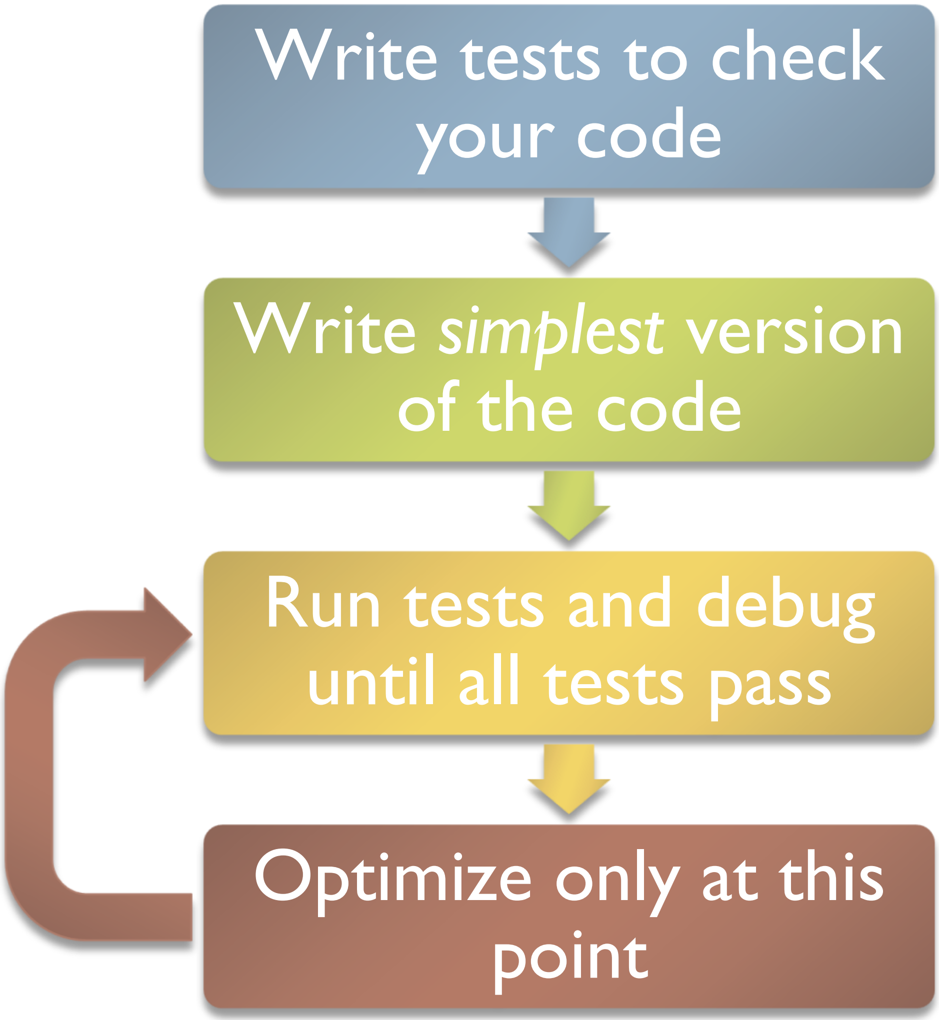
Software carpentry
From theory to practice: Standard tools

Pietro Berkes, Enthought Ltd.

Python tools for agile programming

- There are many tools, based on command line or graphical interface
- I'll present:
  - Python standard "batteries included" tools
  - no graphical interface necessary
  - magic commands for ipython
- Alternatives and cheat sheets are on the wiki

