

# 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.assertEqual(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
```

```
# confest.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

# Test Parameterization



# Test Parameterization

```
@pytest.mark.parametrize('x, y, z', [  
    (1, 2, 3), # first  
    (-1, -2, -3), # neg  
    (0, 0, 0)]) # test 0s  
  
def test_add2(x, y, z):  
    assert adder(x, y) == z
```

# Test Parameterization

Note that the Node Ids change:

```
$ PYTHONPATH=./ pytest tests/*.py -v
===== test session starts =====
platform darwin -- Python 3.6.4, pytest-3.0.6, py-1.4.32, pluggy-0.4.0 --
/Users/matt/.env/36/bin/python3
cachedir: .cache
rootdir: /Users/matt/code_samples/pytest/Project, inifile:
plugins: asyncio-0.8.0
collected 5 items

tests/test_adder.py::test_ints PASSED
tests/test_adder.py::test_big SKIPPED
tests/test_adder.py::test_add2[1-2-3] PASSED
tests/test_adder.py::test_add2[-1--2--3] PASSED
tests/test_adder.py::test_add2[0-0-0] PASSED

===== 4 passed, 1 skipped in 0.03 seconds =====
```

# Assignment

# Assignment 3

# Fixtures

# Fixtures

Provides dependency injection of setup / teardown

# Fixtures

```
@pytest.fixture
def large_num():
    return 1e20

def test_large(large_num):
    assert adder(large_num, 1) ==
large_num
```

# Fixtures Parameterization

```
@pytest.fixture(params=[-1, 0, 100])
def num(request):
    return request.param

def test_num(num_num):
    assert adder(num, 1) == num+1
```

# Method Fixtures

```
class TestAdder:
```

```
    @pytest.fixture
```

```
    def other_num(self):
```

```
        return 42
```

```
    def test_other(self, other_num):
```

```
        assert adder(other_num, 1) == 43
```



# Module Level

```
def setup_module(module):
```

```
...
```

```
def teardown_module(module):
```

```
...
```

# Class Level

```
class TestFoo:  
    @classmethod  
    def setup_class(cls):  
        ...  
  
    @classmethod  
    def teardown_class(cls):  
        ...
```

# Method Level

```
class TestFoo:  
    def setup_method(self, method):  
        ...  
  
    def teardown_method(self, method):  
        ...
```

# Function Level

```
def setup_function(function):
```

```
...
```

```
def teardown_function(function):
```

```
...
```

# Assignment

# Assignment 4

# More Fixtures

# Teardown in Fixtures

- Use request fixture and call `request.addfinalizer(fn)`
- Use generator

# request

Some parts of the request content:

- `r.addfinalizer(f)` - call when done
- `r.applymarker(m)` - dynamically add marker
- `r.config` - pytest config
- `r.keywords` - keywords and markers
- `r.param` - value of parameterization



# Finalizer

```
@pytest.fixture
def db_num(request):
    # connect to db
    num = db.get()
    def fin():
        db.close()
    request.addfinalizer(fin)
    return num
```

Note - can have more than one finalizer function

# Generator

```
@pytest.fixture
def db_num():
    # connect to db
    num = db.get()
    yield num
    db.close()
```

# Generator

Code smell:

```
from contextlib import closing

@pytest.fixture
def db_num():
    # connect to db
    with closing(get_db()) as db:
        num = db.get()
        yield num
```

# Fixture Scope

- `session` - Once per test session
- `module` - Once per module
- `class` - Once per test class
- `function` - Once per test function (default)

# Fixture Scope

```
@pytest.fixture(  
    scope='session')  
def start_time():  
    import time  
    return time.time()
```

# Fixture Scope

```
@pytest.fixture(  
    scope='session')  
  
def session_db():  
    db = get_db()  
    yield db  
    db.close()
```

# Fixture Scope

```
from contextlib import closing

@pytest.fixture(
    scope='session')
def session_db():
    with closing(get_db()) as db:
        yield db
```

# Fixture Scope

Finer grained scope can depend on larger grain,  
but reverse is not true



# Fixture Scope

```
# bad fixture depend  
@pytest.fixture(scope='function')  
def two():  
    return 2  
  
@pytest.fixture(scope='session')  
def four(two):  
    return two * two  
  
def test4(four):  
    assert four == 4
```

# Fixture Scope

```
===== ERRORS =====  
----- ERROR at setup of test4 -----  
ScopeMismatch: You tried to access the 'function'  
scoped fixture 'two' with a 'session' scoped  
request object, involved factories  
tests/test_adder.py:45: def four(two)  
tests/test_adder.py:41: def two()  
== 6 passed, 1 skipped, 1 error in 0.03 seconds ==
```

# Trigger skip from fixture

```
@pytest.fixture
def db_num(request):
    # connect to db
    try:
        num = db.get()
        return num
    except ConnectionError:
        pytest.skip("No DB")
```

# Pass data from marks to fixtures

```
@pytest.fixture
def db_con(request):
    name =
request.node.get_marker('pg_db').args[0]
    return psycopg2.connect("dbname={}".format(
        name))
```

```
@pytest.mark.pg_db('test')
def test_pg(db_con):
    # select from test db
```

# Skip tests on Mac

Use autouse=True to implicitly enable

```
@pytest.mark.nomac
def test_add_nomac():
    # ...

@pytest.fixture(autouse=True)
def skip_mac(request):
    mark = request.node.get_marker('nomac')
    if mark and sys.platform == 'darwin':
        pytest.skip('Skip on Mac')
```

