Unit testing with Python

Source: http://rkd.zgib.net/scicomp/testing-pyladies/testing-unittest.rst

About this talk

This talk discusses the basic concepts of unit testing in Python from a practical side. It is designed not only as an introduction to unit testing in Python, but unit testing in general.

Outline

- First, we will discuss what testing is and why we would want to do it.
- Then, we will discuss basic unit testing frameworks in Python.
- Finally, we will discuss tips and tricks of the `unittest` framework.

Testing is considered one of the cornerstones to good software.

- Benefits according to wikipedia:
  - Find problems early
  - Find regressions when you make big change
  - Simplifies integration
  - Documentation
  - Design
- This talk is about systematically doing so **automatically** and **systematically**, instead of just testing only while developing.
- The key to making testing work is *balance*.

Test driven development

https://en.wikipedia.org/wiki/Test-driven_development

- Testing taken to the extreme
- You write the tests first, then write code to make the test pass.
- Nothing exists without a test.
- You can feel free to change anything and not think about it, as long as the tests pass you are good to go.

Examples of unit tests of different libraries

Let's look at these existing projects and what they do:

- python - http://hg.python.org/cpython/file/tip/Lib/test
  - never-ending screens of huge files.
- networkx - https://github.com/networkx/networkx/tree/master/networkx
  - All tests collected in tests/ directory in each module directory.
- sqlite3 - http://sqlite.org/testing.html
- django web framework - https://docs.djangoproject.com/en/1.6/topics/testing/
  - Frameworks need frameworks for testing, too.

Where to put tests?

- General practice is to put your tests in a module separate from the main code itself.
- The name should include the word "test" at the beginning (to be automatically discoverable automatically by most test runners).
- My recommendation: either call it `NAME_test.py` or `test_NAME.py`. I use the second.
- The modules must be importable (no side effects when run, if importing has any side-effects they should be put in a `if __name__ == "__main__":)` block.

Python `unittest` module

- Basic unit testing framework included in the standard library (based on JUnit/xUnit).
- It is (perhaps too much?) object-oriented and only basic interface to running tests.
- It is a base that other frameworks build on, but unless you need complex inheritance; of setup/teardown code, you probably don't need to write using this.

Basic usage:

- Running module: `python test_NAME.py`
- Discovering all tests in modules named `test*.py` and running them: `python -m unittest discover`

unittest example

```
class TestSequenceFunctions(unittest.TestCase):

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elements
    seq = list(range(10))
        random.shuffle(seq)
        self.seq.sort()
        self.assertEqual(seq, range(10))

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

nosetests framework

- Framework to make running tests easier.
- You can write tests with simple functions.
- Nicer command line interface, that can also do things like automatically start *pdb*.

```
$ nosetests
$ nosetests test_NAME.py
$ nosetests test_NAME.py:test1
$ nosetests --pdb                # start pdb if there are fai
```

Standard output is hidden by default, unless a test fails! Use `-s` to make all standard output be shown.

nose example

```
from nose.tools import *

def test_sorted():
    seq = list(range(10))
    random.shuffle(seq)
    seq.sort()
    assert_equal(seq, range(10))
```

Basic atoms of unit tests

- **Assertion**: wiktionary: a condition expected to be true at a particular point.
- **Test functions:** Code that does stuff and makes **assertions** about expected results.
- **setup** / **teardown**: Code that produces initial data structures/frees resources before/after tests.
- **Test classes:** Combines functions and setup/teardown, allows you to use more inheritance to simplify writing if needed.
- **Mock objects:** Objects which simulate an interface to facilitate testing.

Assertions

- The fundamental unit of a test. One test function or method can have many assertions in it.
- Use `assertions functions` that do the following:
  - Compare the arguments according to some rules to verify the assertion.
  - If the condition is false, raise `AssertionError` and print some useful error message.

Assertion example

Example:

- You can simply use the `assert` keyword:

```
assert func(5) == 1, "function is not 1"
```

- For better detail, you can use special assertion functions:

```
>>> self.assertEqual(set([1, 2, 3]), set([1, 2, 4]) )

AssertionError: Items in the first set but not the second:
3
Items in the second set but not the first:
4
```

Look at how it prints exactly what the difference is. It combines testing and "print debugging".

What assertions are available?

See the list of `assert*` methods at https://docs.python.org/library/unittest.html#assert-methods

- These standard library assertions are *methods* of the `TestCase` class, and thus you have to use `unittest` to have these.

List of Assertions available

```
assertAlmostEqual
assertAlmostEquals
assertDictContainsSubset
assertDictEqual
assertEqual
assertEquals
assertFalse
assertGreater
assertGreaterEqual
assertIn
assertIs
assertIsInstance
assertIsNone
assertIsNot
assertIsNotNone
assertItemsEqual
assertLess
assertLessEqual
assertListEqual
assertMultiLineEqual
assertNotAlmostEqual
assertNotAlmostEquals
assertNotEqual
```

