pytest Documentation

Release 2.3.5

holger krekel, http://merlinux.eu

April 30, 2013

CONTENTS

Download latest version as PDF

CHAPTER

ONE

GETTING STARTED BASICS

1.1 pytest: helps you write better programs

Note: Upcoming: professional testing with pytest and tox , 24th-26th June 2013, Leipzig.

a mature full-featured Python testing tool

- runs on Posix/Windows, Python 2.4-3.3, PyPy and Jython-2.5.1
- *comprehensive online* and PDF documentation
- used in *many projects and organisations*, in test suites ranging from 10 to 10s of thousands of tests
- comes with many *tested examples*

provides easy no-boilerplate testing

- makes it *easy to get started*, many *usage options*
- *Asserting with the assert statement*
- helpful *traceback and failing assertion reporting*
- allows *print debugging* and *the capturing of standard output during test execution*

scales from simple unit to complex functional testing

- (new in 2.3) *modular parametrizeable fixtures*
- *parametrized test functions*
- *Marking test functions with attributes*
- *Skip and xfail: dealing with tests that can not succeed*
- can *distribute tests to multiple CPUs* through *xdist plugin*
- can *continuously re-run failing tests*
- many *builtin helpers* and *plugins*
- flexible *Conventions for Python test discovery*

integrates many common testing methods:

- multi-paradigm: pytest can run many `nose`, `unittest.py` and `doctest.py` style test suites, including running testcases made for Django and trial
- supports *good integration practises*

- supports extended *xUnit style setup*
- supports domain-specific *Working with non-python tests*
- supports the generation of testing coverage reports
- Javascript unit- and functional testing
- supports **PEP 8** compliant coding styles in tests

extensive plugin and customization system:

- all collection, reporting, running aspects are delegated to hook functions
- customizations can be per-directory, per-project or per PyPI released plugins
- it is easy to add command line options or do other kind of add-ons and customizations.

1.2 Installation and Getting Started

Pythons: Python 2.4-3.3, Jython, PyPy

Platforms: Unix/Posix and Windows

PyPI package name: pytest

documentation as PDF: download latest

1.2.1 Installation

Installation options:

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

To check your installation has installed the correct version:

```
$ py.test --version
This is py.test version 2.3.5, imported from /home/hpk/p/pytest/.tox/regen/local/lib/python2.7/site-p
```

If you get an error checkout *Known Installation issues*.

1.2.2 Our first test run

Let's create a first test file with a simple test function:

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

That's it. You can execute the test function now:

```
$ py.test
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items
```

```
test_sample.py F

================================= FAILURES =================================
_____________________________________ test_answer _____________________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:5: AssertionError
========================== 1 failed in 0.01 seconds ==========================
```

py.test found the `test_answer` function by following *standard test discovery rules*, basically detecting the `test_` prefixes. We got a failure report because our little `func(3)` call did not return `5`.

Note: You can simply use the `assert` statement for asserting test expectations. pytest's *Advanced assertion introspection* will intelligently report intermediate values of the assert expression freeing you from the need to learn the many names of JUnit legacy methods.

1.2.3 Asserting that a certain exception is raised

If you want to assert that some code raises an exception you can use the `raises` helper:

```
# content of test_sysexit.py
import pytest
def f():
    raise SystemExit(1)

def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

Running it with, this time in "quiet" reporting mode:

```
$ py.test -q test_sysexit.py
.
```

Todo

For further ways to assert exceptions see the *raises*

1.2.4 Grouping multiple tests in a class

Once you start to have more than a few tests it often makes sense to group tests logically, in classes and modules. Let's write a class containing two tests:

```
# content of test_class.py
class TestClass:
    def test_one(self):
        x = "this"
        assert 'h' in x
```

```
    def test_two(self):
        x = "hello"
        assert hasattr(x, 'check')
```

The two tests are found because of the standard *Conventions for Python test discovery*. There is no need to subclass anything. We can simply run the module by passing its filename:

```
$ py.test -q test_class.py
.F
================================= FAILURES =================================
_______________________________ TestClass.test_two ________________________________

self = <test_class.TestClass instance at 0x315b488>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E       assert hasattr('hello', 'check')

test_class.py:8: AssertionError
```

The first test passed, the second failed. Again we can easily see the intermediate values used in the assertion, helping us to understand the reason for the failure.

1.2.5 Going functional: requesting a unique temporary directory

For functional tests one often needs to create some files and pass them to application objects. pytest provides *Builtin fixtures/function arguments* which allow to request arbitrary resources, for example a unique temporary directory:

```
# content of test_tmpdir.py
def test_needsfiles(tmpdir):
    print tmpdir
    assert 0
```

We list the name `tmpdir` in the test function signature and py.test will lookup and call a fixture factory to create the resource before performing the test function call. Let's just run it:

```
$ py.test -q test_tmpdir.py
F
================================= FAILURES =================================
_______________________________ test_needsfiles ________________________________

tmpdir = local('/tmp/pytest-322/test_needsfiles0')

    def test_needsfiles(tmpdir):
        print tmpdir
>       assert 0
E       assert 0

test_tmpdir.py:3: AssertionError
----------------------------- Captured stdout ------------------------------
/tmp/pytest-322/test_needsfiles0
```

Before the test runs, a unique-per-test-invocation temporary directory was created. More info at *Temporary directories and files*.

You can find out what kind of builtin *pytest fixtures: explicit, modular, scalable* exist by typing:

```
py.test --fixtures   # shows builtin and custom fixtures
```

1.2.6 Where to go next

Here are a few suggestions where to go next:

- *Calling pytest through python -m pytest* for command line invocation examples
- *good practises* for virtualenv, test layout, genscript support
- *pytest fixtures: explicit, modular, scalable* for providing a functional baseline to your tests
- *py.test reference documentation* for documentation and examples on using py.test
- *Working with plugins and conftest files* managing and writing plugins

1.2.7 Known Installation issues

easy_install or pip not found?

Install pip for a state of the art python package installer.

Or consult distribute docs to install the `easy_install` tool on your machine.

You may also use the older setuptools project but it lacks bug fixes and does not work on Python3.

py.test not found on Windows despite installation?

- **Windows**: If "easy_install" or "py.test" are not found you need to add the Python script path to your `PATH`, see here: Python for Windows. You may alternatively use an ActivePython install which does this for you automatically.
- **Jython2.5.1 on Windows XP**: Jython does not create command line launchers so `py.test` will not work correctly. You may install py.test on CPython and type `py.test --genscript=mytest` and then use `jython mytest` to run py.test for your tests to run with Jython.

  Usages and Examples for more complex examples

1.3 Usage and Invocations

1.3.1 Calling pytest through `python -m pytest`

New in version 2.0. If you use Python-2.5 or later you can invoke testing through the Python interpreter from the command line:

```
python -m pytest [...]
```

This is equivalent to invoking the command line script `py.test [...]` directly.

1.3.2 Getting help on version, option names, environment variables

```
py.test --version   # shows where pytest was imported from
py.test --fixtures  # show available builtin function arguments
py.test -h | --help # show help on command line and config file options
```

1.3.3 Stopping after the first (or N) failures

To stop the testing process after the first (N) failures:

```
py.test -x            # stop after first failure
py.test --maxfail=2    # stop after two failures
```

1.3.4 Specifying tests / selecting tests

Several test run options:

```
py.test test_mod.py   # run tests in module
py.test somepath      # run all tests below path
py.test -k string     # only run tests whose names contain a string
```

Import 'pkg' and use its filesystem location to find and run tests:

```
py.test --pyargs pkg # run all tests found below directory of pypkg
```

1.3.5 Modifying Python traceback printing

Examples for modifying traceback printing:

```
py.test --showlocals # show local variables in tracebacks
py.test -l           # show local variables (shortcut)

py.test --tb=long    # the default informative traceback formatting
py.test --tb=native  # the Python standard library formatting
py.test --tb=short   # a shorter traceback format
py.test --tb=line    # only one line per failure
```

1.3.6 Dropping to PDB (Python Debugger) on failures

Python comes with a builtin Python debugger called PDB. `py.test` allows one to drop into the PDB prompt via a command line option:

```
py.test --pdb
```

This will invoke the Python debugger on every failure. Often you might only want to do this for the first failing test to understand a certain failure situation:

```
py.test -x --pdb   # drop to PDB on first failure, then end test session
py.test --pdb --maxfail=3  # drop to PDB for the first three failures
```

1.3.7 Setting a breakpoint / aka `set_trace()`

If you want to set a breakpoint and enter the `pdb.set_trace()` you can use a helper:

```
import pytest
def test_function():
    ...
    pytest.set_trace()     # invoke PDB debugger and tracing
```

In previous versions you could only enter PDB tracing if you disabled capturing on the command line via `py.test -s`.

1.3.8 Profiling test execution duration

To get a list of the slowest 10 test durations:

```
py.test --durations=10
```

1.3.9 Creating JUnitXML format files

To create result files which can be read by Hudson or other Continuous integration servers, use this invocation:

```
py.test --junitxml=path
```

to create an XML file at `path`.

1.3.10 Creating resultlog format files

To create plain-text machine-readable result files you can issue:

```
py.test --resultlog=path
```

and look at the content at the `path` location. Such files are used e.g. by the PyPy-test web page to show test results over several revisions.

1.3.11 Sending test report to online pastebin service

Creating a URL for each test failure:

```
py.test --pastebin=failed
```

This will submit test run information to a remote Paste service and provide a URL for each failure. You may select tests as usual or add for example `-x` if you only want to send one particular failure.

Creating a URL for a whole test session log:

```
py.test --pastebin=all
```

Currently only pasting to the http://bpaste.net service is implemented.

1.3.12 Calling pytest from Python code

New in version 2.0. You can invoke `py.test` from Python code directly:

```
pytest.main()
```

this acts as if you would call "py.test" from the command line. It will not raise `SystemExit` but return the exitcode instead. You can pass in options and arguments:

```
pytest.main(['-x', 'mytestdir'])
```

or pass in a string:

```
pytest.main("-x mytestdir")
```

You can specify additional plugins to `pytest.main`:

```
# content of myinvoke.py
import pytest
class MyPlugin:
    def pytest_sessionfinish(self):
        print("*** test run reporting finishing")

pytest.main("-qq", plugins=[MyPlugin()])
```

Running it will show that `MyPlugin` was added and its hook was invoked:

```
$ python myinvoke.py

*** test run reporting finishing
```

1.4 Good Integration Practises

1.4.1 Work with virtual environments

We recommend to use virtualenv environments and use easy_install (or pip) for installing your application dependencies as well as the `pytest` package itself. This way you will get a much more reproducible environment. A good tool to help you automate test runs against multiple dependency configurations or Python interpreters is tox.

1.4.2 Use tox and Continuous Integration servers

If you frequently release code to the public you may want to look into tox, the virtualenv test automation tool and its pytest support. The basic idea is to generate a JUnitXML file through the `--junitxml=PATH` option and have a continuous integration server like Jenkins pick it up and generate reports.

1.4.3 Create a py.test standalone script

If you are a maintainer or application developer and want others to easily run tests you can generate a completely standalone "py.test" script:

```
py.test --genscript=runtests.py
```

generates a `runtests.py` script which is a fully functional basic `py.test` script, running unchanged under Python2 and Python3. You can tell people to download the script and then e.g. run it like this:

```
python runtests.py
```

1.4.4 Integrating with distutils / `python setup.py test`

You can integrate test runs into your distutils or setuptools based project. Use the genscript method to generate a standalone py.test script:

```
py.test --genscript=runtests.py
```

and make this script part of your distribution and then add this to your `setup.py` file:

```
from distutils.core import setup, Command
# you can also import from setuptools

class PyTest(Command):
    user_options = []
    def initialize_options(self):
        pass
    def finalize_options(self):
        pass
    def run(self):
        import sys,subprocess
        errno = subprocess.call([sys.executable, 'runtests.py'])
        raise SystemExit(errno)
setup(
    #...,
    cmdclass = {'test': PyTest},
    #...,
)
```

If you now type:

```
python setup.py test
```

this will execute your tests using `runtests.py`. As this is a standalone version of `py.test` no prior installation whatsoever is required for calling the test command. You can also pass additional arguments to the subprocess-calls such as your test directory or other options.

1.4.5 Integration with setuptools/distribute test commands

Distribute/Setuptools support test requirements, which means its really easy to extend its test command to support running a pytest from test requirements:

```
from setuptools.command.test import test as TestCommand
import sys

class PyTest(TestCommand):
    def finalize_options(self):
        TestCommand.finalize_options(self)
        self.test_args = []
        self.test_suite = True
    def run_tests(self):
        #import here, cause outside the eggs aren't loaded
        import pytest
        errno = pytest.main(self.test_args)
        sys.exit(errno)

setup(
    #...,
    tests_require=['pytest'],
```

```
    cmdclass = {'test': PyTest},
    )
```

Now if you run:

```
python setup.py test
```

this will download py.test if needed and then run py.test as you would expect it to.

1.4.6 Conventions for Python test discovery

`py.test` implements the following standard test discovery:

- collection starts from the initial command line arguments which may be directories, filenames or test ids.
- recurse into directories, unless they match `norecursedirs`
- `test_*.py` or `*_test.py` files, imported by their package name.
- `Test` prefixed test classes (without an `__init__` method)
- `test_` prefixed test functions or methods are test items

For examples of how to customize your test discovery *Changing standard (Python) test discovery*.

Within Python modules, py.test also discovers tests using the standard *unittest.TestCase* subclassing technique.

1.4.7 Choosing a test layout / import rules

py.test supports common test layouts:

- inlining test directories into your application package, useful if you want to keep (unit) tests and actually tested code close together:

```
mypkg/
    __init__.py
    appmodule.py
    ...
    test/
        test_app.py
        ...
```

- putting tests into an extra directory outside your actual application code, useful if you have many functional tests or want to keep tests separate from actual application code:

```
mypkg/
    __init__.py
    appmodule.py
tests/
    test_app.py
    ...
```

In both cases you usually need to make sure that `mypkg` is importable, for example by using the setuptools `python setup.py develop` method.

You can run your tests by pointing to it:

```
py.test tests/test_app.py       # for external test dirs
py.test mypkg/test/test_app.py  # for inlined test dirs
py.test mypkg                   # run tests in all below test directories
py.test                         # run all tests below current dir
...
```

Note: If py.test finds a "a/b/test_module.py" test file while recursing into the filesystem it determines the import name as follows:

- find `basedir` – this is the first "upward" (towards the root) directory not containing an `__init__.py`. If both the `a` and `b` directories contain an `__init__.py` the basedir will be the parent dir of `a`.
- perform `sys.path.insert(0, basedir)` to make the test module importable under the fully qualified import name.
- `import a.b.test_module` where the path is determined by converting path separators `/` into "." characters. This means you must follow the convention of having directory and file names map directly to the import names.

The reason for this somewhat evolved importing technique is that in larger projects multiple test modules might import from each other and thus deriving a canonical import name helps to avoid surprises such as a test modules getting imported twice.

1.5 Project examples

Here are some examples of projects using py.test (please send notes via *Contact channels*):

- PyPy, Python with a JIT compiler, running over 16000 tests
- the MoinMoin Wiki Engine
- sentry, realtime app-maintenance and exception tracking
- tox, virtualenv/Hudson integration tool
- PIDA framework for integrated development
- PyPM ActiveState's package manager
- Fom a fluid object mapper for FluidDB
- applib cross-platform utilities
- six Python 2 and 3 compatibility utilities
- pediapress MediaWiki articles
- mwlib mediawiki parser and utility library
- The Translate Toolkit for localization and conversion
- execnet rapid multi-Python deployment
- pylib cross-platform path, IO, dynamic code library
- Pacha configuration management in five minutes
- bbfreeze create standalone executables from Python scripts
- pdb++ a fancier version of PDB
- py-s3fuse Amazon S3 FUSE based filesystem

- waskr WSGI Stats Middleware
- guachi global persistent configs for Python modules
- Circuits lightweight Event Driven Framework
- pygtk-helpers easy interaction with PyGTK
- QuantumCore statusmessage and repoze openid plugin
- pydataportability libraries for managing the open web
- XIST extensible HTML/XML generator
- tiddlyweb optionally headless, extensible RESTful datastore
- fancycompleter for colorful tab-completion
- Paludis tools for Gentoo Paludis package manager
- Gerald schema comparison tool
- abjad Python API for Formalized Score control
- bu a microscopic build system
- katcp Telescope communication protocol over Twisted
- kss plugin timer
- pyudev a pure Python binding to the Linux library libudev
- pytest-localserver a plugin for pytest that provides a httpserver and smtpserver
- pytest-monkeyplus a plugin that extends monkeypatch

These projects help integrate py.test into other Python frameworks:

- pytest-django for Django
- zope.pytest for Zope and Grok
- pytest_gae for Google App Engine
- There is some work underway for Kotti, a CMS built in Pyramid/Pylons

1.5.1 Some organisations using py.test

- Square Kilometre Array, Cape Town
- Some Mozilla QA people use pytest to distribute their Selenium tests
- Tandberg
- Shootq
- Stups department of Heinrich Heine University Duesseldorf
- cellzome
- Open End, Gothenborg
- Laboraratory of Bioinformatics, Warsaw
- merlinux, Germany
- many more ... (please be so kind to send a note via *Contact channels*)

1.6 Some Issues and Questions

Note: If you don't find an answer here, you may checkout pytest Q&A at Stackoverflow or other *Contact channels* to get help.

1.6.1 On naming, nosetests, licensing and magic

How does py.test relate to nose and unittest?

py.test and nose share basic philosophy when it comes to running and writing Python tests. In fact, you can run many tests written for nose with py.test. nose was originally created as a clone of `py.test` when py.test was in the `0.8` release cycle. Note that starting with pytest-2.0 support for running unittest test suites is majorly improved.

how does py.test relate to twisted's trial?

Since some time py.test has builtin support for supporting tests written using trial. It does not itself start a reactor, however, and does not handle Deferreds returned from a test in pytest style. If you are using trial's unittest.TestCase chances are that you can just run your tests even if you return Deferreds. In addition, there also is a dedicated pytest-twisted plugin which allows to return deferreds from pytest-style tests, allowing to use *pytest fixtures: explicit, modular, scalable* and other features.

how does py.test work with Django?

In 2012, some work is going into the pytest-django plugin. It substitutes the usage of Django's `manage.py test` and allows to use all pytest features most of which are not available from Django directly.

What's this "magic" with py.test? (historic notes)

Around 2007 (version `0.8`) some people thought that py.test was using too much "magic". It had been part of the pylib which contains a lot of unreleated python library code. Around 2010 there was a major cleanup refactoring, which removed unused or deprecated code and resulted in the new `pytest` PyPI package which strictly contains only test-related code. This relese also brought a complete pluginification such that the core is around 300 lines of code and everything else is implemented in plugins. Thus `pytest` today is a small, universally runnable and customizable testing framework for Python. Note, however, that `pytest` uses metaprogramming techniques and reading its source is thus likely not something for Python beginners.

A second "magic" issue was the assert statement debugging feature. Nowadays, py.test explicitely rewrites assert statements in test modules in order to provide more useful *assert feedback*. This completely avoids previous issues of confusing assertion-reporting. It also means, that you can use Python's `-O` optimization without loosing assertions in test modules.

py.test contains a second mostly obsolete assert debugging technique, invoked via `--assert=reinterpret`, activated by default on Python-2.5: When an `assert` statement fails, py.test re-interprets the expression part to show intermediate values. This technique suffers from a caveat that the rewriting does not: If your expression has side effects (better to avoid them anyway!) the intermediate values may not be the same, confusing the reinterpreter and obfuscating the initial error (this is also explained at the command line if it happens).

You can also turn off all assertion interaction using the `--assertmode=off` option.

Why a `py.test` instead of a `pytest` command?

Some of the reasons are historic, others are practical. `py.test` used to be part of the `py` package which provided several developer utilities, all starting with `py.<TAB>`, thus providing nice TAB-completion. If you install `pip install pycmd` you get these tools from a separate package. These days the command line tool could be called `pytest` but since many people have gotten used to the old name and there is another tool named "pytest" we just decided to stick with `py.test` for now.

1.6.2 Function arguments, parametrized tests and setup

Is using funcarg- versus xUnit setup a style question?

For simple applications and for people experienced with nose or unittest-style test setup using xUnit style setup probably feels natural. For larger test suites, parametrized testing or setup of complex test resources using funcargs may feel more natural. Moreover, funcargs are ideal for writing advanced test support code (like e.g. the monkeypatch, the tmpdir or capture funcargs) because the support code can register setup/teardown functions in a managed class/module/function scope.

Why the `pytest_funcarg__*` name for funcarg factories?

We like Convention over Configuration and didn't see much point in allowing a more flexible or abstract mechanism. Moreover, it is nice to be able to search for `pytest_funcarg__MYARG` in source code and safely find all factory functions for the `MYARG` function argument.

Note: With pytest-2.3 you can use the *Fixtures as Function arguments (funcargs)* decorator to mark a function as a fixture function.

Can I yield multiple values from a fixture function function?

There are two conceptual reasons why yielding from a factory function is not possible:

- Calling factories for obtaining test function arguments is part of setting up and running a test. At that point it is not possible to add new test calls to the test collection anymore.
- If multiple factories yielded values there would be no natural place to determine the combination policy - in real-world examples some combinations often should not run.

However, with pytest-2.3 you can use the *Fixtures as Function arguments (funcargs)* decorator and specify `params` so that all tests depending on the factory-created resource will run multiple times with different parameters.

You can also use the pytest_generate_tests hook to implement the parametrization scheme of your choice.

1.6.3 py.test interaction with other packages

Issues with py.test, multiprocess and setuptools?

On windows the multiprocess package will instantiate sub processes by pickling and thus implicitly re-import a lot of local modules. Unfortunately, setuptools-0.6.11 does not `if __name__=='__main__'` protect its generated command line script. This leads to infinite recursion when running a test that instantiates Processes.

A good solution is to install Distribute as a drop-in replacement for setuptools and then re-install `pytest`. Otherwise you could fix the script that is created by setuptools by inserting an `if __name__ == '__main__'`. Or you can create a "pytest.py" script with this content and invoke that with the python version:

```
import pytest
if __name__ == '__main__':
    pytest.main()
```

CHAPTER

TWO

PY.TEST REFERENCE DOCUMENTATION

2.1 Pytest API and builtin fixtures

This is a list of `pytest.*` API functions and fixtures.

For information on plugin hooks and objects, see *Working with plugins and conftest files*.

For information on the `pytest.mark` mechanism, see *Marking test functions with attributes*.

For the below objects, you can also interactively ask for help, e.g. by typing on the Python interactive prompt something like:

```
import pytest
help(pytest)
```

2.1.1 Invoking pytest interactively

main(*args=None*, *plugins=None*)
return exit code, after performing an in-process test run.

Parameters

- **args** – list of command line arguments.
- **plugins** – list of plugin objects to be auto-registered during initialization.

More examples at *Calling pytest from Python code*

2.1.2 Helpers for assertions about Exceptions/Warnings

raises(*ExpectedException*, **args*, ***kwargs*)
assert that a code block/function call raises @ExpectedException and raise a failure exception otherwise.

If using Python 2.5 or above, you may use this function as a context manager:

```
>>> with raises(ZeroDivisionError):
...     1/0
```

Or you can specify a callable by passing a to-be-called lambda:

```
>>> raises(ZeroDivisionError, lambda: 1/0)
<ExceptionInfo ...>
```

or you can specify an arbitrary callable with arguments:

```
>>> def f(x): return 1/x
...
>>> raises(ZeroDivisionError, f, 0)
<ExceptionInfo ...>
>>> raises(ZeroDivisionError, f, x=0)
<ExceptionInfo ...>
```

A third possibility is to use a string to be executed:

```
>>> raises(ZeroDivisionError, "f(0)")
<ExceptionInfo ...>
```

Examples at *Assertions about expected exceptions*.

deprecated_call (*func*, **args*, ***kwargs*)
assert that calling `func(*args, **kwargs)` triggers a DeprecationWarning.

2.1.3 Raising a specific test outcome

You can use the following functions in your test, fixture or setup functions to force a certain test outcome. Note that most often you can rather use declarative marks, see *Skip and xfail: dealing with tests that can not succeed*.

fail (*msg=''*, *pytrace=True*)
explicitely fail an currently-executing test with the given Message.

Parameters **pytrace** – if false the msg represents the full failure information and no python traceback will be reported.

skip (*msg=''*)
skip an executing test with the given message. Note: it's usually better to use the py.test.mark.skipif marker to declare a test to be skipped under certain conditions like mismatching platforms or dependencies. See the pytest_skipping plugin for details.

importorskip (*modname*, *minversion=None*)
return imported module if it has a higher __version__ than the optionally specified 'minversion' - otherwise call py.test.skip() with a message detailing the mismatch.

xfail (*reason=''*)
xfail an executing test or setup functions with the given reason.

exit (*msg*)
exit testing process as if KeyboardInterrupt was triggered.

2.1.4 fixtures and requests

To mark a fixture function:

fixture (*scope='function'*, *params=None*, *autouse=False*)
(return a) decorator to mark a fixture factory function.

This decorator can be used (with or or without parameters) to define a fixture function. The name of the fixture function can later be referenced to cause its invocation ahead of running tests: test modules or classes can use the

pytest.mark.usefixtures(fixturename) marker. Test functions can directly use fixture names as input arguments in which case the fixture instance returned from the fixture function will be injected.

Parameters

- **scope** – the scope for which this fixture is shared, one of "function" (default), "class", "module", "session".
- **params** – an optional list of parameters which will cause multiple invocations of the fixture function and all of the tests using it.
- **autouse** – if True, the fixture func is activated for all tests that can see it. If False (the default) then an explicit reference is needed to activate the fixture.

Tutorial at *pytest fixtures: explicit, modular, scalable*.

The `request` object that can be used from fixture functions.

class `FixtureRequest`

A request for a fixture from a test or fixture function.

A request object gives access to the requesting test context and has an optional `param` attribute in case the fixture is parametrized indirectly.

`fixturename` = None
fixture for which this request is being performed

`scope` = None
Scope string, one of "function", "cls", "module", "session"

`node`
underlying collection node (depends on current request scope)

`config`
the pytest config object associated with this request.

`function`
test function object if the request has a per-function scope.

`cls`
class (can be None) where the test function was collected.

`instance`
instance (can be None) on which test function was collected.

`module`
python module object where the test function was collected.

`fspath`
the file system path of the test module which collected this test.

`keywords`
keywords/markers dictionary for the underlying node.

`session`
pytest session object.

`addfinalizer`(*finalizer*)
add finalizer/teardown function to be called after the last test within the requesting test context finished execution.

`applymarker`(*marker*)
Apply a marker to a single test function invocation. This method is useful if you don't want to have a keyword/marker on all function invocations.

Parameters **marker** – a _pytest.mark.MarkDecorator object created by a call to py.test.mark.NAME(...).

raiseerror(*msg*)
raise a FixtureLookupError with the given message.

cached_setup(*setup*, *teardown=None*, *scope='module'*, *extrakey=None*)
(deprecated) Return a testing resource managed by setup & teardown calls. scope and extrakey determine when the teardown function will be called so that subsequent calls to setup would recreate the resource. With pytest-2.3 you often do not need cached_setup() as you can directly declare a scope on a fixture function and register a finalizer through request.addfinalizer().

Parameters

- **teardown** – function receiving a previously setup resource.
- **setup** – a no-argument function creating a resource.
- **scope** – a string value out of function, class, module or session indicating the caching lifecycle of the resource.
- **extrakey** – added to internal caching key of (funcargname, scope).

getfuncargvalue(*argname*)
Dynamically retrieve a named fixture function argument.

As of pytest-2.3, it is easier and usually better to access other fixture values by stating it as an input argument in the fixture function. If you only can decide about using another fixture at test setup time, you may use this function to retrieve it inside a fixture function body.

2.1.5 Builtin fixtures/function arguments

You can ask for available builtin or project-custom *fixtures* by typing:

```
$ py.test -q --fixtures
capsys
    enables capturing of writes to sys.stdout/sys.stderr and makes
    captured output available via ``capsys.readouterr()`` method calls
    which return a ``(out, err)`` tuple.

capfd
    enables capturing of writes to file descriptors 1 and 2 and makes
    captured output available via ``capsys.readouterr()`` method calls
    which return a ``(out, err)`` tuple.

monkeypatch
    The returned ``monkeypatch`` funcarg provides these
    helper methods to modify objects, dictionaries or os.environ::

    monkeypatch.setattr(obj, name, value, raising=True)
    monkeypatch.delattr(obj, name, raising=True)
    monkeypatch.setitem(mapping, name, value)
    monkeypatch.delitem(obj, name, raising=True)
    monkeypatch.setenv(name, value, prepend=False)
    monkeypatch.delenv(name, value, raising=True)
    monkeypatch.syspath_prepend(path)
    monkeypatch.chdir(path)

    All modifications will be undone after the requesting
    test function has finished. The ``raising``
```

```
    parameter determines if a KeyError or AttributeError
    will be raised if the set/deletion operation has no target.

pytestconfig
    the pytest config object with access to command line opts.
recwarn
    Return a WarningsRecorder instance that provides these methods:

    * ``pop(category=None)``: return last warning matching the category.
    * ``clear()``: clear list of warnings

    See http://docs.python.org/library/warnings.html for information
    on warning categories.

tmpdir
    return a temporary directory path object
    which is unique to each test function invocation,
    created as a sub directory of the base temporary
    directory.  The returned object is a `py.path.local`_
    path object.
```

2.2 Basic test configuration

2.2.1 Command line options and configuration file settings

You can get help on command line options and values in INI-style configurations files by using the general help option:

```
py.test -h   # prints options _and_ config file settings
```

This will display command line and configuration file settings which were registered by installed plugins.

2.2.2 How test configuration is read from configuration INI-files

py.test searches for the first matching ini-style configuration file in the directories of command line argument and the directories above. It looks for file basenames in this order:

```
pytest.ini
tox.ini
setup.cfg
```

Searching stops when the first `[pytest]` section is found in any of these files. There is no merging of configuration values from multiple files. Example:

```
py.test path/to/testdir
```

will look in the following dirs for a config file:

```
path/to/testdir/pytest.ini
path/to/testdir/tox.ini
path/to/testdir/setup.cfg
path/to/pytest.ini
path/to/tox.ini
path/to/setup.cfg
... # up until root of filesystem
```

If argument is provided to a py.test run, the current working directory is used to start the search.

2.2.3 How to change command line options defaults

It can be tedious to type the same series of command line options every time you use py.test . For example, if you always want to see detailed info on skipped and xfailed tests, as well as have terser "dot" progress output, you can write it into a configuration file:

```
# content of pytest.ini
# (or tox.ini or setup.cfg)
[pytest]
addopts = -rsxX -q
```

From now on, running `py.test` will add the specified options.

2.2.4 Builtin configuration file options

minversion
Specifies a minimal pytest version required for running tests.

minversion = 2.1 # will fail if we run with pytest-2.0

addopts
Add the specified `OPTS` to the set of command line arguments as if they had been specified by the user. Example: if you have this ini file content:

```
[pytest]
addopts = --maxfail=2 -rf  # exit after 2 failures, report fail info
```

issuing `py.test test_hello.py` actually means:

```
py.test --maxfail=2 -rf test_hello.py
```

Default is to add no options.

norecursedirs
Set the directory basename patterns to avoid when recursing for test discovery. The individual (fnmatch-style) patterns are applied to the basename of a directory to decide if to recurse into it. Pattern matching characters:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

Default patterns are `.* _* CVS {args}`. Setting a `norecursedir` replaces the default. Here is an example of how to avoid certain directories:

```
# content of setup.cfg
[pytest]
norecursedirs = .svn _build tmp*
```

This would tell py.test to not look into typical subversion or sphinx-build directories or into any `tmp` prefixed directory.

python_files
One or more Glob-style file patterns determining which python files are considered as test modules.

python_classes
One or more name prefixes determining which test classes are considered as test modules.

python_functions
One or more name prefixes determining which test functions and methods are considered as test modules.

See *Changing naming conventions* for examples.

2.3 The writing and reporting of assertions in tests

2.3.1 Asserting with the `assert` statement

`py.test` allows you to use the standard python `assert` for verifying expectations and values in Python tests. For example, you can write the following:

```
# content of test_assert1.py
def f():
    return 3

def test_function():
    assert f() == 4
```

to assert that your function returns a certain value. If this assertion fails you will see the return value of the function call:

```
$ py.test test_assert1.py
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

test_assert1.py F

================================= FAILURES =================================
_____________________________ test_function ______________________________

    def test_function():
>       assert f() == 4
E       assert 3 == 4
E        +  where 3 = f()

test_assert1.py:5: AssertionError
========================= 1 failed in 0.01 seconds =========================
```

py.test has support for showing the values of the most common subexpressions including calls, attributes, comparisons, and binary and unary operators. (See *Demo of Python failure reports with py.test*). This allows you to use the idiomatic python constructs without boilerplate code while not losing introspection information.

However, if you specify a message with the assertion like this:

```
assert a % 2 == 0, "value was odd, should be even"
```

then no assertion introspection takes places at all and the message will be simply shown in the traceback.

See *Advanced assertion introspection* for more information on assertion introspection.

2.3.2 Assertions about expected exceptions

In order to write assertions about raised exceptions, you can use `pytest.raises` as a context manager like this:

```
import pytest
with pytest.raises(ZeroDivisionError):
    1 / 0
```

and if you need to have access to the actual exception info you may use:

```
with pytest.raises(RuntimeError) as excinfo:
    def f():
        f()
    f()

# do checks related to excinfo.type, excinfo.value, excinfo.traceback
```

If you want to write test code that works on Python 2.4 as well, you may also use two other ways to test for an expected exception:

```
pytest.raises(ExpectedException, func, *args, **kwargs)
pytest.raises(ExpectedException, "func(*args, **kwargs)")
```

both of which execute the specified function with args and kwargs and asserts that the given `ExpectedException` is raised. The reporter will provide you with helpful output in case of failures such as *no exception* or *wrong exception*.

2.3.3 Making use of context-sensitive comparisons

New in version 2.0. py.test has rich support for providing context-sensitive information when it encounters comparisons. For example:

```
# content of test_assert2.py

def test_set_comparison():
    set1 = set("1308")
    set2 = set("8035")
    assert set1 == set2
```

if you run this module:

```
$ py.test test_assert2.py
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

test_assert2.py F

================================= FAILURES =================================
_____________________________ test_set_comparison ______________________________

    def test_set_comparison():
        set1 = set("1308")
        set2 = set("8035")
>       assert set1 == set2
E       assert set(['0', '1', '3', '8']) == set(['0', '3', '5', '8'])
E         Extra items in the left set:
E         '1'
E         Extra items in the right set:
E         '5'

test_assert2.py:5: AssertionError
========================= 1 failed in 0.01 seconds =========================
```

Special comparisons are done for a number of cases:

- comparing long strings: a context diff is shown
- comparing long sequences: first failing indices
- comparing dicts: different entries

See the *reporting demo* for many more examples.

2.3.4 Defining your own assertion comparison

It is possible to add your own detailed explanations by implementing the `pytest_assertrepr_compare` hook.

pytest_assertrepr_compare(*config*, *op*, *left*, *right*)
return explanation for comparisons in failing assert expressions.

Return None for no custom explanation, otherwise return a list of strings. The strings will be joined by newlines but any newlines *in* a string will be escaped. Note that all but the first line will be indented sligthly, the intention is for the first line to be a summary.

As an example consider adding the following hook in a conftest.py which provides an alternative explanation for `Foo` objects:

```
# content of conftest.py
from test_foocompare import Foo
def pytest_assertrepr_compare(op, left, right):
    if isinstance(left, Foo) and isinstance(right, Foo) and op == "==":
     return ['Comparing Foo instances:',
               '   vals: %s != %s' % (left.val, right.val)]
```

now, given this test module:

```
# content of test_foocompare.py
class Foo:
    def __init__(self, val):
         self.val = val

def test_compare():
    f1 = Foo(1)
    f2 = Foo(2)
    assert f1 == f2
```

you can run the test module and get the custom output defined in the conftest file:

```
$ py.test -q test_foocompare.py
F
================================ FAILURES ================================
_____________________________ test_compare _____________________________

    def test_compare():
        f1 = Foo(1)
        f2 = Foo(2)
>       assert f1 == f2
E       assert Comparing Foo instances:
E            vals: 1 != 2

test_foocompare.py:8: AssertionError
```

2.3.5 Advanced assertion introspection

New in version 2.1. Reporting details about a failing assertion is achieved either by rewriting assert statements before they are run or re-evaluating the assert expression and recording the intermediate values. Which technique is used depends on the location of the assert, py.test's configuration, and Python version being used to run py.test. Note that for assert statements with a manually provided message, i.e. `assert expr, message`, no assertion introspection takes place and the manually provided message will be rendered in tracebacks.

By default, if the Python version is greater than or equal to 2.6, py.test rewrites assert statements in test modules. Rewritten assert statements put introspection information into the assertion failure message. py.test only rewrites test modules directly discovered by its test collection process, so asserts in supporting modules which are not themselves test modules will not be rewritten.

Note: py.test rewrites test modules on import. It does this by using an import hook to write a new pyc files. Most of the time this works transparently. However, if you are messing with import yourself, the import hook may interfere. If this is the case, simply use `--assert=reinterp` or `--assert=plain`. Additionally, rewriting will fail silently if it cannot write new pycs, i.e. in a read-only filesystem or a zipfile.

If an assert statement has not been rewritten or the Python version is less than 2.6, py.test falls back on assert reinterpretation. In assert reinterpretation, py.test walks the frame of the function containing the assert statement to discover sub-expression results of the failing assert statement. You can force py.test to always use assertion reinterpretation by passing the `--assert=reinterp` option.

Assert reinterpretation has a caveat not present with assert rewriting: If evaluating the assert expression has side effects you may get a warning that the intermediate values could not be determined safely. A common example of this issue is an assertion which reads from a file:

```
assert f.read() != '...'
```

If this assertion fails then the re-evaluation will probably succeed! This is because `f.read()` will return an empty string when it is called the second time during the re-evaluation. However, it is easy to rewrite the assertion and avoid any trouble:

```
content = f.read()
assert content != '...'
```

All assert introspection can be turned off by passing `--assert=plain`.

For further information, Benjamin Peterson wrote up Behind the scenes of py.test's new assertion rewriting. New in version 2.1: Add assert rewriting as an alternate introspection technique.Changed in version 2.1: Introduce the `--assert` option. Deprecate `--no-assert` and `--nomagic`.

2.4 pytest fixtures: explicit, modular, scalable

New in version 2.0/2.3. The general purpose of test fixtures is to provide a fixed baseline upon which tests can reliably and repeatedly execute. pytest-2.3 fixtures offer dramatic improvements over the classic xUnit style of setup/teardown functions:

- fixtures have explicit names and are activated by declaring their use from test functions, modules, classes or whole projects.
- fixtures are implemented in a modular manner, as each fixture name triggers a *fixture function* which can itself easily use other fixtures.

- fixture management scales from simple unit to complex functional testing, allowing to parametrize fixtures and tests according to configuration and component options, or to re-use fixtures across class, module or whole test session scopes.

In addition, pytest continues to support *classic xunit-style setup*. You can mix both styles, moving incrementally from classic to new style, as you prefer. You can also start out from existing *unittest.TestCase style* or *nose based* projects.

2.4.1 Fixtures as Function arguments (funcargs)

Test functions can receive fixture objects by naming them as an input argument. For each argument name, a fixture function with that name provides the fixture object. Fixture functions are registered by marking them with `@pytest.fixture`. Let's look at a simple self-contained test module containing a fixture and a test function using it:

```
# content of ./test_smtpsimple.py
import pytest

@pytest.fixture
def smtp():
    import smtplib
    return smtplib.SMTP("merlinux.eu")

def test_ehlo(smtp):
    response, msg = smtp.ehlo()
    assert response == 250
    assert "merlinux" in msg
    assert 0 # for demo purposes
```

Here, the `test_ehlo` needs the `smtp` fixture value. pytest will discover and call the `@pytest.fixture` marked `smtp` fixture function. Running the test looks like this:

```
$ py.test test_smtpsimple.py
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

test_smtpsimple.py F

================================= FAILURES =================================
_____________________________________ test_ehlo _____________________________________

smtp = <smtplib.SMTP instance at 0x226cc20>

    def test_ehlo(smtp):
        response, msg = smtp.ehlo()
        assert response == 250
        assert "merlinux" in msg
>       assert 0 # for demo purposes
E       assert 0

test_smtpsimple.py:12: AssertionError
========================= 1 failed in 0.20 seconds =========================
```

In the failure traceback we see that the test function was called with a `smtp` argument, the `smtplib.SMTP()` instance created by the fixture function. The test function fails on our deliberate `assert 0`. Here is an exact protocol of how py.test comes to call the test function this way:

1. pytest *finds* the `test_ehlo` because of the `test_` prefix. The test function needs a function argument named `smtp`. A matching fixture function is discovered by looking for a fixture-marked function named `smtp`.
2. `smtp()` is called to create an instance.
3. `test_ehlo(<SMTP instance>)` is called and fails in the last line of the test function.

Note that if you misspell a function argument or want to use one that isn't available, you'll see an error with a list of available function arguments.

Note: You can always issue:

```
py.test --fixtures test_simplefactory.py
```

to see available fixtures.

In versions prior to 2.3 there was no `@pytest.fixture` marker and you had to use a magic `pytest_funcarg__NAME` prefix for the fixture factory. This remains and will remain supported but is not anymore advertised as the primary means of declaring fixture functions.

2.4.2 Funcargs a prime example of dependency injection

When injecting fixtures to test functions, pytest-2.0 introduced the term "funcargs" or "funcarg mechanism" which continues to be present also in pytest-2.3 docs. It now refers to the specific case of injecting fixture values as arguments to test functions. With pytest-2.3 there are more possibilities to use fixtures but "funcargs" probably will remain as the main way of dealing with fixtures.

As the following examples show in more detail, funcargs allow test functions to easily receive and work against specific pre-initialized application objects without having to care about import/setup/cleanup details. It's a prime example of dependency injection where fixture functions take the role of the *injector* and test functions are the *consumers* of fixture objects.

2.4.3 Working with a module-shared fixture

Fixtures requiring network access depend on connectivity and are usually time-expensive to create. Extending the previous example, we can add a `scope='module'` parameter to the `@pytest.fixture` invocation to cause the decorated `smtp` fixture function to only be invoked once per test module. Multiple test functions in a test module will thus each receive the same `smtp` fixture instance. The next example also extracts the fixture function into a separate `conftest.py` file so that all tests in test modules in the directory can access the fixture function:

```
# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp():
    return smtplib.SMTP("merlinux.eu")
```

The name of the fixture again is `smtp` and you can access its result by listing the name `smtp` as an input parameter in any test or fixture function (in or below the directory where `conftest.py` is located):

```
# content of test_module.py

def test_ehlo(smtp):
    response = smtp.ehlo()
    assert response[0] == 250
```

```
    assert "merlinux" in response[1]
    assert 0  # for demo purposes

def test_noop(smtp):
    response = smtp.noop()
    assert response[0] == 250
    assert 0  # for demo purposes
```

We deliberately insert failing `assert 0` statements in order to inspect what is going on and can now run the tests:

```
$ py.test test_module.py
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_module.py FF

================================ FAILURES ================================
_____________________________________ test_ehlo _____________________________________

smtp = <smtplib.SMTP instance at 0x18a6368>

    def test_ehlo(smtp):
        response = smtp.ehlo()
        assert response[0] == 250
        assert "merlinux" in response[1]
>       assert 0  # for demo purposes
E       assert 0

test_module.py:6: AssertionError
_____________________________________ test_noop _____________________________________

smtp = <smtplib.SMTP instance at 0x18a6368>

    def test_noop(smtp):
        response = smtp.noop()
        assert response[0] == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:11: AssertionError
======================== 2 failed in 0.26 seconds ========================
```

You see the two `assert 0` failing and more importantly you can also see that the same (module-scoped) `smtp` object was passed into the two test functions because pytest shows the incoming argument values in the traceback. As a result, the two test functions using `smtp` run as quick as a single one because they reuse the same instance.

If you decide that you rather want to have a session-scoped `smtp` instance, you can simply declare it:

```
@pytest.fixture(scope=``session``)
def smtp(...):
    # the returned fixture value will be shared for
    # all tests needing it
```

2.4.4 Fixtures can interact with the requesting test context

Fixture functions can themselves use other fixtures by naming them as an input argument just like test functions do, see *Modularity: using fixtures from a fixture function*. Moreover, pytest provides a builtin `request` object, which

fixture functions can use to introspect the function, class or module for which they are invoked or to register finalizing (cleanup) functions which are called when the last test finished execution.

Further extending the previous `smtp` fixture example, let's read an optional server URL from the module namespace and register a finalizer that closes the smtp connection after the last test in a module finished execution:

```
# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp(request):
    server = getattr(request.module, "smtpserver", "merlinux.eu")
    smtp = smtplib.SMTP(server)
    def fin():
        print ("finalizing %s" % smtp)
        smtp.close()
    request.addfinalizer(fin)
    return smtp
```

The registered `fin` function will be called when the last test using it has executed:

```
$ py.test -s -q --tb=no
FF
finalizing <smtplib.SMTP instance at 0x1e10248>
```

We see that the `smtp` instance is finalized after the two tests using it tests executed. If we had specified `scope='function'` then fixture setup and cleanup would occur around each single test. Note that either case the test module itself does not need to change!

Let's quickly create another test module that actually sets the server URL and has a test to verify the fixture picks it up:

```
# content of test_anothersmtp.py

smtpserver = "mail.python.org"  # will be read by smtp fixture

def test_showhelo(smtp):
    assert 0, smtp.helo()
```

Running it:

```
$ py.test -qq --tb=short test_anothersmtp.py
F
================================= FAILURES =================================
_____________________________ test_showhelo ______________________________
test_anothersmtp.py:5: in test_showhelo
>       assert 0, smtp.helo()
E       AssertionError: (250, 'mail.python.org')
```

2.4.5 Parametrizing a fixture

Fixture functions can be parametrized in which case they will be called multiple times, each time executing the set of dependent tests, i. e. the tests that depend on this fixture. Test functions do usually not need to be aware of their re-running. Fixture parametrization helps to write exhaustive functional tests for components which themselves can be configured in multiple ways.

Extending the previous example, we can flag the fixture to create two `smtp` fixture instances which will cause all tests using the fixture to run twice. The fixture function gets access to each parameter through the special `request` object:

```
# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module",
                params=["merlinux.eu", "mail.python.org"])
def smtp(request):
    smtp = smtplib.SMTP(request.param)
    def fin():
        print ("finalizing %s" % smtp)
        smtp.close()
    request.addfinalizer(fin)
    return smtp
```

The main change is the declaration of `params` with `@pytest.fixture`, a list of values for each of which the fixture function will execute and can access a value via `request.param`. No test function code needs to change. So let's just do another run:

```
$ py.test -q test_module.py
FFFF
================================ FAILURES ================================
_____________________________ test_ehlo[merlinux.eu] _____________________________

smtp = <smtplib.SMTP instance at 0x1b38a28>

    def test_ehlo(smtp):
        response = smtp.ehlo()
        assert response[0] == 250
        assert "merlinux" in response[1]
>       assert 0  # for demo purposes
E       assert 0

test_module.py:6: AssertionError
_____________________________ test_noop[merlinux.eu] _____________________________

smtp = <smtplib.SMTP instance at 0x1b38a28>

    def test_noop(smtp):
        response = smtp.noop()
        assert response[0] == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:11: AssertionError
_____________________________ test_ehlo[mail.python.org] _____________________________

smtp = <smtplib.SMTP instance at 0x1b496c8>

    def test_ehlo(smtp):
        response = smtp.ehlo()
        assert response[0] == 250
>       assert "merlinux" in response[1]
E       assert 'merlinux' in 'mail.python.org\nSIZE 25600000\nETRN\nSTARTTLS\nENHANCEDSTATUSCODES\n8I

test_module.py:5: AssertionError
_____________________________ test_noop[mail.python.org] _____________________________

smtp = <smtplib.SMTP instance at 0x1b496c8>
```

```
    def test_noop(smtp):
        response = smtp.noop()
        assert response[0] == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:11: AssertionError
```

We see that our two test functions each ran twice, against the different `smtp` instances. Note also, that with the `mail.python.org` connection the second test fails in `test_ehlo` because a different server string is expected than what arrived.

2.4.6 Modularity: using fixtures from a fixture function

You can not only use fixtures in test functions but fixture functions can use other fixtures themselves. This contributes to a modular design of your fixtures and allows re-use of framework-specific fixtures across many projects. As a simple example, we can extend the previous example and instantiate an object `app` where we stick the already defined `smtp` resource into it:

```
# content of test_appsetup.py

import pytest

class App:
    def __init__(self, smtp):
        self.smtp = smtp

@pytest.fixture(scope="module")
def app(smtp):
    return App(smtp)

def test_smtp_exists(app):
    assert app.smtp
```

Here we declare an `app` fixture which receives the previously defined `smtp` fixture and instantiates an `App` object with it. Let's run it:

```
$ py.test -v test_appsetup.py
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 2 items

test_appsetup.py:12: test_smtp_exists[merlinux.eu] PASSED
test_appsetup.py:12: test_smtp_exists[mail.python.org] PASSED

======================== 2 passed in 5.38 seconds ========================
```

Due to the parametrization of `smtp` the test will run twice with two different `App` instances and respective smtp servers. There is no need for the `app` fixture to be aware of the `smtp` parametrization as pytest will fully analyse the fixture dependency graph.

Note, that the `app` fixture has a scope of `module` and uses a module-scoped `smtp` fixture. The example would still work if `smtp` was cached on a `session` scope: it is fine for fixtures to use "broader" scoped fixtures but not the other way round: A session-scoped fixture could not use a module-scoped one in a meaningful way.

2.4.7 Automatic grouping of tests by fixture instances

pytest minimizes the number of active fixtures during test runs. If you have a parametrized fixture, then all the tests using it will first execute with one instance and then finalizers are called before the next fixture instance is created. Among other things, this eases testing of applications which create and use global state.

The following example uses two parametrized funcargs, one of which is scoped on a per-module basis, and all the functions perform `print` calls to show the setup/teardown flow:

```
# content of test_module.py
import pytest

@pytest.fixture(scope="module", params=["mod1", "mod2"])
def modarg(request):
    param = request.param
    print "create", param
    def fin():
        print "fin", param
    request.addfinalizer(fin)
    return param

@pytest.fixture(scope="function", params=[1,2])
def otherarg(request):
    return request.param

def test_0(otherarg):
    print "  test0", otherarg
def test_1(modarg):
    print "  test1", modarg
def test_2(otherarg, modarg):
    print "  test2", otherarg, modarg
```

Let's run the tests in verbose mode and with looking at the print-output:

```
$ py.test -v -s test_module.py
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 8 items

test_module.py:16: test_0[1] PASSED
test_module.py:16: test_0[2] PASSED
test_module.py:18: test_1[mod1] PASSED
test_module.py:20: test_2[1-mod1] PASSED
test_module.py:20: test_2[2-mod1] PASSED
test_module.py:18: test_1[mod2] PASSED
test_module.py:20: test_2[1-mod2] PASSED
test_module.py:20: test_2[2-mod2] PASSED

========================= 8 passed in 0.01 seconds =========================
  test0 1
  test0 2
create mod1
  test1 mod1
  test2 1 mod1
  test2 2 mod1
fin mod1
create mod2
  test1 mod2
  test2 1 mod2
```

```
  test2 2 mod2
fin mod2
```

You can see that the parametrized module-scoped `modarg` resource caused an ordering of test execution that lead to the fewest possible "active" resources. The finalizer for the `mod1` parametrized resource was executed before the `mod2` resource was setup.

2.4.8 using fixtures from classes, modules or projects

Sometimes test functions do not directly need access to a fixture object. For example, tests may require to operate with an empty directory as the current working directory but otherwise do not care for the concrete directory. Here is how you can can use the standard tempfile and pytest fixtures to achieve it. We separate the creation of the fixture into a conftest.py file:

```
# content of conftest.py

import pytest
import tempfile
import os

@pytest.fixture()
def cleandir():
    newpath = tempfile.mkdtemp()
    os.chdir(newpath)
```

and declare its use in a test module via a `usefixtures` marker:

```
# content of test_setenv.py
import os
import pytest

@pytest.mark.usefixtures("cleandir")
class TestDirectoryInit:
    def test_cwd_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
        with open("myfile", "w") as f:
            f.write("hello")

    def test_cwd_again_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
```

Due to the `usefixtures` marker, the `cleandir` fixture will be required for the execution of each test method, just as if you specified a "cleandir" function argument to each of them. Let's run it to verify our fixture is activated and the tests pass:

```
$ py.test -q
..
```

You can specify multiple fixtures like this:

```
@pytest.mark.usefixtures("cleandir", "anotherfixture")
```

and you may specify fixture usage at the test module level, using a generic feature of the mark mechanism:

```
pytestmark = pytest.mark.usefixtures("cleandir")
```

Lastly you can put fixtures required by all tests in your project into an ini-file:

```
# content of pytest.ini

[pytest]
usefixtures = cleandir
```

2.4.9 autouse fixtures (xUnit setup on steroids)

Occasionally, you may want to have fixtures get invoked automatically without a usefixtures or funcargs reference. As a practical example, suppose we have a database fixture which has a begin/rollback/commit architecture and we want to automatically surround each test method by a transaction and a rollback. Here is a dummy self-contained implementation of this idea:

```
# content of test_db_transact.py

import pytest

class DB:
    def __init__(self):
        self.intransaction = []
    def begin(self, name):
        self.intransaction.append(name)
    def rollback(self):
        self.intransaction.pop()

@pytest.fixture(scope="module")
def db():
    return DB()

class TestClass:
    @pytest.fixture(autouse=True)
    def transact(self, request, db):
        db.begin(request.function.__name__)
        request.addfinalizer(db.rollback)

    def test_method1(self, db):
        assert db.intransaction == ["test_method1"]

    def test_method2(self, db):
        assert db.intransaction == ["test_method2"]
```

The class-level `transact` fixture is marked with *autouse=true* which implies that all test methods in the class will use this fixture without a need to state it in the test function signature or with a class-level `usefixtures` decorator.

If we run it, we get two passing tests:

```
$ py.test -q
..
```

Here is how autouse fixtures work in other scopes:

- if an autouse fixture is defined in a test module, all its test functions automatically use it.
- if an autouse fixture is defined in a conftest.py file then all tests in all test modules belows its directory will invoke the fixture.
- lastly, and **please use that with care**: if you define an autouse fixture in a plugin, it will be invoked for all tests in all projects where the plugin is installed. This can be useful if a fixture only anyway works in the presence

of certain settings e. g. in the ini-file. Such a global fixture should always quickly determine if it should do any work and avoid expensive imports or computation otherwise.

Note that the above `transact` fixture may very well be a fixture that you want to make available in your project without having it generally active. The canonical way to do that is to put the transact definition into a conftest.py file **without** using `autouse`:

```
# content of conftest.py
@pytest.fixture()
def transact(self, request, db):
    db.begin()
    request.addfinalizer(db.rollback)
```

and then e.g. have a TestClass using it by declaring the need:

```
@pytest.mark.usefixtures("transact")
class TestClass:
    def test_method1(self):
        ...
```

All test methods in this TestClass will use the transaction fixture while other test classes or functions in the module will not use it unless they also add a `transact` reference.

2.4.10 Shifting (visibility of) fixture functions

If during implementing your tests you realize that you want to use a fixture function from multiple test files you can move it to a *conftest.py* file or even separately installable *plugins* without changing test code. The discovery of fixtures functions starts at test classes, then test modules, then `conftest.py` files and finally builtin and third party plugins.

2.5 Parametrizing fixtures and test functions

pytest supports test parametrization in several well-integrated ways:

- `pytest.fixture()` allows to define *parametrization at the level of fixture functions*.
- @pytest.mark.parametrize allows to define parametrization at the function or class level, provides multiple argument/fixture sets for a particular test function or class.
- pytest_generate_tests enables implementing your own custom dynamic parametrization scheme or extensions.

2.5.1 `@pytest.mark.parametrize`: parametrizing test functions

New in version 2.2. The builtin `pytest.mark.parametrize` decorator enables parametrization of arguments for a test function. Here is a typical example of a test function that implements checking that a certain input leads to an expected output:

```
# content of test_expectation.py
import pytest
@pytest.mark.parametrize(("input", "expected"), [
    ("3+5", 8),
    ("2+4", 6),
    ("6*9", 42),
])
```

```
def test_eval(input, expected):
    assert eval(input) == expected
```

Here, the `@parametrize` decorator defines three different argument sets for the two `(input, output)` arguments of the `test_eval` function which will thus run three times:

```
$ py.test
========================== test session starts ===========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 3 items

test_expectation.py ..F

================================ FAILURES ================================
_____________________________ test_eval[6*9-42] _____________________________

input = '6*9', expected = 42

    @pytest.mark.parametrize(("input", "expected"), [
        ("3+5", 8),
        ("2+4", 6),
        ("6*9", 42),
    ])
    def test_eval(input, expected):
>       assert eval(input) == expected
E       assert 54 == 42
E        +  where 54 = eval('6*9')

test_expectation.py:8: AssertionError
==================== 1 failed, 2 passed in 0.01 seconds ====================
```

As expected only one pair of input/output values fails the simple test function. And as usual with test function arguments, you can see the `input` and `output` values in the traceback.

Note that there ways how you can mark a class or a module, see *Marking test functions with attributes*.

2.5.2 Basic `pytest_generate_tests` example

Sometimes you may want to implement your own parametrization scheme or implement some dynamism for determining the parameters or scope of a fixture. For this, you can use the `pytest_generate_tests` hook which is called when collecting a test function. Through the passed in *metafunc* object you can inspect the requesting test context and, most importantly, you can call `metafunc.parametrize()` to cause parametrization.

For example, let's say we want to run a test taking string inputs which we want to set via a new py.test command line option. Let's first write a simple test accepting a `stringinput` fixture function argument:

```
# content of test_strings.py

def test_valid_string(stringinput):
    assert stringinput.isalpha()
```

Now we add a `conftest.py` file containing the addition of a command line option and the parametrization of our test function:

```
# content of conftest.py

def pytest_addoption(parser):
    parser.addoption("--stringinput", action="append", default=[],
```

```
        help="list of stringinputs to pass to test functions")

def pytest_generate_tests(metafunc):
    if 'stringinput' in metafunc.fixturenames:
        metafunc.parametrize("stringinput",
                             metafunc.config.option.stringinput)
```

If we now pass two stringinput values, our test will run twice:

```
$ py.test -q --stringinput="hello" --stringinput="world" test_strings.py
..
```

Let's also run with a stringinput that will lead to a failing test:

```
$ py.test -q --stringinput="!" test_strings.py
F
================================ FAILURES ================================
_____________________________ test_valid_string[!] _____________________________

stringinput = '!'

    def test_valid_string(stringinput):
>       assert stringinput.isalpha()
E       assert <built-in method isalpha of str object at 0x2ba729dab300>()
E        +  where <built-in method isalpha of str object at 0x2ba729dab300> = '!'.isalpha

test_strings.py:3: AssertionError
```

As expected our test function fails.

If you don't specify a stringinput it will be skipped because `metafunc.parametrize()` will be called with an empty parameter listlist:

```
$ py.test -q -rs test_strings.py
s
======================== short test summary info =========================
SKIP [1] /home/hpk/p/pytest/.tox/regen/local/lib/python2.7/site-packages/_pytest/python.py:974: got e
```

For further examples, you might want to look at *more parametrization examples*.

2.5.3 The metafunc object

metafunc objects are passed to the `pytest_generate_tests` hook. They help to inspect a testfunction and to generate tests according to test configuration or values specified in the class or module where a test function is defined:

`metafunc.fixturenames`: set of required function arguments for given function

`metafunc.function`: underlying python test function

`metafunc.cls`: class object where the test function is defined in or None.

`metafunc.module`: the module object where the test function is defined in.

`metafunc.config`: access to command line opts and general config

`metafunc.funcargnames`: alias for `fixturenames`, for pre-2.3 compatibility

`Metafunc.`**`parametrize`**(*argnames*, *argvalues*, *indirect=False*, *ids=None*, *scope=None*)

Add new invocations to the underlying test function using the list of argvalues for the given argnames. Parametrization is performed during the collection phase. If you need to setup expensive resources see about setting indirect=True to do it rather at test setup time.

Parameters

- **argnames** – an argument name or a list of argument names
- **argvalues** – The list of argvalues determines how often a test is invoked with different argument values. If only one argname was specified argvalues is a list of simple values. If N argnames were specified, argvalues must be a list of N-tuples, where each tuple-element specifies a value for its respective argname.
- **indirect** – if True each argvalue corresponding to an argname will be passed as request.param to its respective argname fixture function so that it can perform more expensive setups during the setup phase of a test rather than at collection time.
- **ids** – list of string ids each corresponding to the argvalues so that they are part of the test id. If no ids are provided they will be generated automatically from the argvalues.
- **scope** – if specified it denotes the scope of the parameters. The scope is used for grouping tests by parameter instances. It will also override any fixture-function defined scope, allowing to set a dynamic scope using test context or configuration.

`Metafunc.addcall` (*funcargs=None*, *id=_notexists*, *param=_notexists*)

(deprecated, use parametrize) Add a new call to the underlying test function during the collection phase of a test run. Note that request.addcall() is called during the test collection phase prior and independently to actual test execution. You should only use addcall() if you need to specify multiple arguments of a test function.

Parameters

- **funcargs** – argument keyword dictionary used when invoking the test function.
- **id** – used for reporting and identification purposes. If you don't supply an *id* an automatic unique id will be generated.
- **param** – a parameter which will be exposed to a later fixture function invocation through the `request.param` attribute.

2.6 classic xunit-style setup

This section describes a classic and popular way how you can implement fixtures (setup and teardown test state) on a per-module/class/function basis. pytest started supporting these methods around 2005 and subsequently nose and the standard library introduced them (under slightly different names). While these setup/teardown methods are and will remain fully supported you may also use pytest's more powerful *fixture mechanism* which leverages the concept of dependency injection, allowing for a more modular and more scalable approach for managing test state, especially for larger projects and for functional testing. It is safe to mix both fixture mechanisms.

2.6.1 Module level setup/teardown

If you have multiple test functions and test classes in a single module you can optionally implement the following fixture methods which will usually be called once for all the functions:

```
def setup_module(module):
    """ setup any state specific to the execution of the given module."""

def teardown_module(module):
    """ teardown any state that was previously setup with a setup_module
    method.
    """
```

2.6.2 Class level setup/teardown

Similarly, the following methods are called at class level before and after all test methods of the class are called:

```
@classmethod
def setup_class(cls):
    """ setup any state specific to the execution of the given class (which
    usually contains tests).
    """

@classmethod
def teardown_class(cls):
    """ teardown any state that was previously setup with a call to
    setup_class.
    """
```

2.6.3 Method and function level setup/teardown

Similarly, the following methods are called around each method invocation:

```
def setup_method(self, method):
    """ setup any state tied to the execution of the given method in a
    class.  setup_method is invoked for every test method of a class.
    """

def teardown_method(self, method):
    """ teardown any state that was previously setup with a setup_method
    call.
    """
```

If you would rather define test functions directly at module level you can also use the following functions to implement fixtures:

```
def setup_function(function):
    """ setup any state tied to the execution of the given function.
    Invoked for every test function in the module.
    """

def teardown_function(function):
    """ teardown any state that was previously setup with a setup_function
    call.
    """
```

Note that it is possible for setup/teardown pairs to be invoked multiple times per testing process.

2.7 Capturing of the stdout/stderr output

2.7.1 Default stdout/stderr/stdin capturing behaviour

During test execution any output sent to `stdout` and `stderr` is captured. If a test or a setup method fails its according captured output will usually be shown along with the failure traceback.

In addition, `stdin` is set to a "null" object which will fail on attempts to read from it because it is rarely desired to wait for interactive input when running automated tests.

By default capturing is done by intercepting writes to low level file descriptors. This allows to capture output from simple print statements as well as output from a subprocess started by a test.

2.7.2 Setting capturing methods or disabling capturing

There are two ways in which `py.test` can perform capturing:

- file descriptor (FD) level capturing (default): All writes going to the operating system file descriptors 1 and 2 will be captured.
- `sys` level capturing: Only writes to Python files `sys.stdout` and `sys.stderr` will be captured. No capturing of writes to filedescriptors is performed.

You can influence output capturing mechanisms from the command line:

```
py.test -s            # disable all capturing
py.test --capture=sys # replace sys.stdout/stderr with in-mem files
py.test --capture=fd  # also point filedescriptors 1 and 2 to temp file
```

2.7.3 Using print statements for debugging

One primary benefit of the default capturing of stdout/stderr output is that you can use print statements for debugging:

```
# content of test_module.py

def setup_function(function):
    print ("setting up %s" % function)

def test_func1():
    assert True

def test_func2():
    assert False
```

and running this module will show you precisely the output of the failing function and hide the other one:

```
$ py.test
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_module.py .F

================================ FAILURES ================================
_____________________________________ test_func2 _____________________________________

    def test_func2():
>       assert False
E       assert False

test_module.py:9: AssertionError
---------------------------- Captured stdout -----------------------------
setting up <function test_func2 at 0x2d79f50>
==================== 1 failed, 1 passed in 0.01 seconds ====================
```

2.7.4 Accessing captured output from a test function

The *Fixtures as Function arguments (funcargs)* allows test function a very easy way to access the captured output by simply using the names `capsys` or `capfd` in the test function signature. Here is an example test function that performs some output related checks:

```
def test_myoutput(capsys): # or use "capfd" for fd-level
    print ("hello")
    sys.stderr.write("world\n")
    out, err = capsys.readouterr()
    assert out == "hello\n"
    assert err == "world\n"
    print "next"
    out, err = capsys.readouterr()
    assert out == "next\n"
```

The `readouterr()` call snapshots the output so far - and capturing will be continued. After the test function finishes the original streams will be restored. Using `capsys` this way frees your test from having to care about setting/resetting output streams and also interacts well with py.test's own per-test capturing.

If you want to capture on `fd` level you can use the `capfd` function argument which offers the exact same interface.

2.8 Monkeypatching/mocking modules and environments

Sometimes tests need to invoke functionality which depends on global settings or which invokes code which cannot be easily tested such as network access. The `monkeypatch` function argument helps you to safely set/delete an attribute, dictionary item or environment variable or to modify `sys.path` for importing. See the monkeypatch blog post for some introduction material and a discussion of its motivation.

2.8.1 Simple example: monkeypatching functions

If you want to pretend that `os.expanduser` returns a certain directory, you can use the `monkeypatch.setattr()` method to patch this function before calling into a function which uses it:

```
# content of test_module.py
import os.path
def getssh(): # pseudo application code
    return os.path.join(os.path.expanduser("~admin"), '.ssh')

def test_mytest(monkeypatch):
    def mockreturn(path):
        return '/abc'
    monkeypatch.setattr(os.path, 'expanduser', mockreturn)
    x = getssh()
    assert x == '/abc/.ssh'
```

Here our test function monkeypatches `os.path.expanduser` and then calls into an function that calls it. After the test function finishes the `os.path.expanduser` modification will be undone.

2.8.2 Method reference of the monkeypatch function argument

class monkeypatch
object keeping a record of setattr/item/env/syspath changes.

setattr(*obj*, *name*, *value*, *raising=True*)
set attribute `name` on `obj` to `value`, by default raise AttributeEror if the attribute did not exist.

delattr(*obj*, *name*, *raising=True*)
delete attribute `name` from `obj`, by default raise AttributeError it the attribute did not previously exist.

setitem(*dic*, *name*, *value*)
set dictionary entry `name` to value.

delitem(*dic*, *name*, *raising=True*)
delete `name` from dict, raise KeyError if it doesn't exist.

setenv(*name*, *value*, *prepend=None*)
set environment variable `name` to `value`. if `prepend` is a character, read the current environment variable value and prepend the `value` adjoined with the `prepend` character.

delenv(*name*, *raising=True*)
delete `name` from environment, raise KeyError it not exists.

syspath_prepend(*path*)
prepend `path` to `sys.path` list of import locations.

chdir(*path*)
change the current working directory to the specified path path can be a string or a py.path.local object

undo()
undo previous changes. This call consumes the undo stack. Calling it a second time has no effect unless you do more monkeypatching after the undo call.

`monkeypatch.setattr/delattr/delitem/delenv()` all by default raise an Exception if the target does not exist. Pass `raising=False` if you want to skip this check.

2.9 xdist: pytest distributed testing plugin

The pytest-xdist plugin extends py.test with some unique test execution modes:

- Looponfail: run your tests repeatedly in a subprocess. After each run, py.test waits until a file in your project changes and then re-runs the previously failing tests. This is repeated until all tests pass. At this point a full run is again performed.
- multiprocess Load-balancing: if you have multiple CPUs or hosts you can use them for a combined test run. This allows to speed up development or to use special resources of remote machines.
- Multi-Platform coverage: you can specify different Python interpreters or different platforms and run tests in parallel on all of them.

Before running tests remotely, `py.test` efficiently "rsyncs" your program source code to the remote place. All test results are reported back and displayed to your local terminal. You may specify different Python versions and interpreters.

2.9.1 Installation of xdist plugin

Install the plugin with:

```
easy_install pytest-xdist

# or

pip install pytest-xdist
```

or use the package in develop/in-place mode with a checkout of the pytest-xdist repository

```
python setup.py develop
```

2.9.2 Usage examples

Speed up test runs by sending tests to multiple CPUs

To send tests to multiple CPUs, type:

```
py.test -n NUM
```

Especially for longer running tests or tests requiring a lot of I/O this can lead to considerable speed ups.

Running tests in a Python subprocess

To instantiate a Python-2.4 subprocess and send tests to it, you may type:

```
py.test -d --tx popen//python=python2.4
```

This will start a subprocess which is run with the "python2.4" Python interpreter, found in your system binary lookup path.

If you prefix the –tx option value like this:

```
py.test -d --tx 3*popen//python=python2.4
```

then three subprocesses would be created and the tests will be distributed to three subprocesses and run simultanously.

Running tests in looponfailing mode

For refactoring a project with a medium or large test suite you can use the looponfailing mode. Simply add the `--f` option:

```
py.test -f
```

and py.test will run your tests. Assuming you have failures it will then wait for file changes and re-run the failing test set. File changes are detected by looking at `looponfailingroots` root directories and all of their contents (recursively). If the default for this value does not work for you you can change it in your project by setting a configuration option:

```
# content of a pytest.ini, setup.cfg or tox.ini file
[pytest]
looponfailroots = mypkg testdir
```

This would lead to only looking for file changes in the respective directories, specified relatively to the ini-file's directory.

Sending tests to remote SSH accounts

Suppose you have a package `mypkg` which contains some tests that you can successfully run locally. And you also have a ssh-reachable machine `myhost`. Then you can ad-hoc distribute your tests by typing:

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

This will synchronize your `mypkg` package directory with a remote ssh account and then collect and run your tests at the remote side.

You can specify multiple `--rsyncdir` directories to be sent to the remote side.

Sending tests to remote Socket Servers

Download the single-module socketserver.py Python program and run it like this:

```
python socketserver.py
```

It will tell you that it starts listening on the default port. You can now on your home machine specify this new socket host with something like this:

```
py.test -d --tx socket=192.168.1.102:8888 --rsyncdir mypkg mypkg
```

Running tests on many platforms at once

The basic command to run tests on multiple platforms is:

```
py.test --dist=each --tx=spec1 --tx=spec2
```

If you specify a windows host, an OSX host and a Linux environment this command will send each tests to all platforms - and report back failures from all platforms at once. The specifications strings use the xspec syntax.

Specifying test exec environments in an ini file

pytest (since version 2.0) supports ini-style configuration. For example, you could make running with three subprocesses your default:

```
[pytest]
addopts = -n3
```

You can also add default environments like this:

```
[pytest]
addopts = --tx ssh=myhost//python=python2.5 --tx ssh=myhost//python=python2.6
```

and then just type:

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

to run tests in each of the environments.

Specifying "rsync" dirs in an ini-file

In a `tox.ini` or `setup.cfg` file in your root project directory you may specify directories to include or to exclude in synchronisation:

```
[pytest]
rsyncdirs = . mypkg helperpkg
rsyncignore = .hg
```

These directory specifications are relative to the directory where the configuration file was found.

2.10 Temporary directories and files

2.10.1 The 'tmpdir' test function argument

You can use the `tmpdir` function argument which will provide a temporary directory unique to the test invocation, created in the base temporary directory.

`tmpdir` is a py.path.local object which offers `os.path` methods and more. Here is an example test usage:

```
# content of test_tmpdir.py
import os
def test_create_file(tmpdir):
    p = tmpdir.mkdir("sub").join("hello.txt")
    p.write("content")
    assert p.read() == "content"
    assert len(tmpdir.listdir()) == 1
    assert 0
```

Running this would result in a passed test except for the last `assert 0` line which we use to look at values:

```
$ py.test test_tmpdir.py
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

test_tmpdir.py F

================================ FAILURES ================================
_____________________________ test_create_file _____________________________

tmpdir = local('/tmp/pytest-323/test_create_file0')

    def test_create_file(tmpdir):
        p = tmpdir.mkdir("sub").join("hello.txt")
        p.write("content")
        assert p.read() == "content"
        assert len(tmpdir.listdir()) == 1
>       assert 0
E       assert 0

test_tmpdir.py:7: AssertionError
======================== 1 failed in 0.02 seconds ========================
```

2.10.2 The default base temporary directory

Temporary directories are by default created as sub-directories of the system temporary directory. The base name will be `pytest-NUM` where `NUM` will be incremented with each test run. Moreover, entries older than 3 temporary directories will be removed.

You can override the default temporary directory setting like this:

```
py.test --basetemp=mydir
```

When distributing tests on the local machine, `py.test` takes care to configure a basetemp directory for the sub processes such that all temporary data lands below a single per-test run basetemp directory.

2.11 Marking test functions with attributes

By using the `pytest.mark` helper you can easily set metadata on your test functions. There are some builtin markers, for example:

- *skipif* - skip a test function if a certain condition is met
- *xfail* - produce an "expected failure" outcome if a certain condition is met
- *parametrize* to perform multiple calls to the same test function.

It's easy to create custom markers or to apply markers to whole test classes or modules. See *Working with custom markers* for examples which also serve as documentation.

2.11.1 API reference for mark related objects

class MarkGenerator
Factory for `MarkDecorator` objects - exposed as a `py.test.mark` singleton instance. Example:

```
import py
@py.test.mark.slowtest
def test_function():
   pass
```

will set a 'slowtest' `MarkInfo` object on the `test_function` object.

class MarkDecorator(*name*, *args=None*, *kwargs=None*)
A decorator for test functions and test classes. When applied it will create `MarkInfo` objects which may be *retrieved by hooks as item keywords*. MarkDecorator instances are often created like this:

```
mark1 = py.test.mark.NAME               # simple MarkDecorator
mark2 = py.test.mark.NAME(name1=value)  # parametrized MarkDecorator
```

and can then be applied as decorators to test functions:

```
@mark2
def test_function():
    pass
```

class MarkInfo(*name*, *args*, *kwargs*)
Marking object created by `MarkDecorator` instances.

name = None
name of attribute

args = None
positional argument list, empty if none specified

kwargs = None
keyword argument dictionary, empty if nothing specified

add(*args*, *kwargs*)
add a MarkInfo with the given args and kwargs.

2.12 Skip and xfail: dealing with tests that can not succeed

If you have test functions that cannot be run on certain platforms or that you expect to fail you can mark them accordingly or you may call helper functions during execution of setup or test functions.

A *skip* means that you expect your test to pass unless a certain configuration or condition (e.g. wrong Python interpreter, missing dependency) prevents it to run. And *xfail* means that your test can run but you expect it to fail because there is an implementation problem.

py.test counts and lists *skip* and *xfail* tests separately. However, detailed information about skipped/xfailed tests is not shown by default to avoid cluttering the output. You can use the `-r` option to see details corresponding to the "short" letters shown in the test progress:

```
py.test -rxs  # show extra info on skips and xfails
```

(See *How to change command line options defaults*)

2.12.1 Marking a test function to be skipped

Here is an example of marking a test function to be skipped when run on a Python3 interpreter:

```
import sys
@pytest.mark.skipif("sys.version_info >= (3,0)")
def test_function():
    ...
```

During test function setup the skipif condition is evaluated by calling `eval('sys.version_info >= (3,0)', namespace)`. (*New in version 2.0.2*) The namespace contains all the module globals of the test function so that you can for example check for versions of a module you are using:

```
import mymodule

@pytest.mark.skipif("mymodule.__version__ < '1.2'")
def test_function():
    ...
```

The test function will not be run ("skipped") if `mymodule` is below the specified version. The reason for specifying the condition as a string is mainly that py.test can report a summary of skip conditions. For information on the construction of the `namespace` see evaluation of skipif/xfail conditions.

You can of course create a shortcut for your conditional skip decorator at module level like this:

```
win32only = pytest.mark.skipif("sys.platform != 'win32'")

@win32only
def test_function():
    ...
```

2.12.2 Skip all test functions of a class

As with all function *marking* you can skip test functions at the whole class- or module level. Here is an example for skipping all methods of a test class based on the platform:

```
class TestPosixCalls:
    pytestmark = pytest.mark.skipif("sys.platform == 'win32'")

    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

The `pytestmark` special name tells py.test to apply it to each test function in the class. If your code targets python2.6 or above you can more naturally use the skipif decorator (and any other marker) on classes:

```
@pytest.mark.skipif("sys.platform == 'win32'")
class TestPosixCalls:

    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

Using multiple "skipif" decorators on a single function is generally fine - it means that if any of the conditions apply the function execution will be skipped.

2.12.3 Mark a test function as expected to fail

You can use the `xfail` marker to indicate that you expect the test to fail:

```
@pytest.mark.xfail
def test_function():
    ...
```

This test will be run but no traceback will be reported when it fails. Instead terminal reporting will list it in the "expected to fail" or "unexpectedly passing" sections.

By specifying on the commandline:

```
pytest --runxfail
```

you can force the running and reporting of an `xfail` marked test as if it weren't marked at all.

As with skipif you can also mark your expectation of a failure on a particular platform:

```
@pytest.mark.xfail("sys.version_info >= (3,0)")
def test_function():
    ...
```

You can furthermore prevent the running of an "xfail" test or specify a reason such as a bug ID or similar. Here is a simple test file with the several usages:

```
import pytest
xfail = pytest.mark.xfail

@xfail
def test_hello():
    assert 0

@xfail(run=False)
def test_hello2():
    assert 0

@xfail("hasattr(os, 'sep')")
def test_hello3():
    assert 0

@xfail(reason="bug 110")
def test_hello4():
    assert 0

@xfail('pytest.__version__[0] != "17"')
def test_hello5():
    assert 0
```

```
def test_hello6():
    pytest.xfail("reason")
```

Running it with the report-on-xfail option gives this output:

```
example $ py.test -rx xfail_demo.py
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 6 items

xfail_demo.py xxxxxx
======================== short test summary info ========================
XFAIL xfail_demo.py::test_hello
XFAIL xfail_demo.py::test_hello2
  reason: [NOTRUN]
XFAIL xfail_demo.py::test_hello3
  condition: hasattr(os, 'sep')
XFAIL xfail_demo.py::test_hello4
  bug 110
XFAIL xfail_demo.py::test_hello5
  condition: pytest.__version__[0] != "17"
XFAIL xfail_demo.py::test_hello6
  reason: reason

======================== 6 xfailed in 0.05 seconds ========================
```

2.12.4 Evaluation of skipif/xfail expressions

New in version 2.0.2. The evaluation of a condition string in `pytest.mark.skipif(conditionstring)` or `pytest.mark.xfail(conditionstring)` takes place in a namespace dictionary which is constructed as follows:

- the namespace is initialized by putting the `sys` and `os` modules and the pytest `config` object into it.
- updated with the module globals of the test function for which the expression is applied.

The pytest `config` object allows you to skip based on a test configuration value which you might have added:

```
@pytest.mark.skipif("not config.getvalue('db')")
def test_function(...):
    ...
```

2.12.5 Imperative xfail from within a test or setup function

If you cannot declare xfail-conditions at import time you can also imperatively produce an XFail-outcome from within test or setup code. Example:

```
def test_function():
    if not valid_config():
        pytest.xfail("unsupported configuration")
```

2.12.6 Skipping on a missing import dependency

You can use the following import helper at module level or within a test or test setup function:

```
docutils = pytest.importorskip("docutils")
```

If `docutils` cannot be imported here, this will lead to a skip outcome of the test. You can also skip based on the version number of a library:

```
docutils = pytest.importorskip("docutils", minversion="0.3")
```

The version will be read from the specified module's `__version__` attribute.

2.12.7 Imperative skip from within a test or setup function

If for some reason you cannot declare skip-conditions you can also imperatively produce a skip-outcome from within test or setup code. Example:

```
def test_function():
    if not valid_config():
        pytest.skip("unsupported configuration")
```

2.13 Asserting deprecation and other warnings

2.13.1 The recwarn function argument

You can use the `recwarn` funcarg to assert that code triggers warnings through the Python warnings system. Here is a simple self-contained test:

```
# content of test_recwarn.py
def test_hello(recwarn):
    from warnings import warn
    warn("hello", DeprecationWarning)
    w = recwarn.pop(DeprecationWarning)
    assert issubclass(w.category, DeprecationWarning)
    assert 'hello' in str(w.message)
    assert w.filename
    assert w.lineno
```

The `recwarn` function argument provides these methods:

- `pop(category=None)`: return last warning matching the category.
- `clear()`: clear list of warnings

2.13.2 Ensuring a function triggers a deprecation warning

You can also call a global helper for checking that a certain function call triggers a Deprecation warning:

```
import pytest

def test_global():
    pytest.deprecated_call(myfunction, 17)
```

2.14 Support for unittest.TestCase / Integration of fixtures

py.test has support for running Python unittest.py style tests. It's meant for leveraging existing unittest-style projects to use pytest features. Concretely, pytest will automatically collect `unittest.TestCase` subclasses and their `test` methods in test files. It will invoke typical setup/teardown methods and generally try to make test suites written to run on unittest, to also run using `py.test`. We assume here that you are familiar with writing `unittest.TestCase` style tests and rather focus on integration aspects.

2.14.1 Usage

After *Installation* type:

```
py.test
```

and you should be able to run your unittest-style tests if they are contained in `test_*` modules. If that works for you then you can make use of most *pytest features*, for example `--pdb` debugging in failures, using *plain assert-statements*, *more informative tracebacks*, stdout-capturing or distributing tests to multiple CPUs via the `-nNUM` option if you installed the `pytest-xdist` plugin. Please refer to the general pytest documentation for many more examples.

2.14.2 Mixing pytest fixtures into unittest.TestCase style tests

Running your unittest with `py.test` allows you to use its *fixture mechanism* with `unittest.TestCase` style tests. Assuming you have at least skimmed the pytest fixture features, let's jump-start into an example that integrates a pytest `db_class` fixture, setting up a class-cached database object, and then reference it from a unittest-style test:

```
# content of conftest.py

# we define a fixture function below and it will be "used" by
# referencing its name from tests

import pytest

@pytest.fixture(scope="class")
def db_class(request):
    class DummyDB:
        pass
    # set a class attribute on the invoking test context
    request.cls.db = DummyDB()
```

This defines a fixture function `db_class` which - if used - is called once for each test class and which sets the class-level `db` attribute to a `DummyDB` instance. The fixture function achieves this by receiving a special `request` object which gives access to *the requesting test context* such as the `cls` attribute, denoting the class from which the fixture is used. This architecture de-couples fixture writing from actual test code and allows re-use of the fixture by a minimal reference, the fixture name. So let's write an actual `unittest.TestCase` class using our fixture definition:

```
# content of test_unittest_db.py

import unittest
import pytest

@pytest.mark.usefixtures("db_class")
class MyTest(unittest.TestCase):
    def test_method1(self):
        assert hasattr(self, "db")
```

```
        assert 0, self.db   # fail for demo purposes

    def test_method2(self):
        assert 0, self.db   # fail for demo purposes
```

The `@pytest.mark.usefixtures("db_class")` class-decorator makes sure that the pytest fixture function `db_class` is called once per class. Due to the deliberately failing assert statements, we can take a look at the `self.db` values in the traceback:

```
$ py.test test_unittest_db.py
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_unittest_db.py FF

================================= FAILURES =================================
_______________________________ MyTest.test_method1 ________________________________

self = <test_unittest_db.MyTest testMethod=test_method1>

    def test_method1(self):
        assert hasattr(self, "db")
>       assert 0, self.db   # fail for demo purposes
E       AssertionError: <conftest.DummyDB instance at 0x19fdf38>

test_unittest_db.py:9: AssertionError
_______________________________ MyTest.test_method2 ________________________________

self = <test_unittest_db.MyTest testMethod=test_method2>

    def test_method2(self):
>       assert 0, self.db   # fail for demo purposes
E       AssertionError: <conftest.DummyDB instance at 0x19fdf38>

test_unittest_db.py:12: AssertionError
========================== 2 failed in 0.02 seconds ==========================
```

This default pytest traceback shows that the two test methods share the same `self.db` instance which was our intention when writing the class-scoped fixture function above.

2.14.3 autouse fixtures and accessing other fixtures

Although it's usually better to explicitely declare use of fixtures you need for a given test, you may sometimes want to have fixtures that are automatically used in a given context. After all, the traditional style of unittest-setup mandates the use of this implicit fixture writing and chances are, you are used to it or like it.

You can flag fixture functions with `@pytest.fixture(autouse=True)` and define the fixture function in the context where you want it used. Let's look at an `initdir` fixture which makes all test methods of a `TestCase` class execute in a temporary directory with a pre-initialized `samplefile.ini`. Our `initdir` fixture itself uses the pytest builtin *tmpdir* fixture to delegate the creation of a per-test temporary directory:

```
# content of test_unittest_cleandir.py
import pytest
import unittest

class MyTest(unittest.TestCase):
```

```
@pytest.fixture(autouse=True)
def initdir(self, tmpdir):
    tmpdir.chdir() # change to pytest-provided temporary directory
    tmpdir.join("samplefile.ini").write("# testdata")

def test_method(self):
    s = open("samplefile.ini").read()
    assert "testdata" in s
```

Due to the `autouse` flag the `initdir` fixture function will be used for all methods of the class where it is defined. This is a shortcut for using a `@pytest.mark.usefixtures("initdir")` marker on the class like in the previous example.

Running this test module ...:

```
$ py.test -q test_unittest_cleandir.py
.
```

... gives us one passed test because the `initdir` fixture function was executed ahead of the `test_method`.

Note: While pytest supports receiving fixtures via *test function arguments* for non-unittest test methods, `unittest.TestCase` methods cannot directly receive fixture function arguments as implementing that is likely to inflict on the ability to run general unittest.TestCase test suites. Maybe optional support would be possible, though. If unittest finally grows a plugin system that should help as well. In the meanwhile, the above `usefixtures` and `autouse` examples should help to mix in pytest fixtures into unittest suites. And of course you can also start to selectively leave away the `unittest.TestCase` subclassing, use plain asserts and get the unlimited pytest feature set.

2.15 Running tests written for nose

py.test has basic support for running tests written for nose.

2.15.1 Usage

After *Installation* type:

```
py.test  # instead of 'nosetests'
```

and you should be able to run your nose style tests and make use of py.test's capabilities.

2.15.2 Supported nose Idioms

- setup and teardown at module/class/method level
- SkipTest exceptions and markers
- setup/teardown decorators
- yield-based tests and their setup
- general usage of nose utilities

2.15.3 Unsupported idioms / known issues

- nose-style doctests are not collected and executed correctly, also doctest fixtures don't work.
- no nose-configuration is recognized

2.16 Doctest integration for modules and test files

By default all files matching the `test*.txt` pattern will be run through the python standard `doctest` module. You can change the pattern by issuing:

```
py.test --doctest-glob='*.rst'
```

on the command line. You can also trigger running of doctests from docstrings in all python modules (including regular python test modules):

```
py.test --doctest-modules
```

You can make these changes permanent in your project by putting them into a pytest.ini file like this:

```
# content of pytest.ini
[pytest]
addopts = --doctest-modules
```

If you then have a text file like this:

```
# content of example.rst

hello this is a doctest
>>> x = 3
>>> x
3
```

and another like this:

```
# content of mymodule.py
def something():
    """ a doctest in a docstring
    >>> something()
    42
    """
    return 42
```

then you can just invoke `py.test` without command line options:

```
$ py.test
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

mymodule.py .

