Testing Code, As You Code

Unit tests and more

Mike Jarvis

July 10, 2017

LSST DESC DE School

Why write tests?

Why write tests?

- To **find bugs as soon as possible**, before they lead to a lot of wasted time running programs with errors.
- To notice **regressions** -- when you accidentally break previously working code.
- To enable **refactoring** with confidence that you aren't breaking the existing functionality.
- To make it easier to check for **portability** issues.
- To help **design** the user interface of your software.

What kinds of errors?

What kinds of errors?

- **Errors in the math** being implemented.
- Insufficient **numerical accuracy**, maybe only in certain cases.
- Problems with the **output format** or packaging of the results.
- Improper handling of bad user input or **bad data**.
- Improper handling of **exceptional cases**.
- **Discrepancies** from previously validated results or from other sources of truth.
- Non-backwards compatible **changes to the API**.

Levels of software testing

Unit Tests check that small units of code perform the function that they are intended (and documented) to perform, independent of the rest of the program.

Integration Tests check that different components of the code work properly when integrated together in various combinations or in connection to external systems, databases, files, etc.

System Tests check that the entire system is working as intended. Usually involves running the final completed program on some particular data.

What is a unit?

What is a unit?

A "unit" is **a single unit of action** from a user's perspective. Typically, a single line of user code.

- Constructing an instance of a class

```
c = CelestialCoord(ra=4*hours, dec=31*degrees)
```

- A call of a function or method of a class

```
d = c.distanceTo(c2)
```

- A function call with a particular choice for an optional argument

```
u,v = c.project(c2, projection='stereographic')
```

- Use of an object in some other documented way

```
sint = np.sin(theta)
```

What does a unit test do?

What does a unit test do?

1. **Setup**
    - Create any preconditions required to run the unit of code.
    - Typically, make all the objects you will need.
2. **Execution**
    - Run the unit of code being tested.
3. **Verification**
    - Possibly, do some alternate calculation to be used to verify the correctness of the output.
    - **Assert that the result was as expected.**

What makes for a good unit test?

What makes for a good unit test?

- **Clear**ly identifies where the problem is when it fails.
- Is **easy to run** in an automated way.
- Runs **quick**ly.
- Has **consistent** results.
- Is **independent** of other tests, programs, system variables, etc.
- Is **easy to write**.

Examples

```python
def test_init():
    """Test the AngleUnit initializer"""

    gradians = coord.AngleUnit(2.*pi / 400.)
    assert gradians.value == 2.*pi / 400., 'gradians value not 2pi/400'
```

Examples

```python
def test_init():
    """Test the AngleUnit initializer"""

    gradians = coord.AngleUnit(2.*pi / 400.)
    assert gradians.value == 2.*pi / 400., 'gradians value not 2pi/400'

    # Can also use named keyword argument
    np.testing.assert_almost_equal(coord.AngleUnit(value=17).value, 17)

    # Non-float value is ok so long as it is convertible to float.
    np.testing.assert_almost_equal(
        coord.AngleUnit(np.float64(0.17)).value, 0.17, decimal=12,
        err_msg='using np.float64 for value failed')
```

Examples

```python
def test_invalid():
    """Check that invalid arguments raise the appropriate exceptions."""

    # Wrong type of value argument
    np.testing.assert_raises(TypeError, coord.AngleUnit, coord.degrees)

    # Also wrong type, but strings give a ValueError
    np.testing.assert_raises(ValueError, coord.AngleUnit, 'spam')

    # Wrong number of arguments
    np.testing.assert_raises(TypeError, coord.AngleUnit, 1, 3)
    np.testing.assert_raises(TypeError, coord.AngleUnit)

    # Wrong keyword argument
    np.testing.assert_raises(TypeError, coord.AngleUnit, the_value=0.2)
```

Examples

```
def test_pickle():
    """Check that AngleUnits pickle correctly."""

    do_pickle(coord.radians)
    do_pickle(coord.degrees)
    do_pickle(coord.hours)
    do_pickle(coord.arcmin)
    do_pickle(coord.arcsec)

    gradians = coord.AngleUnit(2.*pi / 400.)
    do_pickle(gradians)
```

Examples

```
def test_eq():
    """Check that equal angles are equal, but unequal ones are not."""

    theta1 = pi/4. * coord.radians
    theta2 = 45 * coord.degrees
    assert theta1 == theta2, "pi/4 rad != 45 deg"

    theta3 = coord.Angle(theta1)  # Copy constructor
    assert theta3 == theta1, "copy not == to original"

    # These should all test as unequal.  Note some non-Angles in the list.
    diff_list = [ theta1, 14 * coord.degrees,
                  -theta1, theta1 * 2.,
                  theta1 + 360. * coord.degrees,
                  theta1 - 360. * coord.degrees,
                  pi/4., coord.Angle, None ]
    all_obj_diff(diff_list)
```

