

rapid testing with py.test

Holger Krekel
<http://merlinux.eu>

EuroPython 2009, Birmingham

June 29, 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

What's your background?

- what test tools do you use?
- work test-driven?
- have test-suites with >1000 tests?
- need/want to run tests on >1 platforms?
- have “non-python” tests?

The 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 ...

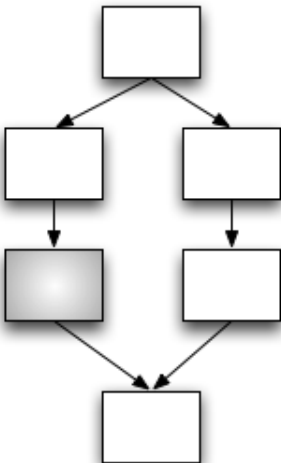
improve the test tool or
write more or write different tests

Developer oriented automated tests

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

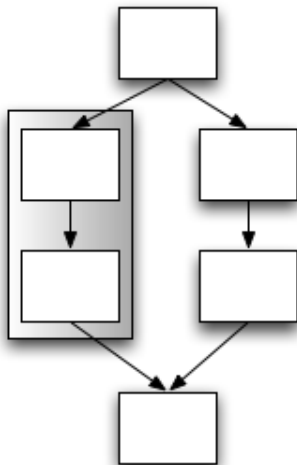
unittest

assert that functions and classes behave as expected



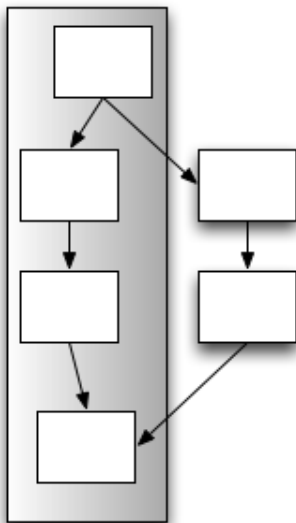
integration

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 fundamental features

- cross-project testing tool
- no-boilerplate test code
- useful information when a test fails
- deep extensibility
- distribute tests 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/dirs or current dir

examples:

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

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

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 test files
- discovers test_ functions and Test classes

automatic discovery avoids boilerplate

no boilerplate python test code

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

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

assert introspection

```
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 test_raises_one():
    py.test.raises(ValueError, int, 'foo')

def test_raises_two():
    py.test.raises(ValueError, "int('foo')")
```

Failure / Traceback Demo

interactive demo: `py.test failure_demo.py`

print() debugging

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

output captured per function run, 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

If you want to mark a test function as “expected to fail”:

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

no traceback shown / reports specially if it passes.

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.
- `-f/--looponfail` - loop on failing test set.

Exercise 1 (max 15 minutes)

`easy_install -U py` or:

- *svn co <http://codespeak.net/svn/py/trunk>*
OR
 - *hg clone <https://bitbucket.org/hpk42/py-trunk/>*
 - *run "python setup.py" with "install" or "develop"*
-
- `create mypkg/test/test_hello.py` (plus `__init__.py` files)
 - write a simple test function
 - `run py.test mypkg`, play around with options

test functions and funcargs

funcargs are new since 1.0, unique py.test feature
let's first look at the motivation ...

Typical test function - viewed abstractly

```
from mypkg.mod import SomeClass
def test_something():
    inst1 = SomeClass("somevalue")
    ...
    assert inst1.method() == "ok"
```

observations and the fixture question

- test value for Instantiation mixes with test code
- the `mypkg.mod.SomeClass` reference may change

What if multiple test functions need to create an instance as well to perform their tests?

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"
```

observations / notes

- unittest-runner automatically invokes `self.setUp()` for each test function invocation.
- test value for Instantiation mixes with test code
- `mypkg.mod.SomeClass` reference may change
- **test module has 7 LOCs for one test function**
- **multiple methods reuse the same setup**
- **test functions group by setup code**

meet “funcargs” - test function arguments

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

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

observations

- test function “requests” arguments it needs
- grouping in test classes independent of setup
- functions/methods re-use same funcarg setup

How to provide the function argument?

```
#!/confptest.py (or test module)

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

funcarg provider observations

- discovered through naming convention
- loosely coupled, separated from test code
- will be called for each test function
- has one `mypkg.mod.SomeClass` reference

Exercise

- write a package “mypkg”
- add mypkg/___init___ and mypkg/test_url.py
- add a def test_path(url) function
- write a pytest_funcarg__url provider in mypkg/conftest.py
- run your test

Bonus: play with wrong funcarg names

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

DEMO `py.test -s test_cachedsetup.py`

funcarg request object attributes

funcarg providers always receive a request object:

`request.function`: python test function

`request.cls`: class of test function or None

`request.module`: module where test function lives

`request.config`: config object (*)

mysetup pattern: avoiding too-many-funcargs

```
# ./test_sample.py
def test_answer(mysetup):
    app = mysetup.myapp()
    answer = app.question()
    assert answer == 42
```

-> mysetup funcarg can easily grow more objects

calling test functions multiple times

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)
```

pytest_generate_tests notes

- test function just takes funcargs
- can live in conftest.py or plugin
- enables various parametrization schemes
- “generative” yield-tests now deprecated

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

test-function aware helpers ...