========================= 1 passed in 0.02 seconds =========================
```

It is possible to use fixtures using the `getfixture` helper:

```
# content of example.rst
>>> tmp = getfixture('tmpdir')
>>> ...
```

CHAPTER

THREE

WORKING WITH PLUGINS AND CONFTEST FILES

py.test implements all aspects of configuration, collection, running and reporting by calling well specified hooks. Virtually any Python module can be registered as a plugin. It can implement any number of hook functions (usually two or three) which all have a `pytest_` prefix, making hook functions easy to distinguish and find. There are three basic location types:

- builtin plugins: loaded from py.test's internal `_pytest` directory.
- external plugins: modules discovered through setuptools entry points
- conftest.py plugins: modules auto-discovered in test directories

3.1 conftest.py: local per-directory plugins

local `conftest.py` plugins contain directory-specific hook implementations. Session and test running activities will invoke all hooks defined in `conftest.py` files closer to the root of the filesystem. Example: Assume the following layout and content of files:

```
a/conftest.py:
    def pytest_runtest_setup(item):
        # called for running each test in 'a' directory
        print ("setting up", item)

a/test_in_subdir.py:
    def test_sub():
        pass

test_flat.py:
    def test_flat():
        pass
```

Here is how you might run it:

```
py.test test_flat.py   # will not show "setting up"
py.test a/test_sub.py  # will show "setting up"
```

Note: If you have `conftest.py` files which do not reside in a python package directory (i.e. one containing an `__init__.py`) then "import conftest" can be ambiguous because there might be other `conftest.py` files as well on your PYTHONPATH or `sys.path`. It is thus good practise for projects to either put `conftest.py` under a package scope or to never import anything from a conftest.py file.

3.2 Installing External Plugins / Searching

Installing a plugin happens through any usual Python installation tool, for example:

```
pip install pytest-NAME
pip uninstall pytest-NAME
```

If a plugin is installed, py.test automatically finds and integrates it, there is no need to activate it. Here is a initial list of known plugins:

- pytest-django: write tests for django apps, using pytest integration.
- pytest-twisted: write tests for twisted apps, starting a reactor and processing deferreds from test functions.
- pytest-capturelog: to capture and assert about messages from the logging module
- pytest-xdist: to distribute tests to CPUs and remote hosts, to run in boxed mode which allows to survive segmentation faults, to run in looponfailing mode, automatically re-running failing tests on file changes, see also *xdist: pytest distributed testing plugin*
- pytest-timeout: to timeout tests based on function marks or global definitions.
- pytest-cache: to interactively re-run failing tests and help other plugins to store test run information across invocations.
- pytest-cov: coverage reporting, compatible with distributed testing
- pytest-pep8: a `--pep8` option to enable PEP8 compliance checking.
- oejskit: a plugin to run javascript unittests in life browsers

You may discover more plugins through a pytest- pypi.python.org search.

3.3 Writing a plugin by looking at examples

If you want to write a plugin, there are many real-life examples you can copy from:

- a custom collection example plugin: *A basic example for specifying tests in Yaml files*
- around 20 builtin plugins which provide py.test's own functionality
- many external plugins providing additional features

All of these plugins implement the documented well specified hooks to extend and add functionality.

3.4 Making your plugin installable by others

If you want to make your plugin externally available, you may define a so-called entry point for your distribution so that `py.test` finds your plugin module. Entry points are a feature that is provided by setuptools or Distribute. py.test looks up the `pytest11` entrypoint to discover its plugins and you can thus make your plugin available by definig it in your setuptools/distribute-based setup-invocation:

```
# sample ./setup.py file
from setuptools import setup

setup(
    name="myproject",
    packages = ['myproject']

    # the following makes a plugin available to py.test
    entry_points = {
        'pytest11': [
            'name_of_plugin = myproject.pluginmodule',
        ]
    },
)
```

If a package is installed this way, py.test will load `myproject.pluginmodule` as a plugin which can define well specified hooks.

3.5 Plugin discovery order at tool startup

py.test loads plugin modules at tool startup in the following way:

- by loading all builtin plugins
- by loading all plugins registered through setuptools entry points.
- by pre-scanning the command line for the `-p name` option and loading the specified plugin before actual command line parsing.
- by loading all `conftest.py` files as inferred by the command line invocation (test files and all of its *parent* directories). Note that `conftest.py` files from *sub* directories are by default not loaded at tool startup.
- by recursively loading all plugins specified by the `pytest_plugins` variable in `conftest.py` files

3.6 Requiring/Loading plugins in a test module or conftest file

You can require plugins in a test module or a conftest file like this:

```
pytest_plugins = "name1", "name2",
```

When the test module or conftest plugin is loaded the specified plugins will be loaded as well. You can also use dotted path like this:

```
pytest_plugins = "myapp.testsupport.myplugin"
```

which will import the specified module as a py.test plugin.

3.7 Accessing another plugin by name

If a plugin wants to collaborate with code from another plugin it can obtain a reference through the plugin manager like this:

```
plugin = config.pluginmanager.getplugin("name_of_plugin")
```

If you want to look at the names of existing plugins, use the `--traceconfig` option.

3.8 Finding out which plugins are active

If you want to find out which plugins are active in your environment you can type:

```
py.test --traceconfig
```

and will get an extended test header which shows activated plugins and their names. It will also print local plugins aka *conftest.py* files when they are loaded.

3.9 Deactivating / unregistering a plugin by name

You can prevent plugins from loading or unregister them:

```
py.test -p no:NAME
```

This means that any subsequent try to activate/load the named plugin will it already existing. See *Finding out which plugins are active* for how to obtain the name of a plugin.

CHAPTER

FOUR

PY.TEST DEFAULT PLUGIN REFERENCE

You can find the source code for the following plugins in the pytest repository.

`_pytest.assertion`	support for presenting detailed information in failing assertions.
`_pytest.capture`	per-test stdout/stderr capturing mechanisms, `capsys` and `capfd` function arguments.
`_pytest.config`	command line options, ini-file and conftest.py processing.
`_pytest.doctest`	discover and run doctests in modules and test files.
`_pytest.genscript`	generate a single-file self-contained version of py.test
`_pytest.helpconfig`	version info, help messages, tracing configuration.
`_pytest.junitxml`	report test results in JUnit-XML format, for use with Hudson and build integration servers.
`_pytest.mark`	generic mechanism for marking and selecting python functions.
`_pytest.monkeypatch`	monkeypatching and mocking functionality.
`_pytest.nose`	run test suites written for nose.
`_pytest.pastebin`	submit failure or test session information to a pastebin service.
`_pytest.pdb`	interactive debugging with PDB, the Python Debugger.
`_pytest.pytester`	(disabled by default) support for testing py.test and py.test plugins.
`_pytest.python`	Python test discovery, setup and run of test functions.
`_pytest.recwarn`	recording warnings during test function execution.
`_pytest.resultlog`	log machine-parseable test session result information in a plain
`_pytest.runner`	basic collect and runtest protocol implementations
`_pytest.main`	core implementation of testing process: init, session, runtest loop.
`_pytest.skipping`	support for skip/xfail functions and markers.
`_pytest.terminal`	terminal reporting of the full testing process.
`_pytest.tmpdir`	support for providing temporary directories to test functions.
`_pytest.unittest`	discovery and running of std-library “unittest” style tests.

CHAPTER

FIVE

PY.TEST HOOK REFERENCE

5.1 Hook specification and validation

py.test calls hook functions to implement initialization, running, test execution and reporting. When py.test loads a plugin it validates that each hook function conforms to its respective hook specification. Each hook function name and its argument names need to match a hook specification. However, a hook function may accept *fewer* parameters by simply not specifying them. If you mistype argument names or the hook name itself you get an error showing the available arguments.

5.2 Initialization, command line and configuration hooks

pytest_cmdline_preparse(*config*, *args*)
modify command line arguments before option parsing.

pytest_cmdline_parse(*pluginmanager*, *args*)
return initialized config object, parsing the specified args.

pytest_namespace()
return dict of name->object to be made globally available in the py.test/pytest namespace. This hook is called before command line options are parsed.

pytest_addoption(*parser*)
register optparse-style options and ini-style config values.

This function must be implemented in a *plugin* and is called once at the beginning of a test run.

Parameters **parser** – To add command line options, call `parser.addoption(...)`. To add ini-file values call `parser.addini(...)`.

Options can later be accessed through the `config` object, respectively:

- `config.getoption(name)` to retrieve the value of a command line option.
- `config.getini(name)` to retrieve a value read from an ini-style file.

The config object is passed around on many internal objects via the `.config` attribute or can be retrieved as the `pytestconfig` fixture or accessed via (deprecated) `pytest.config`.

pytest_cmdline_main(*config*)
called for performing the main command line action. The default implementation will invoke the configure hooks and runtest_mainloop.

pytest_configure(*config*)
called after command line options have been parsed and all plugins and initial conftest files been loaded.

pytest_unconfigure(*config*)
called before test process is exited.

5.3 Generic "runtest" hooks

All all runtest related hooks receive a `pytest.Item` object.

pytest_runtest_protocol(*item*, *nextitem*)
implements the runtest_setup/call/teardown protocol for the given test item, including capturing exceptions and calling reporting hooks.

Parameters

- **item** – test item for which the runtest protocol is performed.
- **nexitem** – the scheduled-to-be-next test item (or None if this is the end my friend). This argument is passed on to `pytest_runtest_teardown()`.

Return boolean True if no further hook implementations should be invoked.

pytest_runtest_setup(*item*)
called before `pytest_runtest_call(item)`.

pytest_runtest_call(*item*)
called to execute the test `item`.

pytest_runtest_teardown(*item*, *nextitem*)
called after `pytest_runtest_call`.

Parameters **nexitem** – the scheduled-to-be-next test item (None if no further test item is scheduled). This argument can be used to perform exact teardowns, i.e. calling just enough finalizers so that nextitem only needs to call setup-functions.

pytest_runtest_makereport(*item*, *call*)
return a `_pytest.runner.TestReport` object for the given `pytest.Item` and `_pytest.runner.CallInfo`.

For deeper understanding you may look at the default implementation of these hooks in `_pytest.runner` and maybe also in `_pytest.pdb` which interacts with `_pytest.capture` and its input/output capturing in order to immediately drop into interactive debugging when a test failure occurs.

The `_pytest.terminal` reported specifically uses the reporting hook to print information about a test run.

5.4 Collection hooks

py.test calls the following hooks for collecting files and directories:

pytest_ignore_collect(*path*, *config*)
return True to prevent considering this path for collection. This hook is consulted for all files and directories prior to calling more specific hooks.

pytest_collect_directory(*path*, *parent*)
called before traversing a directory for collection files.

pytest_collect_file(*path*, *parent*)
return collection Node or None for the given path. Any new node needs to have the specified `parent` as a parent.

For influencing the collection of objects in Python modules you can use the following hook:

pytest_pycollect_makeitem(*collector*, *name*, *obj*)
return custom item/collector for a python object in a module, or None.

pytest_generate_tests(*metafunc*)
generate (multiple) parametrized calls to a test function.

5.5 Reporting hooks

Session related reporting hooks:

And here is the central hook for reporting about test execution:

CHAPTER

SIX

REFERENCE OF OBJECTS INVOLVED IN HOOKS

class `Config`
access to configuration values, pluginmanager and plugin hooks.

`option` = None
access to command line option as attributes. (deprecated), use `getoption()` instead

`pluginmanager` = None
a pluginmanager instance

classmethod `fromdictargs`(*option_dict*, *args*)
constructor useable for subprocesses.

`addinivalue_line`(*name*, *line*)
add a line to an ini-file option. The option must have been declared but might not yet be set in which case the line becomes the the first line in its value.

`getini`(*name*)
return configuration value from an *ini file*. If the specified name hasn't been registered through a prior `parser.addini` call (usually from a plugin), a ValueError is raised.

`getoption`(*name*)
return command line option value.

Parameters **name** – name of the option. You may also specify the literal `--OPT` option instead of the "dest" option name.

`getvalue`(*name*, *path=None*)
return command line option value.

Parameters **name** – name of the command line option

(deprecated) if we can't find the option also lookup the name in a matching conftest file.

`getvalueorskip`(*name*, *path=None*)
(deprecated) return getvalue(name) or call py.test.skip if no value exists.

class `Parser`
Parser for command line arguments and ini-file values.

`getgroup`(*name*, *description=''*, *after=None*)
get (or create) a named option Group.

Name name of the option group.

Description long description for –help output.

After name of other group, used for ordering –help output.

The returned group object has an `addoption` method with the same signature as `parser.addoption` but will be shown in the respective group in the output of `pytest. --help`.

addoption(**opts*, ***attrs*)
register a command line option.

Opts option names, can be short or long options.

Attrs same attributes which the `add_option()` function of the optparse library accepts.

After command line parsing options are available on the pytest config object via `config.option.NAME` where `NAME` is usually set by passing a `dest` attribute, for example `addoption("--long", dest="NAME", ...)`.

addini(*name*, *help*, *type=None*, *default=None*)
register an ini-file option.

Name name of the ini-variable

Type type of the variable, can be `pathlist`, `args` or `linelist`.

Default default value if no ini-file option exists but is queried.

The value of ini-variables can be retrieved via a call to `config.getini(name)`.

class Node
base class for Collector and Item the test collection tree. Collector subclasses have children, Items are terminal nodes.

name = None
a unique name within the scope of the parent node

parent = None
the parent collector node.

config = None
the pytest config object

session = None
the session this node is part of

fspath = None
filesystem path where this node was collected from (can be None)

keywords = None
keywords/markers collected from all scopes

ihook
fspath sensitive hook proxy used to call pytest hooks

nodeid
a ::-separated string denoting its collection tree address.

listchain()
return list of all parent collectors up to self, starting from root of collection tree.

class Collector
Bases: `_pytest.main.Node`

Collector instances create children through collect() and thus iteratively build a tree.

exception CollectError
Bases: `exceptions.Exception`

an error during collection, contains a custom message.

Collector.**collect**()
returns a list of children (items and collectors) for this collection node.

Collector.**repr_failure**(*excinfo*)
represent a collection failure.

class Item
Bases: _pytest.main.Node

a basic test invocation item. Note that for a single function there might be multiple test invocation items.

class Module
Bases: _pytest.main.File, _pytest.python.PyCollector

Collector for test classes and functions.

class Class
Bases: _pytest.python.PyCollector

Collector for test methods.

class Function
Bases: _pytest.python.FunctionMixin, _pytest.main.Item, _pytest.python.FuncargnamesCompatAttr

a Function Item is responsible for setting up and executing a Python test function.

function
underlying python 'function' object

runtest()
execute the underlying test function.

class CallInfo
Result/Exception info a function invocation.

when = None
context of invocation: one of "setup", "call", "teardown", "memocollect"

excinfo = None
None or ExceptionInfo object.

class TestReport
Basic test report object (also used for setup and teardown calls if they fail).

nodeid = None
normalized collection node id

location = None
a (filesystempath, lineno, domaininfo) tuple indicating the actual location of a test item - it might be different from the collected one e.g. if a method is inherited from a different module.

keywords = None
a name -> value dictionary containing all keywords and markers associated with a test invocation.

outcome = None
test outcome, always one of "passed", "failed", "skipped".

longrepr = None
None or a failure representation.

when = None
one of 'setup', 'call', 'teardown' to indicate runtest phase.

sections = None
list of (secname, data) extra information which needs to marshallable

duration = None
time it took to run just the test

CHAPTER

SEVEN

USAGES AND EXAMPLES

Here is a (growing) list of examples. *Contact* us if you need more examples or have questions. Also take a look at the *comprehensive documentation* which contains many example snippets as well. Also, pytest on stackoverflow.com often comes with example answers.

For basic examples, see

- *Installation and Getting Started* for basic introductory examples
- *Asserting with the assert statement* for basic assertion examples
- *pytest fixtures: explicit, modular, scalable* for basic fixture/setup examples
- *Parametrizing fixtures and test functions* for basic test function parametrization
- *Support for unittest.TestCase / Integration of fixtures* for basic unittest integration
- *Running tests written for nose* for basic nosetests integration

The following examples aim at various use cases you might encounter.

7.1 Demo of Python failure reports with py.test

Here is a nice run of several tens of failures and how py.test presents things (unfortunately not showing the nice colors here in the HTML that you get on the terminal - we are working on that):

```
assertion $ py.test failure_demo.py
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 39 items

failure_demo.py FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

================================ FAILURES ================================
_____________________________ test_generative[0] _____________________________

param1 = 3, param2 = 6

    def test_generative(param1, param2):
>       assert param1 * 2 < param2
E       assert (3 * 2) < 6

failure_demo.py:15: AssertionError
____________________________ TestFailing.test_simple ____________________________
```

```
self = <failure_demo.TestFailing object at 0x1445e10>

    def test_simple(self):
        def f():
            return 42
        def g():
            return 43

>       assert f() == g()
E       assert 42 == 43
E        +  where 42 = <function f at 0x137c6e0>()
E        +  and   43 = <function g at 0x137c758>()

failure_demo.py:28: AssertionError
____________________ TestFailing.test_simple_multiline ____________________

self = <failure_demo.TestFailing object at 0x135a1d0>

    def test_simple_multiline(self):
        otherfunc_multi(
                  42,
>                 6*9)

failure_demo.py:33:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 42, b = 54

    def otherfunc_multi(a,b):
>       assert (a ==
                b)
E       assert 42 == 54

failure_demo.py:11: AssertionError
_____________________________ TestFailing.test_not _____________________________

self = <failure_demo.TestFailing object at 0x1458ed0>

    def test_not(self):
        def f():
            return 42
>       assert not f()
E       assert not 42
E        +  where 42 = <function f at 0x137caa0>()

failure_demo.py:38: AssertionError
___________________ TestSpecialisedExplanations.test_eq_text ___________________

self = <failure_demo.TestSpecialisedExplanations object at 0x14451d0>

    def test_eq_text(self):
>       assert 'spam' == 'eggs'
E       assert 'spam' == 'eggs'
E         - spam
E         + eggs

failure_demo.py:42: AssertionError
_____________ TestSpecialisedExplanations.test_eq_similar_text _____________
```

```
self = <failure_demo.TestSpecialisedExplanations object at 0x1458c90>

    def test_eq_similar_text(self):
>       assert 'foo 1 bar' == 'foo 2 bar'
E       assert 'foo 1 bar' == 'foo 2 bar'
E         - foo 1 bar
E         ?     ^
E         + foo 2 bar
E         ?     ^

failure_demo.py:45: AssertionError
_____________ TestSpecialisedExplanations.test_eq_multiline_text _____________

self = <failure_demo.TestSpecialisedExplanations object at 0x1434390>

    def test_eq_multiline_text(self):
>       assert 'foo\nspam\nbar' == 'foo\neggs\nbar'
E       assert 'foo\nspam\nbar' == 'foo\neggs\nbar'
E           foo
E         - spam
E         + eggs
E           bar

failure_demo.py:48: AssertionError
_____________ TestSpecialisedExplanations.test_eq_long_text _____________

self = <failure_demo.TestSpecialisedExplanations object at 0x1459f50>

    def test_eq_long_text(self):
        a = '1'*100 + 'a' + '2'*100
        b = '1'*100 + 'b' + '2'*100
>       assert a == b
E       assert '111111111111...2222222222222' == '1111111111111...2222222222222'
E         Skipping 90 identical leading characters in diff, use -v to show
E         Skipping 91 identical trailing characters in diff, use -v to show
E         - 1111111111a222222222
E         ?           ^
E         + 1111111111b222222222
E         ?           ^

failure_demo.py:53: AssertionError
_____________ TestSpecialisedExplanations.test_eq_long_text_multiline _____________

self = <failure_demo.TestSpecialisedExplanations object at 0x135a790>

    def test_eq_long_text_multiline(self):
        a = '1\n'*100 + 'a' + '2\n'*100
        b = '1\n'*100 + 'b' + '2\n'*100
>       assert a == b
E       assert '1\n1\n1\n1\n...n2\n2\n2\n2\n' == '1\n1\n1\n1\n1...n2\n2\n2\n2\n'
E         Skipping 190 identical leading characters in diff, use -v to show
E         Skipping 191 identical trailing characters in diff, use -v to show
E           1
E           1
E           1
E           1
E           1
E         - a2
```

```
E         + b2
E           2
E           2
E           2
E           2

failure_demo.py:58: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_list _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0x138dfd0>

    def test_eq_list(self):
>       assert [0, 1, 2] == [0, 1, 3]
E       assert [0, 1, 2] == [0, 1, 3]
E         At index 2 diff: 2 != 3

failure_demo.py:61: AssertionError
_________________ TestSpecialisedExplanations.test_eq_list_long _________________

self = <failure_demo.TestSpecialisedExplanations object at 0x135a990>

    def test_eq_list_long(self):
        a = [0]*100 + [1] + [3]*100
        b = [0]*100 + [2] + [3]*100
>       assert a == b
E       assert [0, 0, 0, 0, 0, 0, ...] == [0, 0, 0, 0, 0, 0, ...]
E         At index 100 diff: 1 != 2

failure_demo.py:66: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_dict _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0x1459310>

    def test_eq_dict(self):
>       assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
E       assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
E         Hiding 1 identical items, use -v to show
E         Differing items:
E         {'b': 1} != {'b': 2}
E         Left contains more items:
E         {'c': 0}
E         Right contains more items:
E         {'d': 0}

failure_demo.py:69: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_set _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0x1434310>

    def test_eq_set(self):
>       assert set([0, 10, 11, 12]) == set([0, 20, 21])
E       assert set([0, 10, 11, 12]) == set([0, 20, 21])
E         Extra items in the left set:
E         10
E         11
E         12
E         Extra items in the right set:
E         20
```

```
E         21

failure_demo.py:72: AssertionError
_____________ TestSpecialisedExplanations.test_eq_longer_list _____________

self = <failure_demo.TestSpecialisedExplanations object at 0x138ded0>

    def test_eq_longer_list(self):
>       assert [1,2] == [1,2,3]
E       assert [1, 2] == [1, 2, 3]
E         Right contains more items, first extra item: 3

failure_demo.py:75: AssertionError
_____________________ TestSpecialisedExplanations.test_in_list _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0x1459e10>

    def test_in_list(self):
>       assert 1 in [0, 2, 3, 4, 5]
E       assert 1 in [0, 2, 3, 4, 5]

failure_demo.py:78: AssertionError
___________ TestSpecialisedExplanations.test_not_in_text_multiline ___________

self = <failure_demo.TestSpecialisedExplanations object at 0x1434950>

    def test_not_in_text_multiline(self):
        text = 'some multiline\ntext\nwhich\nincludes foo\nand a\ntail'
>       assert 'foo' not in text
E       assert 'foo' not in 'some multiline\ntext\nw...ncludes foo\nand a\ntail'
E         'foo' is contained here:
E           some multiline
E           text
E           which
E           includes foo
E         ?          +++
E           and a
E           tail

failure_demo.py:82: AssertionError
_____________ TestSpecialisedExplanations.test_not_in_text_single _____________

self = <failure_demo.TestSpecialisedExplanations object at 0x138dbd0>

    def test_not_in_text_single(self):
        text = 'single foo line'
>       assert 'foo' not in text
E       assert 'foo' not in 'single foo line'
E         'foo' is contained here:
E           single foo line
E         ?        +++

failure_demo.py:86: AssertionError
___________ TestSpecialisedExplanations.test_not_in_text_single_long ___________

self = <failure_demo.TestSpecialisedExplanations object at 0x14593d0>

    def test_not_in_text_single_long(self):
```

```
        text = 'head ' * 50 + 'foo ' + 'tail ' * 20
>       assert 'foo' not in text
E       assert 'foo' not in 'head head head head hea...ail tail tail tail '
E         'foo' is contained here:
E           head head foo tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail
E         ?           +++

failure_demo.py:90: AssertionError
_______ TestSpecialisedExplanations.test_not_in_text_single_long_term ________

self = <failure_demo.TestSpecialisedExplanations object at 0x1459650>

    def test_not_in_text_single_long_term(self):
        text = 'head ' * 50 + 'f'*70 + 'tail ' * 20
>       assert 'f'*70 not in text
E       assert 'fffffffffff...fffffffffff' not in 'head head he...l tail tail '
E         'fffffffffffffffff...fffffffffffffffff' is contained here:
E           head head fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail tail
E         ?           ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

failure_demo.py:94: AssertionError
_____________________________ test_attribute ______________________________

    def test_attribute():
        class Foo(object):
            b = 1
        i = Foo()
>       assert i.b == 2
E       assert 1 == 2
E        +  where 1 = <failure_demo.Foo object at 0x1434850>.b

failure_demo.py:101: AssertionError
_____________________________ test_attribute_instance _____________________________

    def test_attribute_instance():
        class Foo(object):
            b = 1
>       assert Foo().b == 2
E       assert 1 == 2
E        +  where 1 = <failure_demo.Foo object at 0x1459dd0>.b
E        +    where <failure_demo.Foo object at 0x1459dd0> = <class 'failure_demo.Foo'>()

failure_demo.py:107: AssertionError
_____________________________ test_attribute_failure ______________________________

    def test_attribute_failure():
        class Foo(object):
            def _get_b(self):
                raise Exception('Failed to get attrib')
            b = property(_get_b)
        i = Foo()
>       assert i.b == 2

failure_demo.py:116:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <failure_demo.Foo object at 0x1434150>
```

```
    def _get_b(self):
>       raise Exception('Failed to get attrib')
E       Exception: Failed to get attrib

failure_demo.py:113: Exception
_____________________________ test_attribute_multiple _____________________________

    def test_attribute_multiple():
        class Foo(object):
            b = 1
        class Bar(object):
            b = 2
>       assert Foo().b == Bar().b
E       assert 1 == 2
E        +  where 1 = <failure_demo.Foo object at 0x14590d0>.b
E        +    where <failure_demo.Foo object at 0x14590d0> = <class 'failure_demo.Foo'>()
E        +  and   2 = <failure_demo.Bar object at 0x1459b10>.b
E        +    where <failure_demo.Bar object at 0x1459b10> = <class 'failure_demo.Bar'>()

failure_demo.py:124: AssertionError
_____________________________ TestRaises.test_raises _____________________________

self = <failure_demo.TestRaises instance at 0x13a0d88>

    def test_raises(self):
        s = 'qwe'
>       raises(TypeError, "int(s)")

failure_demo.py:133:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

>   int(s)
E   ValueError: invalid literal for int() with base 10: 'qwe'

<0-codegen /home/hpk/p/pytest/.tox/regen/local/lib/python2.7/site-packages/_pytest/python.py:858>:1:
____________________ TestRaises.test_raises_doesnt _____________________

self = <failure_demo.TestRaises instance at 0x145fcf8>

    def test_raises_doesnt(self):
>       raises(IOError, "int('3')")
E       Failed: DID NOT RAISE

failure_demo.py:136: Failed
_____________________________ TestRaises.test_raise _____________________________

self = <failure_demo.TestRaises instance at 0x13a9ea8>

    def test_raise(self):
>       raise ValueError("demo error")
E       ValueError: demo error

failure_demo.py:139: ValueError
____________________ TestRaises.test_tupleerror ____________________

self = <failure_demo.TestRaises instance at 0x13843f8>

    def test_tupleerror(self):
```

```
>       a,b = [1]
E       ValueError: need more than 1 value to unpack

failure_demo.py:142: ValueError
_______ TestRaises.test_reinterpret_fails_with_print_for_the_fun_of_it _______

self = <failure_demo.TestRaises instance at 0x14532d8>

    def test_reinterpret_fails_with_print_for_the_fun_of_it(self):
        l = [1,2,3]
        print ("l is %r" % l)
>       a,b = l.pop()
E       TypeError: 'int' object is not iterable

failure_demo.py:147: TypeError
----------------------------- Captured stdout ------------------------------
l is [1, 2, 3]
____________________________ TestRaises.test_some_error ____________________________

self = <failure_demo.TestRaises instance at 0x139d290>

    def test_some_error(self):
>       if namenotexi:
E       NameError: global name 'namenotexi' is not defined

failure_demo.py:150: NameError
_____________________ test_dynamic_compile_shows_nicely _____________________

    def test_dynamic_compile_shows_nicely():
        src = 'def foo():\n assert 1 == 0\n'
        name = 'abc-123'
        module = py.std.imp.new_module(name)
        code = py.code.compile(src, name, 'exec')
        py.builtin.exec_(code, module.__dict__)
        py.std.sys.modules[name] = module
>       module.foo()

failure_demo.py:165:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    def foo():
>    assert 1 == 0
E    assert 1 == 0

<2-codegen 'abc-123' /home/hpk/p/pytest/doc/en/example/assertion/failure_demo.py:162>:2: AssertionEr
____________________ TestMoreErrors.test_complex_error ____________________

self = <failure_demo.TestMoreErrors instance at 0x137d758>

    def test_complex_error(self):
        def f():
            return 44
        def g():
            return 43
>       somefunc(f(), g())

failure_demo.py:175:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

```
x = 44, y = 43

    def somefunc(x,y):
>       otherfunc(x,y)

failure_demo.py:8:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 44, b = 43

    def otherfunc(a,b):
>       assert a==b
E       assert 44 == 43

failure_demo.py:5: AssertionError
_____________________ TestMoreErrors.test_z1_unpack_error ____________________

self = <failure_demo.TestMoreErrors instance at 0x13a5200>

    def test_z1_unpack_error(self):
        l = []
>       a,b  = l
E       ValueError: need more than 0 values to unpack

failure_demo.py:179: ValueError
_____________________ TestMoreErrors.test_z2_type_error ____________________

self = <failure_demo.TestMoreErrors instance at 0x1395290>

    def test_z2_type_error(self):
        l = 3
>       a,b  = l
E       TypeError: 'int' object is not iterable

failure_demo.py:183: TypeError
_________________________ TestMoreErrors.test_startswith _________________________

self = <failure_demo.TestMoreErrors instance at 0x137f200>

    def test_startswith(self):
        s = "123"
        g = "456"
>       assert s.startswith(g)
E       assert <built-in method startswith of str object at 0x143f288>('456')
E        +  where <built-in method startswith of str object at 0x143f288> = '123'.startswith

failure_demo.py:188: AssertionError
____________________ TestMoreErrors.test_startswith_nested ____________________

self = <failure_demo.TestMoreErrors instance at 0x145fb00>

    def test_startswith_nested(self):
        def f():
            return "123"
        def g():
            return "456"
>       assert f().startswith(g())
E       assert <built-in method startswith of str object at 0x143f288>('456')
```

```
E        +  where <built-in method startswith of str object at 0x143f288> = '123'.startswith
E        +    where '123' = <function f at 0x13abaa0>()
E        +  and   '456' = <function g at 0x13ab578>()

failure_demo.py:195: AssertionError
_____________________ TestMoreErrors.test_global_func ____________________

self = <failure_demo.TestMoreErrors instance at 0x139cd40>

    def test_global_func(self):
>       assert isinstance(globf(42), float)
E       assert isinstance(43, float)
E        +  where 43 = globf(42)

failure_demo.py:198: AssertionError
_____________________ TestMoreErrors.test_instance ____________________

self = <failure_demo.TestMoreErrors instance at 0x13593b0>

    def test_instance(self):
        self.x = 6*7
>       assert self.x != 42
E       assert 42 != 42
E        +  where 42 = <failure_demo.TestMoreErrors instance at 0x13593b0>.x

failure_demo.py:202: AssertionError
_____________________ TestMoreErrors.test_compare ____________________

self = <failure_demo.TestMoreErrors instance at 0x1465d40>

    def test_compare(self):
>       assert globf(10) < 5
E       assert 11 < 5
E        +  where 11 = globf(10)

failure_demo.py:205: AssertionError
_____________________ TestMoreErrors.test_try_finally ____________________

self = <failure_demo.TestMoreErrors instance at 0x1456ea8>

    def test_try_finally(self):
        x = 1
        try:
>           assert x == 0
E           assert 1 == 0

failure_demo.py:210: AssertionError
======================== 39 failed in 0.21 seconds =========================
```

7.2 Basic patterns and examples

7.2.1 Pass different values to a test function, depending on command line options

Suppose we want to write a test that depends on a command line option. Here is a basic pattern how to achieve this:

```
# content of test_sample.py
def test_answer(cmdopt):
    if cmdopt == "type1":
        print ("first")
    elif cmdopt == "type2":
        print ("second")
    assert 0 # to see what was printed
```

For this to work we need to add a command line option and provide the `cmdopt` through a *fixture function*:

```
# content of conftest.py
import pytest

def pytest_addoption(parser):
    parser.addoption("--cmdopt", action="store", default="type1",
        help="my option: type1 or type2")

@pytest.fixture
def cmdopt(request):
    return request.config.getoption("--cmdopt")
```

Let's run this without supplying our new option:

```
$ py.test -q test_sample.py
F
================================ FAILURES ================================
_____________________________ test_answer _____________________________

cmdopt = 'type1'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print ("first")
        elif cmdopt == "type2":
            print ("second")
>       assert 0 # to see what was printed
E       assert 0

test_sample.py:6: AssertionError
---------------------------- Captured stdout -----------------------------
first
```

And now with supplying a command line option:

```
$ py.test -q --cmdopt=type2
F
================================ FAILURES ================================
_____________________________ test_answer _____________________________

cmdopt = 'type2'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print ("first")
        elif cmdopt == "type2":
            print ("second")
>       assert 0 # to see what was printed
E       assert 0
```

```
test_sample.py:6: AssertionError
---------------------------- Captured stdout -----------------------------
second
```

You can see that the command line option arrived in our test. This completes the basic pattern. However, one often rather wants to process command line options outside of the test and rather pass in different or more complex objects.

