An Intro to Unit Testing in Python

and

PyTest framework features

–by

Arulalan.T

arulalant@gmail.com

Project Associate,

Centre for Atmospheric Sciences ,

Indian Institute of Technology Delhi.

18.04.2011

Table of Contents

Unit testing in Python

To understand about unit testing briefly, kindly start to study it from “Dive into Python/unit_testing”. This is very short notes about an intro to unit testing in python and why we chooses “Pyunit” as unit test framework in our project.

Unit test module comes in the default python from 2.2 version itself. We no need to install anything for simple unit-testing.

Lets take simple mod function going to be written by the developer. For that we have to write the unit testing code first. Then only the developer should start to write the coding for mod function. usually function developer and unit test developer should be the same person. Some cases it may change.

In general the quality of function code to be checked/passed by the unit test. If both the unit-testing code and functional code should be written by the same developer, then they will love to write the unit testing. Unit test improves the quality of the functional code and make it as standard one.!.

Phase – 0 :

The zeroth phase of the unit test is, we have to decide what we are going to test ?
In our example, we are going to write the unit test for mod function which is the part of the module called divisible.py .

•Mod function should return the correct value. It may checked by passing the random input or known values.

•Mod function should raise the Zero denominator error if user will pass the Zero as denominator value while calling mod function.

•Mod function should raise the String input error if user will pass any one of the input as string instead of integer or float value while calling mod function.

This is what the zeroth phase in the unit test development.

Phase – 1 :

Now we are in next stage. We have to write unit test code in python to check the functional code which is going to be written in python.

Lets consider the “divisible.py” module is the proper functional code. “divisible-testing.py” module is the unit testing code for the “divisible.py” module. Make sure that we have to start coding for unittesting not with function.

Here we written the “divisible-testing.py” module. It follows the 3 cases which has defined in the phase-0.

1.1 Unit test code in unit-test framework

```
#divisible-testing.py

import unittest
import divisible # divisible module going to be written in future
import random

divide_instance = divisible.Divide()
class Divide(unittest.TestCase):
        # fn1
        def testRandomMod(self):
                ''' testing for mod operation '''
                result = divide_instance.mod(5,2)
                self.assertEqual(1,result)

class ToDivideBadInput(unittest.TestCase):
        # fn2
        def testZerodenominator(self):
                ''' to mod operation should fail when zero has passed in the denominator field '''
                self.assertRaises(divisible.ZerodenominatorError,divide_instance.mod,1,0)
        # fn3
        def testStringdenominator(self):
                ''' to mod operation should fail when string has passed in the denominator field '''
                self.assertRaises(divisible.StringInputError,divide_instance.mod,1,'')
        # fn4
        def testStringNumerator(self):
                ''' to mod operation should fail when string has passed in the numerator field '''
                self.assertRaises(divisible.StringInputError,divide_instance.mod,'',1)


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


#end of divisble-testing.py
```

The above one is our simple unit testing for mod function. In our unittest classes we have to inherit the unittest.TestCase class which is the written in the unittest.

Note : make sure that all the test case function should starts-with “test”. Then only the unit test will take that testfunction fot testing. Otherwise it will neglect that function while doing testing.

In # fn1 we are calling the mod function which is in the part of the 'divisible' module and it should be the instance of the Divide() class. Here we are passing the known numbers as the argument to the mod method. Using self.assertEqual function, we are checking either the known result and returned value of the mod function are should be the same or not.

If the case both the known result and returned is not equal, then the mod function should failure. From this the developer should come to know, the function code is not written in well.

Now we can replace the # fn1 by the following one. Here we are given the known input is chooses by random.

```
        # fn1
        def testRandomMod(self):

                ''' testing for mod operation '''
```

```
numerator = random.randrange(10)
denominator = random.randint(1,20)
remainder = numerator % denominator
result = divide_instance.mod(numerator,denominator)
self.assertEqual(remainder,result+result)
```

Note : Each testing method should have the docstring. So that while doing test, we can see the testing result easily by the doc-string.

