py.test II - advanced configs and usages

Holger Krekel
http://merlinux.eu

EuroPython 2010, Birmingham

July 17th, 2010

these slides

If you have time before the tutorial starts, you may make sure you have installed the latest pylib. You need either:

- `setuptools`: http://pypi.python.org/pypi/setuptools
- `distribute`: http://pypi.python.org/pypi/distribute

to be working on your system and can then type:

```
easy_install py   # or
pip install py
```

slides and examples:

```
http://pytest.org/tutorial/pytest-ep2010.zip
```

me

Holger Krekel <holger@merlinux.eu>

- developer of py.test, execnet and PyPy
- founded the PyPy project
- Python since around 2000
- Test-driven development since 2001
- operates *merlinux*, a small company doing open source, Python and test-oriented consulting

you

What is your background?

- Python background
- Testing background
- python testing libraries: unittest, nose, py.test

why automated testing?

- to raise confidence that code works
- to specify and document behaviour
- to allow for later changes

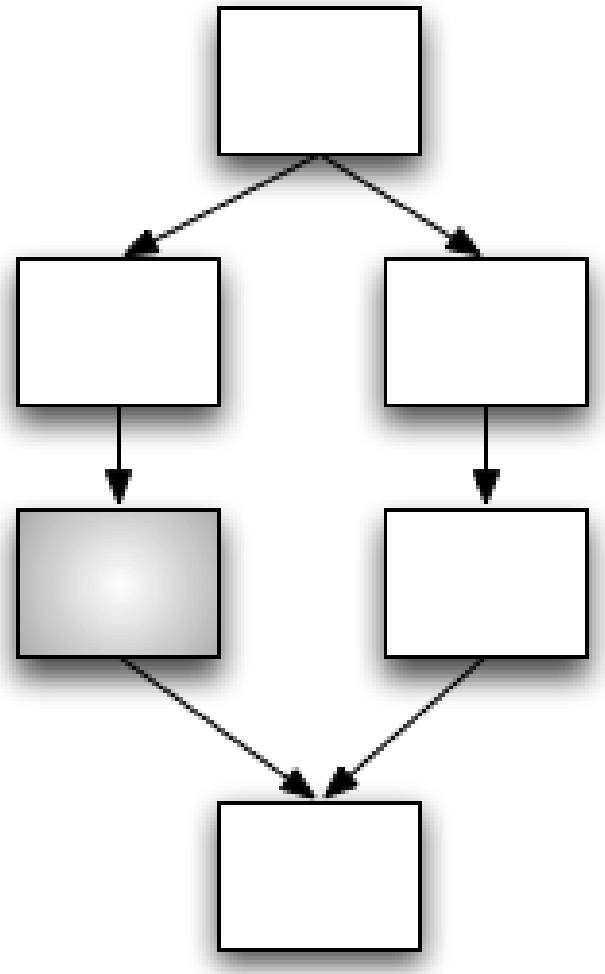
more collaborative and faster development cycles!