7.2.2 Dynamically adding command line options

Through `addopts` you can statically add command line options for your project. You can also dynamically modify the command line arguments before they get processed:

```
# content of conftest.py
import sys
def pytest_cmdline_preparse(args):
    if 'xdist' in sys.modules: # pytest-xdist plugin
        import multiprocessing
        num = max(multiprocessing.cpu_count() / 2, 1)
        args[:] = ["-n", str(num)] + args
```

If you have the *xdist plugin* installed you will now always perform test runs using a number of subprocesses close to your CPU. Running in an empty directory with the above conftest.py:

```
$ py.test
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 0 items

============================  in 0.00 seconds ============================
```

7.2.3 Control skipping of tests according to command line option

Here is a `conftest.py` file adding a `--runslow` command line option to control skipping of `slow` marked tests:

```
# content of conftest.py

import pytest
def pytest_addoption(parser):
    parser.addoption("--runslow", action="store_true",
        help="run slow tests")

def pytest_runtest_setup(item):
    if 'slow' in item.keywords and not item.config.getoption("--runslow"):
        pytest.skip("need --runslow option to run")
```

We can now write a test module like this:

```
# content of test_module.py

import pytest
slow = pytest.mark.slow

def test_func_fast():
    pass

@slow
```

```
def test_func_slow():
    pass
```

and when running it will see a skipped "slow" test:

```
$ py.test -rs    # "-rs" means report details on the little 's'
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_module.py .s
========================== short test summary info ==========================
SKIP [1] /tmp/doc-exec-278/conftest.py:9: need --runslow option to run

=================== 1 passed, 1 skipped in 0.01 seconds ====================
```

Or run it including the `slow` marked test:

```
$ py.test --runslow
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_module.py ..

======================== 2 passed in 0.01 seconds =========================
```

7.2.4 Writing well integrated assertion helpers

If you have a test helper function called from a test you can use the `pytest.fail` marker to fail a test with a certain message. The test support function will not show up in the traceback if you set the `__tracebackhide__` option somewhere in the helper function. Example:

```
# content of test_checkconfig.py
import pytest
def checkconfig(x):
    __tracebackhide__ = True
    if not hasattr(x, "config"):
        pytest.fail("not configured: %s" %(x,))

def test_something():
    checkconfig(42)
```

The `__tracebackhide__` setting influences py.test showing of tracebacks: the `checkconfig` function will not be shown unless the `--fulltrace` command line option is specified. Let's run our little function:

```
$ py.test -q test_checkconfig.py
F
================================ FAILURES =================================
_____________________________ test_something ______________________________

    def test_something():
>       checkconfig(42)
E       Failed: not configured: 42

test_checkconfig.py:8: Failed
```

7.2.5 Detect if running from within a py.test run

Usually it is a bad idea to make application code behave differently if called from a test. But if you absolutely must find out if your application code is running from a test you can do something like this:

```
# content of conftest.py

def pytest_configure(config):
    import sys
    sys._called_from_test = True

def pytest_unconfigure(config):
    del sys._called_from_test
```

and then check for the `sys._called_from_test` flag:

```
if hasattr(sys, '_called_from_test'):
    # called from within a test run
else:
    # called "normally"
```

accordingly in your application. It's also a good idea to use your own application module rather than `sys` for handling flag.

7.2.6 Adding info to test report header

It's easy to present extra information in a py.test run:

```
# content of conftest.py

def pytest_report_header(config):
    return "project deps: mylib-1.1"
```

which will add the string to the test header accordingly:

```
$ py.test
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
project deps: mylib-1.1
collected 0 items

============================  in 0.00 seconds =============================
```

You can also return a list of strings which will be considered as several lines of information. You can of course also make the amount of reporting information on e.g. the value of `config.option.verbose` so that you present more information appropriately:

```
# content of conftest.py

def pytest_report_header(config):
    if config.option.verbose > 0:
        return ["info1: did you know that ...", "did you?"]
```

which will add info only when run with "–v":

```
$ py.test -v
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
```

```
info1: did you know that ...
did you?
collecting ... collected 0 items

============================  in 0.00 seconds =============================
```

and nothing when run plainly:

```
$ py.test
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 0 items

============================  in 0.00 seconds =============================
```

7.2.7 profiling test duration

If you have a slow running large test suite you might want to find out which tests are the slowest. Let's make an artifical test suite:

```
# content of test_some_are_slow.py

import time

def test_funcfast():
    pass

def test_funcslow1():
    time.sleep(0.1)

def test_funcslow2():
    time.sleep(0.2)
```

Now we can profile which test functions execute the slowest:

```
$ py.test --durations=3
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 3 items

test_some_are_slow.py ...

======================== slowest 3 test durations ========================
0.20s call     test_some_are_slow.py::test_funcslow2
0.10s call     test_some_are_slow.py::test_funcslow1
0.00s setup    test_some_are_slow.py::test_funcfast
======================== 3 passed in 0.31 seconds ========================
```

7.2.8 incremental testing - test steps

Sometimes you may have a testing situation which consists of a series of test steps. If one step fails it makes no sense to execute further steps as they are all expected to fail anyway and their tracebacks add no insight. Here is a simple `conftest.py` file which introduces an `incremental` marker which is to be used on classes:

```
# content of conftest.py
```

```
import pytest

def pytest_runtest_makereport(item, call):
    if "incremental" in item.keywords:
        if call.excinfo is not None:
            parent = item.parent
            parent._previousfailed = item

def pytest_runtest_setup(item):
    if "incremental" in item.keywords:
        previousfailed = getattr(item.parent, "_previousfailed", None)
        if previousfailed is not None:
            pytest.xfail("previous test failed (%s)" %previousfailed.name)
```

These two hook implementations work together to abort incremental-marked tests in a class. Here is a test module example:

```
# content of test_step.py

import pytest

@pytest.mark.incremental
class TestUserHandling:
    def test_login(self):
        pass
    def test_modification(self):
        assert 0
    def test_deletion(self):
        pass

def test_normal():
    pass
```

If we run this:

```
$ py.test -rx
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 4 items

test_step.py .Fx.

================================= FAILURES =================================
______________________ TestUserHandling.test_modification ______________________

self = <test_step.TestUserHandling instance at 0x282b8c0>

    def test_modification(self):
>       assert 0
E       assert 0

test_step.py:9: AssertionError
======================== short test summary info ========================
XFAIL test_step.py::TestUserHandling::()::test_deletion
  reason: previous test failed (test_modification)
============== 1 failed, 2 passed, 1 xfailed in 0.01 seconds ===============
```

We'll see that `test_deletion` was not executed because `test_modification` failed. It is reported as an "expected failure".

7.2.9 Package/Directory-level fixtures (setups)

If you have nested test directories, you can have per-directory fixture scopes by placing fixture functions in a `conftest.py` file in that directory You can use all types of fixtures including *autouse fixtures* which are the equivalent of xUnit's setup/teardown concept. It's however recommended to have explicit fixture references in your tests or test classes rather than relying on implicitely executing setup/teardown functions, especially if they are far away from the actual tests.

Here is a an example for making a `db` fixture available in a directory:

```
# content of a/conftest.py
import pytest

class DB:
    pass

@pytest.fixture(scope="session")
def db():
    return DB()
```

and then a test module in that directory:

```
# content of a/test_db.py
def test_a1(db):
    assert 0, db  # to show value
```

another test module:

```
# content of a/test_db2.py
def test_a2(db):
    assert 0, db  # to show value
```

and then a module in a sister directory which will not see the `db` fixture:

```
# content of b/test_error.py
def test_root(db):  # no db here, will error out
    pass
```

We can run this:

```
$ py.test
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 7 items

test_step.py .Fx.
a/test_db.py F
a/test_db2.py F
b/test_error.py E

================================== ERRORS ==================================
_____________________ ERROR at setup of test_root ______________________
file /tmp/doc-exec-278/b/test_error.py, line 1
  def test_root(db):  # no db here, will error out
        fixture 'db' not found
        available fixtures: pytestconfig, recwarn, monkeypatch, capfd, capsys, tmpdir
        use 'py.test --fixtures [testpath]' for help on them.

/tmp/doc-exec-278/b/test_error.py:1
================================= FAILURES =================================
```

```
_____________________ TestUserHandling.test_modification ____________________

self = <test_step.TestUserHandling instance at 0x26145f0>

    def test_modification(self):
>       assert 0
E       assert 0

test_step.py:9: AssertionError
_____________________________________ test_a1 ______________________________________

db = <conftest.DB instance at 0x26211b8>

    def test_a1(db):
>       assert 0, db  # to show value
E       AssertionError: <conftest.DB instance at 0x26211b8>

a/test_db.py:2: AssertionError
_____________________________________ test_a2 ______________________________________

db = <conftest.DB instance at 0x26211b8>

    def test_a2(db):
>       assert 0, db  # to show value
E       AssertionError: <conftest.DB instance at 0x26211b8>

a/test_db2.py:2: AssertionError
========== 3 failed, 2 passed, 1 xfailed, 1 error in 0.03 seconds ==========
```

The two test modules in the `a` directory see the same `db` fixture instance while the one test in the sister-directory `b` doesn't see it. We could of course also define a `db` fixture in that sister directory's `conftest.py` file. Note that each fixture is only instantiated if there is a test actually needing it (unless you use "autouse" fixture which are always executed ahead of the first test executing).

7.2.10 post-process test reports / failures

If you want to postprocess test reports and need access to the executing environment you can implement a hook that gets called when the test "report" object is about to be created. Here we write out all failing test calls and also access a fixture (if it was used by the test) in case you want to query/look at it during your post processing. In our case we just write some informations out to a `failures` file:

```
# content of conftest.py

import pytest
import os.path

@pytest.mark.tryfirst
def pytest_runtest_makereport(item, call, __multicall__):
    # execute all other hooks to obtain the report object
    rep = __multicall__.execute()

    # we only look at actual failing test calls, not setup/teardown
    if rep.when == "call" and rep.failed:
        mode = "a" if os.path.exists("failures") else "w"
        with open("failures", mode) as f:
            # let's also access a fixture for the fun of it
            if "tmpdir" in item.funcargs:
```

```
                extra = " (%s)" % item.funcargs["tmpdir"]
            else:
                extra = ""

            f.write(rep.nodeid + extra + "\n")
    return rep
```

if you then have failing tests:

```
# content of test_module.py
def test_fail1(tmpdir):
    assert 0
def test_fail2():
    assert 0
```

and run them:

```
$ py.test test_module.py
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_module.py FF

================================= FAILURES =================================
_____________________________________ test_fail1 _____________________________________

tmpdir = local('/tmp/pytest-326/test_fail10')

    def test_fail1(tmpdir):
>       assert 0
E       assert 0

test_module.py:2: AssertionError
_____________________________________ test_fail2 _____________________________________

    def test_fail2():
>       assert 0
E       assert 0

test_module.py:4: AssertionError
========================== 2 failed in 0.02 seconds ==========================
```

you will have a "failures" file which contains the failing test ids:

```
$ cat failures
test_module.py::test_fail1 (/tmp/pytest-326/test_fail10)
test_module.py::test_fail2
```

7.2.11 Making test result information available in fixtures

If you want to make test result reports available in fixture finalizers here is a little example implemented via a local plugin:

```
# content of conftest.py

import pytest
```

```
@pytest.mark.tryfirst
def pytest_runtest_makereport(item, call, __multicall__):
    # execute all other hooks to obtain the report object
    rep = __multicall__.execute()

    # set an report attribute for each phase of a call, which can
    # be "setup", "call", "teardown"

    setattr(item, "rep_" + rep.when, rep)
    return rep


@pytest.fixture
def something(request):
    def fin():
        # request.node is an "item" because we use the default
        # "function" scope
        if request.node.rep_setup.failed:
            print "setting up a test failed!", request.node.nodeid
        elif request.node.rep_setup.passed:
            if request.node.rep_call.failed:
                print "executing test failed", request.node.nodeid
    request.addfinalizer(fin)
```

if you then have failing tests:

```
# content of test_module.py

import pytest

@pytest.fixture
def other():
    assert 0

def test_setup_fails(something, other):
    pass

def test_call_fails(something):
    assert 0

def test_fail2():
    assert 0
```

and run it:

```
$ py.test -s test_module.py
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 3 items

test_module.py EFF

================================== ERRORS ===================================
_____________________ ERROR at setup of test_setup_fails ____________________

    @pytest.fixture
    def other():
>       assert 0
E       assert 0
```

```
test_module.py:6: AssertionError
================================ FAILURES ================================
_____________________________ test_call_fails _____________________________

something = None

    def test_call_fails(something):
>       assert 0
E       assert 0

test_module.py:12: AssertionError
_____________________________ test_fail2 ______________________________

    def test_fail2():
>       assert 0
E       assert 0

test_module.py:15: AssertionError
==================== 2 failed, 1 error in 0.01 seconds ====================
setting up a test failed! test_module.py::test_setup_fails
executing test failed test_module.py::test_call_fails
```

You'll see that the fixture finalizers could use the precise reporting information.

7.3 Parametrizing tests

py.test allows to easily parametrize test functions. For basic docs, see *Parametrizing fixtures and test functions*.

In the following we provide some examples using the builtin mechanisms.

7.3.1 Generating parameters combinations, depending on command line

Let's say we want to execute a test with different computation parameters and the parameter range shall be determined by a command line argument. Let's first write a simple (do-nothing) computation test:

```
# content of test_compute.py

def test_compute(param1):
    assert param1 < 4
```

Now we add a test configuration like this:

```
# content of conftest.py

def pytest_addoption(parser):
    parser.addoption("--all", action="store_true",
        help="run all combinations")

def pytest_generate_tests(metafunc):
    if 'param1' in metafunc.fixturenames:
        if metafunc.config.option.all:
            end = 5
        else:
            end = 2
        metafunc.parametrize("param1", range(end))
```

This means that we only run 2 tests if we do not pass `--all`:

```
$ py.test -q test_compute.py
..
```

We run only two computations, so we see two dots. let's run the full monty:

```
$ py.test -q --all
....F
================================= FAILURES =================================
_____________________________ test_compute[4] ______________________________

param1 = 4

    def test_compute(param1):
>       assert param1 < 4
E       assert 4 < 4

test_compute.py:3: AssertionError
```

As expected when running the full range of `param1` values we'll get an error on the last one.

7.3.2 A quick port of "testscenarios"

Here is a quick port to run tests configured with test scenarios, an add-on from Robert Collins for the standard unittest framework. We only have to work a bit to construct the correct arguments for pytest's `Metafunc.parametrize()`:

```
# content of test_scenarios.py

def pytest_generate_tests(metafunc):
    idlist = []
    argvalues = []
    for scenario in metafunc.cls.scenarios:
        idlist.append(scenario[0])
        items = scenario[1].items()
        argnames = [x[0] for x in items]
        argvalues.append(([x[1] for x in items]))
    metafunc.parametrize(argnames, argvalues, ids=idlist, scope="class")

scenario1 = ('basic', {'attribute': 'value'})
scenario2 = ('advanced', {'attribute': 'value2'})

class TestSampleWithScenarios:
    scenarios = [scenario1, scenario2]

    def test_demo1(self, attribute):
        assert isinstance(attribute, str)

    def test_demo2(self, attribute):
        assert isinstance(attribute, str)
```

this is a fully self-contained example which you can run with:

```
$ py.test test_scenarios.py
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 4 items
```

```
test_scenarios.py ....

======================== 4 passed in 0.01 seconds =========================
```

If you just collect tests you'll also nicely see 'advanced' and 'basic' as variants for the test function:

```
$ py.test --collectonly test_scenarios.py
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 4 items
<Module 'test_scenarios.py'>
  <Class 'TestSampleWithScenarios'>
    <Instance '()'>
      <Function 'test_demo1[basic]'>
      <Function 'test_demo2[basic]'>
      <Function 'test_demo1[advanced]'>
      <Function 'test_demo2[advanced]'>

============================  in 0.01 seconds =============================
```

Note that we told `metafunc.parametrize()` that your scenario values should be considered class-scoped. With pytest-2.3 this leads to a resource-based ordering.

7.3.3 Deferring the setup of parametrized resources

The parametrization of test functions happens at collection time. It is a good idea to setup expensive resources like DB connections or subprocess only when the actual test is run. Here is a simple example how you can achieve that, first the actual test requiring a `db` object:

```
# content of test_backends.py

import pytest
def test_db_initialized(db):
    # a dummy test
    if db.__class__.__name__ == "DB2":
        pytest.fail("deliberately failing for demo purposes")
```

We can now add a test configuration that generates two invocations of the `test_db_initialized` function and also implements a factory that creates a database object for the actual test invocations:

```
# content of conftest.py
import pytest

def pytest_generate_tests(metafunc):
    if 'db' in metafunc.fixturenames:
        metafunc.parametrize("db", ['d1', 'd2'], indirect=True)

class DB1:
    "one database object"
class DB2:
    "alternative database object"

@pytest.fixture
def db(request):
    if request.param == "d1":
        return DB1()
    elif request.param == "d2":
```

```
        return DB2()
    else:
        raise ValueError("invalid internal test config")
```

Let's first see how it looks like at collection time:

```
$ py.test test_backends.py --collectonly
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items
<Module 'test_backends.py'>
  <Function 'test_db_initialized[d1]'>
  <Function 'test_db_initialized[d2]'>

============================  in 0.00 seconds ============================
```

And then when we run the test:

```
$ py.test -q test_backends.py
.F
================================= FAILURES =================================
_______________________________ test_db_initialized[d2] ________________________________

db = <conftest.DB2 instance at 0x2038f80>

    def test_db_initialized(db):
        # a dummy test
        if db.__class__.__name__ == "DB2":
>           pytest.fail("deliberately failing for demo purposes")
E           Failed: deliberately failing for demo purposes

test_backends.py:6: Failed
```

The first invocation with `db == "DB1"` passed while the second with `db == "DB2"` failed. Our `db` fixture function has instantiated each of the DB values during the setup phase while the `pytest_generate_tests` generated two according calls to the `test_db_initialized` during the collection phase.

7.3.4 Parametrizing test methods through per-class configuration

Here is an example `pytest_generate_function` function implementing a parametrization scheme similar to Michael Foord's unittest parameterizer but in a lot less code:

```
# content of ./test_parametrize.py
import pytest

def pytest_generate_tests(metafunc):
    # called once per each test function
    funcarglist = metafunc.cls.params[metafunc.function.__name__]
    argnames = list(funcarglist[0])
    metafunc.parametrize(argnames, [[funcargs[name] for name in argnames]
            for funcargs in funcarglist])

class TestClass:
    # a map specifying multiple argument sets for a test method
    params = {
        'test_equals': [dict(a=1, b=2), dict(a=3, b=3), ],
        'test_zerodivision': [dict(a=1, b=0), ],
    }
```

```
    def test_equals(self, a, b):
        assert a == b

    def test_zerodivision(self, a, b):
        pytest.raises(ZeroDivisionError, "a/b")
```

Our test generator looks up a class-level definition which specifies which argument sets to use for each test function. Let's run it:

```
$ py.test -q
F..
================================ FAILURES ================================
____________________________ TestClass.test_equals[1-2] _____________________________

self = <test_parametrize.TestClass instance at 0x1338f80>, a = 1, b = 2

    def test_equals(self, a, b):
>       assert a == b
E       assert 1 == 2

test_parametrize.py:18: AssertionError
```

7.3.5 Indirect parametrization with multiple fixtures

Here is a stripped down real-life example of using parametrized testing for testing serialization of objects between different python interpreters. We define a `test_basic_objects` function which is to be run with different sets of arguments for its three arguments:

- `python1`: first python interpreter, run to pickle-dump an object to a file
- `python2`: second interpreter, run to pickle-load an object from a file
- `obj`: object to be dumped/loaded

```
"""
module containing a parametrized tests testing cross-python
serialization via the pickle module.
"""
import py, pytest

pythonlist = ['python2.4', 'python2.5', 'python2.6', 'python2.7', 'python2.8']
@pytest.fixture(params=pythonlist)
def python1(request, tmpdir):
    picklefile = tmpdir.join("data.pickle")
    return Python(request.param, picklefile)

@pytest.fixture(params=pythonlist)
def python2(request, python1):
    return Python(request.param, python1.picklefile)

class Python:
    def __init__(self, version, picklefile):
        self.pythonpath = py.path.local.sysfind(version)
        if not self.pythonpath:
            py.test.skip("%r not found" %(version,))
        self.picklefile = picklefile
    def dumps(self, obj):
```

```
        dumpfile = self.picklefile.dirpath("dump.py")
        dumpfile.write(py.code.Source("""
            import pickle
            f = open(%r, 'wb')
            s = pickle.dump(%r, f)
            f.close()
        """ % (str(self.picklefile), obj)))
        py.process.cmdexec("%s %s" %(self.pythonpath, dumpfile))

    def load_and_is_true(self, expression):
        loadfile = self.picklefile.dirpath("load.py")
        loadfile.write(py.code.Source("""
            import pickle
            f = open(%r, 'rb')
            obj = pickle.load(f)
            f.close()
            res = eval(%r)
            if not res:
                raise SystemExit(1)
        """ % (str(self.picklefile), expression)))
        print (loadfile)
        py.process.cmdexec("%s %s" %(self.pythonpath, loadfile))

@pytest.mark.parametrize("obj", [42, {}, {1:3},])
def test_basic_objects(python1, python2, obj):
    python1.dumps(obj)
    python2.load_and_is_true("obj == %s" % obj)
```

Running it results in some skips if we don't have all the python interpreters installed and otherwise runs all combinations (5 interpreters times 5 interpreters times 3 objects to serialize/deserialize):

```
. $ py.test -rs -q multipython.py
............sss............sss............sss............ssssssssssssssssss
========================== short test summary info ==========================
SKIP [27] /home/hpk/p/pytest/doc/en/example/multipython.py:21: 'python2.8' not found
```

7.3.6 Indirect parametrization of optional implementations/imports

If you want to compare the outcomes of several implementations of a given API, you can write test functions that receive the already imported implementations and get skipped in case the implementation is not importable/available. Let's say we have a "base" implementation and the other (possibly optimized ones) need to provide similar results:

```
# content of conftest.py

import pytest

@pytest.fixture(scope="session")
def basemod(request):
    return pytest.importorskip("base")

@pytest.fixture(scope="session", params=["opt1", "opt2"])
def optmod(request):
    return pytest.importorskip(request.param)
```

And then a base implementation of a simple function:

```
# content of base.py
def func1():
    return 1
```

And an optimized version:

```
# content of opt1.py
def func1():
    return 1.0001
```

And finally a little test module:

```
# content of test_module.py

def test_func1(basemod, optmod):
    assert round(basemod.func1(), 3) == round(optmod.func1(), 3)
```

If you run this with reporting for skips enabled:

```
$ py.test -rs test_module.py
============================= test session starts ==============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_module.py .s
========================= short test summary info ==========================
SKIP [1] /tmp/doc-exec-275/conftest.py:10: could not import 'opt2'

==================== 1 passed, 1 skipped in 0.01 seconds ====================
```

You'll see that we don't have a `opt2` module and thus the second test run of our `test_func1` was skipped. A few notes:

- the fixture functions in the `conftest.py` file are "session-scoped" because we don't need to import more than once
- if you have multiple test functions and a skipped import, you will see the `[1]` count increasing in the report
- you can put *@pytest.mark.parametrize* style parametrization on the test functions to parametrize input/output values as well.

7.4 Working with custom markers

Here are some example using the *Marking test functions with attributes* mechanism.

7.4.1 Marking test functions and selecting them for a run

You can "mark" a test function with custom metadata like this:

```
# content of test_server.py

import pytest
@pytest.mark.webtest
def test_send_http():
    pass # perform some webtest test for your app
def test_something_quick():
    pass
```

```
def test_another():
    pass
```

New in version 2.2. You can then restrict a test run to only run tests marked with `webtest`:

```
$ py.test -v -m webtest
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 3 items

test_server.py:3: test_send_http PASSED

==================== 2 tests deselected by "-m 'webtest'" ====================
================== 1 passed, 2 deselected in 0.01 seconds ==================
```

Or the inverse, running all tests except the webtest ones:

```
$ py.test -v -m "not webtest"
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 3 items

test_server.py:6: test_something_quick PASSED
test_server.py:8: test_another PASSED

================ 1 tests deselected by "-m 'not webtest'" ================
================== 2 passed, 1 deselected in 0.01 seconds ==================
```

7.4.2 Using `-k expr` to select tests based on their name

You can use the `-k` command line option to specify an expression which implements a substring match on the test names instead of the exact match on markers that `-m` provides. This makes it easy to select tests based on their names:

```
$ py.test -v -k http  # running with the above defined example module
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 3 items

test_server.py:3: test_send_http PASSED

====================== 2 tests deselected by '-khttp' ======================
================== 1 passed, 2 deselected in 0.01 seconds ==================
```

And you can also run all tests except the ones that match the keyword:

```
$ py.test -k "not send_http" -v
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 3 items

test_server.py:6: test_something_quick PASSED
test_server.py:8: test_another PASSED

================ 1 tests deselected by '-knot send_http' ================
================== 2 passed, 1 deselected in 0.01 seconds ==================
```

Or to select "http" and "quick" tests:

```
$ py.test -k "http or quick" -v
========================== test session starts ===========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 3 items

test_server.py:3: test_send_http PASSED
test_server.py:6: test_something_quick PASSED

================= 1 tests deselected by '-khttp or quick' =================
================== 2 passed, 1 deselected in 0.01 seconds ==================
```

7.4.3 Registering markers

New in version 2.2. Registering markers for your test suite is simple:

```
# content of pytest.ini
[pytest]
markers =
    webtest: mark a test as a webtest.
```

You can ask which markers exist for your test suite - the list includes our just defined `webtest` markers:

```
$ py.test --markers
@pytest.mark.webtest: mark a test as a webtest.

@pytest.mark.skipif(condition): skip the given test function if eval(condition) results in a True val

@pytest.mark.xfail(condition, reason=None, run=True): mark the the test function as an expected failu

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times passing in multipl

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all of the specified

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin machinery will try to

@pytest.mark.trylast: mark a hook implementation function such that the plugin machinery will try to
```

For an example on how to add and work with markers from a plugin, see *Custom marker and command line option to control test runs*.

Note: It is recommended to explicitely register markers so that:

- there is one place in your test suite defining your markers
- asking for existing markers via `py.test --markers` gives good output
- typos in function markers are treated as an error if you use the `--strict` option. Later versions of py.test are probably going to treat non-registered markers as an error.

7.4.4 Marking whole classes or modules

If you are programming with Python 2.6 or later you may use `pytest.mark` decorators with classes to apply markers to all of its test methods:

```
# content of test_mark_classlevel.py
import pytest
@pytest.mark.webtest
class TestClass:
    def test_startup(self):
        pass
    def test_startup_and_more(self):
        pass
```

This is equivalent to directly applying the decorator to the two test functions.

To remain backward-compatible with Python 2.4 you can also set a `pytestmark` attribute on a TestClass like this:

```
import pytest

class TestClass:
    pytestmark = pytest.mark.webtest
```

or if you need to use multiple markers you can use a list:

```
import pytest

class TestClass:
    pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]
```

You can also set a module level marker:

```
import pytest
pytestmark = pytest.mark.webtest
```

in which case it will be applied to all functions and methods defined in the module.