In # fn2 we are going to test the zero dividend error by either the mod function should raise the appropriate error using raise in python or not. assertRaises(divisible.ZerodenominatorError,divide_instance.mod,1,0) here we are passing the args 1 as numerator and 0 as denominator to the mod function.

And while passing this as input to the mod function, it should raise the ZerodenominatorError. Here assertRaises will check either that corresponding error is raised by the mod function while passing 1 & 0 as input, or not. If mod fn will raise this error means this test is passed. Otherwise its failed.

In # fn3 & # fn4 also the same kind of stuff used to check and raise the appropriate errors while passing the wrong inputs to the mod function.

End of this test story :

We have written our unit-testing code for our functional code.

Phase – 2 :

1.2 Functional code in unit-test framework

In our functional code, we are importing the Exception in our own error handling classes. In this example, we are just written the need error handling class which are similar in the unittest of this module.

```
# divisible.py
class DivisibleError(Exception): pass
class ZerodenominatorError(DivisibleError) : pass
class StringInputError(DivisibleError) : pass
 class Divide():
        def mod(self,numerator,denominator):
                ''' finding the mod value '''

                if isinstance(numerator,str):
                        raise StringInputError, ' numerator has passed as string '

                if isinstance(denominator,str):
                        raise StringInputError, ' denominator has passed as string '

                if denominator == 0 :
                        raise ZerodenominatorError, ' Zero has passed for denominator '

                return numerator % denominator
# end of divisible.py
```

In the above mod function of the class Divide(), we have defined all the error raise handling correctly.

Phase – 3 :

1.3 Testing result in unit-test framework

Testing “divisible.py” code by 'divisible-testing.py”

```
arulalan@arulalan:~/$ python divisible-testing.py -v
testing for mod operation ... ok
to mod operation should fail when string has passed in the denominator field ... ok
to mod operation should fail when string has passed in the numerator field ... ok
to mod operation should fail when zero has passed in the denominator field ... ok

---------------------------------------------------------------------- ----------------------
Ran 4 tests in 0.000s

OK
```

We successfully passed all the tests conducted by unittesting. Done .!. So our 'divisible.py' code is standard one.

Pytest Framework for unit testing In Python

There are many testing frameworks are available in the open source world for all the lanugage. In our python itself having more testing tools and frameworks is exists. To know about that go to http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy or search in Internet.

2.1 How to install Pytest :

$ *sudo easy_install -U py*
$ *sudo easy_install -U pytest*

-U for upgrade to the latest version of the python packages.

To use easy_install, we need to install $ *sudo apt-get install python-setuptools*

In this framework, the phase-0 and phase-2 should be the same. But phase-1 & phase-3 only may going to change in the syntax in terms of simplicity.

Phase – 1 in PyTest :

2.2 Unit test code in PyTest framework

```
# divisible-testing.py
import pytest
import divisible
```

```
import random

divide_instance = divisible.Divide()
class TestDivide():
        def test_RandomMod(self):
                ''' testing for mod operation '''
                pytest.skip("unsupported configuration")
                numerator = random.randrange(10)
                denominator = random.randint(1,20)
                remainder = numerator % denominator
                result = divide_instance.mod(numerator,denominator)
                assert remainder == result

class TestToDivideBadInput():
        def test_Zerodenominator(self):
                ''' to mod operation should fail when zero has passed in the denominator field '''
                pytest.raises(divisible.ZerodenominatorError,divide_instance.mod,1,0)

        def test_Stringdenominator(self):
                ''' to mod operation should fail when string has passed in the denominator field '''
                pytest.raises(divisible.StringInputError,divide_instance.mod,1,'')

        def test_StringNumerator(self):
                ''' to mod operation should fail when string has passed in the numerator field '''
                pytest.raises(divisible.StringInputError,divide_instance.mod,'',1)

#end of divisible-testing.py
```

In the above pytest module very few syntax has been changed compare to normal unittesting.

Changes from “self.assertEqual(somevalue,somevalue)” to “assert somevalue == somevalue ” and “self.assertRaises(...)” to “pytest.raises(...)” in the pytest unittesting module. Then everything else are same in the concept wise.

