Nosetests → Pytest

This presentation covers the major changes that I am making to transition the code base to pytest from nosetests. I focus on the impacts this change will have on your workflow when writing and running our unit tests.

Motivation

Writing Tests

Running Tests

Motivation

Writing Tests

Running Tests

Pytest is Actively Maintained

"… [Nose] will likely cease without a new person/team to take over maintainership."

– Nose Maintainers

nose.readthedocs.io/en/latest/index.html#note-to-users

Nosetests is currently in maintenance mode, so there is little development activity aside from bug fixes. In the future, the project is likely to shutdown entirely. We don't want to be relying on it when that happens, so we're transitioning to pytest, which is actively being developed.

Pytest Found Deprecation Warnings

```
- np.random.random_integers(a, b)
+ np.random.randint(a, b + 1)


- with self.assertRaises(Exception) as c:
-      ...
-      self.assertEqual(c.exception.message, …)
+ with self.assertRaisesRegexp(Exception, "^msg$"):
+      ...
```


When I was making the changes needed to transition to pytest, Jerry pointed out that pytest was catching deprecation warnings that nosetests had been missing. These are the two kinds of warnings raised, which I have already fixed in my PR.

Pytest Asserts Are More Pythonic

unittest	pytest
`self.assertEqual(a, b)`	`assert a == b`

The syntax in pytest is much nicer! I compare with unittest here since that’s the predominant syntax in our code.

Motivation

Writing Tests

Running Tests

Assertions

unittest	pytest
`self.assertEqual(a, b)`	`assert a == b`
`self.assertLessThan(a, b)`	`assert a < b`
`self.assertIn(a, my_list)`	`assert a in my_list`
`self.assertRaises(Exception):`	`pytest.raises(Exception):`

Here is a sample of differences in assertion syntax. In general, you can pretty safely bet that the pytest syntax is just an assert followed by the normal test you would write in python, e.g. in an if condition.

No Constructors in Tests

```
class TestTmp(object):

    def __init__(self):
        # setup logic

    def test_0(self):
        x = func_under_test()
        assert x == 2
```


In nosetests, you could do this.

No Constructors in Tests

```
class TestTmp(object):

    def __init__(self):
        # setup logic

    def test_0(self):
        x = func_under_test()
        assert x == 2
```

In pytest, tests are not allowed to have constructors, ever. Note that this is a problem whenever pytest tries to load a class with a constructor, whether or not there are actually tests there. This is a problem I will address in more detail later.

Setup and Teardown

UnitTest Style

```
class TestA(TestCase):

    def setUp(self):
        ...
    def tearDown(self):
        ...
    def setUpClass(self):
        ...
    def tearDownClass(self):
        ...
```

Pytest / Nose Style

```
class TestA:

    def setup_method(self):
        ...
    def teardown_method(self):
        ...
    def setup_class(self):
        ...
    def teardown_class(self):
        ...
```


Instead, you can use one of these two styles for similar functionality. Notice that to use the unittest style camelcase methods, your test class must inherit from unittest.TestCase

Pytest Fixtures

```
@pytest.fixture(scope="class")  # created once per class
def load_data():
    # setup code
    yield '{"foo": "bar"}'  # if no teardown, use return
    # teardown code

def test_analyze(load_data):
    analysis_results = analyze(load_data())
    assert results == ["myResults"]
```


An even better option is to use pytest fixtures. Broadly speaking, they are a way to inject dependencies into your code. For example, if we want to test an analysis script, we can use a fixture to generate dummy data and then pass that data into our analysis function. This fixture can run setup code and teardown code, replacing the need for the methods on the previous slide. If fixtures are expensive to setup, you can set a scope to limit how often they are re-created. Fixtures can also be shared across classes and files by defining them in conftest.py files or in __init__.py files, which is a great way to re-use setup code.

Test Discovery

- Pytest loads based on:
  - Modules: `test_*.py` or `*_test.py`
  - Classes: `Test*`
  - Functions (outside class or in test class): `test_*`

Pytest's discovery rules are a little different from nose, though in general our code is compatible still. Modules must start or end with test, separated from the rest of the name by an underscore. Classes must begin with uppercase Test, and functions must begin with test_. Note that test functions can be either not inside any class or inside a test class.

Test Discovery

- Pytest loads based on:
  - Modules: `test_*.py` or `*_test.py`
  - Classes: `Test*`
  - Functions (outside class or in test class): `test_*`
- Example: Naming integration tests file
  - `test_workflow.py`
  - `workflow_test.py`
  - `integration_test_workflow.py`

For example, consider the integration test file test_workflow.py (runscripts/cloud/util/test_workflow.py). It has a constructor, so we can’t let pytest import it, but it looks like a test based on its naming. To fix this, I renamed it as integration_test_workflow.py. With the test in the middle, it is ignored by pytest.

Motivation

Writing Tests

Running Tests

Running Tests

```
$ pytest
```


Execute pytest instead of nosetests

Test Output: Passing

```
$ pytest
=================================== test session starts ===================================
platform darwin -- Python 2.7.16, pytest-4.6.5, py-1.5.4, pluggy-0.13.0
benchmark: 3.2.2 (defaults: timer=time.time disable_gc=False min_rounds=5 min_time=0.000005
max_time=1.0 calibration_precision=10 warmup=False warmup_iterations=100000)
rootdir: /path/to/repository/root/, inifile: pytest.ini
plugins: benchmark-3.2.2, cov-2.8.1
collected 111 items

models/ecoli/tests/test_arrow.py .                                          [  0%]
path/to/second/test.py .....                                                [ 10%]
...
============================== 109 passed, 2 skipped in 7.29 seconds ==============================
```


This is what it looks like when your tests pass. Note that I’ve omitted a bunch of test output for simplicity.

Failures

```
class TestTmp(object):

    def test_0(self):
        x = 1
        assert x == 2

    def test_1(self):
        x = 1
        assert 0 < x < 1

    def test_2(self):
        x = {1: 2}
        assert 5 in x
```

```
    def test_0(self):
      x = 1
>     assert x == 2
E     assert 1 == 2
test_tmp.py:5: AssertionError

    def test_1(self):
      x = 1
>     assert 0 < x < 1
E     assert 1 < 1
test_tmp.py:9: AssertionError

    def test_2(self):
      x = {1: 2}
>     assert 5 in x
E     assert 5 in {1: 2}
test_tmp.py:13: AssertionError
```


Here are some examples of test failures. Notice that pytest shows some code context, the line that failed, what in particular caused the assertion to fail, and a filename and line number.

Key Takeaways

- Writing Tests
  - Use `assert`
  - Use `setup_method`, not `__init__`
- Running Tests
  - Update dependencies
  - Run `pytest`, not `nosetests` (change in PyCharm)

More Information

pytest.org

PR#728

Documentation at pytest.org. My changes are in PR#728.