7.4.5 Custom marker and command line option to control test runs

Plugins can provide custom markers and implement specific behaviour based on it. This is a self-contained example which adds a command line option and a parametrized test function marker to run tests specifies via named environments:

```
# content of conftest.py

import pytest
def pytest_addoption(parser):
    parser.addoption("-E", action="store", metavar="NAME",
        help="only run tests matching the environment NAME.")

def pytest_configure(config):
    # register an additional marker
    config.addinivalue_line("markers",
        "env(name): mark test to run only on named environment")

def pytest_runtest_setup(item):
    envmarker = item.keywords.get("env", None)
    if envmarker is not None:
        envname = envmarker.args[0]
        if envname != item.config.getoption("-E"):
            pytest.skip("test requires env %r" % envname)
```

A test file using this local plugin:

```
# content of test_someenv.py

import pytest
@pytest.mark.env("stage1")
def test_basic_db_operation():
    pass
```

and an example invocations specifying a different environment than what the test needs:

```
$ py.test -E stage2
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

test_someenv.py s

======================== 1 skipped in 0.01 seconds =========================
```

and here is one that specifies exactly the environment needed:

```
$ py.test -E stage1
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items

test_someenv.py .

========================= 1 passed in 0.01 seconds =========================
```

The `--markers` option always gives you a list of available markers:

```
$ py.test --markers
@pytest.mark.env(name): mark test to run only on named environment

@pytest.mark.skipif(condition): skip the given test function if eval(condition) results in a True val

@pytest.mark.xfail(condition, reason=None, run=True): mark the the test function as an expected failu

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times passing in multipl

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all of the specifiec

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin machinery will try to

@pytest.mark.trylast: mark a hook implementation function such that the plugin machinery will try to
```

7.4.6 Reading markers which were set from multiple places

If you are heavily using markers in your test suite you may encounter the case where a marker is applied several times to a test function. From plugin code you can read over all such settings. Example:

```
# content of test_mark_three_times.py
import pytest
pytestmark = pytest.mark.glob("module", x=1)

@pytest.mark.glob("class", x=2)
class TestClass:
    @pytest.mark.glob("function", x=3)
```

```
    def test_something(self):
        pass
```

Here we have the marker "glob" applied three times to the same test function. From a conftest file we can read it like this:

```
# content of conftest.py
import sys

def pytest_runtest_setup(item):
    g = item.keywords.get("glob", None)
    if g is not None:
        for info in g:
            print ("glob args=%s kwargs=%s" %(info.args, info.kwargs))
            sys.stdout.flush()
```

Let's run this without capturing output and see what we get:

```
$ py.test -q -s
glob args=('function',) kwargs={'x': 3}
glob args=('class',) kwargs={'x': 2}
glob args=('module',) kwargs={'x': 1}
.
```

7.4.7 marking platform specific tests with pytest

Consider you have a test suite which marks tests for particular platforms, namely `pytest.mark.osx`, `pytest.mark.win32` etc. and you also have tests that run on all platforms and have no specific marker. If you now want to have a way to only run the tests for your particular platform, you could use the following plugin:

```
# content of conftest.py
#
import sys
import pytest

ALL = set("osx linux2 win32".split())

def pytest_runtest_setup(item):
    if isinstance(item, item.Function):
        plat = sys.platform
        if plat not in item.keywords:
            if ALL.intersection(item.keywords):
                pytest.skip("cannot run on platform %s" %(plat))
```

then tests will be skipped if they were specified for a different platform. Let's do a little test file to show how this looks like:

```
# content of test_plat.py

import pytest

@pytest.mark.osx
def test_if_apple_is_evil():
    pass

@pytest.mark.linux2
def test_if_linux_works():
```

```
    pass

@pytest.mark.win32
def test_if_win32_crashes():
    pass

def test_runs_everywhere():
    pass
```

then you will see two test skipped and two executed tests as expected:

```
$ py.test -rs # this option reports skip reasons
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 4 items

test_plat.py s.s.
========================= short test summary info ==========================
SKIP [2] /tmp/doc-exec-273/conftest.py:12: cannot run on platform linux2

=================== 2 passed, 2 skipped in 0.01 seconds ====================
```

Note that if you specify a platform via the marker-command line option like this:

```
$ py.test -m linux2
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 4 items

test_plat.py .

==================== 3 tests deselected by "-m 'linux2'" ====================
================== 1 passed, 3 deselected in 0.01 seconds ==================
```

then the unmarked-tests will not be run. It is thus a way to restrict the run to the specific tests.

7.4.8 Automatically adding markers based on test names

If you a test suite where test function names indicate a certain type of test, you can implement a hook that automatically defines markers so that you can use the `-m` option with it. Let's look at this test module:

```
# content of test_module.py

def test_interface_simple():
    assert 0

def test_interface_complex():
    assert 0

def test_event_simple():
    assert 0

def test_something_else():
    assert 0
```

We want to dynamically define two markers and can do it in a `conftest.py` plugin:

```
# content of conftest.py

import pytest
def pytest_collection_modifyitems(items):
    for item in items:
        if "interface" in item.nodeid:
            item.keywords["interface"] = pytest.mark.interface
        elif "event" in item.nodeid:
            item.keywords["event"] = pytest.mark.event
```

We can now use the `-m option` to select one set:

```
$ py.test -m interface --tb=short
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 4 items

test_module.py FF

================================ FAILURES ================================
_____________________________ test_interface_simple ______________________________
test_module.py:3: in test_interface_simple
>       assert 0
E       assert 0
_____________________________ test_interface_complex _____________________________
test_module.py:6: in test_interface_complex
>       assert 0
E       assert 0
================== 2 tests deselected by "-m 'interface'" ==================
================== 2 failed, 2 deselected in 0.01 seconds ==================
```

or to select both "event" and "interface" tests:

```
$ py.test -m "interface or event" --tb=short
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 4 items

test_module.py FFF

================================ FAILURES ================================
_____________________________ test_interface_simple ______________________________
test_module.py:3: in test_interface_simple
>       assert 0
E       assert 0
_____________________________ test_interface_complex _____________________________
test_module.py:6: in test_interface_complex
>       assert 0
E       assert 0
_____________________________ test_event_simple _____________________________
test_module.py:9: in test_event_simple
>       assert 0
E       assert 0
============= 1 tests deselected by "-m 'interface or event'" =============
================== 3 failed, 1 deselected in 0.02 seconds ==================
```

7.5 A sesssion-fixture which can look at all collected tests

A session-scoped fixture effectively has access to all collected test items. Here is an example of a fixture function which walks all collected tests and looks if their test class defines a `callme` method and calls it:

```
# content of conftest.py

import pytest

@pytest.fixture(scope="session", autouse=True)
def callattr_ahead_of_alltests(request):
    print "callattr_ahead_of_alltests called"
    seen = set([None])
    session = request.node
    for item in session.items:
        cls = item.getparent(pytest.Class)
        if cls not in seen:
            if hasattr(cls.obj, "callme"):
               cls.obj.callme()
            seen.add(cls)
```

test classes may now define a `callme` method which will be called ahead of running any tests:

```
# content of test_module.py

class TestHello:
    @classmethod
    def callme(cls):
        print "callme called!"

    def test_method1(self):
        print "test_method1 called"

    def test_method2(self):
        print "test_method1 called"

class TestOther:
    @classmethod
    def callme(cls):
        print "callme other called"
    def test_other(self):
        print "test other"

# works with unittest as well ...
import unittest

class SomeTest(unittest.TestCase):
    @classmethod
    def callme(self):
        print "SomeTest callme called"

    def test_unit1(self):
        print "test_unit1 method called"
```

If you run this without output capturing:

```
$ py.test -q -s test_module.py
....
callattr_ahead_of_alltests called
```

```
callme called!
callme other called
SomeTest callme called
test_method1 called
test_method1 called
test other
test_unit1 method called
```

7.6 Changing standard (Python) test discovery

7.6.1 Changing directory recursion

You can set the `norecursedirs` option in an ini-file, for example your `setup.cfg` in the project root directory:

```
# content of setup.cfg
[pytest]
norecursedirs = .svn _build tmp*
```

This would tell py.test to not recurse into typical subversion or sphinx-build directories or into any `tmp` prefixed directory.

7.6.2 Changing naming conventions

You can configure different naming conventions by setting the `python_files`, `python_classes` and `python_functions` configuration options. Example:

```
# content of setup.cfg
# can also be defined in in tox.ini or pytest.ini file
[pytest]
python_files=check_*.py
python_classes=Check
python_functions=check
```

This would make py.test look for `check_` prefixes in Python filenames, `Check` prefixes in classes and `check` prefixes in functions and classes. For example, if we have:

```
# content of check_myapp.py
class CheckMyApp:
    def check_simple(self):
        pass
    def check_complex(self):
        pass
```

then the test collection looks like this:

```
$ py.test --collectonly
========================== test session starts ==========================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items
<Module 'check_myapp.py'>
  <Class 'CheckMyApp'>
    <Instance '()'>
      <Function 'check_simple'>
      <Function 'check_complex'>
```

```
============================  in 0.01 seconds ============================
```

7.6.3 Interpreting cmdline arguments as Python packages

You can use the `--pyargs` option to make py.test try interpreting arguments as python package names, deriving their file system path and then running the test. For example if you have unittest2 installed you can type:

```
py.test --pyargs unittest2.test.test_skipping -q
```

which would run the respective test module. Like with other options, through an ini-file and the `addopts` option you can make this change more permanently:

```
# content of pytest.ini
[pytest]
addopts = --pyargs
```

Now a simple invocation of `py.test NAME` will check if NAME exists as an importable package/module and otherwise treat it as a filesystem path.

7.6.4 Finding out what is collected

You can always peek at the collection tree without running tests like this:

```
. $ py.test --collectonly pythoncollection.py
=========================== test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 3 items
<Module 'pythoncollection.py'>
  <Function 'test_function'>
  <Class 'TestClass'>
    <Instance '()'>
      <Function 'test_method'>
      <Function 'test_anothermethod'>

============================  in 0.01 seconds ============================
```

7.6.5 customizing test collection to find all .py files

You can easily instruct py.test to discover tests from every python file:

```
# content of pytest.ini
[pytest]
python_files = *.py
```

However, many projects will have a `setup.py` which they don't want to be imported. Moreover, there may files only importable by a specific python version. For such cases you can dynamically define files to be ignored by listing them in a `conftest.py` file:

```
# content of conftest.py
import sys

collect_ignore = ["setup.py"]
if sys.version_info[0] > 2:
    collect_ignore.append("pkg/module_py2.py")
```

And then if you have a module file like this:

```
# content of pkg/module_py2.py
def test_only_on_python2():
    try:
        assert 0
    except Exception, e:
        pass
```

and a setup.py dummy file like this:

```
# content of setup.py
0/0  # will raise exeption if imported
```

then a pytest run on python2 will find the one test when run with a python2 interpreters and will leave out the setup.py file:

```
$ py.test --collectonly
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 1 items
<Module 'pkg/module_py2.py'>
  <Function 'test_only_on_python2'>

============================  in 0.01 seconds =============================
```

If you run with a Python3 interpreter the moduled added through the conftest.py file will not be considered for test collection.

7.7 Working with non-python tests

7.7.1 A basic example for specifying tests in Yaml files

Here is an example `conftest.py` (extracted from Ali Afshnars special purpose pytest-yamlwsgi plugin). This `conftest.py` will collect `test*.yml` files and will execute the yaml-formatted content as custom tests:

```
# content of conftest.py

import pytest

def pytest_collect_file(parent, path):
    if path.ext == ".yml" and path.basename.startswith("test"):
        return YamlFile(path, parent)

class YamlFile(pytest.File):
    def collect(self):
        import yaml # we need a yaml parser, e.g. PyYAML
        raw = yaml.load(self.fspath.open())
        for name, spec in raw.items():
            yield YamlItem(name, self, spec)

class YamlItem(pytest.Item):
    def __init__(self, name, parent, spec):
        super(YamlItem, self).__init__(name, parent)
        self.spec = spec

    def runtest(self):
```

```
        for name, value in self.spec.items():
            # some custom test execution (dumb example follows)
            if name != value:
                raise YamlException(self, name, value)

    def repr_failure(self, excinfo):
        """ called when self.runtest() raises an exception. """
        if isinstance(excinfo.value, YamlException):
            return "\n".join([
                "usecase execution failed",
                "   spec failed: %r: %r" % excinfo.value.args[1:3],
                "   no further details known at this point."
            ])

    def reportinfo(self):
        return self.fspath, 0, "usecase: %s" % self.name

class YamlException(Exception):
    """ custom exception for error reporting. """
```

You can create a simple example file:

```
# test_simple.yml
ok:
    sub1: sub1

hello:
    world: world
    some: other
```

and if you installed PyYAML or a compatible YAML-parser you can now execute the test specification:

```
nonpython $ py.test test_simple.yml
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items

test_simple.yml .F

================================= FAILURES =================================
______________________________ usecase: hello ______________________________
usecase execution failed
   spec failed: 'some': 'other'
   no further details known at this point.
==================== 1 failed, 1 passed in 0.05 seconds ====================
```

You get one dot for the passing `sub1: sub1` check and one failure. Obviously in the above `conftest.py` you'll want to implement a more interesting interpretation of the yaml-values. You can easily write your own domain specific testing language this way.

Note: `repr_failure(excinfo)` is called for representing test failures. If you create custom collection nodes you can return an error representation string of your choice. It will be reported as a (red) string.

`reportinfo()` is used for representing the test location and is also consulted when reporting in `verbose` mode:

```
nonpython $ py.test -v
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5 -- /home/hpk/p/pytest/.tox/regen/bin/python
```

```
collecting ... collected 2 items

test_simple.yml:1: usecase: ok PASSED
test_simple.yml:1: usecase: hello FAILED

================================= FAILURES =================================
_______________________________ usecase: hello ________________________________
usecase execution failed
   spec failed: 'some': 'other'
   no further details known at this point.
==================== 1 failed, 1 passed in 0.05 seconds ====================
```

While developing your custom test collection and execution it's also interesting to just look at the collection tree:

```
nonpython $ py.test --collectonly
============================ test session starts ============================
platform linux2 -- Python 2.7.3 -- pytest-2.3.5
collected 2 items
<YamlFile 'test_simple.yml'>
  <YamlItem 'ok'>
  <YamlItem 'hello'>

============================  in 0.05 seconds =============================
```

CHAPTER

EIGHT

TALKS AND TUTORIALS

Note: Upcoming: professional testing with pytest and tox , 24th-26th June 2013, Leipzig.

8.1 Tutorial examples and blog postings

Basic usage and funcargs:

- pytest introduction from Brian Okken (January 2013)
- pycon australia 2012 pytest talk from Brianna Laugher (video, slides, code)
- pycon 2012 US talk video from Holger Krekel
- pycon 2010 tutorial PDF and tutorial1 repository

Fixtures and Function arguments:

- *pytest fixtures: explicit, modular, scalable*
- monkey patching done right (blog post, consult monkeypatch plugin for up-to-date API)

Test parametrization:

- generating parametrized tests with funcargs (uses deprecated `addcall()` API.
- test generators and cached setup
- parametrizing tests, generalized (blog post)
- putting test-hooks into local or global plugins (blog post)

Assertion introspection:

- (07/2011) Behind the scenes of py.test's new assertion rewriting

Distributed testing:

- simultaneously test your code on all platforms (blog entry)

Plugin specific examples:

- skipping slow tests by default in py.test (blog entry)
- many examples in the docs for plugins

8.2 Older conference talks and tutorials

- ep2009-rapidtesting.pdf tutorial slides (July 2009):
  - testing terminology
  - basic py.test usage, file system layout
  - test function arguments (funcargs) and test fixtures
  - existing plugins
  - distributed testing
- ep2009-pytest.pdf 60 minute py.test talk, highlighting unique features and a roadmap (July 2009)
- pycon2009-pytest-introduction.zip slides and files, extended version of py.test basic introduction, discusses more options, also introduces old-style xUnit setup, looponfailing and other features.
- pycon2009-pytest-advanced.pdf contain a slightly older version of funcargs and distributed testing, compared to the EuroPython 2009 slides.

CHAPTER

NINE

FEEDBACK AND CONTRIBUTE TO PY.TEST

9.1 Contact channels

- pytest issue tracker to report bugs or suggest features (for version 2.0 and above).
- pytest on stackoverflow.com to post questions with the tag `pytest`. New Questions will usually be seen by pytest users or developers and answered quickly.
- Testing In Python: a mailing list for Python testing tools and discussion.
- pytest-dev at python.org (mailing list) pytest specific announcements and discussions.
- pytest-commit at python.org (mailing list): for commits and new issues
- #pylib on irc.freenode.net IRC channel for random questions.
- private mail to Holger.Krekel at gmail com if you want to communicate sensitive issues
- merlinux.eu offers pytest and tox-related professional teaching and consulting.

9.2 Working from version control or a tarball

To follow development or start experiments, checkout the complete code and documentation source with mercurial:

```
hg clone https://bitbucket.org/hpk42/pytest/
```

You can also go to the python package index and download and unpack a TAR file:

```
http://pypi.python.org/pypi/pytest/
```

9.2.1 Activating a checkout with setuptools

With a working Distribute or setuptools installation you can type:

```
python setup.py develop
```

in order to work inline with the tools and the lib of your checkout.

If this command complains that it could not find the required version of "py" then you need to use the development pypi repository:

```
python setup.py develop -i http://pypi.testrun.org
```

CHAPTER

TEN

PYTEST-2.3: REASONING FOR FIXTURE/FUNCARG EVOLUTION

Target audience: Reading this document requires basic knowledge of python testing, xUnit setup methods and the (previous) basic pytest funcarg mechanism, see http://pytest.org/2.2.4/funcargs.html If you are new to pytest, then you can simply ignore this section and read the other sections.

10.1 Shortcomings of the previous `pytest_funcarg__` mechanism

The pre pytest-2.3 funcarg mechanism calls a factory each time a funcarg for a test function is required. If a factory wants to re-use a resource across different scopes, it often used the `request.cached_setup()` helper to manage caching of resources. Here is a basic example how we could implement a per-session Database object:

```
# content of conftest.py
class Database:
    def __init__(self):
        print ("database instance created")
    def destroy(self):
        print ("database instance destroyed")

def pytest_funcarg__db(request):
    return request.cached_setup(setup=DataBase,
                                teardown=lambda db: db.destroy,
                                scope="session")
```

There are several limitations and difficulties with this approach:

1. Scoping funcarg resource creation is not straight forward, instead one must understand the intricate cached_setup() method mechanics.
2. parametrizing the "db" resource is not straight forward: you need to apply a "parametrize" decorator or implement a `pytest_generate_tests()` hook calling `parametrize()` which performs parametrization at the places where the resource is used. Moreover, you need to modify the factory to use an `extrakey` parameter containing `request.param` to the `cached_setup()` call.
3. Multiple parametrized session-scoped resources will be active at the same time, making it hard for them to affect global state of the application under test.
4. there is no way how you can make use of funcarg factories in xUnit setup methods.
5. A non-parametrized fixture function cannot use a parametrized funcarg resource if it isn't stated in the test function signature.

All of these limitations are addressed with pytest-2.3 and its improved *fixture mechanism*.

10.2 Direct scoping of fixture/funcarg factories

Instead of calling cached_setup() with a cache scope, you can use the *@pytest.fixture* decorator and directly state the scope:

```
@pytest.fixture(scope="session")
def db(request):
    # factory will only be invoked once per session -
    db = DataBase()
    request.addfinalizer(db.destroy)  # destroy when session is finished
    return db
```

This factory implementation does not need to call `cached_setup()` anymore because it will only be invoked once per session. Moreover, the `request.addfinalizer()` registers a finalizer according to the specified resource scope on which the factory function is operating.

10.3 Direct parametrization of funcarg resource factories

Previously, funcarg factories could not directly cause parametrization. You needed to specify a `@parametrize` decorator on your test function or implement a `pytest_generate_tests` hook to perform parametrization, i.e. calling a test multiple times with different value sets. pytest-2.3 introduces a decorator for use on the factory itself:

```
@pytest.fixture(params=["mysql", "pg"])
def db(request):
    ... # use request.param
```

Here the factory will be invoked twice (with the respective "mysql" and "pg" values set as `request.param` attributes) and and all of the tests requiring "db" will run twice as well. The "mysql" and "pg" values will also be used for reporting the test-invocation variants.

This new way of parametrizing funcarg factories should in many cases allow to re-use already written factories because effectively `request.param` was already used when test functions/classes were parametrized via `parametrize(indirect=True)()` calls.

Of course it's perfectly fine to combine parametrization and scoping:

```
@pytest.fixture(scope="session", params=["mysql", "pg"])
def db(request):
    if request.param == "mysql":
        db = MySQL()
    elif request.param == "pg":
        db = PG()
    request.addfinalizer(db.destroy)  # destroy when session is finished
    return db
```

This would execute all tests requiring the per-session "db" resource twice, receiving the values created by the two respective invocations to the factory function.

10.4 No pytest_funcarg__ prefix when using @fixture decorator

When using the `@fixture` decorator the name of the function denotes the name under which the resource can be accessed as a function argument:

```
@pytest.fixture()
def db(request):
    ...
```

The name under which the funcarg resource can be requested is `db`.

You can still use the "old" non-decorator way of specifying funcarg factories aka:

```
def pytest_funcarg__db(request):
    ...
```

But it is then not possible to define scoping and parametrization. It is thus recommended to use the factory decorator.

10.5 solving per-session setup / autouse fixtures

pytest for a long time offered a pytest_configure and a pytest_sessionstart hook which are often used to setup global resources. This suffers from several problems:

1. in distributed testing the master process would setup test resources that are never needed because it only co-ordinates the test run activities of the slave processes.
2. if you only perform a collection (with "–collectonly") resource-setup will still be executed.
3. If a pytest_sessionstart is contained in some subdirectories conftest.py file, it will not be called. This stems from the fact that this hook is actually used for reporting, in particular the test-header with platform/custom information.

Moreover, it was not easy to define a scoped setup from plugins or conftest files other than to implement a `pytest_runtest_setup()` hook and caring for scoping/caching yourself. And it's virtually impossible to do this with parametrization as `pytest_runtest_setup()` is called during test execution and parametrization happens at collection time.

It follows that pytest_configure/session/runtest_setup are often not appropriate for implementing common fixture needs. Therefore, pytest-2.3 introduces *autouse fixtures (xUnit setup on steroids)* which fully integrate with the generic *fixture mechanism* and obsolete many prior uses of pytest hooks.

10.6 funcargs/fixture discovery now happens at collection time

pytest-2.3 takes care to discover fixture/funcarg factories at collection time. This is more efficient especially for large test suites. Moreover, a call to "py.test –collectonly" should be able to in the future show a lot of setup-information and thus presents a nice method to get an overview of fixture management in your project.

10.7 Conclusion and compatibility notes

funcargs were originally introduced to pytest-2.0. In pytest-2.3 the mechanism was extended and refined and is now described as fixtures:

- previously funcarg factories were specified with a special `pytest_funcarg__NAME` prefix instead of using the `@pytest.fixture` decorator.

- Factories received a `request` object which managed caching through `request.cached_setup()` calls and allowed using other funcargs via `request.getfuncargvalue()` calls. These intricate APIs made it hard to do proper parametrization and implement resource caching. The new `pytest.fixture'()` decorator allows to declare the scope and let pytest figure things out for you.
- if you used parametrization and funcarg factories which made use of `request.cached_setup()` it is recommeneded to invest a few minutes and simplify your fixture function code to use the *Fixtures as Function arguments (funcargs)* decorator instead. This will also allow to take advantage of the automatic per-resource grouping of tests.

CHAPTER

ELEVEN

RELEASE ANNOUNCEMENTS

11.1 pytest-2.3.4: stabilization, more flexible selection via "-k expr"

pytest-2.3.4 is a small stabilization release of the py.test tool which offers uebersimple assertions, scalable fixture mechanisms and deep customization for testing with Python. This release comes with the following fixes and features:

- make "-k" option accept an expressions the same as with "-m" so that one can write: -k "name1 or name2" etc. This is a slight usage incompatibility if you used special syntax like "TestClass.test_method" which you now need to write as -k "TestClass and test_method" to match a certain method in a certain test class.
- allow to dynamically define markers via item.keywords[...]=assignment integrating with "-m" option
- yielded test functions will now have autouse-fixtures active but cannot accept fixtures as funcargs - it's anyway recommended to rather use the post-2.0 parametrize features instead of yield, see: http://pytest.org/latest/example/parametrize.html
- fix autouse-issue where autouse-fixtures would not be discovered if defined in a a/conftest.py file and tests in a/tests/test_some.py
- fix issue226 - LIFO ordering for fixture teardowns
- fix issue224 - invocations with >256 char arguments now work
- fix issue91 - add/discuss package/directory level setups in example
- fixes related to autouse discovery and calling

Thanks in particular to Thomas Waldmann for spotting and reporting issues.

See

    http://pytest.org/

for general information. To install or upgrade pytest:

    pip install -U pytest # or easy_install -U pytest

best, holger krekel

11.2 pytest-2.3.3: integration fixes, py24 suport, */** shown in traceback

pytest-2.3.3 is a another stabilization release of the py.test tool which offers uebersimple assertions, scalable fixture mechanisms and deep customization for testing with Python. Particularly, this release provides:

- integration fixes and improvements related to flask, numpy, nose, unittest, mock
- makes pytest work on py24 again (yes, people sometimes still need to use it)
- show `*,**` args in pytest tracebacks

Thanks to Manuel Jacob, Thomas Waldmann, Ronny Pfannschmidt, Pavel Repin and Andreas Taumoefolau for providing patches and all for the issues.

See

http://pytest.org/

for general information. To install or upgrade pytest:

pip install -U pytest # or easy_install -U pytest

best, holger krekel

11.2.1 Changes between 2.3.2 and 2.3.3

- fix issue214 - parse modules that contain special objects like e. g. flask's request object which blows up on getattr access if no request is active. thanks Thomas Waldmann.
- fix issue213 - allow to parametrize with values like numpy arrays that do not support an __eq__ operator
- fix issue215 - split test_python.org into multiple files
- fix issue148 - @unittest.skip on classes is now recognized and avoids calling setUpClass/tearDownClass, thanks Pavel Repin
- fix issue209 - reintroduce python2.4 support by depending on newer pylib which re-introduced statement-finding for pre-AST interpreters
- nose support: only call setup if its a callable, thanks Andrew Taumoefolau
- fix issue219 - add py2.4-3.3 classifiers to TROVE list
- in tracebacks *,** arg values are now shown next to normal arguments (thanks Manuel Jacob)
- fix issue217 - support mock.patch with pytest's fixtures - note that you need either mock-1.0.1 or the python3.3 builtin unittest.mock.
- fix issue127 - improve documentation for pytest_addoption() and add a `config.getoption(name)` helper function for consistency.

11.3 pytest-2.3.2: some fixes and more traceback-printing speed

pytest-2.3.2 is a another stabilization release:

- issue 205: fixes a regression with conftest detection
- issue 208/29: fixes traceback-printing speed in some bad cases
- fix teardown-ordering for parametrized setups
- fix unittest and trial compat behaviour with respect to runTest() methods
- issue 206 and others: some improvements to packaging
- fix issue127 and others: improve some docs

See

http://pytest.org/

for general information. To install or upgrade pytest:

pip install -U pytest # or easy_install -U pytest

best, holger krekel

11.3.1 Changes between 2.3.1 and 2.3.2

- fix issue208 and fix issue29 use new py version to avoid long pauses when printing tracebacks in long modules
- fix issue205 - conftests in subdirs customizing pytest_pycollect_makemodule and pytest_pycollect_makeitem now work properly
- fix teardown-ordering for parametrized setups
- fix issue127 - better documentation for pytest_addoption and related objects.
- fix unittest behaviour: TestCase.runtest only called if there are test methods defined
- improve trial support: don't collect its empty unittest.TestCase.runTest() method
- "python setup.py test" now works with pytest itself
- fix/improve internal/packaging related bits:
  - exception message check of test_nose.py now passes on python33 as well
  - issue206 - fix test_assertrewrite.py to work when a global PYTHONDONTWRITEBYTECODE=1 is present
  - add tox.ini to pytest distribution so that ignore-dirs and others config bits are properly distributed for maintainers who run pytest-own tests

11.4 pytest-2.3.1: fix regression with factory functions

pytest-2.3.1 is a quick follow-up release:

- fix issue202 - regression with fixture functions/funcarg factories: using "self" is now safe again and works as in 2.2.4. Thanks to Eduard Schettino for the quick bug report.
- disable pexpect pytest self tests on Freebsd - thanks Koob for the quick reporting
- fix/improve interactive docs with –markers

See

http://pytest.org/

for general information. To install or upgrade pytest:

pip install -U pytest # or easy_install -U pytest

best, holger krekel

11.4.1 Changes between 2.3.0 and 2.3.1

- fix issue202 - fix regression: using "self" from fixture functions now works as expected (it's the same "self" instance that a test method which uses the fixture sees)

- skip pexpect using tests (test_pdb.py mostly) on freebsd* systems due to pexpect not supporting it properly (hanging)
- link to web pages from –markers output which provides help for pytest.mark.* usage.

11.5 pytest-2.3: improved fixtures / better unittest integration

pytest-2.3 comes with many major improvements for fixture/funcarg management and parametrized testing in Python. It is now easier, more efficient and more predicatable to re-run the same tests with different fixture instances. Also, you can directly declare the caching "scope" of fixtures so that dependent tests throughout your whole test suite can re-use database or other expensive fixture objects with ease. Lastly, it's possible for fixture functions (formerly known as funcarg factories) to use other fixtures, allowing for a completely modular and re-useable fixture design.

For detailed info and tutorial-style examples, see:

http://pytest.org/latest/fixture.html

Moreover, there is now support for using pytest fixtures/funcargs with unittest-style suites, see here for examples:

http://pytest.org/latest/unittest.html

Besides, more unittest-test suites are now expected to "simply work" with pytest.

All changes are backward compatible and you should be able to continue to run your test suites and 3rd party plugins that worked with pytest-2.2.4.

If you are interested in the precise reasoning (including examples) of the pytest-2.3 fixture evolution, please consult http://pytest.org/latest/funcarg_compare.html

For general info on installation and getting started:

http://pytest.org/latest/getting-started.html

Docs and PDF access as usual at:

http://pytest.org

and more details for those already in the knowing of pytest can be found in the CHANGELOG below.

