# Python Testing

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

November 14, 2011

# Outline

# `unittest` – Unit testing framework

## Description

- The Python unit testing framework (PyUnit) is a Python language version of JUnit.
- `unittest` supports:
  - test automation,
  - sharing of setup and shutdown code tests,
  - aggregation of tests into collections, and
  - independence of the tests from the reporting framework.
- `unittest` provides classes that make it easy to support these qualifies for a set of tests.

# `unittest` – Unit testing framework

## Concepts

- **test fixture**: A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.
- **test case**: A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.
- **test suite**: A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.
- **test runner**: A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

# A simple permutation class

## Example

```
class PermutationException(Exception):
    pass

class Permutation(object):
    def __init__(self, n):
        self._seq = range(n+1)

    def __str__(self):
        return ' '.join(str(x) for x in self._seq[1:])

    def __getitem__(self, i):
        if i <= 0 or i >= len(self._seq):
            raise PermutationException
        return self._seq[i]
```

# A simple permutation class

## Example

```
In [1]: from permutation1 import Permutation
In [2]: p = Permutation(5)
In [3]: print p
1 2 3 4 5
In [4]: p[1], p[2], p[3], p[4], p[5]
Out[4]: (1, 2, 3, 4, 5)
In [5]: p[0]
---------------------------------------------------------
PermutationException  Traceback (most recent call last)
...
In [6]: p[6]
---------------------------------------------------------
PermutationException  Traceback (most recent call last)
...
In [7]:
```

# Testing for equality

## `assertEqual`

`assertEqual(first, second, msg=None)`

- Test that first and second are equal. If the values do not compare equal, the test will fail.
- In addition, if first and second are the exact same type and one of list, tuple, dict, set, frozenset or unicode or any type that a subclass registers with `addTypeEqualityFunc()` the type specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods).

# Testing a simple permutation class

## Example

```
import unittest
from permutation1 import Permutation

class PermutationTest(unittest.TestCase):
    def test_correct_getitem(self):
        p = Permutation(5)
        for i in range(1, 6):
            self.assertEqual(p[i], i)

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

# Testing a simple permutation class

## Running the tests . . .

```
barbalala:$ python test-permutation1-a.py
----------
Ran 1 test in 0.000s

OK
barbalala:$
```

# Other tests

## `assertNotEqual`

`assertNotEqual(first, second, msg=None)`

- Test that first and second are not equal. If the values do compare equal, the test will fail.

## `assertTrue` and `AssertFalse`

`assertTrue(expr, msg=None)` **and** `assertFalse(expr, msg=None)`

- Test that expr is true (or false).
- Note that this is equivalent to `bool(expr)` is True and not to `expr is True` (use `assertIs(expr, True)` for the latter).
- This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

# What about exceptions?

## `assertRaises`

`assertRaises(exception)`

- The test passes if exception is raised, is an error if another exception is raised, or fails if no exception is raised.

# Testing a simple permutation class

## Example

```
import unittest
from permutation1 import Permutation, PermutationException

class PermutationTest(unittest.TestCase):
    def test_correct_getitem(self):
        p = Permutation(5)
        for i in range(1, 6):
            self.assertEqual(p[i], i)

    def test_exception_getitem0(self):
        with self.assertRaises(PermutationException):
            p = Permutation(5)
            p[0]

    def test_exception_getitem(self):
        with self.assertRaises(PermutationException):
            p = Permutation(5)
            p[6]

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

# Testing a simple permutation class

## Running the tests . . .

```
barbalala:$ python test-permutation1-b.py
Ran 3 tests in 0.000s

OK
barbalala:$
```

# A finer level of control

## Introducing suites

Instead of `unittest.main()`, there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line.

# Testing a simple permutation class

## Example

```
import unittest
from permutation1 import Permutation, PermutationException

class PermutationTest(unittest.TestCase):
    def test_correct_getitem(self):
        ...

    def test_exception_getitem0(self):
        ...

    def test_exception_getitem(self):
        ...

if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(PermutationTest)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

# Testing a simple permutation class

## Running the tests ...

```
barbalala:$ python test-permutation1-c.py
test_correct_getitem (__main__.PermutationTest) ... ok
test_exception_getitem (__main__.PermutationTest) ... ok
test_exception_getitem0 (__main__.PermutationTest) ... ok

----------
Ran 3 tests in 0.000s

OK
barbalala:$
```

# `setUp` and `tearDown`

## Desription

- The testing framework will automatically call for us `setUp()` when we run the test.
- We can provide a `tearDown()` method that tidies up after the runTest() method has been run

# Testing a simple permutation class

## Example

```
import unittest
from permutation1 import Permutation, PermutationException

class PermutationTest(unittest.TestCase):
    def setUp(self):
        print 'constructing a permutation of size 5'
        self.p = Permutation(5)

    def tearDown(self):
        print "not much to do ... but I'm here"

    def test_correct_getitem(self):
        ...

    def test_exception_getitem0(self):
        ...

    def test_exception_getitem(self):
        ...
```

# Testing a simple permutation class

## Running the tests . . .

```
barbalala:$ python test-permutation1-c.py
test_correct_getitem (__main__.PermutationTest) ...
constructing a permutation of size 5
not much to do ... but I'm here
ok
test_exception_getitem (__main__.PermutationTest) ...
constructing a permutation of size 5
not much to do ... but I'm here
ok
test_exception_getitem0 (__main__.PermutationTest) ...
constructing a permutation of size 5
not much to do ... but I'm here
ok
----------
Ran 3 tests in 0.000s
OK
barbalala:$
```

# Skipping tests and expected failures

## Example

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

# Skipping tests and expected failures

## Running the tests . . .

```
test_format (__main__.MyTestCase) ...
skipped 'not supported in this library version'

test_nothing (__main__.MyTestCase) ...
skipped 'demonstrating skipping'

test_windows_support (__main__.MyTestCase) ...
skipped 'requires Windows'

----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK (skipped=3)
```

## Description

The `doctest` module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use `doctest`

- To check that a modules docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of literate testing or executable documentation.

# Doctest

## Example

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
```

# Doctest

## Example

```
    """
    ...
    >>> factorial(30)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000L
```

# Doctest

## Example

```
    """
    ...
    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0: raise ValueError("n must be >= 0")
    if math.floor(n) != n: raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result
```

# Doctest

## Example

```
# main program
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

## Running . . .

```
barbalala:$ python factorial.py
barbalala:$
```

Theres no output! Thats normal, and it means all the examples worked

# Verbose doctest

## Example

```
$ python factorial.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    [factorial(long(n)) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

# Verbose doctest

## Example

```
Trying:
    factorial(30)
Expecting:
    265252859812191058636308480000000L
ok
Trying:
    factorial(30L)
Expecting:
    265252859812191058636308480000000L
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0
ok
```

# Verbose doctest

## Example

```
Trying:
    factorial(30.1)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
ok
Trying:
    factorial(30.0)
Expecting:
    265252859812191058636308480000000L
ok
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
```

# Verbose doctest

## Example

```
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

# Checking Examples in a Text File

## Example

```
import doctest
doctest.testfile("example.txt")
```

## example.txt

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from factorial import factorial

Now use it:

    >>> factorial(6)
    120
```

# Checking Examples in a Text File

## Running . . .

```
barbalala:$ python runningfromexample.py
**********************************************************************
File "example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
**********************************************************************
1 items had failures:
   1 of   2 in example.txt
***Test Failed*** 1 failures.
barbalala:$
```