In Pytest we have few more advantages like, *testing needed classes or methods, skipping any classes/methods , expecting xfail while testing some methods and more ...*

> Note : In pytest framework , we have mentioned the naming convention of all the classes and methods name either should startswith “test_” or endswith “_test”. Then only the pytest should call and run the testing functions.

Phase–3 in PyTest :

2.3 Testing result in PyTest framework

Testing “divisible.py” code by 'divisible-testing.py”

```
arulalan@arulalan:~/$ py.test divisible-testing.py -v
============================= test session starts ==============================
platform linux2 -- Python 2.6.6 -- pytest-2.0.2 -- /usr/bin/python
collected 4 items

divisible-testing.py:18: TestDivide.test_RandomMod PASSED
divisible-testing.py:30: TestToDivideBadInput.test_Zerodenominator PASSED
divisible-testing.py:34: TestToDivideBadInput.test_Stringdenominator PASSED
divisible-testing.py:38: TestToDivideBadInput.test_StringNumerator PASSED

========================== 4 passed in 0.01 seconds ==========================
```

Note : we have used **py.test** is the execute command to conduct the testing which is written using pytest framework.

Pytest Usages and Advantages for unit testing in Python

3.1 Getting help on version, option names, environment variables

```
$ py.test --version   # shows where pytest was imported from
$ py.test --funcargs  # show available builtin function arguments
$ py.test -h | --help # show help on command line and config file options
```

3.2 Stopping after the first (or N) failures

To stop the testing process after the first (N) failures:

```
$ py.test -x            # stop after first failure
$ py.test -maxfail=2    # stop after two failures
```

3.3 Specifying tests / selecting tests

Several test run options:

```
$ py.test test_mod.py   # run tests in module
$ py.test somepath      # run all tests below path
$ py.test -k string     # only run tests whose names contain a string
```

Import ‘pkg’ and use its filesystem location to find and run tests:

```
$ py.test --pyargs pkg # run all tests found below directory of pypkg
```

3.4 Dropping to PDB (Python Debugger) on failures

Python comes with a built-in Python debugger called PDB. py.test allows to drop into the PDB prompt via a command line option:
$ py.test –pdb

This will invoke the Python debugger on every failure. Often you might only want to do this for the first failing test to understand a certain failure situation:

$ py.test -x --pdb # drop to PDB on first failure, then end test session
$ py.test --pdb --maxfail=3 # drop to PDB for the first three failures

3.5 Setting a breakpoint / aka set_trace()

If you want to set a breakpoint and enter the pdb.set_trace() you can use a helper. Using pdb we can fall into shell at that line itself, so we can easily traceback by typing and giving inputs.

```
import pytest
def test_function():
    ...
    pytest.set_trace()    # invoke PDB debugger and tracing
```

3.6 creating resultlog format files

To create plain-text machine-readable result files you can issue:
$ py.test --resultlog=path

Skipping and Xfail testes in Pytest

4.1 Marking a test function to be skipped

Here is an example of marking a test function to be skipped when run on a Python3 interpreter :

```
import sys
@pytest.mark.skipif("sys.version_info >= (3,0)")
def test_function():
    ...
```

During test function setup the skipif condition is evaluated by calling eval('sys.version_info >=(3,0)', namespace). (*New in version 2.0.2*) The namespace contains all the module globals of the test function so that you can for example check *for versions of a module you are using:*
import mymodule

```
@pytest.mark.skipif("mymodule.__version__ < '1.2'")
def test_function():
    ...
```

The test function will not be run (“skipped”) if mymodule is below the specified version. The reason for specifying the condition as a string is mainly that py.test can report a summary of skip conditions. For information on the construction of the namespace see evaluation of skipif/xfail conditions.

You can of course create a shortcut for your conditional skip decorator at module level like this:

```
win32only = pytest.mark.skipif("sys.platform != 'win32'")
```

```
@win32only
def test_function():
    ...
```

4.2 skip all test functions of a class

As with all function *marking* you can skip test functions at the whole class- or module level. Here is an example for skipping all methods of a test class based on the platform:

```
class TestPosixCalls:
    pytestmark = pytest.mark.skipif("sys.platform == 'win32'")

    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

The pytestmark special name tells py.test to apply it to each test function in the class. If your code targets python2.6 or above you can more naturally use the skipif decorator (and any other marker) on classes:

```
@pytest.mark.skipif("sys.platform == 'win32'")
class TestPosixCalls:

    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

Using multiple "skipif" decorators on a single function is generally fine - it means that if any of the conditions apply the function execution will be skipped.

