

pytest advanced testing tutorial

Holger Krekel
<http://merlinux.eu>

Pycon US 2009, Chicago

March 26, 2009



If you have time, please install

- svn checkout <http://codespeak.net/svn/py/dist>
- run “python setup.py” with “install” or “develop”
- if need be: easy_install “py”
- slides: <http://tinyurl.com/c76gve>

my technical background

- programming since 20 years
- Python since around 2000
- released projects: pypy, py.test/py lib, rlcompleter2
- other: mailwitness, shpy, vadm, codespeak, ...
- merlinux GmbH since 2004
- PyPy EU-project 2004-2007

my current background



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

my testing background

- learned Python 2001
- “test-driven developer” (TDD) since 2002
- founded PyPy, based on TDD principles
- developed unittest/stdtest, now py.test
- consulted to help with 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?

Python

Python has no compile-time type security.

Testing to the rescue!

automated tests are better than declaring types.

The test tool question

what is the job of automated testing tools?

my current answer

- verify that my code changes work out
- be helpful when test scenarios fail

If failures are not helpful ...

improve the test tool or
write more (different) tests

The Automated Test question

what are automated tests there for?

my current answer

to make sure that

- units react well to input.
- components co-operate nicely
- code changes work out in the end

What does “code changes work out” mean?



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

my current answer

- various operating systems
- various Python Interpreters
- cloud environments
- web browsers etc.

Common Test terminology

- developer and customer tests
- unit tests
- functional tests
- acceptance tests
- integration tests

you may discuss a long time about categorizations ...

py.test strives to test it all

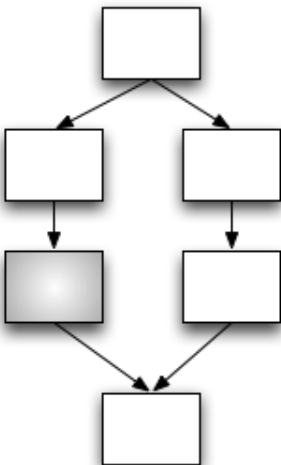
current focus:

- unittesting
- functional
- integration tests

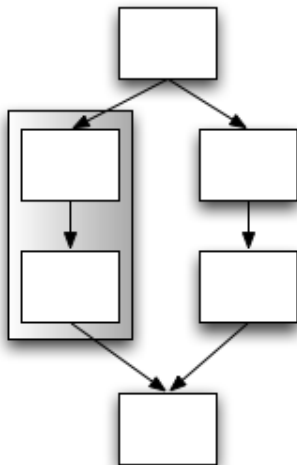
A pragmatic view on test types

let's talk about small, medium or large tests.

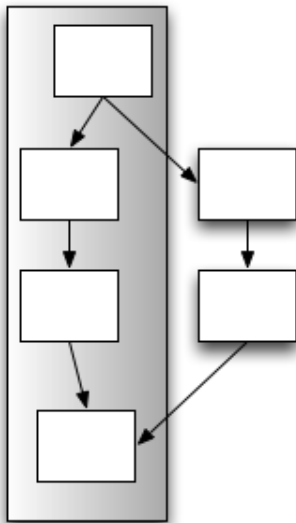
Small Tests: one aspect



Medium Tests: two aspects



Large Tests: end-to-end



generally speaking

what is the vision of automated testing?

my current answer

merge with real-life deployment.

pytest advanced features (30 minutes)



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

Python test function viewed abstractly

```
from app.pkg import SomeClass
def test_something():
    inst1 = SomeClass("somevalue")
    ...
    assert "things are ok"
```

observations

- test configuration mixes with code instantiation
- importing 'app' may fail
- the “app.pkg.SomeClass” reference may change

setting up state close to code

```
from app.pkg import SomeClass
class TestGroup:
    def setup_method(self, method):
        self.inst1 = SomeClass("somevalue")

    def test_something(self):
        ... use self.inst1 ...
        assert "things are ok"
```

observations

- test configuration mixes with code instantiation
- importing 'app' may fail
- the "app.pkg.SomeClass" reference may change
- **functions tend to group by setup methods**
- **multiple methods can reuse the same setup**

meet py.test “funcargs”

goals:

- fully separate test setup from test code
- setup app bootstraps in one place
- allow to group tests in classes or files logically

basic funcarg mechanism

- have a test function specify its needed setup
- lookup and call a function that provides setup

Example test function

```
def test_somefunction(myarg):  
    assert myarg == 'test_somefunction'
```

Example conftest.py

```
class ConftestPlugin:
    def pytest_funcarg__myarg(self, pyfuncitem):
        return pyfuncitem.name
```

observations

funcargs:

- test functions receive value by specifying a name
- makers are registered after command line parsing
- makers have meta-access to “collected function item”

notes on “named based lookup”

py.test often looks up names:

- discover test_*.py test files
- discover test_ functions or Test classes
- and now: discover test function arguments

automatic discovery avoids boilerplate

Exercise

- write a new package “mypkg”
- add mypkg/___init___ and mypkg/test_url.py
- add a test_path function that needs an “url” argument
- write the provider in mypkg/conftest.py
- run your test

Bonus: add more tests, play with wrongly named args.

