Testing

Olav Vahtras

Leiden 2015-04-13

Why testing?

Why testing?

- Make sure the code is correct.

Why testing?

- Make sure the code is correct.
- Testing is a design tool

Why testing?

- Make sure the code is correct.
- Testing is a design tool
- Testing is a documentation tool

TDD = Test-Driven Development

What's special about TDD

There are several aspects

What's special about TDD

There are several aspects

- It makes for a better design

What's special about TDD

There are several aspects

- It makes for a better design
- Code will never be written that is difficult to test

What's special about TDD

There are several aspects

- It makes for a better design
- Code will never be written that is difficult to test
- Code tends get better structure

What's special about TDD

There are several aspects

- It makes for a better design
- Code will never be written that is difficult to test
- Code tends get better structure
- Unconciously you do not want to find bugs - testing existing code tends to focus on the parts which work best.

What's special about TDD

There are several aspects

- It makes for a better design
- Code will never be written that is difficult to test
- Code tends get better structure
- Unconciously you do not want to find bugs - testing existing code tends to focus on the parts which work best.
- It's what professionals do

What's special about TDD

There are several aspects

- It makes for a better design
- Code will never be written that is difficult to test
- Code tends get better structure
- Unconciously you do not want to find bugs - testing existing code tends to focus on the parts which work best.
- It's what professionals do
- Because it saves time and money

What is TDD in practice?

To add new functionality, e.g. a function

- Make up in your mind what the function should do
- What input
- What output
- Do not code the function

What is TDD in practice?

To add new functionality, e.g. a function

- Make up in your mind what the function should do
- What input
- What output
- Do not code the function

Initialize

- First Write a test that
    - Calls the function
    - Compares the actual output to the desired output
    - Report if they differ

The work cycle

- Run the test

The work cycle

- Run the test
- Did it fail?

The work cycle

- Run the test
- Did it fail?
    - The first time it fails because the function does not exist

The work cycle

- Run the test
- Did it fail?
  - The first time it fails because the function does not exist
  - Add code to the function with the purpose of passing the test and nothing more

The work cycle

- Run the test
- Did it fail?
  - The first time it fails because the function does not exist
  - Add code to the function with the purpose of passing the test and nothing more
  - Start over

The work cycle

- Run the test
- Did it fail?
    - The first time it fails because the function does not exist
    - Add code to the function with the purpose of passing the test and nothing more
    - Start over
- Did it pass?

The work cycle

- Run the test
- Did it fail?
    - The first time it fails because the function does not exist
    - Add code to the function with the purpose of passing the test and nothing more
    - Start over
- Did it pass?
    - Stop

The work cycle

- Run the test
- Did it fail?
    - The first time it fails because the function does not exist
    - Add code to the function with the purpose of passing the test and nothing more
    - Start over
- Did it pass?
    - Stop
    - Do not write more code. You are done.

Tools

Tools for testing python code

- doctest: a simple way of including tests in a doc-string of a function
- unittest: a module of the standard python library to provide advanced testing
- nosetests: a commonly used third-party tool for running tests

Doctest

- Simple test cases can be provided as part of the documentation string
- Cut and paste an interactive session known to give true result
- The doctest module executes the example from the interactive session as a test case
- Test on string output and not values - small changes in formatting will case tests to fail

Example

```
    #calculate.py
    def add(a, b):
        """Return sum of two arguments
        >>> add(1, 1)
        2
        >>>
        """
        return a + b

    def sub(a, b):
        """Return difference of two arguments
        >>> sub(1, 1)
        0
        """
        return a + b
```

can you see the bug?

Running doctest

At the end of the file

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

On the command line

```
$ python calculate.py
```

- All code in the file is executed
    - Functions are defined
    - `__name__ == "__main__"` evaluates to `True`
    - Test are run

The output

```
$ python calculate.py
**********************************************************************
File "calculate.py", line 14, in __main__.sub
Failed example:
    sub(1, 1)
Expected:
    0
Got:
    2
**********************************************************************
1 items had failures:
   1 of   1 in __main__.sub
***Test Failed*** 1 failures.
```

Correct the bug

```
        return a + b -> return a - b
```

Rerun

```
    $ python calculate.py
    $
```

silent - all ok

Conclusion - doctests

- Very easy to include testning into your code
- The test serves as documentation as well
- Typically tests only one aspect of the function
- But could clutter your code and may not be the best for extensive testing
- Extensive testing is best separated from production code
- More information on http://docs.python.org/library/doctest

The `unittest` module

Unit testing

Unit testing in program development refers to testing the behaviour of smallest possible units of code in a program with a well defined task

- A unit test module exist for this purpose: `unittest`
- The tests can be written in a separate file
- One defines a class which is a subclass of `unittest.TestCase`
- The unittest framework executes and checks everything that begins with test
- Part of the standard library and provides very portable testing