Particular thanks for this release go to Floris Bruynooghe, Alex Okrushko Carl Meyer, Ronny Pfannschmidt, Benjamin Peterson and Alex Gaynor for helping to get the new features right and well integrated. Ronny and Floris also helped to fix a number of bugs and yet more people helped by providing bug reports.

have fun, holger krekel

11.5.1 Changes between 2.2.4 and 2.3.0

- fix issue202 - better automatic names for parametrized test functions
- fix issue139 - introduce @pytest.fixture which allows direct scoping and parametrization of funcarg factories. Introduce new @pytest.setup marker to allow the writing of setup functions which accept funcargs.
- fix issue198 - conftest fixtures were not found on windows32 in some circumstances with nested directory structures due to path manipulation issues
- fix issue193 skip test functions with were parametrized with empty parameter sets
- fix python3.3 compat, mostly reporting bits that previously depended on dict ordering
- introduce re-ordering of tests by resource and parametrization setup which takes precedence to the usual file-ordering

- fix issue185 monkeypatching time.time does not cause pytest to fail
- fix issue172 duplicate call of pytest.setup-decoratored setup_module functions
- fix junitxml=path construction so that if tests change the current working directory and the path is a relative path it is constructed correctly from the original current working dir.
- fix "python setup.py test" example to cause a proper "errno" return
- fix issue165 - fix broken doc links and mention stackoverflow for FAQ
- catch unicode-issues when writing failure representations to terminal to prevent the whole session from crashing
- fix xfail/skip confusion: a skip-mark or an imperative pytest.skip will now take precedence before xfail-markers because we can't determine xfail/xpass status in case of a skip. see also: http://stackoverflow.com/questions/11105828/in-py-test-when-i-explicitly-skip-a-test-that-is-marked-as-xfail-how-can-i-get
- always report installed 3rd party plugins in the header of a test run
- fix issue160: a failing setup of an xfail-marked tests should be reported as xfail (not xpass)
- fix issue128: show captured output when capsys/capfd are used
- fix issue179: propperly show the dependency chain of factories
- pluginmanager.register(...) now raises ValueError if the plugin has been already registered or the name is taken
- fix issue159: improve http://pytest.org/latest/faq.html especially with respect to the "magic" history, also mention pytest-django, trial and unittest integration.
- make request.keywords and node.keywords writable. All descendant collection nodes will see keyword values. Keywords are dictionaries containing markers and other info.
- fix issue 178: xml binary escapes are now wrapped in py.xml.raw
- fix issue 176: correctly catch the builtin AssertionError even when we replaced AssertionError with a subclass on the python level
- factory discovery no longer fails with magic global callables that provide no sane __code__ object (mock.call for example)
- fix issue 182: testdir.inprocess_run now considers passed plugins
- **fix issue 188: ensure sys.exc_info is clear on python2** before calling into a test
- fix issue 191: add unittest TestCase runTest method support
- fix issue 156: monkeypatch correctly handles class level descriptors
- reporting refinements:
  - pytest_report_header now receives a "startdir" so that you can use startdir.bestrelpath(yourpath) to show nice relative path
  - allow plugins to implement both pytest_report_header and pytest_sessionstart (sessionstart is invoked first).
  - don't show deselected reason line if there is none
  - py.test -vv will show all of assert comparisations instead of truncating

11.6 pytest-2.2.4: bug fixes, better junitxml/unittest/python3 compat

pytest-2.2.4 is a minor backward-compatible release of the versatile py.test testing tool. It contains bug fixes and a few refinements to junitxml reporting, better unittest- and python3 compatibility.

For general information see here:

http://pytest.org/

To install or upgrade pytest:

pip install -U pytest # or easy_install -U pytest

Special thanks for helping on this release to Ronny Pfannschmidt and Benjamin Peterson and the contributors of issues.

best, holger krekel

11.6.1 Changes between 2.2.3 and 2.2.4

- fix error message for rewritten assertions involving the % operator
- fix issue 126: correctly match all invalid xml characters for junitxml binary escape
- fix issue with unittest: now @unittest.expectedFailure markers should be processed correctly (you can also use @pytest.mark markers)
- document integration with the extended distribute/setuptools test commands
- fix issue 140: propperly get the real functions of bound classmethods for setup/teardown_class
- fix issue #141: switch from the deceased paste.pocoo.org to bpaste.net
- fix issue #143: call unconfigure/sessionfinish always when configure/sessionstart where called
- fix issue #144: better mangle test ids to junitxml classnames
- upgrade distribute_setup.py to 0.6.27

11.7 pytest-2.2.2: bug fixes

pytest-2.2.2 (updated to 2.2.3 to fix packaging issues) is a minor backward-compatible release of the versatile py.test testing tool. It contains bug fixes and a few refinements particularly to reporting with "–collectonly", see below for betails.

For general information see here:

http://pytest.org/

To install or upgrade pytest:

pip install -U pytest # or easy_install -U pytest

Special thanks for helping on this release to Ronny Pfannschmidt and Ralf Schmitt and the contributors of issues.

best, holger krekel

11.7.1 Changes between 2.2.1 and 2.2.2

- fix issue101: wrong args to unittest.TestCase test function now produce better output
- fix issue102: report more useful errors and hints for when a test directory was renamed and some pyc/__pycache__ remain
- fix issue106: allow parametrize to be applied multiple times e.g. from module, class and at function level.
- fix issue107: actually perform session scope finalization
- don't check in parametrize if indirect parameters are funcarg names
- add chdir method to monkeypatch funcarg
- fix crash resulting from calling monkeypatch undo a second time
- fix issue115: make –collectonly robust against early failure (missing files/directories)
- "-qq –collectonly" now shows only files and the number of tests in them
- "-q –collectonly" now shows test ids
- allow adding of attributes to test reports such that it also works with distributed testing (no upgrade of pytest-xdist needed)

11.8 pytest-2.2.1: bug fixes, perfect teardowns

pytest-2.2.1 is a minor backward-compatible release of the the py.test testing tool. It contains bug fixes and little improvements, including documentation fixes. If you are using the distributed testing pluginmake sure to upgrade it to pytest-xdist-1.8.

For general information see here:

http://pytest.org/

To install or upgrade pytest:

pip install -U pytest # or easy_install -U pytest

Special thanks for helping on this release to Ronny Pfannschmidt, Jurko Gospodnetic and Ralf Schmitt.

best, holger krekel

11.8.1 Changes between 2.2.0 and 2.2.1

- fix issue99 (in pytest and py) internallerrors with resultlog now produce better output - fixed by normalizing pytest_internalerror input arguments.
- fix issue97 / traceback issues (in pytest and py) improve traceback output in conjunction with jinja2 and cython which hack tracebacks
- fix issue93 (in pytest and pytest-xdist) avoid "delayed teardowns": the final test in a test node will now run its teardown directly instead of waiting for the end of the session. Thanks Dave Hunt for the good reporting and feedback. The pytest_runtest_protocol as well as the pytest_runtest_teardown hooks now have "nextitem" available which will be None indicating the end of the test run.
- fix collection crash due to unknown-source collected items, thanks to Ralf Schmitt (fixed by depending on a more recent pylib)

11.9 py.test 2.2.0: test marking++, parametrization++ and duration profiling

pytest-2.2.0 is a test-suite compatible release of the popular py.test testing tool. Plugins might need upgrades. It comes with these improvements:

- easier and more powerful parametrization of tests:
  - new @pytest.mark.parametrize decorator to run tests with different arguments
  - new metafunc.parametrize() API for parametrizing arguments independently
  - see examples at http://pytest.org/latest/example/parametrize.html
  - NOTE that parametrize() related APIs are still a bit experimental and might change in future releases.
- improved handling of test markers and refined marking mechanism:
  - "-m markexpr" option for selecting tests according to their mark
  - a new "markers" ini-variable for registering test markers for your project
  - the new "–strict" bails out with an error if using unregistered markers.
  - see examples at http://pytest.org/latest/example/markers.html
- duration profiling: new "–duration=N" option showing the N slowest test execution or setup/teardown calls. This is most useful if you want to find out where your slowest test code is.
- also 2.2.0 performs more eager calling of teardown/finalizers functions resulting in better and more accurate reporting when they fail

Besides there is the usual set of bug fixes along with a cleanup of pytest's own test suite allowing it to run on a wider range of environments.

For general information, see extensive docs with examples here:

http://pytest.org/

If you want to install or upgrade pytest you might just type:

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

Thanks to Ronny Pfannschmidt, David Burns, Jeff Donner, Daniel Nouri, Alfredo Deza and all who gave feedback or sent bug reports.

best, holger krekel

11.9.1 notes on incompatibility

While test suites should work unchanged you might need to upgrade plugins:

- You need a new version of the pytest-xdist plugin (1.7) for distributing test runs.
- Other plugins might need an upgrade if they implement the `pytest_runtest_logreport` hook which now is called unconditionally for the setup/teardown fixture phases of a test. You may choose to ignore setup/teardown failures by inserting "if rep.when != 'call': return" or something similar. Note that most code probably "just" works because the hook was already called for failing setup/teardown phases of a test so a plugin should have been ready to grok such reports already.

11.9.2 Changes between 2.1.3 and 2.2.0

- fix issue90: introduce eager tearing down of test items so that teardown function are called earlier.
- add an all-powerful metafunc.parametrize function which allows to parametrize test function arguments in multiple steps and therefore from independent plugins and places.
- add a @pytest.mark.parametrize helper which allows to easily call a test function with different argument values.
- Add examples to the "parametrize" example page, including a quick port of Test scenarios and the new parametrize function and decorator.
- introduce registration for "pytest.mark.*" helpers via ini-files or through plugin hooks. Also introduce a "–strict" option which will treat unregistered markers as errors allowing to avoid typos and maintain a well described set of markers for your test suite. See examples at http://pytest.org/latest/mark.html and its links.
- issue50: introduce "-m marker" option to select tests based on markers (this is a stricter and more predictable version of "-k" in that "-m" only matches complete markers and has more obvious rules for and/or semantics.
- new feature to help optimizing the speed of your tests: –durations=N option for displaying N slowest test calls and setup/teardown methods.
- fix issue87: –pastebin now works with python3
- fix issue89: –pdb with unexpected exceptions in doctest work more sensibly
- fix and cleanup pytest's own test suite to not leak FDs
- fix issue83: link to generated funcarg list
- fix issue74: pyarg module names are now checked against imp.find_module false positives
- fix compatibility with twisted/trial-11.1.0 use cases

11.10 py.test 2.1.3: just some more fixes

pytest-2.1.3 is a minor backward compatible maintenance release of the popular py.test testing tool. It is commonly used for unit, functional- and integration testing. See extensive docs with examples here:

http://pytest.org/

The release contains another fix to the perfected assertions introduced with the 2.1 series as well as the new possibility to customize reporting for assertion expressions on a per-directory level.

If you want to install or upgrade pytest, just type one of:

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

Thanks to the bug reporters and to Ronny Pfannschmidt, Benjamin Peterson and Floris Bruynooghe who implemented the fixes.

best, holger krekel

11.10.1 Changes between 2.1.2 and 2.1.3

- fix issue79: assertion rewriting failed on some comparisons in boolops,
- correctly handle zero length arguments (a la pytest '')
- fix issue67 / junitxml now contains correct test durations

- fix issue75 / skipping test failure on jython
- fix issue77 / Allow assertrepr_compare hook to apply to a subset of tests

11.11 py.test 2.1.2: bug fixes and fixes for jython

pytest-2.1.2 is a minor backward compatible maintenance release of the popular py.test testing tool. pytest is commonly used for unit, functional- and integration testing. See extensive docs with examples here:

http://pytest.org/

Most bug fixes address remaining issues with the perfected assertions introduced in the 2.1 series - many thanks to the bug reporters and to Benjamin Peterson for helping to fix them. pytest should also work better with Jython-2.5.1 (and Jython trunk).

If you want to install or upgrade pytest, just type one of:

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

best, holger krekel / http://merlinux.eu

11.11.1 Changes between 2.1.1 and 2.1.2

- fix assertion rewriting on files with windows newlines on some Python versions
- refine test discovery by package/module name (–pyargs), thanks Florian Mayer
- fix issue69 / assertion rewriting fixed on some boolean operations
- fix issue68 / packages now work with assertion rewriting
- fix issue66: use different assertion rewriting caches when the -O option is passed
- don't try assertion rewriting on Jython, use reinterp

11.12 py.test 2.1.1: assertion fixes and improved junitxml output

pytest-2.1.1 is a backward compatible maintenance release of the popular py.test testing tool. See extensive docs with examples here:

http://pytest.org/

Most bug fixes address remaining issues with the perfected assertions introduced with 2.1.0 - many thanks to the bug reporters and to Benjamin Peterson for helping to fix them. Also, junitxml output now produces system-out/err tags which lead to better displays of tracebacks with Jenkins.

Also a quick note to package maintainers and others interested: there now is a "pytest" man page which can be generated with "make man" in doc/.

If you want to install or upgrade pytest, just type one of:

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

best, holger krekel / http://merlinux.eu

11.12.1 Changes between 2.1.0 and 2.1.1

- fix issue64 / pytest.set_trace now works within pytest_generate_tests hooks
- fix issue60 / fix error conditions involving the creation of __pycache__
- fix issue63 / assertion rewriting on inserts involving strings containing '%'
- fix assertion rewriting on calls with a ** arg
- don't cache rewritten modules if bytecode generation is disabled
- fix assertion rewriting in read-only directories
- fix issue59: provide system-out/err tags for junitxml output
- fix issue61: assertion rewriting on boolean operations with 3 or more operands
- you can now build a man page with "cd doc ; make man"

11.13 py.test 2.1.0: perfected assertions and bug fixes

Welcome to the release of pytest-2.1, a mature testing tool for Python, supporting CPython 2.4-3.2, Jython and latest PyPy interpreters. See the improved extensive docs (now also as PDF!) with tested examples here:

http://pytest.org/

The single biggest news about this release are **perfected assertions** courtesy of Benjamin Peterson. You can now safely use `assert` statements in test modules without having to worry about side effects or python optimization ("-OO") options. This is achieved by rewriting assert statements in test modules upon import, using a PEP302 hook. See http://pytest.org/assert.html#advanced-assertion-introspection for detailed information. The work has been partly sponsored by my company, merlinux GmbH.

For further details on bug fixes and smaller enhancements see below.

If you want to install or upgrade pytest, just type one of:

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

best, holger krekel / http://merlinux.eu

11.13.1 Changes between 2.0.3 and 2.1.0

- fix issue53 call nosestyle setup functions with correct ordering
- fix issue58 and issue59: new assertion code fixes
- merge Benjamin's assertionrewrite branch: now assertions for test modules on python 2.6 and above are done by rewriting the AST and saving the pyc file before the test module is imported. see doc/assert.txt for more info.
- fix issue43: improve doctests with better traceback reporting on unexpected exceptions
- fix issue47: timing output in junitxml for test cases is now correct
- fix issue48: typo in MarkInfo repr leading to exception
- fix issue49: avoid confusing error when initialization partially fails
- fix issue44: env/username expansion for junitxml file path
- show releaselevel information in test runs for pypy

- reworked doc pages for better navigation and PDF generation
- report KeyboardInterrupt even if interrupted during session startup
- fix issue 35 - provide PDF doc version and download link from index page

11.14 py.test 2.0.3: bug fixes and speed ups

Welcome to pytest-2.0.3, a maintenance and bug fix release of pytest, a mature testing tool for Python, supporting CPython 2.4-3.2, Jython and latest PyPy interpreters. See the extensive docs with tested examples here:

http://pytest.org/

If you want to install or upgrade pytest, just type one of:

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

There also is a bugfix release 1.6 of pytest-xdist, the plugin that enables seemless distributed and "looponfail" testing for Python.

best, holger krekel

11.14.1 Changes between 2.0.2 and 2.0.3

- fix issue38: nicer tracebacks on calls to hooks, particularly early configure/sessionstart ones
- fix missing skip reason/meta information in junitxml files, reported via http://lists.idyll.org/pipermail/testing-in-python/2011-March/003928.html
- fix issue34: avoid collection failure with "test" prefixed classes deriving from object.
- don't require zlib (and other libs) for genscript plugin without –genscript actually being used.
- speed up skips (by not doing a full traceback represenation internally)
- fix issue37: avoid invalid characters in junitxml's output

11.15 py.test 2.0.2: bug fixes, improved xfail/skip expressions, speed ups

Welcome to pytest-2.0.2, a maintenance and bug fix release of pytest, a mature testing tool for Python, supporting CPython 2.4-3.2, Jython and latest PyPy interpreters. See the extensive docs with tested examples here:

http://pytest.org/

If you want to install or upgrade pytest, just type one of:

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

Many thanks to all issue reporters and people asking questions or complaining, particularly Jurko for his insistence, Laura, Victor and Brianna for helping with improving and Ronny for his general advise.

best, holger krekel

11.15.1 Changes between 2.0.1 and 2.0.2

- tackle issue32 - speed up test runs of very quick test functions by reducing the relative overhead
- fix issue30 - extended xfail/skipif handling and improved reporting. If you have a syntax error in your skip/xfail expressions you now get nice error reports.

  Also you can now access module globals from xfail/skipif expressions so that this for example works now:

  ```
  import pytest
  import mymodule
  @pytest.mark.skipif("mymodule.__version__[0] == "1")
  def test_function():
      pass
  ```

  This will not run the test function if the module's version string does not start with a "1". Note that specifying a string instead of a boolean expressions allows py.test to report meaningful information when summarizing a test run as to what conditions lead to skipping (or xfail-ing) tests.
- fix issue28 - setup_method and pytest_generate_tests work together The setup_method fixture method now gets called also for test function invocations generated from the pytest_generate_tests hook.
- fix issue27 - collectonly and keyword-selection (-k) now work together Also, if you do "py.test –collectonly -q" you now get a flat list of test ids that you can use to paste to the py.test commandline in order to execute a particular test.
- fix issue25 avoid reported problems with –pdb and python3.2/encodings output
- fix issue23 - tmpdir argument now works on Python3.2 and WindowsXP Starting with Python3.2 os.symlink may be supported. By requiring a newer py lib version the py.path.local() implementation acknowledges this.
- fixed typos in the docs (thanks Victor Garcia, Brianna Laugher) and particular thanks to Laura Creighton who also revieved parts of the documentation.
- fix slighly wrong output of verbose progress reporting for classes (thanks Amaury)
- more precise (avoiding of) deprecation warnings for node.Class|Function accesses
- avoid std unittest assertion helper code in tracebacks (thanks Ronny)

11.16 py.test 2.0.1: bug fixes

Welcome to pytest-2.0.1, a maintenance and bug fix release of pytest, a mature testing tool for Python, supporting CPython 2.4-3.2, Jython and latest PyPy interpreters. See extensive docs with tested examples here:

> http://pytest.org/

If you want to install or upgrade pytest, just type one of:

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

Many thanks to all issue reporters and people asking questions or complaining. Particular thanks to Floris Bruynooghe and Ronny Pfannschmidt for their great coding contributions and many others for feedback and help.

best, holger krekel

11.16.1 Changes between 2.0.0 and 2.0.1

- refine and unify initial capturing so that it works nicely even if the logging module is used on an early-loaded conftest.py file or plugin.
- fix issue12 - show plugin versions with "–version" and "–traceconfig" and also document how to add extra information to reporting test header
- fix issue17 (import-* reporting issue on python3) by requiring py>1.4.0 (1.4.1 is going to include it)
- fix issue10 (numpy arrays truth checking) by refining assertion interpretation in py lib
- fix issue15: make nose compatibility tests compatible with python3 (now that nose-1.0 supports python3)
- remove somewhat surprising "same-conftest" detection because it ignores conftest.py when they appear in several subdirs.
- improve assertions ("not in"), thanks Floris Bruynooghe
- improve behaviour/warnings when running on top of "python -OO" (assertions and docstrings are turned off, leading to potential false positives)
- introduce a pytest_cmdline_processargs(args) hook to allow dynamic computation of command line arguments. This fixes a regression because py.test prior to 2.0 allowed to set command line options from conftest.py files which so far pytest-2.0 only allowed from ini-files now.
- fix issue7: assert failures in doctest modules. unexpected failures in doctests will not generally show nicer, i.e. within the doctest failing context.
- fix issue9: setup/teardown functions for an xfail-marked test will report as xfail if they fail but report as normally passing (not xpassing) if they succeed. This only is true for "direct" setup/teardown invocations because teardown_class/ teardown_module cannot closely relate to a single test.
- fix issue14: no logging errors at process exit
- refinements to "collecting" output on non-ttys
- refine internal plugin registration and –traceconfig output
- introduce a mechanism to prevent/unregister plugins from the command line, see http://pytest.org/latest/plugins.html#cmdunregister
- activate resultlog plugin by default
- fix regression wrt yielded tests which due to the collection-before-running semantics were not setup as with pytest 1.3.4. Note, however, that the recommended and much cleaner way to do test parametrization remains the "pytest_generate_tests" mechanism, see the docs.

11.17 py.test 2.0.0: asserts++, unittest++, reporting++, config++, docs++

Welcome to pytest-2.0.0, a major new release of "py.test", the rapid easy Python testing tool. There are many new features and enhancements, see below for summary and detailed lists. A lot of long-deprecated code has been removed, resulting in a much smaller and cleaner implementation. See the new docs with examples here:

http://pytest.org/2.0.0/index.html

A note on packaging: pytest used to part of the "py" distribution up until version py-1.3.4 but this has changed now: pytest-2.0.0 only contains py.test related code and is expected to be backward-compatible to existing test code. If you want to install pytest, just type one of:

```
pip install -U pytest
easy_install -U pytest
```

Many thanks to all issue reporters and people asking questions or complaining. Particular thanks to Floris Bruynooghe and Ronny Pfannschmidt for their great coding contributions and many others for feedback and help.

best, holger krekel

11.17.1 New Features

- new invocations through Python interpreter and from Python:

  ```
  python -m pytest      # on all pythons >= 2.5
  ```

  or from a python program:

  ```
  import pytest ; pytest.main(arglist, pluginlist)
  ```

  see http://pytest.org/2.0.0/usage.html for details.

- new and better reporting information in assert expressions if comparing lists, sequences or strings.

  see http://pytest.org/2.0.0/assert.html#newreport

- new configuration through ini-files (setup.cfg or tox.ini recognized), for example:

  ```
  [pytest]
  norecursedirs = .hg data*  # don't ever recurse in such dirs
  addopts = -x --pyargs      # add these command line options by default
  ```

  see http://pytest.org/2.0.0/customize.html

- improved standard unittest support. In general py.test should now better be able to run custom unittest.TestCases like twisted trial or Django based TestCases. Also you can now run the tests of an installed 'unittest' package with py.test:

  ```
  py.test --pyargs unittest
  ```

- new "-q" option which decreases verbosity and prints a more nose/unittest-style "dot" output.
- many many more detailed improvements details

11.17.2 Fixes

- fix issue126 - introduce py.test.set_trace() to trace execution via PDB during the running of tests even if capturing is ongoing.
- fix issue124 - make reporting more resilient against tests opening files on filedescriptor 1 (stdout).
- fix issue109 - sibling conftest.py files will not be loaded. (and Directory collectors cannot be customized anymore from a Directory's conftest.py - this needs to happen at least one level up).
- fix issue88 (finding custom test nodes from command line arg)
- fix issue93 stdout/stderr is captured while importing conftest.py
- fix bug: unittest collected functions now also can have "pytestmark" applied at class/module level

11.17.3 Important Notes

- The usual way in pre-2.0 times to use py.test in python code was to import "py" and then e.g. use "py.test.raises" for the helper. This remains valid and is not planned to be deprecated. However, in most examples and internal code you'll find "import pytest" and "pytest.raises" used as the recommended default way.
- pytest now first performs collection of the complete test suite before running any test. This changes for example the semantics of when pytest_collectstart/pytest_collectreport are called. Some plugins may need upgrading.
- The pytest package consists of a 400 LOC core.py and about 20 builtin plugins, summing up to roughly 5000 LOCs, including docstrings. To be fair, it also uses generic code from the "pylib", and the new "py" package to help with filesystem and introspection/code manipulation.

11.17.4 (Incompatible) Removals

- py.test.config is now only available if you are in a test run.
- the following (mostly already deprecated) functionality was removed:
  - removed support for Module/Class/... collection node definitions in conftest.py files. They will cause nothing special.
  - removed support for calling the pre-1.0 collection API of "run()" and "join"
  - removed reading option values from conftest.py files or env variables. This can now be done much much better and easier through the ini-file mechanism and the "addopts" entry in particular.
  - removed the "disabled" attribute in test classes. Use the skipping and pytestmark mechanism to skip or xfail a test class.
- py.test.collect.Directory does not exist anymore and it is not possible to provide an own "Directory" object. If you have used this and don't know what to do, get in contact. We'll figure something out.

  Note that pytest_collect_directory() is still called but any return value will be ignored. This allows to keep old code working that performed for example "py.test.skip()" in collect() to prevent recursion into directory trees if a certain dependency or command line option is missing.

see *Changelog history* for more detailed changes.

CHAPTER

TWELVE

CHANGELOG HISTORY

12.1 Changes between 2.3.4 and 2.3.5dev

- never consider a fixture function for test function collection
- allow re-running of test items / helps to fix pytest-reruntests plugin and also help to keep less fixture/resource references alive
- put captured stdout/stderr into junitxml output even for passing tests (thanks Adam Goucher)
- Issue 265 - integrate nose setup/teardown with setupstate so it doesnt try to teardown if it did not setup
- issue 271 - dont write junitxml on slave nodes
- Issue 274 - dont try to show full doctest example when doctest does not know the example location
- issue 280 - disable assertion rewriting on buggy CPython 2.6.0
- inject "getfixture()" helper to retrieve fixtures from doctests, thanks Andreas Zeidler
- issue 259 - when assertion rewriting, be consistent with the default source encoding of ASCII on Python 2
- issue 251 - report a skip instead of ignoring classes with init
- issue250 unicode/str mixes in parametrization names and values now works
- issue257, assertion-triggered compilation of source ending in a comment line doesn't blow up in python2.5 (fixed through py>=1.4.13.dev6)
- fix –genscript option to generate standalone scripts that also work with python3.3 (importer ordering)
- issue171 - in assertion rewriting, show the repr of some global variables
- fix option help for "-k"
- move long description of distribution into README.rst
- improve docstring for metafunc.parametrize()
- fix bug where using capsys with pytest.set_trace() in a test function would break when looking at capsys.readouterr()
- allow to specify prefixes starting with "_" when customizing python_functions test discovery. (thanks Graham Horler)
- improve PYTEST_DEBUG tracing output by puting extra data on a new lines with additional indent
- ensure OutcomeExceptions like skip/fail have initialized exception attributes
- issue 260 - don't use nose special setup on plain unittest cases

- fix issue134 - print the collect errors that prevent running specified test items
- fix issue266 - accept unicode in MarkEvaluator expressions

12.2 Changes between 2.3.3 and 2.3.4

- yielded test functions will now have autouse-fixtures active but cannot accept fixtures as funcargs - it's anyway recommended to rather use the post-2.0 parametrize features instead of yield, see: http://pytest.org/latest/example/parametrize.html
- fix autouse-issue where autouse-fixtures would not be discovered if defined in a a/conftest.py file and tests in a/tests/test_some.py
- fix issue226 - LIFO ordering for fixture teardowns
- fix issue224 - invocations with >256 char arguments now work
- fix issue91 - add/discuss package/directory level setups in example
- allow to dynamically define markers via item.keywords[...]=assignment integrating with "-m" option
- make "-k" accept an expressions the same as with "-m" so that one can write: -k "name1 or name2" etc. This is a slight incompatibility if you used special syntax like "TestClass.test_method" which you now need to write as -k "TestClass and test_method" to match a certain method in a certain test class.

12.3 Changes between 2.3.2 and 2.3.3

- fix issue214 - parse modules that contain special objects like e. g. flask's request object which blows up on getattr access if no request is active. thanks Thomas Waldmann.
- fix issue213 - allow to parametrize with values like numpy arrays that do not support an __eq__ operator
- fix issue215 - split test_python.org into multiple files
- fix issue148 - @unittest.skip on classes is now recognized and avoids calling setUpClass/tearDownClass, thanks Pavel Repin
- fix issue209 - reintroduce python2.4 support by depending on newer pylib which re-introduced statement-finding for pre-AST interpreters
- nose support: only call setup if its a callable, thanks Andrew Taumoefolau
- fix issue219 - add py2.4-3.3 classifiers to TROVE list
- in tracebacks *,** arg values are now shown next to normal arguments (thanks Manuel Jacob)
- fix issue217 - support mock.patch with pytest's fixtures - note that you need either mock-1.0.1 or the python3.3 builtin unittest.mock.
- fix issue127 - improve documentation for pytest_addoption() and add a `config.getoption(name)` helper function for consistency.

12.4 Changes between 2.3.1 and 2.3.2

- fix issue208 and fix issue29 use new py version to avoid long pauses when printing tracebacks in long modules

- fix issue205 - conftests in subdirs customizing pytest_pycollect_makemodule and pytest_pycollect_makeitem now work properly
- fix teardown-ordering for parametrized setups
- fix issue127 - better documentation for pytest_addoption and related objects.
- fix unittest behaviour: TestCase.runtest only called if there are test methods defined
- improve trial support: don't collect its empty unittest.TestCase.runTest() method
- "python setup.py test" now works with pytest itself
- fix/improve internal/packaging related bits:
  - exception message check of test_nose.py now passes on python33 as well
  - issue206 - fix test_assertrewrite.py to work when a global PYTHONDONTWRITEBYTECODE=1 is present
  - add tox.ini to pytest distribution so that ignore-dirs and others config bits are properly distributed for maintainers who run pytest-own tests

12.5 Changes between 2.3.0 and 2.3.1

- fix issue202 - fix regression: using "self" from fixture functions now works as expected (it's the same "self" instance that a test method which uses the fixture sees)
- skip pexpect using tests (test_pdb.py mostly) on freebsd* systems due to pexpect not supporting it properly (hanging)
- link to web pages from –markers output which provides help for pytest.mark.* usage.

