

Testing in Python with py.test

Holger Krekel
<http://merlinux.eu>

EuroPython 2009, Birmingham

June 30, 2009



my testing background

- programming since 20 years
- Python since around 2000
- founded PyPy on TDD principles
- py.test since 2003
- couple of merlinux projects with TDD
- consultancies on testing

The developer test tool question

what is the job of automated testing tools?

my current answer

- verify code changes work out
- be helpful when tests fail

If failures are not helpful ...

1. improve the test tool
2. write more or write different tests

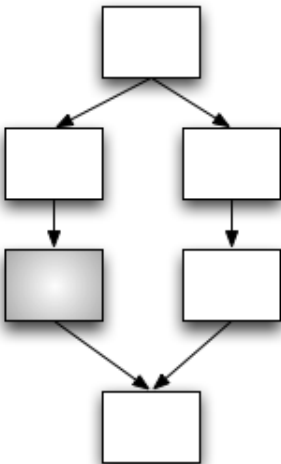
Developer oriented automated tests

- unittest: units react well to input.
- integration: components co-operate nicely
- functional: code changes work out in user environments

targets of py.test 1.0!

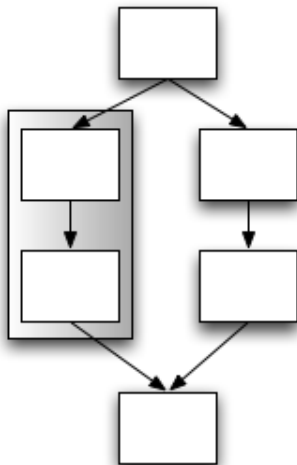
unittest

assert that functions and classes behave as expected



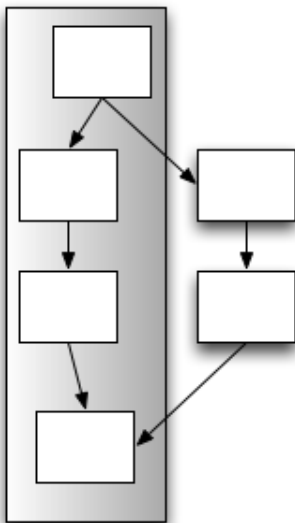
integration

assert units/components co-operate nicely



functional / system

things work in user target environment



py.test basics



<http://marikaz.deviantart.com/> CC 3.0 AN-ND

py.test aims and goals

- cross-project testing tool
- no-boilerplate concise test code
- useful information when a test fails
- deep extensibility (since 1.0)
- dynamic test distribution to multiple hosts

cross-project test tool

- tests are run via `py.test` command line tool.
- project specific `conftest.py` files
- testing starts from files or directories

examples:

```
py.test test_file.py
```

```
py.test path/to/tests
```

```
py.test # current directory
```

A Typical Python test layout

```
mypkg/__init__.py
```

```
...
```

```
mypkg/tests/test_module.py
```

```
...
```

example invocation:

```
py.test mypkg
```

Another typical test layout

```
mypkg/__init__.py
```

```
...
```

```
test/test_module.py
```

example invocations:

```
py.test test
```

```
py.test # curdir == parent of mypkg
```

mind the tests/___init___.py files

- provide `___init___.py` with your tests
- test files are imported as normal python modules
- `sys.path` is amended to contain the first non-`___init___.py` dir

automatic test discovery

py.test walks over your **whole** source tree and:

- discovers `test_*.py` and `*_test.py` test files
- discovers `test_` functions and `Test` classes

automatic discovery avoids boilerplate

no boilerplate python test code

```
# test_module.py

def test_something():
    x = 3
    assert x == 4

class TestSomething:
    def test_something(self):
        x = 1
        assert x == 5
```

assert re-interpretation

no boilerplate assertions:

```
# test_module.py
def test_assert_introspection():
    # unittest.py
    assert x          # assertTrue()
    assert x == 1     # assertEquals(x, 1)
    assert x != 2     # assertNotEqual(x, 2)
    assert not x      # assertFalse(x)
```

testing for exceptions

```
import py

def func(x):
    raise ValueError

def test_raises_one():
    py.test.raises(ValueError, func, 10)

def test_raises_two():
    py.test.raises(ValueError, "func(10)")
```

Failure / Traceback Demo

interactive demo: `py.test failure_demo.py`

print() debugging

```
def test_something1():  
    print "Useful debugging information."  
    assert False
```

print captured by default, only shown on failure.

