

# Pytest

@\_\_mharrison\_\_



# pytest

# pytest

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

# Installation

Create a virtualenv

```
(venv) $ pip install pytest
```

# Command line

Installs an executable called `py.test` (previously part of `py` library). With 3.0 can run `pytest` or `py.test`

# Assignment

# Assignment 1

# Basics

# Code Layout

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



# 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)
```

# Run Tests

```
$ PYTHONPATH=. 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 =====
```

# 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
```

# pytest style

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

# Assignment

# Assignment 2



# 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)
```

# Catching Exceptions (2)

Can specify an exception in decorator

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

# Failing a Test

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

# Approximations

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

# 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')
```

# Care needed (2)

You will get a warning:

```
$ py.test 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

```
$ py.test 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

# Assignment 3

# Test Runner

# Test Runner

For unittest add:

```
if __name__ == '__main__':  
    unittest.main()
```

or run:

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

# Test Runner

For pytest add:

```
if __name__ == '__main__':  
    import pytest  
    pytest.main()
```

or run:

```
$ py.test test_adder2.TestAdder
```

# Test Discovery

- Recurse current directory or testpaths from `pytest.ini`
- 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`
- `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
- `NODE IDS` - Run tests with NODE IDS

# Assignment

# Assignment 4

# 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)
- `-v` - Show node ids
- `-x` - Exit after first fail

# 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
```

# Assignment

# Assignment 5

# Test Selection & Marking

# Listing Tests

```
$ PYTHONPATH=./ 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

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



# Marking Tests

```
$ py.test -m num
```

or

```
$ py.test -m "not num"
```

# 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:

```
$ py.test --markers
```

```
@pytest.mark.small: Tests with small  
numbers
```

```
@pytest.mark.num: Tests on integers
```

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

# Register Markers

If you run with `--strict` it will complain if a marker isn't registered

# Named Tests

To run tests with "int" in name:

```
$ py.test -k int
```

# 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

# Assignment 6

# Thanks

Go forth and test!