```
assertNotEquals
assertNotIn
assertNotIsInstance
assertNotRegexpMatches
assertRaises
assertRaisesRegexp
assertRegexpMatches
assertSequenceEqual
assertSetEqual
assertTrue
assertTupleEqual
assert_
```

Full example: A working test (permutations)

Get the perm.py and test_perm_ut.py files from the repository. This is a simple permutations function.

Instructions:

- Run these unit tests (`python test_perm_ut.py`).
- Write a *factorial* function.
- ... and test for that factorial function.

How to debug a failing test

- Is the test correct? (side point: do you make tests for tests?)
- Run just that one test: `python module_name.py ClassName.MethodName`.
- Use the debugger (next slide), add in print statements, or debug however you normally do.

Testing and debugging

- When using other testing packages, you can give options like `--pdb` to cause the Python debugger to start when there are exceptions (or failures).
- This is not easy with `unittest` without weird hacks, so this talk excludes it (slides moved to the end).

Testing and debugging

- When using other testing packages, you can give options like `--pdb` to cause the Python debugger to start when there are exceptions (or failures).
- This is not easy with `unittest` without weird hacks, so this talk excludes it (slides moved to the end).

Example: Test inheritance (Fibonacci numbers)

- fib.py contains two functions to calculate the nth Fibonacci number.
- In test_fib.py you see a class-based method of testing both the functions. This module compatible with both `unittest` and `nose`.
    - Notice that both functions are expected to pass the exact same tests. This is a case of using inheritance to simplify writing.

Instructions:

- Use `nosetests` to run `TestFib1` only. Does it pass?
- Use `nosetests` to run `TestFib2` only. Does it pass?
- If any don't pass, use `--pdb` or `--pdb-fail` to examine the situation, if you think it will help.
- Fix the problem until the test suite passes.

Example: Test-driven development (counting function)

- A function that returns the counts of items in an iterable as a dictionary.
  - Example: `[1, 1, 5, ]  -->  {1:2, 5:1}`
- Get count.py and test_count_ut.py from the repository.

Instructions:

- Run the test module. Notice it fails because `count.py` is empty but there is one test.
- Write a `count` function to make the test pass.
- Do the following over and over until you are satisfied:
  - Think: What else should this function return (hint: the example above)
  - Write a test script for that example.
  - Run the test script: notice it fails.
  - Fix the function so that it passes.

Recommendations for making tests

- Think about what axes can be used to simplify the problem. For example, if the problem scales as a function of n, write tests for low n where the solution is easily checked in your head.
- Try to think of all important boundary cases to handle.
- Testing is easiest for `pure functions`: the return value depends only on arguments and the function does not have any side effects.
- You will be tempted to run the code over and over during development as part of your iterative development cycle. Instead,
    - Put it in a test instead - it's the same amount of work.
    - If there is an exception or `AssertionError`, then use `--pdb` or `--pdb-fail` to drop to a Python shell at that point and figure out what the problem is.
- Have two windows open: one with the editor, and one to run `nosetests` over and over again.

Conclusions

- Testing is a concept that spans all languages and programming paradigms.
- Tests should be:
  - Fast
  - Automatic
  - Extensive
- We have looked at the `unittest` and `nose` frameworks for testing in Python.
- Many standard development processes integrate into testing, and can save you a lot of time: debugging, profiling, release, ...

Extensions we haven't covered

- Testing non-pure functions: You'll need to make initial data, run function, and test side-effects.
    - **Mock objects** can be used to test the effect a function has on another object. `unittest.mock` and other libraries automate this.
- Code coverage: automatic tools to show you what lines have been run by tests.
- Levels of testing: unit testing, integration testing, system testing, etc.
- Doctests: tests in docstrings automatically run. Serve as documentation.

Extra: Invoking the python debugger (not with `unittest`)

If a test fails, you can automatically invoke the debugger:

- `nosetests --pdb` starts pdb when there is an exception or assertion failure.

  Note: for older versions, you must use `-pdb` or `--pdb-failures`.

Useful pdb commands:

- `l` or `list` - list lines of code around the point
- `bt` or `backtrace` - list full call stack.
- `u` or `up` and `d` or `down` - Go up or down the call stack
- `p` or `print` - print any variable or expression
- Any other input: evaluate that line at that point (i.e. evaluate an expression).

Full list of commands: https://docs.python.org/2/library/pdb.html#debugger-commands

Invoking the python debugger (unittest)

- There is not an automatic way to do this with `unittest`.
- Option 1) add `import pdb ; pdb.set_trace()` in the function before the error you want to debug.
- Option 2) To emulate `--pdb` of nose, you can monkey-patch `unittest` to make it work, by adding this line before unittest.main():

```
import unittest; unittest.TestCase.run = lambda self,*args,**kw:
```

Then, run the test under `pdb`:

```
$ pdb test_NAME.py
```

Example: Debugging (prime numbers) (not with `unittest`)

- prime.py contains a function for testing for primality of numbers.
- Run test_prime_ut.py in nosetests.
- When it fails, add the magic line from the last slide and run with `pdb` instead of `python` to invoke the debugger and examine the situation.
- Try to fix the line in the debugger so that it works.
- Copy your fix to the module, then repeat.