Skipping tests

Skipping a test because of **platform mismatch**:

```
def test_function_win32():
    if sys.platform != "win32":
        py.test.skip("win32 needed")
    ...
```

or because of **missing dependency**:

```
# ./test_module.py
docutils = py.test.importorskip("docutils")
```

“XFailing” tests

mark a test function as “expected to fail”:

```
import py
@pytest.mark.xfail
def test_function():
    assert f() == "thisresult"
```

some important options

see also `py.test -h`:

- `-s` disable catching of stdout/stderr
- `-x` exit instantly on first failure
- `-k` only run tests matching the given keyword.
- `-l` show locals in tracebacks.
- `--pdb` start Python debugger on errors
- `--tb/fulltrace` - control traceback generation.

nose and py.test

“nose provides an alternate test discovery and running process for unittest, one that is intended to mimic the behavior of py.test as much as is reasonably possible without resorting to too much magic.” -- from nose home page

nosetests and py.test share basic philosophy and features

“magic” bits

regarding `py.test` maybe this:

- assert re-interpretation (disable with “`--nomagic`”)
- lazy importing from explicit exports
(`py/__init__.py`)
- usage of `setuptools`
- “`py`” naming?

unique py.test features

- funcargs (since 1.0)
- per-project configurability
- simple hook mechanism (since 1.0)
- assert re-interpretation
- testing non-python items
- distributing tests multi-platform (since 1.0)

Test Function arguments

- new way to manage test fixtures
- also enables test-function aware helpers

old-style strategy: xUnit-style fixtures

```
from mypkg.mod import SomeClass
import unittest
class TestGroup(unittest.TestCase):
    def setUp(self):
        self.inst1 = SomeClass("somevalue")

    def test_something(self):
        ... use self.inst1 ...
        assert self.inst1.method() == "ok"
```

meet “funcargs” - test function arguments

```
# test_module.py

def test_something(inst1):
    assert inst1.method() == "ok"

class TestClass:
    def test_something(self, inst1):
        assert inst1.method() == "ok"
```

providing funcargs

```
# conftest.py (or test_module.py)

from mypkg.mod import SomeClass
def pytest_funcarg__inst1(request):
    return SomeClass("somevalue")
```

Adding funcarg finalizers / teardown

```
def pytest_funcarg__file(request):  
    f = open(...)  
    request.addfinalizer(lambda: f.close())  
    return f
```

DEMO `py.test -s test_addfinalizer.py`

cached_setup(): long-living funcargs

```
def pytest_funcarg__myarg(request):  
    return request.cached_setup(  
        setup=lambda: Myarg()  
        teardown=lambda arg: arg.finalize(),  
        scope="module"  
    """)
```

DEMO `py.test -s test_cachedsetup.py`

parametrizing tests

why not run a test function with multiple funcarg values?

pytest_generate_tests FTW

self-contained example:

```
def test_function(param):  
    assert param < 3  
  
def pytest_generate_tests(metafunc):  
    if "param" not in metafunc.funcargnames:  
        return  
    for i in range(3):  
        funcargs = {'param': i}  
        metafunc.addcall(funcargs=funcargs)
```

py.test's new parametrizing tests

- can live in `conftest.py` or plugin
- enables all common parametrization schemes

see blog post: <http://tinyurl.com/ofry94>

funcargs as test-function-aware helpers

for example monkeypatching:

```
def test_envreading(inst1, monkeypatch):
    monkeypatch.setattr(os.path, 'listdir',
                        lambda x: return ["x"])
    l = inst1.contents()
    assert len(l) == 1
    ...
```

see post <http://tinyurl.com/ob87o6>

summary py.test funcargs

- readable no-boilerplate test code
- complexity of fixtures does not affect test code
- one place to manage fixtures for tests
- enables new “test-function aware” helpers