The basic agile development cycle

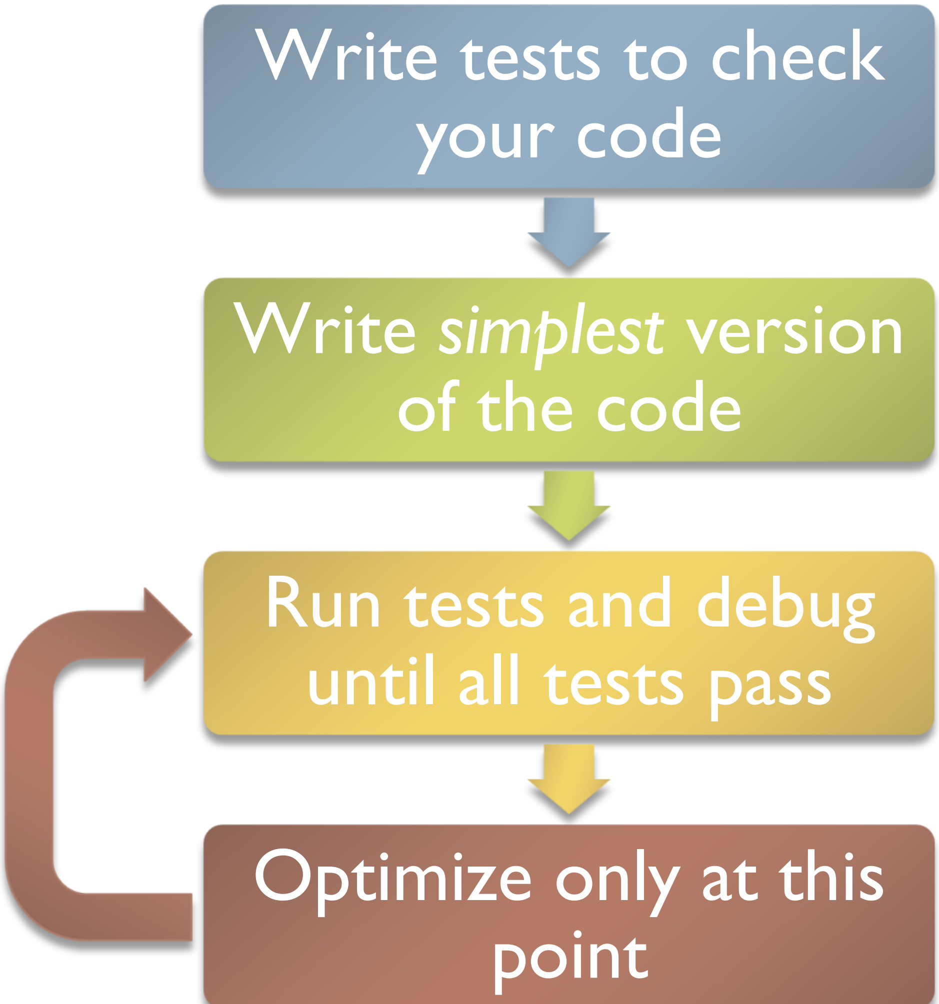



```
unittest
coverage.py

pdb

cProfile
timeit
runSnake
```

The basic agile development cycle



unittest
coverage.py

pdb

cProfile
timeit
runSnake

Test-driven development: reminder

- Tests are *crucial* for scientific programming:
  - Your research results depend on the code working as advertised
  - You need to know if you're not getting the expected results because of a bug or because your working hypothesis is wrong
  - Unchecked code usually contains errors (some small, some not)
  - Hopefully in the future tests will be requested for paper review
- Write test suite (collection of tests) in parallel with your code
- **External software runs the tests and provides reports and statistics**

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()
```

Multiple TestCases

```
import unittest

class FirstTestCase(unittest.TestCase):

    def test_truisms(self):
        self.assertTrue(True)
        self.assertFalse(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()
```

Fixtures

- Tests suites often require an initial state or *test context* to execute a number of test cases. 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 an interaction with complex objects

setUp and tearDown

```
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

    # ... all tests here ...
```

`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`

Formula for almost-equality is

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

and so

```
assertAlmostEqual(1.126, 1.12, 2) => fail
```

TestCase.assertSomething

- Check that an exception is raised:

```
assertRaises(exception_class, function,
             arg1, arg2, kwarg1=None, kwarg2=None)
```

  executes

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

  and passes if an exception of the appropriate class is raised

- For example:

```
assertRaises(IOError,
             file, 'inexistent', 'r')    => pass
```

Use the most specific exception class, or the test may pass because of collateral damage:

```
assertRaises(IOError, file, 1, 'r')    => fail
assertRaises(Exception, file, 1, 'r') => pass
```

`TestCase.assertSomething`

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

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

- Python 2.7 introduced many new features in unittest, see documentation online

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**
  - **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))
```

What to test and how

- Test with hard-coded inputs for which you know the output:
  - use simple but general cases
  - test special or boundary cases

```
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

- 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

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 optima, 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 `polyomial_expansion(data, degree)` with `quadratic_expansion(data)`
- Test optimized routines with brute-force approaches
  - Example: test `z = outer(x, y)` with

```
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 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?

