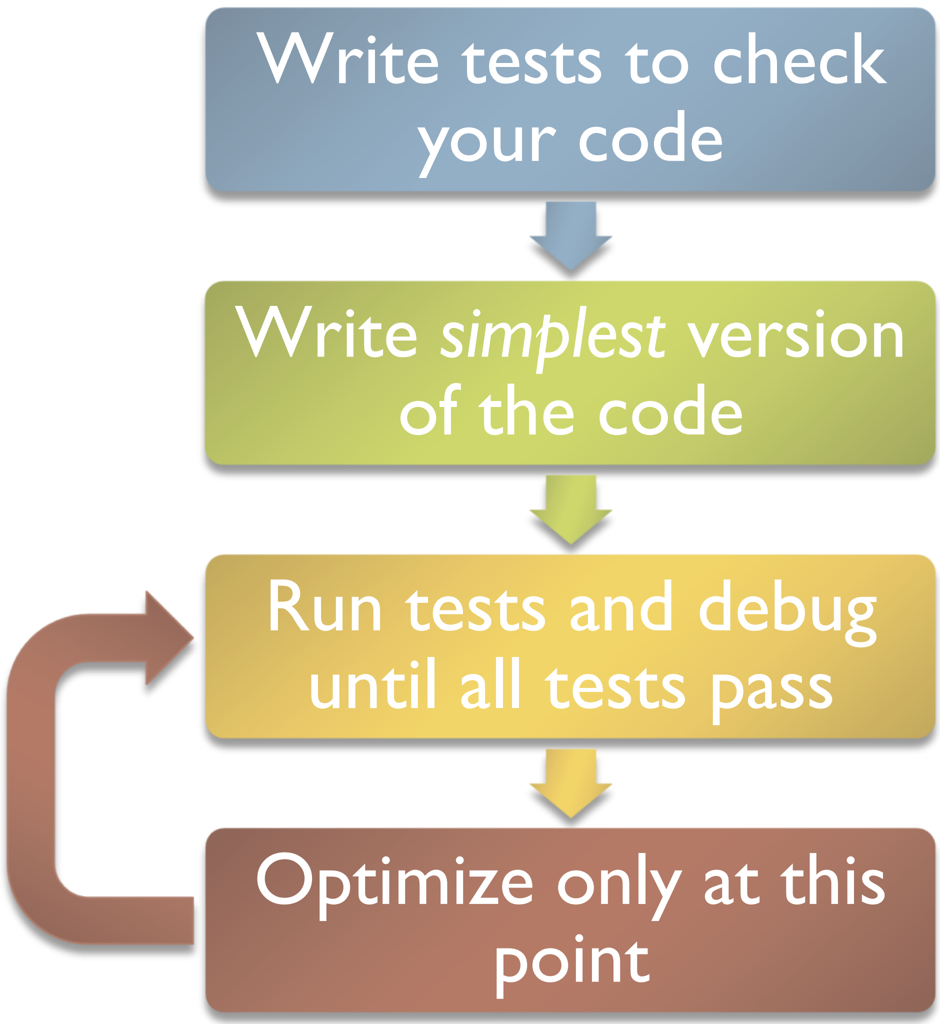
Programming needs of scientists

- Experimental study:
  - [display stimuli for experiment]
  - collect / store data
  - manipulate and process data (statistics, …)
  - visualize and prepare figures for publication
- Computational study:
  - implement model
  - run simulations on various datasets/parameters
  - visualize and prepare figures for publication

Requirements for scientific programming

- Main requirement: 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:
  - easy to read, should not compile only on special architecture
  - no need for somebody else to re-implement your algorithm

