Pytest

@__mharrison__

METASNAKE

`unittest`

Implements Kent Beck's *xUnit* paradigm

xUnit workflow

- Setup
- Call "Unit"
- Make assertion
- Teardown

py.test

pytest

- Easy test creation (less boilerplate)
- Test runner
- Test selection
- Test parameterization
- Fixtures
- Plugins

Installation (Windows)

(Don't type >)

```
> python -m venv env
> env\Scripts\activate
(env) > pip install pytest
```

Installation (Mac/Linux)

(Don't type $)

```
$ python3 -m venv env
$ source env/bin/activate
(env) $ pip install pytest
```

Command Line

Installs `pytest` executable

Assignment 1

- Install pytest in a virtual environment
- Run:

```
pytest -h
```

Basics

Code Layout

```
Project/
    proj/
        __init__.py
        adder.py
    tests/
        conftest.py
        test_adder.py
```

package

module

pytest

Code Layout

Notes

- If test subdirectories don't have `__init__.py`, you can't use the same filename in different directories
- If file named `testadder.py` instead of `test_adder.py`, pytest won't find it

Simple Code

Basic but fits on slides (`adder.py`)

```
# adder.py
def adder(x, y):
    return x + y
```

Test Creation

Unittest style (`test_adder.py`)

```
# test_adder.py
from proj.adder import adder
import unittest

class TestAdder(unittest.TestCase):
    def test_simple(self):
        res = adder(2, 3)
        self.assertEquals(res, 5)
```

2.0 - 2004 - junit

OO - java C#C++

- imperative - C, basic
- functional - lisp

#2

#3

? camelcase

PEP8 → assert_equals

Run Tests

`$ pytest` ignores current directory. (To aid in ensuring testing installed code).

`$ python -m pytest` inserts current directory in `sys.path`.

look for a module that I can load but run it

Run Tests (2)

```
$ python -m pytest tests/*.py
================= test session starts =================
platform darwin -- Python 3.6.4, pytest-3.0.6, py-1.4.32, pluggy-
0.4.0
rootdir: /Users/matt/code_samples/pytest, inifile:
plugins: asyncio-0.8.0
collected 1 items

tests/test_adder.py .

============== 1 passed in 0.01 seconds ==============
```

RESULT

Output

- `.` - test passed
- `F` - test failed
- `E` - Exception
- `s` - test skipped
- `x` - expected failure (broken now but will fix)
- `X` - unexpected pass (should have failed)

Unittest style

- Non-PEP 8 compliant
- "Classy"
- Need to remember which `assert...` method to call

Test Creation

pytest style (`test_adder2.py`)

```
# test_adder2.py
from proj.adder import adder

def test_add():
    res = adder(2, 3)
    assert res == 5
```

#2

#3

Function!

pytest style

- Just a function that starts with "test"
- Use the `assert` statement

Assignment 2

- Create a directory/module `Integer/integr.py` (note spelling).
  - Create a function, `parse`, that accepts a string of the form `"1,3,5,9"` that returns a list of integers (`[1, 3, 5, 9]`)
- Create a test directory and test file `Integer/test/test_integr.py`
  - Create a test function, `test_basic` that asserts that `integr.parse` works with the input `"1,3,5,9"`
- Run pytest on `test/test_integr.py`

More Test Creation

Can specify a message

```
from proj.adder import adder

def test_add():
    res = adder(2, 3)
    assert res == 5, "Value should be 5"
```

Catching Exceptions

Can specify an exception

```
import pytest
def test_exc():
    with pytest.raises(TypeError):
        adder('', 3)
```

context manager

block context

Catching Exceptions (2)

Can include a regular expression as a `match` parameter

```
import pytest
def test_exc():
    with pytest.raises(TypeError,
               match='unsupported operand'):
        adder('', 3)
```

Catching Exceptions (3)

Similar to `pytest.raises` use `pytest.warns` context manager for catching `DeprecationWarning`

Catching Exceptions (4)

Can specify an exception in decorator (status XFAIL or x)

```
@pytest.mark.xfail(raises=TypeError)
def test_exc2():
    adder('', 3)
```

Expected Fail?