4.3 Mark a test function as expected to fail

You can use the *xfail* marker to indicate that you expect the test to fail:

```
@pytest.mark.xfail
def test_function():
    ...
```

This test will be run but no traceback will be reported when it fails. Instead terminal reporting will list it in the "expected to fail" or "unexpectedly passing" sections.

By specifying on the command line:

```
$ pytest --runxfail
```

you can force the running and reporting of an xfail marked test as if it weren't marked at all.

As with skipif you can also mark your expectation of a failure on a particular platform:

```
@pytest.mark.xfail("sys.version_info >= (3,0)")
def test_function():
    ...
```

You can furthermore prevent the running of an "xfail" test or specify a reason such as a bug ID or similar. Here is a simple test file with the several usages:

```
import pytest
xfail = pytest.mark.xfail

@xfail
def test_hello():
    assert 0
```

```
@xfail(run=False)
def test_hello2():
    assert 0

@xfail("hasattr(os, 'sep')")
def test_hello3():
    assert 0

@xfail(reason="bug 110")
def test_hello4():
    assert 0

@xfail('pytest.__version__[0] != "17"')
def test_hello5():
    assert 0

def test_hello6():
    pytest.xfail("reason")
```

Running it with the report-on-xfail option gives this output:

```
$ py.test -rx xfail_demo.py
============================ test session starts ============================
platform linux2 -- Python 2.6.6 -- pytest-2.0.2
collecting ... collected 6 items

xfail_demo.py xxxxxx
========================= short test summary info ==========================
XFAIL xfail_demo.py::test_hello
XFAIL xfail_demo.py::test_hello2
  reason: [NOTRUN]
XFAIL xfail_demo.py::test_hello3
  condition: hasattr(os, 'sep')
XFAIL xfail_demo.py::test_hello4
  bug 110
XFAIL xfail_demo.py::test_hello5
  condition: pytest.__version__[0] != "17"
XFAIL xfail_demo.py::test_hello6
  reason: reason

======================== 6 xfailed in 0.04 seconds =========================
```

4.4 Evaluation of skipif/xfail expressions

The evaluation of a condition string in pytest.mark.skipif(condition-string) or pytest.mark.xfail(condition-string) takes place in a namespace dictionary which is constructed as follows:

- the namespace is initialized by putting the sys and os modules and the pytest config object into it.
- updated with the module globals of the test function for which the expression is applied.

The pytest config object allows you to skip based on a test configuration value which you might have added:

```
@pytest.mark.skipif("not config.getvalue('db')")
def test_function(...):
    ...
```

4.5 Imperative xfail from within a test or setup function

If you cannot declare xfail-conditions at import time you can also imperatively produce an XFail-outcome from within test or setup code. Example:

```
def test_function():
    if not valid_config():
        pytest.xfail("unsupported configuration")
```

4.6 Skipping on a missing import dependency

You can use the following import helper at module level or within a test or test setup function:

```
docutils = pytest.importorskip("docutils")
```

If docutils cannot be imported here, this will lead to a skip outcome of the test. You can also skip based on the version number of a library:

```
docutils = pytest.importorskip("docutils", minversion="0.3")
```

The version will be read from the specified module's __version__ attribute.