Hooks and Customizing



<http://marikaz.deviantart.com/> CC 3.0 AN-ND

Customizing py.test runs

- implement hooks in `conftest.py` or named plugin
- implement new funcargs
- invent new test items
- specify plugins / command line opts in `conftest.py`

conftest.py

- **project specific test configuration**
- auto-discovered in *test* directory or higher up
- can contain hook implementations
- can provide default values for command line options
- can specify plugins to load

a simple config: ignoring paths

```
# conftest.py  
collect_ignore = ['data', 'test_myfile.py']
```

paths are relative to directory of conftest.py

a simple default option: verbose

```
# conftest.py
pytest_option_verbose = True
pytest_option_exitfirst = True
```

a simple hook: adding a command line opt

```
# conftest.py
def pytest_addoption(parser):
    parser.addoption("--url",
                    action="store", default=None)

def pytest_funcarg__url(request):
    url = request.getvalue("url")
    if url is None:
        py.test.skip("need --url")
    return url
```

Hooks are Python functions

- hooks are python functions with a `pytest_` prefix
- your hook functions need to use *exact argument names*
- you can just provide a single hook

Named Plugins

- normal python modules or packages
- fixed “pytest_NAME” naming scheme
- contains hooks

pytest plugins often single-file and include tests

specifying plugins

- command line: `py.test -p NAME`
- env variable: `PYTEST_PLUGINS=NAME`
- test module / `conftest.py`: `pytest_plugins=[...]`

Example “new test item” plugins

pytest_unittest: run traditional `unittest.py` modules

pytest_doctest: run traditional doctests

pytest_restdoc: verify ReST syntax and local/remote URLs

testing javascript in live browsers: see the next talk!

runtest/report plugin examples

- `pytest_xfail.py`: display xpassed/xfailed specially
- `pytest_figleaf.py`: generate coverage report at end
- `pytest_pdb.py`: invoke PDB on python test failure
- `pytest_resultlog.py`: produces buildbot friendly output

Summary customization

- `conftest.py` for per-project test config and hooks
- named plugins for global test support code
- many hooks for interacting with `py.test`

hook specs: `py/test/plugin/hookspec.py`

default plugin examples: `py/test/plugin/pytest_*`

Distributed Testing



<http://marikaz.deviantart.com/> CC 3.0 AN-ND

send test to specific python interpreter

```
py.test --dist=each \  
    --tx popen//python=python2.4
```

send test to three different interpreters

```
py.test --dist=each \  
    --tx=popen//python=python2.4 \  
    --tx=popen//python=python2.5 \  
    --tx=popen//python=python2.6
```

Example: Speed up test runs

To send tests to multiple CPUs, type:

```
py.test --dist=load \  
  --tx popen --tx popen --tx popen
```

or in short:

```
py.test -n 3
```

Especially for longer running tests or tests requiring a lot of IO this can lead to considerable speed ups!

send tests to remote SSH account

```
py.test --d --tx ssh=myhostpopen \  
--rsyncdir mypkg mypkg
```

Distribution modes

```
py.test --dist=each mypkg
```

```
py.test --dist=load mypkg
```

how does py.test distribute tests?

it uses py.execnet:

- instantiates local or remote Python Processes
- send code for execution in one or many processes
- asynchronously send and receive data between processes through channels
- completely avoid manual installation steps on remote places

see talk Thursday morning for intro and discussion

xspec distribution specification

```
popen//python=2.5//nice=20  
ssh=wyvern//python=python2.4//chdir=mycache  
socket=192.168.1.4:8888
```

“running tests until all pass”

```
py.test --looponfailing
```

“running tests even if they crash”

```
py.test --boxed  
(requires os.fork)
```

Roadmap

- 1.0.0b7 out a few days ago
- 1.0 next week (barring setuptools issues)
- 1.1 planned for end September:
 - ▶ full Python3 compatibility
 - ▶ plugin for virtualenv integration
 - ▶ move away from setuptools
 - ▶ complete move to mercurial

please mail issues/needs, **need-driven development**

<http://pytest.org>

Feedback round

How did you like the tutorial?

Thanks for taking part!

holger krekel at merlinux eu

<http://pytest.org>