Use to specify a test that should work that isn't (ie planning to implement or known bug without a fix)

Expected Fail? (2)

If an expected failure passes it will have a status of XPASS (`X`), unless you give it a `strict=True` option in the decorator. Then it will FAIL (`F`).

Failing a Test

```
def test_missing_dep():
    try:
        import foo
    except ImportError:
        pytest.fail("No foo import")
```

Approximations

Floating point limitations:

```
>>> .7 + .6 == 1.3
False
```

Approximations (2)

`pytest.approx` dynamically adds tolerance:

```
def test_small():
    assert adder(1e-10, 2e-10) == \
        pytest.approx(3e-10)
```

Approximations (3)

`pytest.approx` works with lists, dictionary values, and numpy arrays of floats

How `assert` works

pytest uses an *import hook* (PEP 302) to rewrite assert statements by introspecting code (AST) the runner has collected.

Care needed

Don't wrap assertion in parentheses (truthy tuple):

```
def test_almost_false():
    assert (False == True, 'Should be false')
```

tuple → True

Care needed (2)

You will get a warning:

```
$ pytest test_adder.py
test_adder.py s..x                                          [100%]

==================== warnings summary ====================
test_adder.py:15
  assertion is always true, perhaps remove parentheses?

-- Docs: http://doc.pytest.org/en/latest/warnings.html
 2 passed, 1 skipped, 1 xfailed, 1 warnings in 0.11 seconds
```

Context-sensitive Comparisons

- Inlining function/variable results
- Diffs in similar text
- Lines in multiline texts
- List/Dict/Set diffs (`-vv` for full diff)
- In (`__contains__`) statements

Customize Assert

In conftest.py:

```
def pytest_assertrepr_compare(op, left, right):
    if (isinstance(left, str) and
        isinstance(right, int) and op == '=='):
        return ['"{}" should be an int'.format(left)]
```

In test_adder.py:

```
def test_custom():
    assert "1" == 1
```

Result

```
$ pytest test_adder.py
test_adder.py F.x                                                    [100%]

======================= FAILURES =======================
_____________________ test_custom _____________________

    def test_custom():
>       assert "1" == 1
E       assert "1" should be an int

test_adder.py:11: AssertionError
===== 1 failed, 1 passed, 1 xfailed in 0.08 seconds =====
```

Assignment 3

- Create a test function, `test_bad1` that asserts that an error is raised when `integr.parse` is called with `'bad input'`. Use a context manager (`with`)
- Create a test function, `test_bad2` that asserts that an error is raised when `integr.parse` is called with `'1-3,12-15'`. Use the `@pytest.mark.xfail` decorator.

Test Runner

Test Runner

For `unittest` run:

```
$ python3 -m unittest test_adder.TestAdder
```

Test Runner

For `pytest` run:

```
$ pytest test_adder2.TestAdder
```

Test Discovery

- Recurse current directory or `testpaths` from `pytest.ini` (ignores the `norecursedirs` and virtual environments)
- Files with `test_*.py` or `*_test.py`
- Functions starting with `test*`
- Methods starting with `test*` in class named `Test*` without a `__init__` method

Can customize

- `--ignore path` - Tell pytest to ignore modules or paths
- `norecursedirs` - Dirs to not recurse in `pytest.ini` (default `.*`, `build`, `dist`, `CVS`, `_darcs`, `{arch}`, `*.egg`, `venv`)
- `testpaths` - Force to look in these locations
- `python_files` - Glob (`validate_*.py`) to discover in `pytest.ini`
- `python_classes`, `python_methods` - More discovery

Options

- `--doctest-modules` - Run doctests
- `--doctest-glob='*.rst'` - Capture rst files (instead of default `*.txt`)
- `--pdb` - Drop into debugger on fail
- `--collect-only` - Don't run tests, just collect
- `-v` - Verbose (show test ids)
- `-m EXPR` - Run marks
- `-k EXPR` - Run tests with names (*keyword expression*)
- `NODE IDS` - Run tests with NODE IDS

Assignment 4

- Run only the `test_~~parse~~` test from the command line.

basic

Debugging

Debugging

Options:

- `import pdb;pdb.set_trace()`
- `assert 0` (in code) + `--pdb` (command line)
- Use `-s` to see stdout for successful tests