```
def test_envreading(inst1, monkeypatch):  
    monkeypatch.setenv('INST1', 'myval')  
    val = inst1._readenv()  
    assert val == "myval"
```

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

summary py.test funcargs

- readable no-boilerplate test code
- complexity of fixtures does not affect test code
- can easily be configured
- one place to manage fixtures for tests
- also enable “test-function aware” helpers

(questions / demo'ing as time permits)

- showcase `py.test`'s `testdir` usage
- showcase `request.getfuncargvalue()`
- showcase/discuss `mysetup` in the docs

Break



Hooks and Extensions (45 minutes)



<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(s)
- 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 directories

```
# 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
- hook functions need to use *exact argument names*

Named Plugins

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

specifying plugins

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

hook categories

- configuration hooks
- collection hooks
- runtest hooks
- reporting hooks
- testing process (“event”) hooks

see `py/test/plugin/hookspec.py`

Configuration Hooks

```
def pytest_addoption(parser):  
    "called before commandline parsing."  
  
def pytest_configure(config):  
    "called after commandline parsing."  
  
def pytest_unconfigure(config):  
    "called before test process is exited."
```

config object / tempdirs

- `config.getvalue(optname)`: return cmdline option value
- `config.mktemp(basename)`: create and returns a new tempdir
- `config.ensuretemp(basename)`: create or return a new tempdir

tempdirs are created as subdirs of a per-session testdir

sidenote: py.path objects

py.test often uses `py.path.local` objects for accessing the filesystem:

```
>>> p = py.path.local() # curdir
>>> newp = p.join("hello.txt")
>>> newp.write("file content of hello.txt")
>>> s = newp.read()
>>> ...
```

exercise

write a named plugin `pytest_tmpfs.py` which contains a `pytest_funcarg__tmpfs` provider which creates a `TmpFS` instance. The `TmpFS` class should contain this method:

makefile(basename, content) -> py.path.local object

and your test module should contain a simple test function that uses `tmpfs` to create a file and checks that the content has been written.

py.test collection tree

- collection tree is built iteratively
- test collection starts from directories or files
- `collector.collect()` returns “colitem” list

py.test.collect.* objects

- **Directory**
- **File**
- Python specific collection objects:
 - ▶ Module
 - ▶ Class
 - ▶ Instance
 - ▶ Function

Demo/Exercise “collection tree”

inspecting the test collection tree of your own examples:

```
py.test --collectonly
```

Collection Hooks

```
def pytest_collect_directory(path, parent):  
    "return Collection node or None. "
```

```
def pytest_collect_file(path, parent):  
    "return Collection node or None."
```

```
def pytest_pycollect_makeitem(collector, name, c:  
    "return custom item/collector or none."
```

Example “new item” plugins

pytest_unittest: run traditional unittest.py modules

pytest_doctest: run doctests

pytest_restdoc: check ReST syntax and remote URLs for .txt files

testing javascript in live browsers: see talk tuesday afternoon!

runtest hooks

```
def pytest_runtest_setup(item):  
    "called before pytest_runtest_call()."  
  
def pytest_runtest_call(item):  
    "execute test item."  
  
def pytest_runtest_teardown(item):  
    "called after pytest_runtest_call()."  
  
def pytest_pyfunc_call(pyfuncitem):  
    "perform python function call."
```

Reporting hooks

```
def pytest_runtest_makereport(item, call):  
    "make report for call object."
```

```
def pytest_runtest_logreport(rep):  
    "process report for logging."
```

```
def pytest_report_teststatus(self, event):  
    "return cat, shortletter, verbose word."
```

```
def pytest_terminal_summary(self, terminalre-  
porter):  
    "add section to summary reporting."
```

runtest/report related examples

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

the `py.test.*` namespace hook

if you need a global helper:

```
#!/conftest.py or named plugin
```

```
def mycheck(val):  
    assert val == 42
```

```
def pytest_namespace(config):  
    return {'mycheck': mycheck}
```

```
#!/test_module.py
```

```
def test_something():  
    py.test.mycheck(41)
```

pytest_namespace examples

`py.test.mark` for settings keywords

`py.test.deprecated_call` for checking for deprecation calls

doing helpers in py.test

funcargs: test-function run aware

py.test.NAME: globally available

test code does not need to import/know implementation location

Summary customization

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

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

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

Distributed Testing (30 minutes)



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

the basic idea

1. collect tests locally
2. run tests in one/more execution processes.

test distribution modes

--dist=load # load-balance tests to exec environments

--dist=each # send each test to each exec environment

send test to one other 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
```

Sending tests to remote socket servers

Download and start `py/execnet/script/socketserver.py`
or

`http://codespeak.net/~hpk/socketserver.py`

Assuming an IP address, you can now tell `py.test` to distribute:

```
py.test --dist=each --  
tx socket=192.168.1.102:8888 \  
    --rsyncdir mypkg mypkg
```

Exercise: Move settings into conftest

mypkg/conftest.py:

```
rsyncdirs = ['.']
```

```
pytest_option_tx = ['ssh=myhost', 'socket=...']
```

Distribution modes

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

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

how does py.test distribute tests?

it uses **py.execnet**

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

xspecs: Exec environment specs

general form:

```
key1=value1//key2=value2//key3=value3
```

xspec key meanings

- `popen` for a `PopenGateway`
- `ssh=host` for a `SshGateway`
- `socket=address:port` for a `SocketGateway`
- `python=executable` for specifying Python executables
- `chdir=path` change remote working dir to given relative or absolute path
- `nice=value` decrease remote nice level if platforms supports it

Exercise

- Play with running your tests in multiple interpreters or processes
- see if you can get access to your neighbour's PC

xspec examples:

```
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)
```

Feedback round

How did you like the tutorial?

Thanks for taking part!

holger krekel at merlinux eu

<http://pytest.org>