Unit Tests and Continuous Integration

Erik Tollerud

@eteq

Space Telescope Science Institute

Giacconi Fellow

Astropy Coordination Committee

What Testing Is Needed For Science Code?

What Testing Is Needed For Science Code?

- Functional testing: "Does it run from beginning to end?" or "Does my example work?"

What Testing Is Needed For Science Code?

- Functional testing: "Does it run from beginning to end?" or "Does my example work?"
- Result-oriented testing: "Do I get out the science I expect if I run the whole thing on known inputs?"

What Testing Is Needed For Science Code?

- Functional testing: "Does it run from beginning to end?" or "Does my example work?"
- Result-oriented testing: "Do I get out the science I expect if I run the whole thing on known inputs?"
- Unit testing: "Does this function give me something sensible when I call it?"

Unit Testing Principals

Unit Testing Principals

- Whenever you write a "chunk" of code, write corresponding tests.

Unit Testing Principals

- Whenever you write a "chunk" of code, write corresponding tests.
  - Modular code is *critical* here.

Unit Testing Principals

- Whenever you write a "chunk" of code, write corresponding tests.
  - Modular code is *critical* here.
  - Make them as specific and fine-grained as you can.

Unit Testing Principals

- Whenever you write a "chunk" of code, write corresponding tests.
  - Modular code is *critical* here.
  - Make them as specific and fine-grained as you can.
- When you find a bug, write a "regression test".

Unit Testing Principals

- Whenever you write a "chunk" of code, write corresponding tests.
  - Modular code is *critical* here.
  - Make them as specific and fine-grained as you can.
  - When you find a bug, write a "regression test".
- Run them early and often!

(Silly) Example

```
def silly_walk(step):
    if 'left' in step:
        return 'twirl-right'
    elif step == 'twirl-right':
        return 'right'
    elif step == 'right':
        return 'stomp-left'
else:
    return 'left'
```

(Silly) Example

```
def silly_walk(step):
    if 'left' in step:
        return 'twirl-right'
    elif step == 'twirl-right':
        return 'right'
    elif step == 'right':
        return 'stomp-left'
    else:
        return 'left'
```

```
assert silly_walk('right') == 'stomp-left'
```

(Silly) Example

```
def silly_walk(step):
    if 'left' in step:
        return 'twirl-right'
    elif step == 'twirl-right':
        return 'right'
    elif step == 'right':
        return 'stomp-left'
    else:
        return 'left'
```

```
step = 'arg'
step = silly_walk(step)
start = step = silly_walk(step)
for i in range(3):
    step = silly_walk(step)
assert step == start
```

How do you Do this? Use a Testing Framework

Documentation » The Python Standard Library » 26. Development Tools »

26.4. unittest — Unit testing

pytest

nose

is nicer testing for python

How do you Do this? Use a Simple Testing Framework

nose

is nicer testing for python

How do you Do this? Use a Simple Testing Framework

Both use a very similar naming/using convention:

```python
# In the file test_code.py

def test_something():
    assert this_should_be == to_this
```

How do you Do this? Use a Simple Testing Framework

Packages are generally laid out as:

mypackage/__init__.py
mypackage/mymodule.py
mypackage/secondmodule.py
mypackage/tests/__init__.py
mypackage/tests/test_mymodule.py
mypackage/tests/test_secondmodule.py
mypackage/tests/test_some_use_case.py

How do you Do this? Use a Simple Testing Framework

And you can run both very easily with a nice pretty output:

```
% py.test
```

```
============================= test session starts ==============================
platform darwin -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /Users/erik/tmp/testtest/pkg, inifile:
collected 9 items

tests/test_a.py ....F
tests/test_b.py ....

=================================== FAILURES ===================================
_____________________________ test_something_wrong _____________________________

    def test_something_wrong():
>       assert False
E       assert False

tests/test_a.py:15: AssertionError
========================== 1 failed, 8 passed in 0.02 seconds ==========================
```

```
% nosetests
```

```
....F....
======================================================================
FAIL: test_a.test_something_wrong
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/erik/miniconda3/lib/python3.5/site-packages/nose/case.py", line 1
98, in runTest
    self.test(*self.arg)
  File "/Users/erik/tmp/testtest/pkg/tests/test_a.py", line 15, in test_somethin
g_wrong
    assert False
AssertionError

----------------------------------------------------------------------
Ran 9 tests in 0.006s
```