DEMO

Code coverage

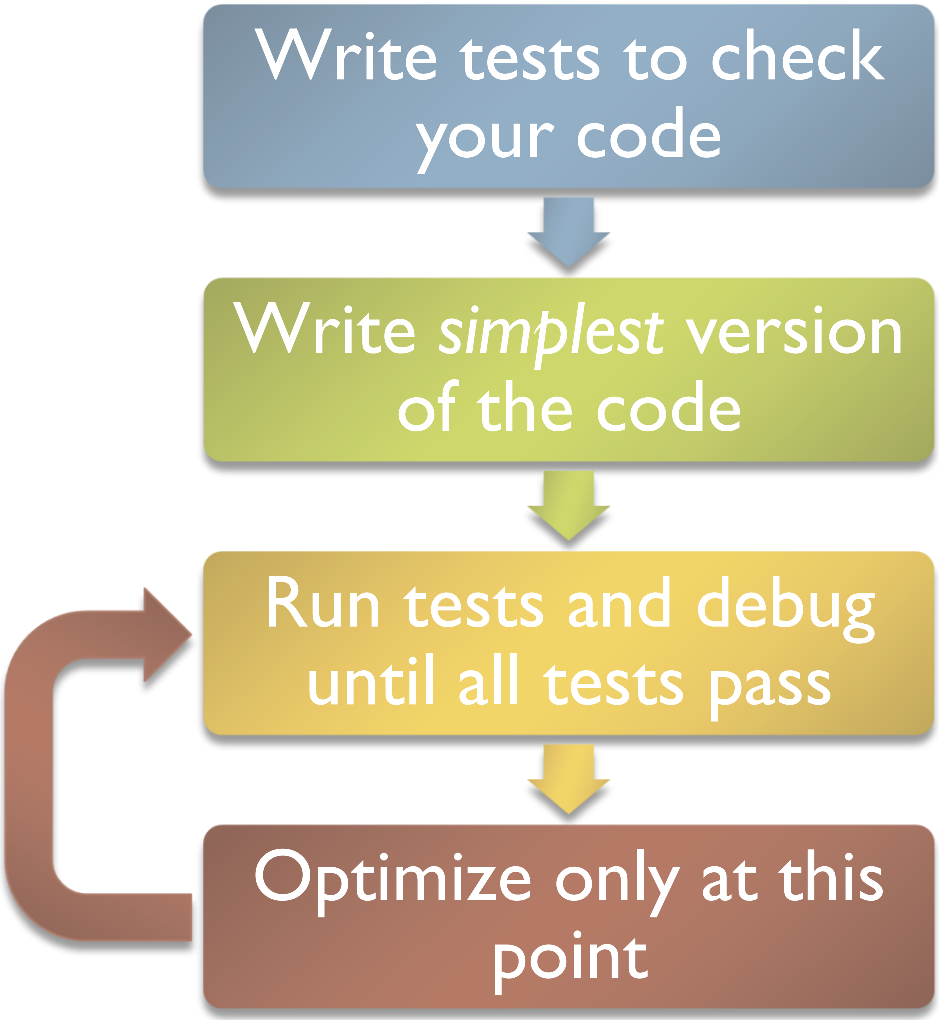
- It's easy to leave part of the code untested
  Classics: feature activated by keyword argument, exception raised for invalid input
- Coverage tools mark the lines visited during execution
- Use together with test framework to make sure all your code is covered

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

The basic agile development cycle



`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 -m pdb myscript.py
```

- Enter in a statement or function:

```
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`):