Developer oriented automated tests

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

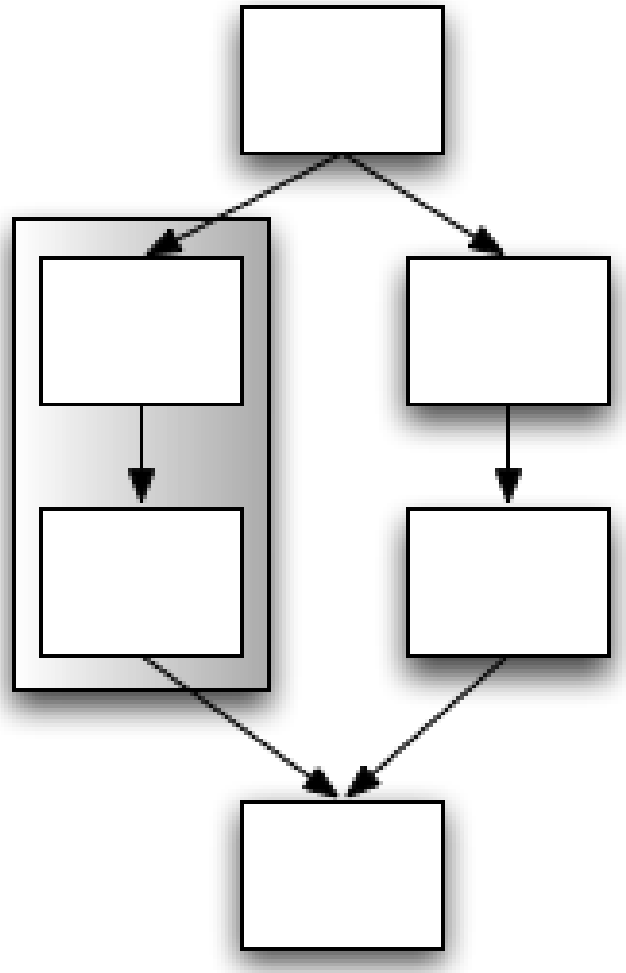
unittest

assert that functions and classes behave as expected



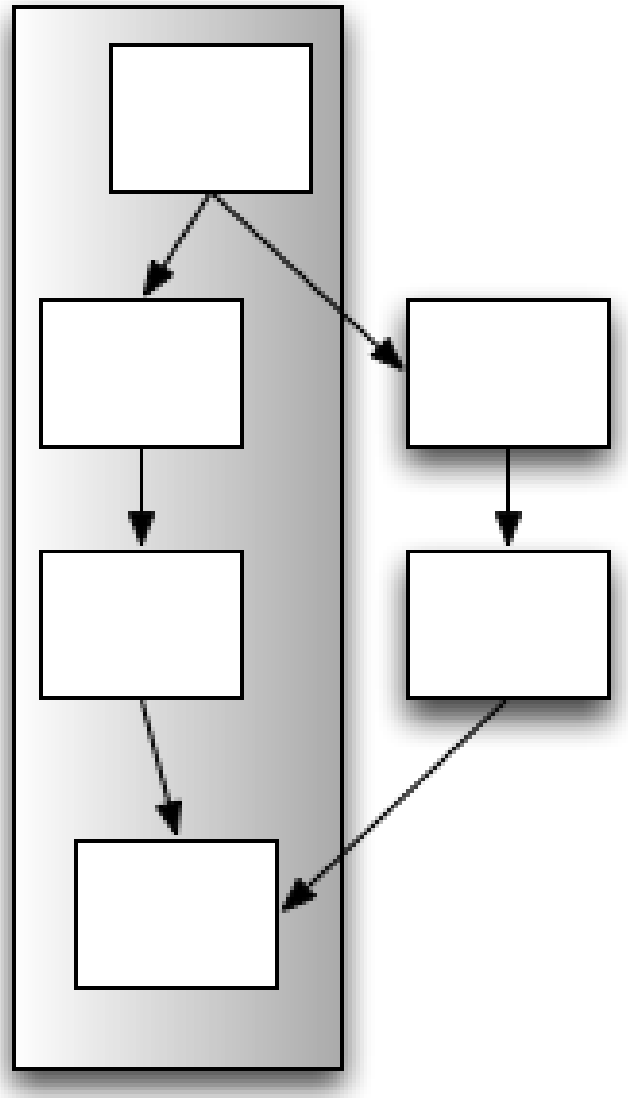
integration

assert units/components co-operate nicely



functional / system

assert things work in user target environment



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
- make writing tests easy and fun

If failures are not helpful ...

improve the test tool or
write more or write different tests

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.
- you may have 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 # starts discovery in cwd
```

Choosing a test layout

inlined test directories:

- associates code to tests
- useful for unit-tests

external test directories:

- separates tests from code
- useful for integration and functional testing
- more freedom to organise and group tests

my preference these days: external

automatic test discovery

py.test walks over the filesystem and:

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

automatic discovery avoids boilerplate

mind the __init__.py files

- test files are imported as normal python modules
- make test directories a package to disambiguate

Bullet list ends without a blank line; unexpected unindent.
first non ‘__init__.py‘ dir is inserted into sys.path

no boilerplate python test code

example test functions/methods:

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

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

assert introspection

use the plain assert statement:

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

py.test details the assert expression values if an assertion fails

asserting expected exceptions

Two ways to assert for expected exceptions:

```
import py

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

or if you don't have >=Python2.5:

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

print() debugging / output capturing

you can use the print statement in tests:

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

standard output is captured per function run and only shown on failure.