# Assignment

# Assignment 5

# Monkey Patch Fixture

# Monkey Patch

Builtin fixture monkeypatch can:

- `chdir` - change current working directory
- `delattr` - remove attribute
- `delenv` - remove environment variable
- `delitem` - remove via index operation
- `setattr` - set attribute
- `setenv` - set environment variable
- `setitem` - set with index operation
- `syspath_prepend` - insert path into `sys.path`



# Monkey Patch

```
def test_mp(monkeypatch):  
    from proj import adder  
    def new_add(x, y):  
        return x - y  
    monkeypatch.setattr(adder, 'adder',  
new_add)  
    assert adder.adder(1,3) == -2
```

# Assignment

# Assignment 6

# Configuration

# Configuration

- Rootdir
  - Node ids determined from root
  - Plugins may store data there
- `pytest.ini` (or `tox.ini` or `setup.cfg`)
  - Must have `[pytest]` section

# Some INI Options

Run to get all of `pytest.ini` settings:

```
$ pytest --help
```

# Some INI Options

- `minversion = 4.0` - Fail if `pytest < 4.0`
- `addopts = -v` - Add verbose flag
- `norecursedirs = .git` - Don't look in `.git` directory
- `testpaths = regression` - Look in `regression` folder if no locations specified on command
- `python_files = regtest_*.py` - Execute files starting with `regtest_` (`test_*.py` and `*_test.py` default)
- `python_classes = RegTest*` - Use class starting with `RegTest` as a test (default `Test*`)
- `python_functions = *_regtest` - Use function ending with `regtest` as test (default `_test`)

# Conftest

Can create a `conftest.py` in a root directory or subdirectory. You can put fixtures in here. You don't import this module. Pytest loads it for you

# Assignment

# Assignment 7



# Plugins

# Plugins

You can have local plugins and installable plugins

# Many Hooks

- Bootstrap - for `setup.py` plugins
- Initialization hooks - for `conftest.py`
- `runtest` hooks - for execution
- Collection hooks
- Reporting hooks
- Debugging hooks

# Examples

- `pytest_addoption(parser)`
- `pytest_ignore_collect(path, config)`
- `pytest_sessionstart(session)`
- `pytest_sessionfinish(session, exitstatus)`
- `pytest_assertrepr_compare(config, op, left, right)`

[https://docs.pytest.org/en/latest/writing\\_plugins.html#writing-hook-functions](https://docs.pytest.org/en/latest/writing_plugins.html#writing-hook-functions)

# Plugin Boilerplate

<https://github.com/pytest-dev/cookiecutter-pytest-plugin>

# Installable Plugin

Need to implement `pytest11` entrypoint in `setup.py`, so `pytest` finds it.

# Installable Plugin

```
entry_points={  
    'pytest11': [  
        'pytest_cov = pytest_cov.plugin',  
    ],  
    'console_scripts': [  
    ]  
},
```

<https://github.com/pytest-dev/pytest-cov/blob/master/setup.py>

# Installable Plugin

```
def pytest_addoption(parser):  
    # Register argparse and INI options  
  
@pytest.mark.tryfirst  
def pytest_load_initial_conftests(early_config, parser,  
args):  
    # Bootstrap setuptools plugin  
  
def pytest_configure(config):  
    # Perform initial configuration
```

[https://github.com/pytest-dev/pytest-cov/blob/master/src/pytest\\_cov/plugin.py](https://github.com/pytest-dev/pytest-cov/blob/master/src/pytest_cov/plugin.py)



# Adding Commandline Options

In `conftest.py`:

```
def pytest_addoption(parser):  
    parser.addoption('--mac', action='store_true',  
                     help='Run Mac tests')
```

In tests:

```
@pytest.fixture  
def a_fixture(request):  
    mac = request.config.getoption('mac')  
  
def test_foo(pytestconfig):  
    mac = pytestconfig.getoption('mac')
```

# Assignment

# Assignment 8

# 3rd Party Plugins

# List

Python 2 & 3 compatibility

<http://plugincompat.herokuapp.com/>

# pytest-xdist

Distribute tests among (7) CPUs

```
$ pip install pytest-xdist
```

```
$ py.test -n 7
```

# pytest-flake8

Run flake8 on all py files

```
$ pip install pytest-flake8
```

```
$ py.test --flake8
```

# pytest-cov

## Run coverage

```
$ pip install pytest-cov
$ py.test --cov=adder --cov-report=html
tests/
# look at htmlcov/index.html
```

# Assignment

# Assignment 9



# Tox

# Tox

3rd party tool for running tests on different  
pythons

# Install

```
pip install tox
```

# Configuration

```
# tox.ini
[tox]
envlist = py27,py36
[testenv]
deps=pytest # use pytest
commands=pytest
```

# Configuration

Run `tox-quickstart` to generate config for you

# Running

At this point, if you run tox, it will:

- Create Python 2.7 venv
  - Install pytest
  - Create sdist and install package
  - Run package tests with pytest
- Create Python 3.6 venv
  - Install pytest
  - Create sdist and install package
  - Run package tests with pytest

# Jenkins CI

Can integrate with Jenkins by having Tox installed and having pytest output JunitXML files (with the `--junitxml` option)

# CircleCI

Contents of `circle.yml`:

```
dependencies:
```

```
  pre:
```

```
    - pip install tox
```

```
test:
```

```
  override:
```

```
    - tox
```



# Travis CI

Contents of `.travis.yml`:

```
language: python
```

```
python:
```

```
  - "2.7"
```

```
  - "3.6"
```

```
install: pip install tox-travis
```

```
script: tox
```

# Thanks

Go forth and test!