AST1501 - Introduction to Research

Jo Bovy

Testing your code

Basics of good testing

- Presumably you test whether each piece of code that you write works in some way, but you probably

  - only run it when developing the code and then assume all is fine

  - don't write the tests as a set of functions, but rather run them in a Python terminal / `jupyter notebook`

  - don't exhaustively test how new parts of your code work together with older parts of your code

- This means that your code is very vulnerable to big and small issues that come up, making it hard to use and maintain

Basics of good testing

- Better to use a *test suite*:

  - A set of Python functions (or classes) with checks on your code's functionality

  - Becomes part of your code repository, so it can be saved and changes tracked, and applied to future changes to the code

  - Consists of *unit tests* and *integration tests*

- If designed well, can be run with standard commands, keep track of *test coverage*, and be run automatically by online services every time you update your code

Example

```python
def test_square_direct():
    # Direct test that the square works based on known solutions
    import math
    import exampy
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(3.)-10.) < tol, \
        "exampy.square does not agree with known solution"
    return None
```

Unit and integration tests

- *Unit tests:*

  - Test whether a small unit of your code works as expected:

  - Break your code into smallest unit (e.g., function) that makes sense and build more complex functionality from these smallest units

  - Unit tests check that each of the smallest units does what it is supposed to do

- *Integration tests:*

  - Even if each unit works as expected, they may not work together as they should

  - Integration tests check that different units of your code work together well and give correct results

Good qualities of a test suite

- Should be as minimal, short, and atomic as possible: Keep tests as simple as you can and still achieve a useful test

- Should run in as little time as possible: Once you add many tests, the time the text suite takes to run will get long…

- For code intended to be shared: you should test expected outputs, but also errors and warnings raised: If you are raising exceptions or warnings upon certain behavior, test that that is correctly done as well

- Should test setting non-default keywords for functions and methods to make sure that works as expected

- Should test different invocations of functions: don't just test one, perhaps special case, but make sure the function works for different cases of the inputs

- Should be well-commented, just like your regular code!

What are good things to test?

- Context specific!

- Can check known answers: often we write code to solve problems that can only be solved analytically in certain special cases. Test that the special cases work.

- Can check known properties of the answer: even if we don't know any solution, we may require that the solution has certain properties (e.g., follows a conservation law or symmetry). Test that known properties work.

- Can check against alternative ways of getting the answer:

  - Alternative method that only applies in certain cases (but not analytic)

  - More straightforward method that is more time-consuming

  - Alternative code implementation, e.g., in an external package

Writing a test suite

Where does my test suite go?

- Inside your package:

  - Include the tests in the package itself and distribute them with the code

  - Advantage: users can easily run the test suite on their own machine and convince themselves that the code works for them

  - Disadvantages: Adds a lot of code to your package that is not necessary for it to work, typically not well-documented, and not tested itself (who tests the tests?)

  - Not typical to get a problem when code works as expected on one machine and it and its dependencies install fine on the user's machine

- Not recommended

Where does my test suite go?

- Outside your package:

  - Include the tests in a sub-directory of your top-level directory, outside of the package itself

    ```
    TOP-LEVEL_DIRECTORY/
        docs/
        exampy/
        tests/
        README.md
        setup.py
    ```

  - This way tests are part of your package's `git` repository without being part of your package's distribution

  - One advantage: Your tests can depend on hard-to-install dependencies, as long as you can get them to install, without having to worry about user complaints

- My recommendation

What does my test suite look like?

- Python standard: `pytest`

- `pytest` automatically detects tests, provided that

  - Files start with `test_` and end in `.py`

  - Functions that are tests start with `test`, classes start with `Test`

```
TOP-LEVEL_DIRECTORY/
    docs/
        build/
        source/
        Makefile
        make.bat
    exampy/
        integrate/
        __init__.py
        _math.py
    tests/
        test_basic_math.py
    README.md
    setup.py
```

What do my tests look like?

- A test statement is a simple `assert` statement

- These are the *only* statements that make up formal tests, don't do something like if `res < tol: print("Didn't work")`

- Add a message to display when the `assert` fails

```python
def test_square_direct():
    # Direct test that the square works based on known solutions
    import math
    import exampy
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(3.)-10.) < tol, \
        "exampy.square does not agree with known solution"
    return None
```

Example: known value

```
>>> import test_basic_math
>>> test_basic_math.test_square_direct()
```

```
AssertionError: exampy.square does not agree with known solution
```

```
def test_square_direct():
    # Direct test that the square works based on known solutions
    import math
    import exampy
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(3.)-10.) < tol, \
        "exampy.square does not agree with known solution"
    return None
```