Getting Started, basic usage [0]

use an installation tool:

```
easy_install -U py  # or
pip install py
```

or use the source:

```
hg clone https://bitbucket.org/hpk42/py-trunk/
cd py-trunk
python setup.py develop
```

more info: http://pylib.org/install.html
slides and examples:

```
http://pytest.org/tutorial/pytest-ep2010.zip
```

Failure / Traceback Demo [0]

exercise: go to directory `pytest-basic/0` and play with:

```
py.test test_failures.py
```

particularly play with options (`py.test -h`):

- -s disable catching of stdout/stderr
- -x exit instantly on first failure
- -l show locals in tracebacks.
- --pdb start Python debugger on errors
- --tb/fulltrace - control traceback generation.

Skipping tests

when to skip a test?

- if it runs on an unfitting platform
- if an (optional) package is missing
- if test configuration (e.g. remote host address) is missing

note:

- a test skip is always conditional
- it will show up as 's' in progress output

Dynamically skipping tests

Skipping a test because environment setting is missing:

```
def test_mysetting():
    if "MYSETTING" not in os.environ:
        py.test.skip("set MYSETTING env-var")
    ...
```

or because of missing import dependency:

```
def test_missing_import_dependency():
    mod = py.test.importorskip("notexist")
```

Marking test functions

builtin markers for skipping/xfail-ed tests:

```
py.test.mark.skipif(expr)
py.test.mark.xfail(expr)
```

or use your own custom marking:

```
py.test.mark.YOURNAME
```

all mark decorators set an according attribute on the function object which you can check later on in your test support code.

Marking a test for conditional skip

example of skipping a test on non-win32 platforms:

```
@py.test.mark.skipif("sys.platform != 'win32'")
def test_function():
    ... # will only run on win32 platform
```

NOTE: `"sys.platform != 'win32'"` is evaluated in a dict that provides `os`, `sys` and `config` (the py.test config object)

Marking a test as expected to fail

mark a test as “expect-to-fail”:

```
@py.test.mark.xfail
def test_function():
    ...
```

notes:

- if it fails, reported as ’x’, no traceback shown
- if it passes, it is reported as “unexpectedly passing”
- nice for testing known bugs or future features
- useful for incremental refactoring of larger test suites

Conditional expected to fail

a test may only fail on a certain platform:

```
@py.test.mark.xfail("sys.platform == 'win32'")
def test_function():
    ...
```

notes:

- if run on non-win32 platform the test runs "normal"
- if run on win32 the test is marked expected-to-fail

Marking / Skipping exercise [0]

go to directory `pytest-basic/0` and play with:

```
py.test test_mark_and_skip.py
```

particularly play with these options:

```
-k "mysetting"
-k "-mysetting"
-k "functional"
-rs # report skipped
-rx # report xfailed
-rsx # report skipped and xfailed
```

Applying Marks to classes

If you need to apply a marker (skipif,xfail,custom) to all test methods of a class you can use `py.test.mark` as a class decorator (new in Python2.6):

```
@py.test.mark.functional
class TestClass
    ...
```

And if you need your tests to work on Python2.5 or below:

```
class TestClass:
    pytestmark = py.test.mark.functional
```

Applying Marks to modules

If you want to apply a marker to all test functions of a certain test module, you can set `pytestmark` likes this:

```
pytestmark = py.test.mark.skipif(...)
```

`py.test` will recognize the `pytestmark` name and apply the marker to all test functions in the module.

break

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

warm up: boilerplate test code

boilerplate test code is repetetive code, often copy-pasted and slightly modified:

- makes it harder to understand the core idea of a test
- makes it tedious to refactor your application later
- makes it tedious to refactor your tests
- makes it hard to recognize little differences in tests

We would like to avoid boilerplate / redundant test code

“myscan” Exercises of this tutorial

- we’ll step-by-step build up a simple `myscan` project
- purpose of `myscan`: scan text files for URLs and check them remotely
- purpose of exercises: getting used to basic idioms and testing work style

sketching our “myscan” project

Application Objects:

- Config object: read configuration from an .ini file
- Scanner object: scan text files for urls and check accessability

Tests will be about:

- config file parsing
- using the scanner object and its method(s)
- separating test setup (fixtures) from test code

Starting Exercise [1]

- look at `1/myscan/config.py` and `1/test/test_myscan.py`
- run `py.test -v`
- maybe introduce a bug to `config.py`
- then use `py.test --pdb`, `-x` etc.
- use `py.test -k fails` to select the exception-test

The example Solution [1]

here is how the test function looks like:

```
def test_config_extensions():
  tmpdir = tempfile.mkdtemp()
  path = os.path.join(tmpdir, 'config.ini')
  f = open(path, 'w')
  f.write("[myscan]\nextensions = .txt .rst")
  f.close()
  cfg = Config(path)
  assert cfg.extensions == ['.txt', '.rst']
```

only three lines deal with app-specific info, the rest is "setup" and preparation code

The "test setup" and fixture Problem

consider:

- to test more configs and variants we need to repeat setup code
- code redundancy is bad, also in tests
- refactoring later likely will require test changes

py.test style test fixtures:

- xUnit/unittest.py/nosetests style setup supported
- but it can do better :)

test functions and funcargs

funcargs are a unique py.test feature:

- a mechanism for managing your test fixtures
- individual fixtures can be managed separately
- full separation between test code and fixtures
- can easily depend on command line options
- convenient “builtin” funcargs provided

How do funcargs work?

- a factory creates an instance of a funcarg
- a test function accepts the instance as a parameter

full self-contained example:

```
# ./content of extra/test/test_funcarg_simple.py

def test_function(myarg):
    assert myarg == 42

# the 'myarg' factory function
def pytest_funcarg__myarg(request):
    return 42
```

funcarg factory notes

The factory function:

```
def pytest_funcarg__myarg(request):
    return 42
```

- is discovered through naming convention
- can live in `conftest.py` or in test module
- loosely coupled, separatable from test code
- will be called for each requesting test function
- the `request` object provides information and interaction related to test_function invocation.

builtin funcargs

py.test provides builtin funcargs which you can use in any test function:

- `tmpdir` provide a unique temporary directory
- `monkeypatch` temporarily patch objects and ENVs
- `recwarn` assert that a warning was issued
- `pytestconfig` the py.test config object

to show builtin and your own funcargs:

```
py.test --funcargs [path/to/my/tests]
```

usage example: per-test temporary directory

previous test function rewritten to use `tmpdir`:

```
def test_config_extensions(tmpdir):
    path = tmpdir.join("test.ini")
    path.write("[myscan]\nextensions = .txt")
    config = myscan.config.Config(path)
    assert config.extensions == ['.txt']
```

notes:

- `tmpdir` points to a fresh per-test-invocation tmpdir
- we use `py.path.local` objects here, see http://pylib.org/path.html

next step: adding more tests [2]

now we can add more tests using `tmpdir`:

- test_config_urlprefix: allow to configure more url-prefixes via a config option
- test_extract_urls: call a Scanner which is to extract http/s urls from a text file
- test_extract_urls_custom: call a Scanner which is to extract custom-urls from a text file

exercise:

- play with `py.test 2/test/test_myscan.py`
- also use `py.test --basetemp=somepath` and inspect

observation on new tmpdir-using tests

- boilerplate is reduced compared to previous approach
- but we have new boilerplate/repetetive code!
- we want to factor out this code into a single place

exactly the purpose of the “mysetup” pattern!