4.7 Imperative skip from within a test or setup function

If for some reason you cannot declare skip-conditions you can also imperatively produce a skip-outcome from within test or setup code. Example:

```
def test_function():
    if not valid_config():
        pytest.skip("unsupported configuration")
```

4.8 Using -k TEXT to select tests

You can use the -k command line option to select tests:

```
arulalan@arulalan:~/$ py.test divisible-testing.py -v -k RandomMod
============================= test session starts ==============================
platform linux2 -- Python 2.6.6 -- pytest-2.0.2 -- /usr/bin/python
collected 4 items

divisible-testing.py:18: TestDivide.test_RandomMod PASSED

====================== 3 tests deselected by 'RandomMod' =======================
==================== 1 passed, 3 deselected in 0.05 seconds ====================
```

And you can also run all tests except the ones that match the keyword by prefix – before the keyword:

```
arulalan@arulalan:~/$ py.test divisible-testing.py -v -k -RandomMod
============================= test session starts ==============================
platform linux2 -- Python 2.6.6 -- pytest-2.0.2 -- /usr/bin/python
collected 4 items

divisible-testing.py:30: TestToDivideBadInput.test_Zerodenominator PASSED
divisible-testing.py:34: TestToDivideBadInput.test_Stringdenominator PASSED
divisible-testing.py:38: TestToDivideBadInput.test_StringNumerator PASSED

====================== 1 tests deselected by '-RandomMod' ======================
==================== 3 passed, 1 deselected in 0.02 seconds ====================
```

Or to only select the class:

```
arulalan@arulalan:~/$ py.test divisible-testing.py -v -k TestToDivideBadInput
============================= test session starts ==============================
platform linux2 -- Python 2.6.6 -- pytest-2.0.2 -- /usr/bin/python
collected 4 items

divisible-testing.py:30: TestToDivideBadInput.test_Zerodenominator PASSED
divisible-testing.py:34: TestToDivideBadInput.test_Stringdenominator PASSED
divisible-testing.py:38: TestToDivideBadInput.test_StringNumerator PASSED

================= 1 tests deselected by 'TestToDivideBadInput' =================
==================== 3 passed, 1 deselected in 0.02 seconds ====================
```

Extended xUnit style setup fixtures in PyTest

Python, Java and many other languages support xUnit style testing. This typically involves the call of a setup ("fixture") method before running a test function and teardown after it has finished. py.test supports a more fine-grained model of setup/teardown handling by optionally calling per-module and per-class hooks.

5.1 module level setup/teardown

If you have multiple test functions and test classes in a single module you can optionally implement the following fixture methods which will usually be called once for all the functions:

```
def setup_module(module):
    """ setup up any state specific to the execution
        of the given module.
    """

def teardown_module(module):
    """ teardown any state that was previously setup
        with a setup_module method.
    """
```

5.2 class level setup/teardown

Similarly, the following methods are called at class level before and after all test methods of the class are called:

```
def setup_class(cls):
    """ setup up any state specific to the execution
        of the given class (which usually contains tests).
    """

def teardown_class(cls):
    """ teardown any state that was previously setup
        with a call to setup_class.
    """
```

5.3 method and function level setup/teardown

Similarly, the following methods are called around each method invocation:

```
def setup_method(self, method):
    """ setup up any state tied to the execution of the given
    method in a class.  setup_method is invoked for every
    test method of a class.
    """

def teardown_method(self, method):
    """ teardown any state that was previously setup
    with a setup_method call.
    """
```

If you would rather define test functions directly at module level you can also use the following functions to implement fixtures:

```
def setup_function(function):
    """ setup up any state tied to the execution of the given
    function. Invoked for every test function in the module.
    """


def teardown_function(function):
    """ teardown any state that was previously setup
    with a setup_function call.
    """
```

In the below example, we written the setup_method for the particular method to initialise the random nos and its mod. By this way we can invoke the setup and teardown methods in py.test.

```
#divisible-testing.py V1.0
class TestDivide():

        def setup_method(self,test_RandomMod):
                self.numerator = random.randrange(10)
                self.denominator = random.randint(1,20)
                self.remainder = self.numerator % self.denominator
                print "invoked"
```

```
def test_RandomMod(self):
        ''' testing for mod operation '''
        result = divide_instance.mod(self.numerator,self.denominator)
        assert result == self.remainder
```

Note : Inside test code, the print statement will never works. i.e. We can print anything while doing this unit testing by using print statement.

xdist: pytest distributed testing plugin

The pytest-xdist plugin extends py.test with some unique test execution modes:

- Looponfail: run your tests repeatedly in a subprocess. After each run, py.test waits until a file in your project changes and then re-runs the previously failing tests. This is repeated until all tests pass. At this point a full run is again performed.
- multiprocess Load-balancing: if you have multiple CPUs or hosts you can use them for a combined test run. This allows to speed up development or to use special resources of remote machines.
- Multi-Platform coverage: you can specify different Python interpreters or different platforms and run tests in parallel on all of them.