Example: symmetry property

```python
def test_cube_oddfunction():
    # Test of the cube function by checking that it is an odd function
    tol= 1e-10
    assert math.fabs(exampy.cube(1.)+exampy.cube(-1.)) < tol, \
        "exampy.cube is not an odd function"
    assert math.fabs(exampy.cube(2.)+exampy.cube(-2.)) < tol, \
        "exampy.cube is not an odd function"
    assert math.fabs(exampy.cube(3.)+exampy.cube(-3.)) < tol, \
        "exampy.cube is not an odd function"
    return None
```

```python
>>> import test_basic_math
>>> test_basic_math.test_cube_oddfunction()
```

or

```python
def test_cube_oddfunction():
    # Test of the cube function by checking that it is an odd function
    tol= 1e-10
    for nn in range(1,10):
        assert math.fabs(exampy.cube(nn)+exampy.cube(-nn)) < tol, \
            "exampy.cube is not an odd function"
    return None
```

Example: test against alternative

```python
def test_simps_against_scipy():
    # Test that exampy.integrate.simps integration agrees with
    # scipy.integrate.quad
    from scipy import integrate as sc_integrate
    complicated_func= lambda x: x*np.cos(x**2)/(1+np.exp(-x))
    tol= 1e-14
    n_int= 1000
    assert np.fabs(exampy.integrate.simps(complicated_func,0,1,n=n_int)
                   -sc_integrate.quad(complicated_func,0,1)[0])\
                   < tol, \
                   """exampy.integrate.simps gives a different result from """\
                   """scipy.integrate.quad for a complicated function"""
    return None
```

Running a test suite with `pytest`

Running a test suite with `pytest`

- `pytest` is the preferred test runner for Python code

- Automatically detects your tests (see before), prints overview of what happens

- Many options for running, skipping, verbosity of output, etc.

- + additional functionality for testing errors and warnings, labeling known failures, etc.

pytest example

```
pytest -v tests/test_basic_math.py
```

in a regular terminal in the top-level of the package. This produces output that looks like

```
============================ test session starts =============================
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 -- /PATH/
TO/PYTHON/BINARY
cachedir: .pytest_cache
rootdir: /PATH/TO/exampy
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.4.0, remotedata-0.3.1
collected 2 items

tests/test_basic_math.py::test_square_direct PASSED                     [ 50%]
tests/test_basic_math.py::test_cube_oddfunction PASSED                  [100%]

============================= 2 passed in 0.07s ==============================
```

```
============================ test session starts ============================
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 -- /PATH/
TO/PYTHON/BINARY
cachedir: .pytest_cache
rootdir: /PATH/TO/exampy
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.4.0, remotedata-0.3.1
collected 2 items


tests/test_basic_math.py::test_square_direct FAILED                     [ 50%]
tests/test_basic_math.py::test_cube_oddfunction PASSED                  [100%]


================================= FAILURES =================================
_____ test_square_direct _____

    def test_square_direct():
        # Direct test that the square works based on known solutions
        tol = 1e-10
        assert math.fabs(exampy.square(1.)-1.) < tol, \
            "exampy.square does not agree with known solution"
        assert math.fabs(exampy.square(2.)-4.) < tol, \
            "exampy.square does not agree with known solution"
>       assert math.fabs(exampy.square(3.)-10.) < tol, \
            "exampy.square does not agree with known solution"
E       AssertionError: exampy.square does not agree with known solution
E       assert 1.0 < 1e-10
E        +  where 1.0 = <built-in function fabs>((9.0 - 10.0))
E        +    where <built-in function fabs> = math.fabs
E        +    and   9.0 = <function square at 0x1080fad90>(3.0)
E        +      where <function square at 0x1080fad90> = exampy.square

tests/test_basic_math.py:12: AssertionError
========================= 1 failed, 1 passed in 0.10s =========================
```

Useful `pytest` options

- `-x`: Exit upon the first failure (default is to run all tests)

- `-s`: Print `stdout` and `stderr` outputs (default is to not print these)

- `-k EXPRESSION`: Only run tests with EXPRESSION in their name (can also do things like `-k 'not EXPRESSION'`)

- `--lf:` only run the last-failed test (also `--sw` for step-wise running)

- `--disable-pytest-warnings`: don't print all warnings (as a summary at the end)

Testing errors

- You can test whether your code correctly raises an exception using `pytest.raises`

```python
def test_simps_typerror():
    # Test that exampy.integrate.simps properly raises a TypeError
    # when called with a non-array function
    import math
    import pytest
    with pytest.raises(TypeError):
        out= exampy.integrate.simps(lambda x: math.exp(x),0,1)
    return None
```

Testing errors

- You can test whether your code correctly raises an exception using `pytest.raises`

  ▪ You can test the entire error string as well:

```python
def test_simps_typerror():
    # Test that exampy.integrate.simps properly raises a TypeError
    # when called with a non-array function
    import math
    import pytest
    with pytest.raises(TypeError) as excinfo:
        out= exampy.integrate.simps(lambda x: math.exp(x),0,1)
    assert str(excinfo.value) == "Provided func needs to be callable on arrays of inputs"
    return None
```

Test coverage

What is test coverage?