The mysetup pattern

- implemented as a funcargs
- is a glue object between test code and application
- incrementally grows helper methods to avoid redundancy in tests
- serves to make tests minimal and robust against future changes

most useful for integration and functional testing

How the mysetup pattern works

A "mysetup" factory provides an instance on which we can define further app-specific test helper methods:

```
class MySetup:
    def __init__(self, request):
        self.tmpdir = request.getfuncargvalue("tmpdir")

def pytest_funcarg__mysetup(request):
    return MySetup(request)
```

see also: http://pytest.org/funcargs.html#funcarg-factory-request

next step: introducing "mysetup" pattern [3]

new base `3/myscan` exercise directory has:

- test functions now receive `mysetup`
- `mysetup.makeconfig(**kwargs)` creates a config file

exercise:

- look at `MySetup` definition in `3/test/conftest.py`
- look at tests in `3/test/test_myscan.py`
- run tests
- create a `mysetup.writefile(basename, content)` helper to write text files and refactor tests to use it
- (bonus: extend a test to also check for 'https' urls)

The new test function using “mysetup”

Here is how our previous config-checking now looks like:

```
def test_config_extensions(mysetup):
    config = mysetup.makeconfig(extensions=".txt .rst")
    assert config.extensions == ['.txt', '.rst']
```

notes:

- tempfile creation/management factored out
- config instantiation factored out
- “mysetup” object fully responsible for app <-> test glue
- rather safe against future refactorings!

a simple hook: adding a command line opt

You can add project-specific command line opts:

```
# conftest.py
def pytest_addoption(parser):
    parser.addoption("--checkremote",
        dest="checkremote",
        action="store_true", default=False)
```

accessible via `config.getvalue('checkremote')` or `request.config.getvalue('checkremote')` from funcarg factories.

the py.test config object

- inside funcarg factories: `request.config`
- as a direct funcarg: `pytestconfig`
- provides access to command line option values
- also provides access to the pluginmanager (advanced)

exercise: implement a new command line option [4]

use `4/myscan` as base and write a new test that:

- is decorated with `@py.test.mark.remote`
- creates a test file with some example urls content
- calls a new (to be implemented afterwards) `scanner.checkremote(path)` method that is to extract urls and verify that the urls are accessible.

extend `conftest.py`:

- introduce a `--checkremote` boolean option option
- in the `pytest_funcarg__mysetup` factory check existence of the `remote` marker with `hasattr(request.function, 'remote')` and skip the test if the command line option was not provided

myscan example completed

what we have done:

- implemented tests for config and scanner behaviour
- factored app-specific test fixtures into conftest.py and the MySetup object (easy-to-extend)
- implemented a mechanism to mark tests that involve remote accesses and only run if the command line `--checkremote` is specified

questions, remarks?

Installing “global” plugins

use common package installation tools:

```
easy_install pytest-figleaf
pip install pytest-figleaf
```

with pip you can also uninstall:

```
pip uninstall pytest-figleaf
```

note: installed plugins are integrated automatically through the setuptools' entrypoint mechanism.

exercise: use the “figleaf” plugin

- install `pytest-figleaf`
- run `py.test --figleaf 5/test` with and without `--checkremote`
- inspect “html” directory created by figleaf
- add an untested method to the app code and re-run

quick discussion of advanced usages

If you install the `pytest-xdist` plugin you get:
looponfailing: `py.test -f` will loop over failing tests until they pass
distributed testing: distribute tests to multiple platforms
Other interesting topics:

- parametrized tests - call the same test function with multiple funcarg values
- doctests - integrate the running of python doctests
- `oejskit` plugin: run javascript unit-tests against wsgi apps

see the following tutorial and http://pytest.org documentation :)

Summary / questions and answers

- py.test thoroughly enables a no-boilerplate testing approach
- makes natural use of the assert statement
- has many debugging options
- support for marking and skipping tests
- funcarg mechanism for integration and functional testing
- easy-to-manage plugins for coverage and others purposes

hope you enjoyed it, improvement ideas welcome!

mail: holger at merlinux.eu
twitter: http://twitter.com/hpk42