Best practices: The "agile development" cycle



Test-driven 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

```
testchoice (__main__.TestSequenceFunctions) ... ok
testsample (__main__.TestSequenceFunctions) ... ok
testshuffle (__main__.TestSequenceFunctions) ... ok

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

Write tests to check your code

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

What to test and how

- Test with hard-coded inputs for which you know the output:
  - use simple but general cases
    E.g., test `lower('Text') -> 'text'`
    with `'Hi tHerE' :`
    `assertEqual(lower('Hi tHerE'), 'hi there')`
  - test special or boundary cases
    `'another test'` -> already lowercase
    `'?[{ 012'` -> lowercase undefined
    `''` -> empty string

Write tests to check your code

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 for which you know the answer
- E.g.: test a function that computes the variance with random data from a normal distribution

Write tests to check your code

Testing learning algorithms

- Learning algorithms can get stuck in local maxima, the solution for general cases might not be known
- 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

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

How to handle bugs

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
3. Solve the bug
4. Run *all tests* and check that they pass

Run tests and debug until all tests pass

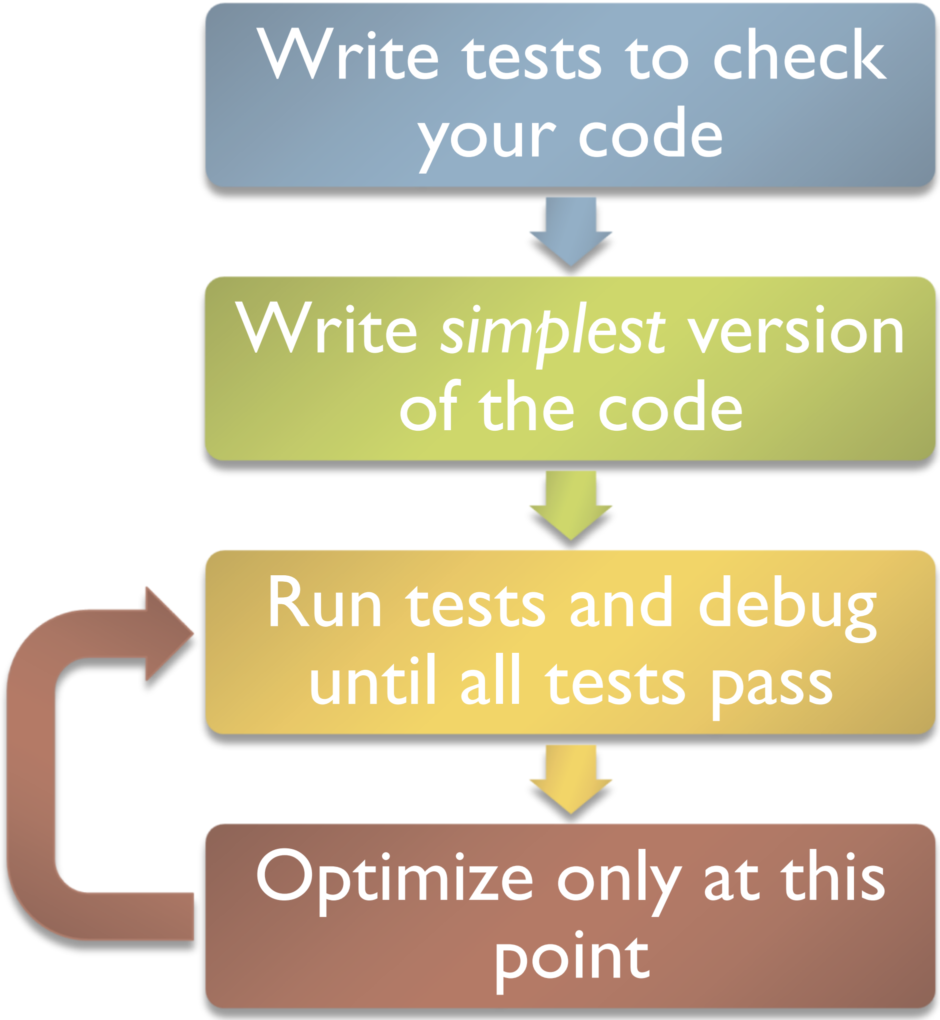
How to optimize

- 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

The "agile development" cycle (again)



Very brief intro to Python

- Why Python?
  - much research is exploratory
    - interpreted language
    - high-level, dynamical language -> writing prototypes of ideas is easy and fast
  - great support for numerical algorithms (numpy and scipy) and visualization (matplotlib)
  - large standard library (file management, data types, …)
  - large scientific community (fun, too)
  - Matlab is a popular alternative, but it is expensive and makes it really difficult to apply the best practices mentioned before

Variables and data types

- [first: ipython command line; how to get help]
- integers, floats
- strings
- lists (slices, range)
- dictionaries
- [tuples, sets, files]

Control statements

- if-else statement (indentation matters)
- cycles:
  - for statements (you can iterate over any sequence: lists, strings, dictionary keys, ...)
  - while statements

Functions

- defining functions
- optional arguments
- docstrings

Objects

- Objects are collections of variables (attributes) and functions (methods)
  - create object through constructor -> initialize attributes
  - interact through methods
  - inheritance: sub-classes inherit methods
- Lists, strings, etc. are objects

Scripts vs. modules

- Scripts: a Python file that will be executed
- Module: a Python file that defines useful functions, objects, or constants
  (a Python library)

- import statement
- Python standard library

Numerical libraries

- **NumPy** is a Python extension module, written mostly in C, that defines the numerical array and matrix types and basic operations on them
- **SciPy** is another Python library that uses NumPy to do advanced math, signal processing, optimization, statistics and much more
- **matplotlib** is a Python library that facilitates publication-quality interactive plotting

numpy arrays

- arrays (multidimensional), shape
- array operations: sum, prod, mean, max, min, …
- arange, linspace
- zeros, ones
- slicing, fancy indexing

Content of SciPy

cluster	Vector quantization / Kmeans
fftpack	Fourier transform
integrate	Integration routines
interpolate	Interpolation
io	Data input and output
linalg	Linear algebra routines
maxentropy	Routines for fitting maximum entropy models
ndimage	n-dimensional image package
odr	Orthogonal distance regression
optimize	Optimization
signal	Signal processing
sparse	Sparse matrices
spatial	Spatial data structures and algorithms
special	Any special mathematical functions
stats	Statistics

random Generate random numbers

matplotlib

- plot (xlabel, title, …)
- imshow

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`:

```
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 the Eclipse IDE"""
        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
```

`TestCase.assertSomething`

- 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
  ```

Testing with numpy arrays

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

```
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/unitt
est.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)
 numpy.testing.assert_array_less(x, y)
```

- 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

```
# test that all elements of x are larger than 1.0
assertTrue(all(x > 1.0))
```

Basic tests – example

- Test with hard-coded inputs:
  - use simple but general cases
  - test special or boundary cases

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

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

        # test all cases
        for arg, expected in test_cases:
            output = arg.lower()
            self.assertEqual(output, expected)
```

Numerical fuzzing – example

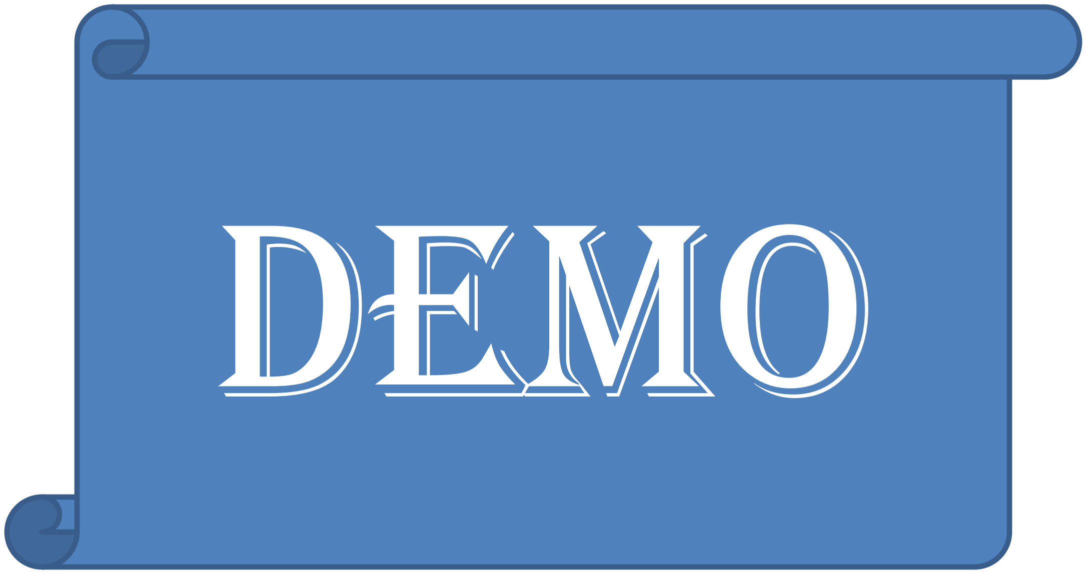
```python
class VarianceTestCase(unittest.TestCase):

    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)
```

Agile development



Thanks!

- Exercises next….
- Where to get help:
  - Python: http://docs.python.org/release/2.6/library/index.html
  - Numpy docs: http://docs.scipy.org/doc/numpy/reference/index.html
  - Matplotlib gallery: http://matplotlib.sourceforge.net/gallery.html
- 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