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

Entering the debugger from ipython

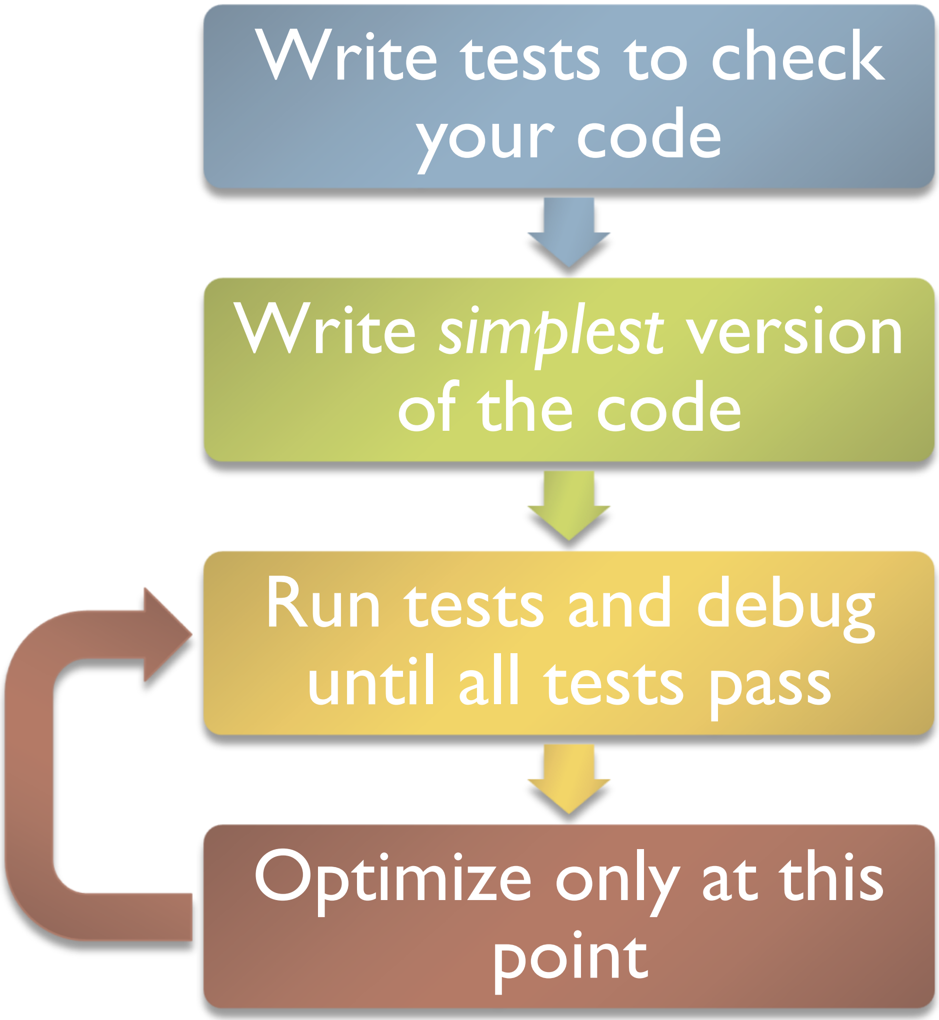
- From ipython:

  `%pdb` – preventive

  `%debug` – post-mortem

DEMO

The basic agile development cycle



`unittest`
`coverage.py`

`pdb`

`cProfile`
`timeit`
`runSnake`

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

```
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"])
```

`cProfile`, analyzing profiling results

- From interactive shell/code:

```
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
- High number of calls per functions
  => bad implementation? consider refactoring
- High time per call
  => consider caching
- High times, but valid
  => consider parallelizing or rewriting with Cython

DEMO

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
```

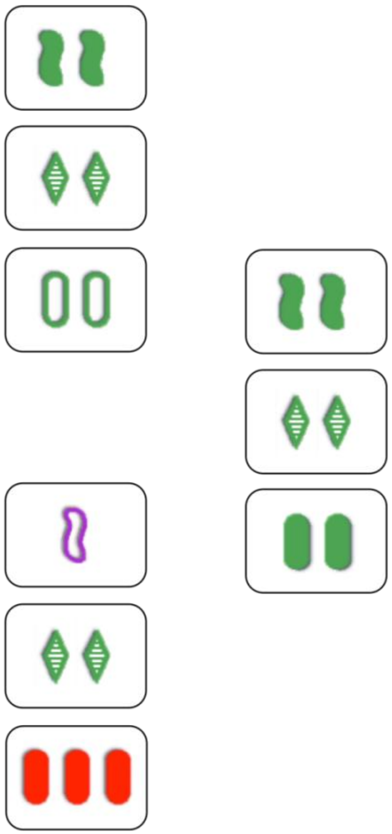
- In a script:

```
import doctest
doctest.testfile("example.txt”) # test examples in a file
doctest.testmod([module])      # test docstrings in module
```

DEMO

The End

- Exercises after the lunch break…



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