Command Line

- `-l` - Show local values
- `--lf` - Run *last failed* test first
- `--maxfail=N` - Stop after N failures
- `--tb=` - Control traceback (auto/long/short/no)
- `-v` - Show node ids
- `-x` - Exit after first fail (`--maxfail=1`)

Hint

Careful with `-l` (`--showlocals`) if running in CI and you have secrets you are using and don't want exposed

Hint

Consider combining `-x --lf` (exit after first fail and run with last fail first)

Hint

If you have hierarchical test directories, use `__init__.py` files (make them packages), otherwise you can't have two test files with the same name (ie `unit/test_name.py` & `reg/test_name.py`)

Doctest

Doctest

Update `pytest.ini` to permanently run doctests, with certain flags:

```
[pytest]
addopts = --doctest-modules

doctest_optionflags= NORMALIZE_WHITESPACE IGNORE_EXCEPTION_DETAIL
```

Doctest

Can use pytest fixtures with `get_fixture`:

```
# file.py
"""
>>> req = get_fixture('request')
>>> req.cache.get('bad_key')
None
"""
```

Injecting into Namespace

Python module that we typically import with shortened name `lf`:

```
# longfilename.py
"""
>>> lf.foo()
"""
def foo(): pass

# conftest.py
import longfilename
@pytest.fixture(autouse=True)
def add_lf(doctest_namespace):
    doctest_namespace['lf'] = longfilename
```

import pandas as pd

pd.read_csv

pd.DataFrame

pandas

'pd' = pandas

Assignment 5

- Create a doctest on the `integr.py` module that shows an example of running the `parse` function.
- Run the doctest via pytest with a command line option
- Create a `pytest.ini` file. Add an option to the configuration file to run the doctests when `pytest` is invoked
- Run the doctest via pytest without a command line option

Test Selection & Marking

Listing Tests

```
$ python -m pytest tests/*.py --collect-only
================= test session starts =================
platform darwin -- Python 3.6.4, pytest-3.0.6, py-1.4.32, pluggy-
0.4.0
rootdir: /Users/matt/code_samples/pytest/Project, inifile:
plugins: asyncio-0.8.0
collected 1 items
<Module 'tests/test_adder.py'>
  <Function 'test_add'>

============ no tests ran in 0.00 seconds ============
```

Test Selection

- Marking tests
- Skip tests

Marking Tests

Can create multiple "marks" with a decorator:

```
@pytest.mark.small
@pytest.mark.num
def test_ints():
    assert adder(1, 3) == 4
```

-m

Marking Tests (2)

Execute any tests that have *num* mark:

```
$ pytest -m num
```

Execute any tests that do not have *num* mark:

```
$ pytest -m "not num"
```

Marking Tests (3)

Can mark a class instead of marking every method

Marking Tests (4)

Can mark a module by creating a `pytestmark` global variable:

```
pytestmark = pytest.mark.num

# or a list of marks
pytestmark = [pytest.mark.num,
              pytest.mark.other]
```

Register Markers

To avoid typos, *register* markers in `pytest.ini` with:

```
[pytest]
markers =
  small: Tests with small numbers
  num: Tests on integers
```

Register Markers

Get *registered* markers:

```
$ pytest --markers
@pytest.mark.small: Tests with small numbers

@pytest.mark.num: Tests on integers

@pytest.mark.asyncio: mark...
```

Register Markers

Pytest will complain if a marker isn't registered. Will raise error with `--strict` flag

Named Tests

To run tests with "int" in name:

```
$ pytest -k int
```

Built-in Marks

- skipif
- xfail

Skipping tests

```
@pytest.mark.skipif(
    not os.environ.get("SLOWTEST"),
    reason="Don't run slow tests")
def test_big():
    assert adder(1e10, 3e10) == 4e10
```

Assignment 6

- Run the tests that have `bad` in the name
- Mark `test_bad1` and `test_bad2` with the `wrong` name.
- Run only tests that are marked with `wrong`.
- Run pytest with `--strict`
- Register `wrong` as a marker in `pytest.ini`
- Run pytest with `--strict`

Thanks!

Go forth and test!

Let's connect on:

- Twitter `@__mharrison__`
- LinkedIn