Examples

```
def test_distance():
    """Test calculations of distances on the sphere."""

    # First, let's test some distances that are easy to figure out.
    eq1 = coord.CelestialCoord(0 * radians, 0 * radians)
    eq2 = coord.CelestialCoord(1 * radians, 0 * radians)
    eq3 = coord.CelestialCoord(pi * radians, 0 * radians)
    npole = coord.CelestialCoord(0 * radians, pi/2 * radians)
    spole = coord.CelestialCoord(0 * radians, -pi/2 * radians)

    np.testing.assert_almost_equal(eq1.distanceTo(eq2).rad, 1, decimal=12)
    np.testing.assert_almost_equal(eq2.distanceTo(eq1).rad, 1, decimal=12)
    np.testing.assert_almost_equal(eq1.distanceTo(eq3).rad, pi, decimal=12)
    np.testing.assert_almost_equal(eq2.distanceTo(eq3).rad, pi-1, decimal=12)
    np.testing.assert_almost_equal(npole.distanceTo(spole).rad, pi, decimal=12)
    np.testing.assert_almost_equal(eq3.distanceTo(npole).rad, pi/2, decimal=12)
    np.testing.assert_almost_equal(eq2.distanceTo(spole).rad, pi/2, decimal=12)
```

Examples

```
def test_distance():  (continued)

    # Some random point
    c1 = coord.CelestialCoord(0.234 * radians, 0.342 * radians)
    # Same meridian
    c2 = coord.CelestialCoord(0.234 * radians, -1.093 * radians)
    # Antipode
    c3 = coord.CelestialCoord((pi + 0.234) * radians, -0.342 * radians)
    # Different point on opposite meridian
    c4 = coord.CelestialCoord((pi + 0.234) * radians, 0.832 * radians)

    for c, d in zip( (c1,c2,c3,c4), (0., 1.435, pi, pi-1.174) ):
        np.testing.assert_almost_equal(c1.distanceTo(c).rad, d,
                                       decimal=12)
```

Examples

```
def test_distance():  (continued)

    # Now some that require spherical trig calculations.
    c1 = coord.CelestialCoord(0.234 * radians, 0.342 * radians)
    c5 = coord.CelestialCoord(1.832 * radians, -0.723 * radians)

    # The standard formula is:
    # cos(d) = sin(dec1) sin(dec2) + cos(dec1) cos(dec2) cos(delta ra)
    d = acos(sin(c1.dec) * sin(c2.dec) +
             cos(c1.dec) * cos(c2.dec) * cos(c1.ra - c2.ra))

    # Note: the CelestialCoord class uses a different formula that is
    # more stable for very small distances
    np.testing.assert_almost_equal(c1.distanceTo(c5).rad, d, decimal=12)
```

Examples

```
def test_distance():  (continued)

    # Tiny displacements should have dsq = (dra^2 cos^2 dec) + (ddec^2)
    c1 = coord.CelestialCoord(0.234 * radians, 0.342 * radians)
    c8 = coord.CelestialCoord(c1.ra + 2.3e-9 * radians,
                              c1.dec + 1.2e-9 * radians)

    # Note that the standard formula gets this one wrong.  d is 0.0
    d = acos(sin(c1.dec) * sin(c8.dec) +
             cos(c1.dec) * cos(c8.dec) * cos(c1.ra - c8.ra))
    true_d = sqrt( (2.3e-9 * cos(c1.dec))**2 + 1.2e-9**2)
    print('d(c7) = ',true_d, c1.distanceTo(c8), d)

    np.testing.assert_allclose(c1.distanceTo(c8).rad, true_d, rtol=1.e-7)
```

Examples

```
def test_distance():  (continued)

    # Near antipodes, the usual formula becomes somewhat inaccurate.
    antipode = coord.CelestialCoord(c1.ra + pi*radians, -c1.dec)
    np.testing.assert_almost_equal(c1.distanceTo(antipode).rad, pi,
                                   decimal=12)

    # Also some near, but not quite antipodes
    eq3 = coord.CelestialCoord(pi * radians, 0 * radians)
    # Note: this range crosses the point where the formula changes
    for delta in range(500):
        eq4 = coord.CelestialCoord(delta * arcmin, 0 * radians)
        np.testing.assert_allclose(pi - eq3.distanceTo(eq4).rad,
                                   eq4.ra.rad, rtol=1.e-7)
```