Howto

- Define a class with a name beginning with `Test` as a subclass of `unittest.TestCase`
- Define class methods that begin with `test` using the test functions of the `unittest` module
- Optionally one may define a `setUp` and a `tearDown` method which are run before and after every test.
- In the main section run `unittest.main()`

```
class TestSomething(unittest.TestCase):
    ...
    def test_this(self):
        ...
    def test_that(self):
        ...
if __name__ == "__main__":
    unittest.main()
```

Example

```
#test_calculate.py
import unittest
import calculate

class TestCalculate(unittest.TestCase):

    def testadd(self):
        res = calculate.add(1, 1)
        self.assertEqual(res, 2)

    def testsub(self):
        res = calculate.sub(1, 1)
        self.assertEqual(res, 0)

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

Run test

```
$ python test_calculate.py
.F
======================================================================
FAIL: testsub (__main__.TestCalculate)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_calculate.py", line 18, in testsub
    self.assertEqual(res, 0)
AssertionError: 2 != 0

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Run verbose test

```
$ python test_calculate.py -v
testadd (__main__.TestCalculate) ... ok
testsub (__main__.TestCalculate) ... FAIL

======================================================================
FAIL: testsub (__main__.TestCalculate)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_calculate.py", line 18, in testsub
    self.assertEqual(res, 0)
AssertionError: 2 != 0

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Fix the bug

```
    return a + b -> return a - b
```

Fix the bug

```
    return a + b -> return a - b
```

Rerun test

```
    $ python test_calculate.py -v
    testadd (__main__.TestCalculate) ... ok
    testsub (__main__.TestCalculate) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Other helper functions tests

- `assertNotEqual`
- `assertTrue`
- `assertFalse`
- `assertAlmostEqual`
  - Most numerical testing is within a threshold, e.g.

```
    def testdiv(self):
        res = calculate.div(1., 3)
        self.assertAlmostEqual(res, 0.333333, 6)
```

- see also http://docs.python.org/2/library/unittest.html

`nosetests`

Another testing framework

- Nosetests is a third-party unit-testing tool for python (from http://ivory.idyll.org/articles/nose-intro.html)
- It looks for all tests in the current directory (and subdirectories and executes functions containing test)
- It is compatible with the `unittest` framework so it executes those tests as well
- Not as strict about setting up tests (as class members)
- nosetests understands unittest style tests and executes them as well.
- Without arguments all test are carried out which it can find
- It couples to the python debugger
- It supports **coverage** - shows which lines of codes were not executed during the tests

Example

```
#testdiv_alt.py
from calculate import div

def test_div3():
    """Test integer division"""
    res = div(1, 3)
    assert res == 0

def test_div4():
    """Test floating point division"""
    res = div(1., 3)
    assert abs(res - 0.333333) < 1e-6
```

Running `nosetests`

```
$ nosetests  testdiv_alt.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Each dot represents a passed test, an `F` is a failed test

or verbose

```
$ nosetests -v testdiv_alt.py
Test integer division ... ok
Test floating point division ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

Note: docstring is used in the error report

Nosetests and the debugger

- By running nosetests with a debug option, it runs all tests.
- When a test fails the program stops and launches the debugger where the error condition was detected

```
$ nosetests  test_calculate.py  --pdb
.> /usr/lib/python2.7/unittest/case.py(508)_baseAssertEqual()
-> raise self.failureException(msg)
(Pdb)
```

Once in the debugger it is possible to examine variables, execute functions

```
(Pdb) print res
2
(Pdb) print calculate.sub(2, 1)
3
```

Nosetest and coverage

coverage is a relative measure of how many of your lines of codes have been executed during the tests

```
$ nosetests test_calculate.py --with-coverage
.F
======================================================================
FAIL: testsub (test_calculate.TestCalculate)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/olav/Dropbox/Python/hieroglyph/test_calculate.py", line 18, in testsub
self.assertEqual(res, 0)
AssertionError: 2 != 0

Name        Stmts   Miss  Cover   Missing
-----------------------------------------
calculate      10      3    70%   25, 28-29
----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

We get a list over all modules that have been executed and, how many lines, and which lines that we missed

note: In this case there was a function not tested.

Recommendation

- Use `doctest` for small illustrations, if any
- Use `unittest` to code your tests,
- Use `nosetests` to execute your tests, optionally with debugging and coverage

Final tip

- Embrace the TDD philosphy, write test before code.
- Document code and modules - be kind to your future self.
- For good programming style, consider PEP 8, http://www.python.org/dev/peps/pep-0008/
- Be obsessive about testing
- If your test code is larger that your production code, you are on the right track
- This takes initially a little more time but the rewards in the long run are huge