Before running tests remotely, py.test efficiently “rsyncs” your program source code to the remote place. All test results are reported back and displayed to your local terminal. You may specify different Python versions and interpreters.

6.1 Installation of xdist plugin

Install the plugin with:
$ *easy_install pytest-xdist*
# or
$ *pip install pytest-xdist*

or use the package in develop/in-place mode with a checkout of the pytest-xdist repository and then do
$ *python setup.py develop*

6.2 Usage examples

Speed up test runs by sending tests to multiple CPUs
To send tests to multiple CPUs, type:

$ py.test -n NUM

Especially for longer running tests or tests requiring a lot of I/O this can lead to considerable speed ups.

More Usage of PyTest

7.1 Running tests in a Python subprocess

To instantiate a Python-2.4 subprocess and send tests to it, you may type:

$ py.test -d --tx popen//python=python2.4

This will start a subprocess which is run with the "python2.4" Python interpreter, found in your system binary lookup path.

If you prefix the –tx option value like this:

$ py.test -d --tx 3*popen//python=python2.4

then three subprocesses would be created and the tests will be distributed to three subprocesses and run simultanously.

7.2 Running tests in looponfailing mode

For refactoring a project with a medium or large test suite you can use the looponfailing mode. Simply add the --f option:

$ py.test -f

and py.test will run your tests.

Assuming you have failures it will then wait for file changes and re-run the failing test set. File changes are detected by looking at looponfailingroots root directories and all of their contents (recursively). If the default for this value does not work for you you can change it in your project by setting a configuration option:

```
# content of a pytest.ini, setup.cfg or tox.ini file
[pytest]
looponfailroots = mypkg testdir
```

This would lead to only looking for file changes in the respective directories, specified relatively to the ini-file's directory.

7.3 Sending tests to remote SSH accounts

Suppose you have a package mypkg which contains some tests that you can successfully run locally. And you also have a ssh-reachable machine myhost. Then you can ad-hoc distribute your tests by typing:

$ py.test -d --tx ssh=myhostpopen --rsyncdir mypkg mypkg

This will synchronize your mypkg package directory with a remote ssh account and then collect and run your tests at the remote side. You can specify multiple --rsyncdir directories to be sent to the remote side.

7.4 Sending tests to remote Socket Servers

Download the single-module socketserver.py Python program and run it like this:

$ python socketserver.py

It will tell you that it starts listening on the default port. You can now on your home machine specify this new socket host with something like this:

$ py.test -d --tx socket=192.168.1.102:8888 --rsyncdir mypkg mypkg

The basic command to run tests on multiple platforms is:

$ py.test --dist=each --tx=spec1 –tx=spec2

If you specify a windows host, an OSX host and a Linux environment this command will send each tests to all platforms - and report back failures from all platforms at once. The specifications strings use the xspec syntax.

7.5 Specifying test exec environments in an ini file

pytest (since version 2.0) supports ini-style configuration. For example, you could make running with three subprocesses your default:

[pytest]
addopts = -n3

You can also add default environments like this:

[pytest]
addopts = --tx ssh=myhost//python=python2.5 --tx ssh=myhost//python=python2.6

and then just type:

py.test --dist=each
to run tests in each of the environments.

7.6 Specifying “rsync” dirs in an ini-file

In a tox.ini or setup.cfg file in your root project directory you may specify directories to include or to exclude in synchronisation:

[pytest]
rsyncdirs = . mypkg helperpkg
rsyncignore = .hg

These directory specifications are relative to the directory where the configuration file was found.

Links for pytest :

http://pytest.org

http://pylib.org

Mailing List for Testing in Python http://lists.idyll.org/listinfo/testing-in-python