Examples

```
def test_precess():
    """Compare precess output to corresponding astropy function"""

    c1 = coord.CelestialCoord(0.234 * coord.radians, 0.342 * coord.radians)
    c2 = c1.precess(2000, 1950)
    c3 = c2.precess(1950, 1900)

    a1 = astropy.coordinates.SkyCoord(0.234, 0.342, unit=units.radian,
                                      frame=FK5(equinox='J2000'))
    a2 = a1.transform_to(FK5(equinox='J1950'))
    a3 = a2.transform_to(FK5(equinox='J1900'))

    np.testing.assert_allclose(c2.rad, [a2.ra.rad, a2.dec.rad], rtol=1.e-5,
                               err_msg='astropy differs after 2000->1950')
    np.testing.assert_allclose(c3.rad, [a3.ra.rad, a3.dec.rad], rtol=1.e-5,
                               err_msg='astropy differs after 1950->1900')
```

Examples

```
def test_precess():  (continued)

    print('Compare times for precession calculations:')
    print('  Make CelestialCoord: t = ', t1-t0)
    print('  Precess with Coord: t = ', t2-t1)
    print('  Make SkyCoord: t = ', t3-t2)
    print('  Precess with Astropy: t = ', t4-t3)
    # On my laptop, these times are
    #   Make CelestialCoord: t =  9.10758972168e-05
    #   Precess with Coord: t =  0.000560998916626
    #   Make SkyCoord: t =  0.00361394882202
    #   Precess with Astropy: t =  0.0377740859985

    # Make sure we don't get slow like astropy.  ;)
    assert t1-t0 < 0.001, 'Building CelestialCoord is too slow'
    assert t2-t1 < 0.01, 'CelestialCoord.precess is too slow'
```

When should you write unit tests?

When should you write unit tests?

Writing the tests **as you code** will...

- Help you find bugs quickly, so they don't bite you later during integration or system testing (or in production runs).
- Help you design a user-friendly interface.
- Be easier to write, since you're already thinking about the bit of code that you need to write a test for.

When should you write unit tests?

The most extreme form of this is called **Test Driven Design**

- Write the test first.
- Run it. Make sure it fails.
- Write just enough code to make the test pass.
- Repeat (with another test)

When should you write unit tests?

More realistic workflow for most of us:

- Write **basic unit tests** ("normal usage") as you write the code.
- Add more unit tests to check **edge cases**, exceptions, etc.
- Add more unit tests for **regression** purposes.
- Add appropriate **integration tests** that combine the new code with other related components.
- On pull requests add tests to **improve coverage** of new code.
- Once all components are individually tested, develop a comprehensive **system test** using typical input data.

When should you write unit tests?

For bug reports, absolutely do follow TDD!

- Write a unit test that reproduces the reported bug.
- Make sure it fails!
- Then fix the code so it doesn't fail.

When should you run unit tests?

When should you run unit tests?

- **When working on new code**, run the relevant tests whenever you think they and the code are ready. (Repeat often)
- **Before pushing your commits**, run the full test suite to make sure you didn't break anything elsewhere in the code.
- Have **Travis** run the tests for every commit to the master branch as well as all pull requests.
- **Before tagging an official release**, run the test suite on as many systems as possible to check for portability problems.

LSSTDESC / Coord

build passing

Current Branches Build History Pull Requests More options

✓ **master** Provide a bit more cushion on the timing tests

Commit 71a90e7

Compare 90c7f78..71a90e7

Branch master

Mike Jarvis authored and committed

#58 passed

Ran for 1 min 6 sec

Total time 1 min 33 sec

about 2 hours ago

Restart build

Build Jobs

✓	# 58.1	Python: 2.7	no environment variables set	30 sec
✓	# 58.2	Python: 3.4	no environment variables set	32 sec
✓	# 58.3	Python: 3.5	no environment variables set	31 sec

Provide a bit more cushion on the timing tests

rmjarvis 2 hours ago ✔ CI Passed

71a90e7 master 90c7f78

63.74% ø ø

Diff | **Files** | Build | Graphs

/ coord

Files					Coverage
__init__.py	14	14	0	0	100.00%
_version.py	2	2	0	0	100.00%
angle.py	146	96	22	28	65.75%
angleunit.py	53	36	1	16	67.92%
celestial.py	324	196	23	105	60.49%
util.py	32	20	0	12	62.50%
Folder Totals (6 files)	571	364	46	161	63.74%
Project Totals (6 files)	571	364	46	161	63.74%

Your turn to practice

- Go to the Coord GitHub page:

  https://github.com/LSSTDESC/Coord

- Follow the instructions at the bottom of the page.
- Work in groups or individually as you prefer.

Things to consider when writing tests

- Include **all valid ways** to intialize an object or call a function.
- Check that appropriate **exceptions** are raised for user errors or exceptional circumstances (e.g. singular matrices)
- Check cases where the results are **trivial to calculate**.
- Do the same calculation using a **different formula** or algorithm.
- Try to find **edge cases** that could make a result wrong or less accurate.
- Compare to **other programs** and/or **external** sources of truth.
- Check that the code handles **bad inputs** appropriately (e.g. NaNs).