- Once you have a test suite, you will wonder "how much of my code is actually used when running the test suite"

- This question has different answers depending on what you mean:

  - *Function coverage*: what fraction of functions is used by the test suite —> should aim for 100%

  - *Statement coverage*: what fraction of statements is used by the test suite —> aim for 100%, can be difficult to get because of edge cases

  - *Branch coverage*: when my code branches, does the test suite cover all possibilities (if … then… else…)?

  - *Condition coverage*: for complex conditional statements, does the test suite produce True/False for *each boolean sub-expression*? (if x > 0 and y < 0)

`coverage.py`: measuring test coverage in your code

- `coverage.py` is a Python package that will report the test coverage of your test suite, most easily *statement coverage*

- Simply run your test suite as

```
coverage run -m pytest ...
```

- instead of

```
pytest ...
```

- This collects the coverage info, but does not yet display it

`coverage.py`: measuring test coverage in your code

- Once collected, you can display the results in different ways

  - `coverage report`: text report

  - `coverage html`: HTML output

Example report

```
coverage run -m pytest -v tests/
coverage report
```

we get

```
Name                               Stmts   Miss   Cover   Missing
-----------------------------------------------------------------
exampy/__init__.py                     1      0    100%
exampy/_math.py                        9      2     78%   73, 88
exampy/integrate/__init__.py           2      0    100%
exampy/integrate/_integrate.py         8      0    100%
tests/test_basic_math.py              13      0    100%
tests/test_integrate.py               27      1     96%   52
-----------------------------------------------------------------
TOTAL                                 60      3     95%
```

Use - -source to specify the package

```
coverage run --source=exampy/ -m pytest -v tests/
coverage report
```

we now get

```
Name                              Stmts   Miss  Cover   Missing
---------------------------------------------------------------
exampy/__init__.py                    1      0   100%
exampy/_math.py                       9      2    78%   73, 88
exampy/integrate/__init__.py          2      0   100%
exampy/integrate/_integrate.py        8      0   100%
---------------------------------------------------------------
TOTAL                                20      2    90%
```

Example HTML report

| Coverage report: 90% | | | filter... | ⌨ |

Module ↓	statements	missing	excluded	coverage
exampy/__init__.py	1	0	0	100%
exampy/_math.py	9	2	0	78%
exampy/integrate/__init__.py	2	0	0	100%
exampy/integrate/_integrate.py	8	0	0	100%
Total	**20**	**2**	**0**	**90%**

coverage.py v5.0.3, created at 2020-03-06 15:40

Excluding code from test coverage statistics

- Sometimes you want to exclude some parts of your code from the test-coverage statistics

  - Lines that you don't think have to be tested (use sparingly!)

  - Lines that cannot be executed by the test suite

    - `if False:`

    - `if __name__ == .__main__.:`

- For a single line, use `# pragma: no cover`

Excluding code from test coverage statistics

- More complex patterns, use `.coveragerc` in the directory where you run the tests

```
[run]
source= exampy/

[report]
# Regexes for lines to exclude from consideration
exclude_lines =
    # Have to re-enable the standard pragma
    pragma: no cover

    # Don't complain if tests don't hit defensive assertion code:
    raise AssertionError
    raise NotImplementedError

    # Don't complain if non-runnable code isn't run:
    if 0:
    if __name__ == .__main__.:

omit =
    exampy/__init__.py
    exampy/integrate/*

ignore_errors = True

[html]
directory = coverage_html_report
```

Continuous integration

What is continuous integration?

- Refers to running 'integration tests' on a regular basis, at high cadence

- 'Integration tests' in this context is the combination of building your package and running the test suite, making sure that all parts of the code package work as expected (incl. installation)

- Nowadays largely done by online services like GitHub Actions whenever:

  - You push a commit or set of commits to GitHub for any branch

  - Somebody opens or updates a pull request

- Try to catch changes to the code (easy) and to dependencies (harder) that may cause your code's installation or tests to fail

Setting up GitHub actions for your repository

or just add a `.yml` file under `.github/workflows`

```
name: Test exampy

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.7, 3.8]
        numpy-version: [1.16,1.17,1.18]
        exclude:
          - python-version: 3.8
            numpy-version: 1.16
          - python-version: 3.8
            numpy-version: 1.17
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v1
      with:
        python-version: ${{ matrix.python-version }}
```

Note: action versions out of date on this slide

```yaml
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install numpy==${{ matrix.numpy-version }}
    - name: Install package
      run: |
        pip install -e .
    - name: Test with pytest
      run: |
        pip install pytest
        pip install pytest-cov
        pip install scipy
        pytest -v tests/ --cov=exampy/
```