Using Plugins and Extensions



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

Historic view

- 0.9.x uses `conftest`'s for extension and configuration
- 1.0 uses “plugins” for extending and `conftest.py` for configuration
- we do “smooth” transition because of existing test code base

Customizing py.test

- configuration values go to conftest.py files
- write local or global plugins
- provide funcargs

Conftest.py

- can be put into test directory or higher up
- contains test configuration values
- specifies plugins to use
- can provide default values for command line options

Specifying plugins

- `-p NAME`: load comma-separated list of plugins
- plugins are always named “`pytest_NAME`” and can be anywhere in your import path

Writing a local conftest plugin

you can write a local conftest.py based plugin:

```
class ConftestPlugin:
    def pytest_addoption(self, parser):
        parser.addoption("--
myworld", action="store_true")
    ...
```

Exercise

- add an option for specifying url on the command line
- look at “`py.test -h`”

other Plugin Examples

- integrate collection/run of traditional unit-tests
- run functions in their own tempdir
- testing ReST documents
- running Prolog tests (old: conftest based)
- running Javascript tests (old: conftest based)
- html reporting for nightly runs (old: conftest-based)

if time permits ...

- let's play with “pytest_unittest” plugin

Break



Writing Plugins (30 minutes)

- test collection objects
- plugin hooks and events
- event system for custom reporting
- test collection hooks
- test running hooks

py.test collection objects

- collectors, `collect()` returns list of “colitems”
- items, implement `runtest()` for running a test

test collection tree

- collection tree is built iteratively
- test collection starts from directories or files (via cmdline)

py.test.collect.* filesystem objects

- **Directory**
- **File**

always available Attributes

parent a reference to the collector that produced us

name: a name that is unique within the scope of our parent

config: test configuration

Python object collectors

obj points to the underlying python object.

- **Module**
- **Class/Instance**
- **Generator**
- **Function**

Exercise “collection tree”

inspecting the test collection tree:

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

Hooks and Events

- Plugin hook methods implement interaction with configuration/collection/running of tests
- Events are called for notification purposes, are not asked for return values.

Configuration Hooks

```
def pytest_addoption(self, parser):
    """ called before commandline parsing. """

def pytest_configure(self, config):
    """ called after command line options have been parsed. """

def pytest_unconfigure(self, config):
    """ called before test process is exited. """
```

Collection Hooks

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

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

def pytest_collect_recurse(self, path, parent):
    """ return True/False to cause/prevent recursion. """
```

Python collection hooks

::

```
def pytest_pymodule_makeitem(self, modcol, name, obj):  
    """ return custom item/collector for  
    a python object in a module, or  
    None. """
```

function test run hooks

::

```
def pytest_pyfunc_call(self, pyfuncitem, args, kwargs):
    """ return True if we consumed/did
    the call to the python function
    item. """

def pytest_item_makereport(self, item, excinfo, when):
    """ return ItemTestReport event for
    the given test outcome. """
```

Reporting hooks

::

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

```
def pytest_terminal_summary(self, terminalreporter):  
    """ add additional section in  
    terminal summary reporting. """
```

Event methods

are only called, no interaction.

see `py/test/plugin/pytest_terminal.py` for a full selection of events.

Warning

naming changes might take place before 1.0 final

Writing cross-project plugins

- put plugin into `pytest_name.py` or package
- make a “NamePlugin” class available
- release or copy to somewhere importable

Exercise

- port confstest plugin to global “pytest_myapp” plugin.
- put `pytest_myapp` somewhere where it's importable (or release it :)
- put “`pytest_plugins = 'pytest_myapp'`” into your test module

Distributed Testing (45 minutes)



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

the basic idea

collect tests locally, run tests in separated test 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

<http://codespeak.net/svn/py/dist/py/execnet/script/socket>

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

```
py.test -d --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=...']  
mypkg/conftest.py:: rsyncdirs = ['.']
```

Distribution modes

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

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

how does `py.test` do all this?

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

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

xspec examples

::

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

Exercise

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

Feedback round

How did you like the tutorial?

Thanks for taking part!

holger krekel at merlinux eu