The Primary (Dis-?) Advantage: Flexibility

pytest_plugins.py

```python
def parse(self, s, name=None):
    result = doctest.DocTestParser.parse(self, s, name=name)

    # result is a sequence of alternating text chunks and
    # doctest.Example objects.  We need to look in the text
    # chunks for the special directives that help us determine
    # whether the following examples should be skipped.

    required = []
    skip_next = False
    skip_all = False

    for entry in result:
        if isinstance(entry, six.string_types) and entry:
            required = []
            skip_next = False
            lines = entry.strip().splitlines()

            if '.. doctest-skip-all' in (x.strip() for x in lines):
                skip_all = True
                continue

            if not len(lines):
                continue

            last_line = lines[-1]
            match = re.match(
                r'\.\.\s+doctest-skip\s*::(\s+.*)?', last_line)
            if match:
                marker = match.group(1)
                if (marker is None or
                        (marker.strip() == 'win32' and
                         sys.platform == 'win32')):
                    skip_next = True
                    continue

            match = re.match(
                r'\.\.\s+doctest-requires\s*::\s+(.*)',
                last_line)
            if match:
                required = re.split(r'\s*,?\s*', match.group(1))
        elif isinstance(entry, doctest.Example):
            if (skip_all or skip_next or
```

```python
# set up for parametrized test
rt_sets = []
rt_frames = [ICRS, FK4, FK5, Galactic]
for rt_frame0 in rt_frames:
    for rt_frame1 in rt_frames:
        for equinox0 in (None, 'J1975.0'):
            for obstime0 in (None, 'J1980.0'):
                for equinox1 in (None, 'J1975.0'):
                    for obstime1 in (None, 'J1980.0'):
                        rt_sets.append((rt_frame0, rt_frame1,
                                        equinox0, equinox1,
                                        obstime0, obstime1))
rt_args = ('frame0','frame1','equinox0','equinox1','obstime0','obstime1')


@pytest.mark.parametrize(rt_args, rt_sets)
def test_round_tripping(frame0, frame1, equinox0, equinox1, obstime0, obstime1):
    """
    Test round tripping out and back using transform_to in every combination.
    """
    attrs0 = {'equinox': equinox0, 'obstime': obstime0}
    attrs1 = {'equinox': equinox1, 'obstime': obstime1}

    # Remove None values
    attrs0 = dict((k, v) for k, v in attrs0.items() if v is not None)
    attrs1 = dict((k, v) for k, v in attrs1.items() if v is not None)

    # Go out and back
    sc = SkyCoord(frame0, RA, DEC, **attrs0)

    # Keep only frame attributes for frame1
    attrs1 = dict((attr, val) for attr, val in attrs1.items()
                  if attr in frame1.get_frame_attr_names())
    sc2 = sc.transform_to(frame1(**attrs1))

    # When coming back only keep frame0 attributes for transform_to
    attrs0 = dict((attr, val) for attr, val in attrs0.items()
                  if attr in frame0.get_frame_attr_names())
```

How Do you Ensure Your Tests Are complete?

Coverage.py

(with plugins), works well with

pytest

nose

is nicer testing for python

Just remember to check!

How do you Make Sure you Do Them?

How do you Make Sure you Do Them?

- Consider it part of doing the code.

How do you Make Sure you Do Them?

- Consider it part of doing the code.
- You probably do tests anyway. Just write ‘em up.

How do you Make Sure you Do Them?

- Consider it part of doing the code.
- You probably do tests anyway. Just write 'em up.
- Adopt test-driven design (although can be hard for science).

How do you Make Sure you Do Them?

- Consider it part of doing the code.
- You probably do tests anyway. Just write 'em up.
- Adopt test-driven design (although can be hard for science).
- Use continuous integration to make it worth your time.

Continuous Integration

Run the tests *whenever* something changes (online).

Continuous Integration

Run the tests *whenever* something changes (online).

Especially when done with Pull Requests, this drastically reduces the number of “Oops, sorry, didn’t realize that would happen” incidences.

Continuous Integration

Run the tests *whenever* something changes (online).

Continuous Integration

Run the tests *whenever* something changes (online).

Go Do that Voodoo that you do... So well!