12.6 Changes between 2.2.4 and 2.3.0

- fix issue202 - better automatic names for parametrized test functions
- fix issue139 - introduce @pytest.fixture which allows direct scoping and parametrization of funcarg factories.
- fix issue198 - conftest fixtures were not found on windows32 in some circumstances with nested directory structures due to path manipulation issues
- fix issue193 skip test functions with were parametrized with empty parameter sets
- fix python3.3 compat, mostly reporting bits that previously depended on dict ordering
- introduce re-ordering of tests by resource and parametrization setup which takes precedence to the usual file-ordering
- fix issue185 monkeypatching time.time does not cause pytest to fail
- fix issue172 duplicate call of pytest.fixture decoratored setup_module functions
- fix junitxml=path construction so that if tests change the current working directory and the path is a relative path it is constructed correctly from the original current working dir.
- fix "python setup.py test" example to cause a proper "errno" return
- fix issue165 - fix broken doc links and mention stackoverflow for FAQ
- catch unicode-issues when writing failure representations to terminal to prevent the whole session from crashing

- fix xfail/skip confusion: a skip-mark or an imperative pytest.skip will now take precedence before xfail-markers because we can't determine xfail/xpass status in case of a skip. see also: http://stackoverflow.com/questions/11105828/in-py-test-when-i-explicitly-skip-a-test-that-is-marked-as-xfail-how-can-i-get
- always report installed 3rd party plugins in the header of a test run
- fix issue160: a failing setup of an xfail-marked tests should be reported as xfail (not xpass)
- fix issue128: show captured output when capsys/capfd are used
- fix issue179: propperly show the dependency chain of factories
- pluginmanager.register(...) now raises ValueError if the plugin has been already registered or the name is taken
- fix issue159: improve http://pytest.org/latest/faq.html especially with respect to the "magic" history, also mention pytest-django, trial and unittest integration.
- make request.keywords and node.keywords writable. All descendant collection nodes will see keyword values. Keywords are dictionaries containing markers and other info.
- fix issue 178: xml binary escapes are now wrapped in py.xml.raw
- fix issue 176: correctly catch the builtin AssertionError even when we replaced AssertionError with a subclass on the python level
- factory discovery no longer fails with magic global callables that provide no sane __code__ object (mock.call for example)
- fix issue 182: testdir.inprocess_run now considers passed plugins
- **fix issue 188: ensure sys.exc_info is clear on python2** before calling into a test
- fix issue 191: add unittest TestCase runTest method support
- fix issue 156: monkeypatch correctly handles class level descriptors
- reporting refinements:
  - pytest_report_header now receives a "startdir" so that you can use startdir.bestrelpath(yourpath) to show nice relative path
  - allow plugins to implement both pytest_report_header and pytest_sessionstart (sessionstart is invoked first).
  - don't show deselected reason line if there is none
  - py.test -vv will show all of assert comparisations instead of truncating

12.7 Changes between 2.2.3 and 2.2.4

- fix error message for rewritten assertions involving the % operator
- fix issue 126: correctly match all invalid xml characters for junitxml binary escape
- fix issue with unittest: now @unittest.expectedFailure markers should be processed correctly (you can also use @pytest.mark markers)
- document integration with the extended distribute/setuptools test commands
- fix issue 140: propperly get the real functions of bound classmethods for setup/teardown_class
- fix issue #141: switch from the deceased paste.pocoo.org to bpaste.net
- fix issue #143: call unconfigure/sessionfinish always when configure/sessionstart where called

- fix issue #144: better mangle test ids to junitxml classnames
- upgrade distribute_setup.py to 0.6.27

12.8 Changes between 2.2.2 and 2.2.3

- fix uploaded package to only include neccesary files

12.9 Changes between 2.2.1 and 2.2.2

- fix issue101: wrong args to unittest.TestCase test function now produce better output
- fix issue102: report more useful errors and hints for when a test directory was renamed and some pyc/__pycache__ remain
- fix issue106: allow parametrize to be applied multiple times e.g. from module, class and at function level.
- fix issue107: actually perform session scope finalization
- don't check in parametrize if indirect parameters are funcarg names
- add chdir method to monkeypatch funcarg
- fix crash resulting from calling monkeypatch undo a second time
- fix issue115: make –collectonly robust against early failure (missing files/directories)
- "-qq –collectonly" now shows only files and the number of tests in them
- "-q –collectonly" now shows test ids
- allow adding of attributes to test reports such that it also works with distributed testing (no upgrade of pytest-xdist needed)

12.10 Changes between 2.2.0 and 2.2.1

- fix issue99 (in pytest and py) internallerrors with resultlog now produce better output - fixed by normalizing pytest_internalerror input arguments.
- fix issue97 / traceback issues (in pytest and py) improve traceback output in conjunction with jinja2 and cython which hack tracebacks
- fix issue93 (in pytest and pytest-xdist) avoid "delayed teardowns": the final test in a test node will now run its teardown directly instead of waiting for the end of the session. Thanks Dave Hunt for the good reporting and feedback. The pytest_runtest_protocol as well as the pytest_runtest_teardown hooks now have "nextitem" available which will be None indicating the end of the test run.
- fix collection crash due to unknown-source collected items, thanks to Ralf Schmitt (fixed by depending on a more recent pylib)

12.11 Changes between 2.1.3 and 2.2.0

- fix issue90: introduce eager tearing down of test items so that teardown function are called earlier.

- add an all-powerful metafunc.parametrize function which allows to parametrize test function arguments in multiple steps and therefore from indepdenent plugins and palces.
- add a @pytest.mark.parametrize helper which allows to easily call a test function with different argument values
- Add examples to the "parametrize" example page, including a quick port of Test scenarios and the new parametrize function and decorator.
- introduce registration for "pytest.mark.*" helpers via ini-files or through plugin hooks. Also introduce a "–strict" option which will treat unregistered markers as errors allowing to avoid typos and maintain a well described set of markers for your test suite. See exaples at http://pytest.org/latest/mark.html and its links.
- issue50: introduce "-m marker" option to select tests based on markers (this is a stricter and more predictable version of '-k' in that "-m" only matches complete markers and has more obvious rules for and/or semantics.
- new feature to help optimizing the speed of your tests: –durations=N option for displaying N slowest test calls and setup/teardown methods.
- fix issue87: –pastebin now works with python3
- fix issue89: –pdb with unexpected exceptions in doctest work more sensibly
- fix and cleanup pytest's own test suite to not leak FDs
- fix issue83: link to generated funcarg list
- fix issue74: pyarg module names are now checked against imp.find_module false positives
- fix compatibility with twisted/trial-11.1.0 use cases
- simplify Node.listchain
- simplify junitxml output code by relying on py.xml
- add support for skip properties on unittest classes and functions

12.12 Changes between 2.1.2 and 2.1.3

- fix issue79: assertion rewriting failed on some comparisons in boolops
- correctly handle zero length arguments (a la pytest '')
- fix issue67 / junitxml now contains correct test durations, thanks ronny
- fix issue75 / skipping test failure on jython
- fix issue77 / Allow assertrepr_compare hook to apply to a subset of tests

12.13 Changes between 2.1.1 and 2.1.2

- fix assertion rewriting on files with windows newlines on some Python versions
- refine test discovery by package/module name (–pyargs), thanks Florian Mayer
- fix issue69 / assertion rewriting fixed on some boolean operations
- fix issue68 / packages now work with assertion rewriting
- fix issue66: use different assertion rewriting caches when the -O option is passed
- don't try assertion rewriting on Jython, use reinterp

12.14 Changes between 2.1.0 and 2.1.1

- fix issue64 / pytest.set_trace now works within pytest_generate_tests hooks
- fix issue60 / fix error conditions involving the creation of __pycache__
- fix issue63 / assertion rewriting on inserts involving strings containing '%'
- fix assertion rewriting on calls with a ** arg
- don't cache rewritten modules if bytecode generation is disabled
- fix assertion rewriting in read-only directories
- fix issue59: provide system-out/err tags for junitxml output
- fix issue61: assertion rewriting on boolean operations with 3 or more operands
- you can now build a man page with "cd doc ; make man"

12.15 Changes between 2.0.3 and 2.1.0.DEV

- fix issue53 call nosestyle setup functions with correct ordering
- fix issue58 and issue59: new assertion code fixes
- merge Benjamin's assertionrewrite branch: now assertions for test modules on python 2.6 and above are done by rewriting the AST and saving the pyc file before the test module is imported. see doc/assert.txt for more info.
- fix issue43: improve doctests with better traceback reporting on unexpected exceptions
- fix issue47: timing output in junitxml for test cases is now correct
- fix issue48: typo in MarkInfo repr leading to exception
- fix issue49: avoid confusing error when initizaliation partially fails
- fix issue44: env/username expansion for junitxml file path
- show releaselevel information in test runs for pypy
- reworked doc pages for better navigation and PDF generation
- report KeyboardInterrupt even if interrupted during session startup
- fix issue 35 - provide PDF doc version and download link from index page

12.16 Changes between 2.0.2 and 2.0.3

- fix issue38: nicer tracebacks on calls to hooks, particularly early configure/sessionstart ones
- fix missing skip reason/meta information in junitxml files, reported via http://lists.idyll.org/pipermail/testing-in-python/2011-March/003928.html
- fix issue34: avoid collection failure with "test" prefixed classes deriving from object.
- don't require zlib (and other libs) for genscript plugin without –genscript actually being used.
- speed up skips (by not doing a full traceback represenation internally)
- fix issue37: avoid invalid characters in junitxml's output

12.17 Changes between 2.0.1 and 2.0.2

- tackle issue32 - speed up test runs of very quick test functions by reducing the relative overhead
- fix issue30 - extended xfail/skipif handling and improved reporting. If you have a syntax error in your skip/xfail expressions you now get nice error reports.

  Also you can now access module globals from xfail/skipif expressions so that this for example works now:

  ```
  import pytest
  import mymodule
  @pytest.mark.skipif("mymodule.__version__[0] == "1")
  def test_function():
      pass
  ```

  This will not run the test function if the module's version string does not start with a "1". Note that specifying a string instead of a boolean expressions allows py.test to report meaningful information when summarizing a test run as to what conditions lead to skipping (or xfail-ing) tests.
- fix issue28 - setup_method and pytest_generate_tests work together The setup_method fixture method now gets called also for test function invocations generated from the pytest_generate_tests hook.
- fix issue27 - collectonly and keyword-selection (-k) now work together Also, if you do "py.test –collectonly -q" you now get a flat list of test ids that you can use to paste to the py.test commandline in order to execute a particular test.
- fix issue25 avoid reported problems with –pdb and python3.2/encodings output
- fix issue23 - tmpdir argument now works on Python3.2 and WindowsXP Starting with Python3.2 os.symlink may be supported. By requiring a newer py lib version the py.path.local() implementation acknowledges this.
- fixed typos in the docs (thanks Victor Garcia, Brianna Laugher) and particular thanks to Laura Creighton who also revieved parts of the documentation.
- fix slighly wrong output of verbose progress reporting for classes (thanks Amaury)
- more precise (avoiding of) deprecation warnings for node.Class|Function accesses
- avoid std unittest assertion helper code in tracebacks (thanks Ronny)

12.18 Changes between 2.0.0 and 2.0.1

- refine and unify initial capturing so that it works nicely even if the logging module is used on an early-loaded conftest.py file or plugin.
- allow to omit "()" in test ids to allow for uniform test ids as produced by Alfredo's nice pytest.vim plugin.
- fix issue12 - show plugin versions with "–version" and "–traceconfig" and also document how to add extra information to reporting test header
- fix issue17 (import-* reporting issue on python3) by requiring py>1.4.0 (1.4.1 is going to include it)
- fix issue10 (numpy arrays truth checking) by refining assertion interpretation in py lib
- fix issue15: make nose compatibility tests compatible with python3 (now that nose-1.0 supports python3)
- remove somewhat surprising "same-conftest" detection because it ignores conftest.py when they appear in several subdirs.
- improve assertions ("not in"), thanks Floris Bruynooghe

- improve behaviour/warnings when running on top of "python -OO" (assertions and docstrings are turned off, leading to potential false positives)
- introduce a pytest_cmdline_processargs(args) hook to allow dynamic computation of command line arguments. This fixes a regression because py.test prior to 2.0 allowed to set command line options from conftest.py files which so far pytest-2.0 only allowed from ini-files now.
- fix issue7: assert failures in doctest modules. unexpected failures in doctests will not generally show nicer, i.e. within the doctest failing context.
- fix issue9: setup/teardown functions for an xfail-marked test will report as xfail if they fail but report as normally passing (not xpassing) if they succeed. This only is true for "direct" setup/teardown invocations because teardown_class/ teardown_module cannot closely relate to a single test.
- fix issue14: no logging errors at process exit
- refinements to "collecting" output on non-ttys
- refine internal plugin registration and –traceconfig output
- introduce a mechanism to prevent/unregister plugins from the command line, see http://pytest.org/plugins.html#cmdunregister
- activate resultlog plugin by default
- fix regression wrt yielded tests which due to the collection-before-running semantics were not setup as with pytest 1.3.4. Note, however, that the recommended and much cleaner way to do test parametraization remains the "pytest_generate_tests" mechanism, see the docs.

12.19 Changes between 1.3.4 and 2.0.0

- pytest-2.0 is now its own package and depends on pylib-2.0
- new ability: python -m pytest / python -m pytest.main ability
- new python invcation: pytest.main(args, plugins) to load some custom plugins early.
- try harder to run unittest test suites in a more compatible manner by deferring setup/teardown semantics to the unittest package. also work harder to run twisted/trial and Django tests which should now basically work by default.
- introduce a new way to set config options via ini-style files, by default setup.cfg and tox.ini files are searched. The old ways (certain environment variables, dynamic conftest.py reading is removed).
- add a new "-q" option which decreases verbosity and prints a more nose/unittest-style "dot" output.
- fix issue135 - marks now work with unittest test cases as well
- fix issue126 - introduce py.test.set_trace() to trace execution via PDB during the running of tests even if capturing is ongoing.
- fix issue123 - new "python -m py.test" invocation for py.test (requires Python 2.5 or above)
- fix issue124 - make reporting more resilient against tests opening files on filedescriptor 1 (stdout).
- fix issue109 - sibling conftest.py files will not be loaded. (and Directory collectors cannot be customized anymore from a Directory's conftest.py - this needs to happen at least one level up).
- introduce (customizable) assertion failure representations and enhance output on assertion failures for comparisons and other cases (Floris Bruynooghe)
- nose-plugin: pass through type-signature failures in setup/teardown functions instead of not calling them (Ed Singleton)

- remove py.test.collect.Directory (follows from a major refactoring and simplification of the collection process)
- majorly reduce py.test core code, shift function/python testing to own plugin
- fix issue88 (finding custom test nodes from command line arg)
- refine 'tmpdir' creation, will now create basenames better associated with test names (thanks Ronny)
- "xpass" (unexpected pass) tests don't cause exitcode!=0
- fix issue131 / issue60 - importing doctests in __init__ files used as namespace packages
- fix issue93 stdout/stderr is captured while importing conftest.py
- fix bug: unittest collected functions now also can have "pytestmark" applied at class/module level
- add ability to use "class" level for cached_setup helper
- fix strangeness: mark.* objects are now immutable, create new instances

12.20 Changes between 1.3.3 and 1.3.4

- fix issue111: improve install documentation for windows
- fix issue119: fix custom collectability of __init__.py as a module
- fix issue116: –doctestmodules work with __init__.py files as well
- fix issue115: unify internal exception passthrough/catching/GeneratorExit
- fix issue118: new –tb=native for presenting cpython-standard exceptions

12.21 Changes between 1.3.2 and 1.3.3

- fix issue113: assertion representation problem with triple-quoted strings (and possibly other cases)
- make conftest loading detect that a conftest file with the same content was already loaded, avoids surprises in nested directory structures which can be produced e.g. by Hudson. It probably removes the need to use –confcutdir in most cases.
- fix terminal coloring for win32 (thanks Michael Foord for reporting)
- fix weirdness: make terminal width detection work on stdout instead of stdin (thanks Armin Ronacher for reporting)
- remove trailing whitespace in all py/text distribution files

12.22 Changes between 1.3.1 and 1.3.2

12.22.1 New features

- fix issue103: introduce py.test.raises as context manager, examples:

```
with py.test.raises(ZeroDivisionError):
    x = 0
    1 / x

with py.test.raises(RuntimeError) as excinfo:
```

```
    call_something()

# you may do extra checks on excinfo.value|type|traceback here
```

(thanks Ronny Pfannschmidt)

- Funcarg factories can now dynamically apply a marker to a test invocation. This is for example useful if a factory provides parameters to a test which are expected-to-fail:

```
def pytest_funcarg__arg(request):
    request.applymarker(py.test.mark.xfail(reason="flaky config"))
    ...

def test_function(arg):
    ...
```

- improved error reporting on collection and import errors. This makes use of a more general mechanism, namely that for custom test item/collect nodes `node.repr_failure(excinfo)` is now uniformly called so that you can override it to return a string error representation of your choice which is going to be reported as a (red) string.
- introduce '–junitprefix=STR' option to prepend a prefix to all reports in the junitxml file.

12.22.2 Bug fixes / Maintenance

- make tests and the `pytest_recwarn` plugin in particular fully compatible to Python2.7 (if you use the `recwarn` funcarg warnings will be enabled so that you can properly check for their existence in a cross-python manner).
- refine –pdb: ignore xfailed tests, unify its TB-reporting and don't display failures again at the end.
- fix assertion interpretation with the ** operator (thanks Benjamin Peterson)
- fix issue105 assignment on the same line as a failing assertion (thanks Benjamin Peterson)
- fix issue104 proper escaping for test names in junitxml plugin (thanks anonymous)
- fix issue57 -fl–looponfail to work with xpassing tests (thanks Ronny)
- fix issue92 collectonly reporter and –pastebin (thanks Benjamin Peterson)
- fix py.code.compile(source) to generate unique filenames
- fix assertion re-interp problems on PyPy, by defering code compilation to the (overridable) Frame.eval class. (thanks Amaury Forgeot)
- fix py.path.local.pyimport() to work with directories
- streamline py.path.local.mkdtemp implementation and usage
- don't print empty lines when showing junitxml-filename
- add optional boolean ignore_errors parameter to py.path.local.remove
- fix terminal writing on win32/python2.4
- py.process.cmdexec() now tries harder to return properly encoded unicode objects on all python versions
- install plain py.test/py.which scripts also for Jython, this helps to get canonical script paths in virtualenv situations
- make path.bestrelpath(path) return ".", note that when calling X.bestrelpath the assumption is that X is a directory.

- make initial conftest discovery ignore "–" prefixed arguments
- fix resultlog plugin when used in an multicpu/multihost xdist situation (thanks Jakub Gustak)
- perform distributed testing related reporting in the xdist-plugin rather than having dist-related code in the generic py.test distribution
- fix homedir detection on Windows
- ship distribute_setup.py version 0.6.13

12.23 Changes between 1.3.0 and 1.3.1

12.23.1 New features

- issue91: introduce new py.test.xfail(reason) helper to imperatively mark a test as expected to fail. Can be used from within setup and test functions. This is useful especially for parametrized tests when certain configurations are expected-to-fail. In this case the declarative approach with the @py.test.mark.xfail cannot be used as it would mark all configurations as xfail.
- issue102: introduce new –maxfail=NUM option to stop test runs after NUM failures. This is a generalization of the '-x' or '–exitfirst' option which is now equivalent to '–maxfail=1'. Both '-x' and '–maxfail' will now also print a line near the end indicating the Interruption.
- issue89: allow py.test.mark decorators to be used on classes (class decorators were introduced with python2.6) and also allow to have multiple markers applied at class/module level by specifying a list.
- improve and refine letter reporting in the progress bar: . pass f failed test s skipped tests (reminder: use for dependency/platform mismatch only) x xfailed test (test that was expected to fail) X xpassed test (test that was expected to fail but passed)

  You can use any combination of 'fsxX' with the '-r' extended reporting option. The xfail/xpass results will show up as skipped tests in the junitxml output - which also fixes issue99.
- make py.test.cmdline.main() return the exitstatus instead of raising SystemExit and also allow it to be called multiple times. This of course requires that your application and tests are properly teared down and don't have global state.

12.23.2 Fixes / Maintenance

- improved traceback presentation: - improved and unified reporting for "–tb=short" option - Errors during test module imports are much shorter, (using –tb=short style) - raises shows shorter more relevant tracebacks - –fulltrace now more systematically makes traces longer / inhibits cutting
- improve support for raises and other dynamically compiled code by manipulating python's linecache.cache instead of the previous rather hacky way of creating custom code objects. This makes it seemlessly work on Jython and PyPy where it previously didn't.
- fix issue96: make capturing more resilient against Control-C interruptions (involved somewhat substantial refactoring to the underlying capturing functionality to avoid race conditions).
- fix chaining of conditional skipif/xfail decorators - so it works now as expected to use multiple @py.test.mark.skipif(condition) decorators, including specific reporting which of the conditions lead to skipping.
- fix issue95: late-import zlib so that it's not required for general py.test startup.

- fix issue94: make reporting more robust against bogus source code (and internally be more careful when presenting unexpected byte sequences)

12.24 Changes between 1.2.1 and 1.3.0

- deprecate –report option in favour of a new shorter and easier to remember -r option: it takes a string argument consisting of any combination of 'xfsX' characters. They relate to the single chars you see during the dotted progress printing and will print an extra line per test at the end of the test run. This extra line indicates the exact position or test ID that you directly paste to the py.test cmdline in order to re-run a particular test.
- allow external plugins to register new hooks via the new pytest_addhooks(pluginmanager) hook. The new release of the pytest-xdist plugin for distributed and looponfailing testing requires this feature.
- add a new pytest_ignore_collect(path, config) hook to allow projects and plugins to define exclusion behaviour for their directory structure - for example you may define in a conftest.py this method:

```
def pytest_ignore_collect(path):
    return path.check(link=1)
```

  to prevent even a collection try of any tests in symlinked dirs.
- new pytest_pycollect_makemodule(path, parent) hook for allowing customization of the Module collection object for a matching test module.
- extend and refine xfail mechanism: `@py.test.mark.xfail(run=False)` do not run the decorated test `@py.test.mark.xfail(reason="...")` prints the reason string in xfail summaries specifiying `--runxfail` on command line virtually ignores xfail markers
- expose (previously internal) commonly useful methods: py.io.get_terminal_with() -> return terminal width py.io.ansi_print(...) -> print colored/bold text on linux/win32 py.io.saferepr(obj) -> return limited representation string
- expose test outcome related exceptions as py.test.skip.Exception, py.test.raises.Exception etc., useful mostly for plugins doing special outcome interpretation/tweaking
- (issue85) fix junitxml plugin to handle tests with non-ascii output
- fix/refine python3 compatibility (thanks Benjamin Peterson)
- fixes for making the jython/win32 combination work, note however: jython2.5.1/win32 does not provide a command line launcher, see http://bugs.jython.org/issue1491 . See pylib install documentation for how to work around.
- fixes for handling of unicode exception values and unprintable objects
- (issue87) fix unboundlocal error in assertionold code
- (issue86) improve documentation for looponfailing
- refine IO capturing: stdin-redirect pseudo-file now has a NOP close() method
- ship distribute_setup.py version 0.6.10
- added links to the new capturelog and coverage plugins

12.25 Changes between 1.2.1 and 1.2.0

- refined usage and options for "py.cleanup":

```
py.cleanup     # remove "*.pyc" and "*$py.class" (jython) files
py.cleanup -e .swp -e .cache # also remove files with these extensions
py.cleanup -s  # remove "build" and "dist" directory next to setup.py files
py.cleanup -d  # also remove empty directories
py.cleanup -a  # synonym for "-s -d -e 'pip-log.txt'"
py.cleanup -n  # dry run, only show what would be removed
```

- add a new option "py.test –funcargs" which shows available funcargs and their help strings (docstrings on their respective factory function) for a given test path
- display a short and concise traceback if a funcarg lookup fails
- early-load "conftest.py" files in non-dot first-level sub directories. allows to conveniently keep and access test-related options in a `test` subdir and still add command line options.
- fix issue67: new super-short traceback-printing option: "–tb=line" will print a single line for each failing (python) test indicating its filename, lineno and the failure value
- fix issue78: always call python-level teardown functions even if the according setup failed. This includes refinements for calling setup_module/class functions which will now only be called once instead of the previous behaviour where they'd be called multiple times if they raise an exception (including a Skipped exception). Any exception will be re-corded and associated with all tests in the according module/class scope.
- fix issue63: assume <40 columns to be a bogus terminal width, default to 80
- fix pdb debugging to be in the correct frame on raises-related errors
- update apipkg.py to fix an issue where recursive imports might unnecessarily break importing
- fix plugin links

12.26 Changes between 1.2 and 1.1.1

- moved dist/looponfailing from py.test core into a new separately released pytest-xdist plugin.
- new junitxml plugin: –junitxml=path will generate a junit style xml file which is processable e.g. by the Hudson CI system.
- new option: –genscript=path will generate a standalone py.test script which will not need any libraries installed. thanks to Ralf Schmitt.
- new option: –ignore will prevent specified path from collection. Can be specified multiple times.
- new option: –confcutdir=dir will make py.test only consider conftest files that are relative to the specified dir.
- new funcarg: "pytestconfig" is the pytest config object for access to command line args and can now be easily used in a test.
- install 'py.test' and *py.which* with a `-$VERSION` suffix to disambiguate between Python3, python2.X, Jython and PyPy installed versions.
- new "pytestconfig" funcarg allows access to test config object
- new "pytest_report_header" hook can return additional lines to be displayed at the header of a test run.
- (experimental) allow "py.test path::name1::name2::..." for pointing to a test within a test collection directly. This might eventually evolve as a full substitute to "-k" specifications.
- streamlined plugin loading: order is now as documented in customize.html: setuptools, ENV, commandline, conftest. also setuptools entry point names are turned to canonical namees ("pytest_*")
- automatically skip tests that need 'capfd' but have no os.dup

- allow pytest_generate_tests to be defined in classes as well
- deprecate usage of 'disabled' attribute in favour of pytestmark
- deprecate definition of Directory, Module, Class and Function nodes in conftest.py files. Use pytest collect hooks instead.
- collection/item node specific runtest/collect hooks are only called exactly on matching conftest.py files, i.e. ones which are exactly below the filesystem path of an item
- change: the first pytest_collect_directory hook to return something will now prevent further hooks to be called.
- change: figleaf plugin now requires –figleaf to run. Also change its long command line options to be a bit shorter (see py.test -h).
- change: pytest doctest plugin is now enabled by default and has a new option –doctest-glob to set a pattern for file matches.
- change: remove internal py._* helper vars, only keep py._pydir
- robustify capturing to survive if custom pytest_runtest_setup code failed and prevented the capturing setup code from running.
- make py.test.* helpers provided by default plugins visible early - works transparently both for pydoc and for interactive sessions which will regularly see e.g. py.test.mark and py.test.importorskip.
- simplify internal plugin manager machinery
- simplify internal collection tree by introducing a RootCollector node
- fix assert reinterpreation that sees a call containing "keyword=..."
- fix issue66: invoke pytest_sessionstart and pytest_sessionfinish hooks on slaves during dist-testing, report module/session teardown hooks correctly.
- fix issue65: properly handle dist-testing if no execnet/py lib installed remotely.
- skip some install-tests if no execnet is available
- fix docs, fix internal bin/ script generation

12.27 Changes between 1.1.1 and 1.1.0

- introduce automatic plugin registration via 'pytest11' entrypoints via setuptools' pkg_resources.iter_entry_points
- fix py.test dist-testing to work with execnet >= 1.0.0b4
- re-introduce py.test.cmdline.main() for better backward compatibility
- svn paths: fix a bug with path.check(versioned=True) for svn paths, allow '%' in svn paths, make svnwc.update() default to interactive mode like in 1.0.x and add svnwc.update(interactive=False) to inhibit interaction.
- refine distributed tarball to contain test and no pyc files
- try harder to have deprecation warnings for py.compat.* accesses report a correct location

12.28 Changes between 1.1.0 and 1.0.2

- adjust and improve docs

- remove py.rest tool and internal namespace - it was never really advertised and can still be used with the old release if needed. If there is interest it could be revived into its own tool i guess.
- fix issue48 and issue59: raise an Error if the module from an imported test file does not seem to come from the filepath - avoids "same-name" confusion that has been reported repeatedly
- merged Ronny's nose-compatibility hacks: now nose-style setup_module() and setup() functions are supported
- introduce generalized py.test.mark function marking
- reshuffle / refine command line grouping
- deprecate parser.addgroup in favour of getgroup which creates option group
- add –report command line option that allows to control showing of skipped/xfailed sections
- generalized skipping: a new way to mark python functions with skipif or xfail at function, class and modules level based on platform or sys-module attributes.
- extend py.test.mark decorator to allow for positional args
- introduce and test "py.cleanup -d" to remove empty directories
- fix issue #59 - robustify unittest test collection
- make bpython/help interaction work by adding an __all__ attribute to ApiModule, cleanup initpkg
- use MIT license for pylib, add some contributors
- remove py.execnet code and substitute all usages with 'execnet' proper
- fix issue50 - cached_setup now caches more to expectations for test functions with multiple arguments.
- merge Jarko's fixes, issue #45 and #46
- add the ability to specify a path for py.lookup to search in
- fix a funcarg cached_setup bug probably only occuring in distributed testing and "module" scope with teardown.
- many fixes and changes for making the code base python3 compatible, many thanks to Benjamin Peterson for helping with this.
- consolidate builtins implementation to be compatible with >=2.3, add helpers to ease keeping 2 and 3k compatible code
- deprecate py.compat.doctest|subprocess|textwrap|optparse
- deprecate py.magic.autopath, remove py/magic directory
- move pytest assertion handling to py/code and a pytest_assertion plugin, add "–no-assert" option, deprecate py.magic namespaces in favour of (less) py.code ones.
- consolidate and cleanup py/code classes and files
- cleanup py/misc, move tests to bin-for-dist
- introduce delattr/delitem/delenv methods to py.test's monkeypatch funcarg
- consolidate py.log implementation, remove old approach.
- introduce py.io.TextIO and py.io.BytesIO for distinguishing between text/unicode and byte-streams (uses underlying standard lib io.* if available)
- make py.unittest_convert helper script available which converts "unittest.py" style files into the simpler assert/direct-test-classes py.test/nosetests style. The script was written by Laura Creighton.
- simplified internal localpath implementation

12.29 Changes between 1.0.1 and 1.0.2

- fixing packaging issues, triggered by fedora redhat packaging, also added doc, examples and contrib dirs to the tarball.
- added a documentation link to the new django plugin.

12.30 Changes between 1.0.0 and 1.0.1

- added a 'pytest_nose' plugin which handles nose.SkipTest, nose-style function/method/generator setup/teardown and tries to report functions correctly.
- capturing of unicode writes or encoded strings to sys.stdout/err work better, also terminalwriting was adapted and somewhat unified between windows and linux.
- improved documentation layout and content a lot
- added a "–help-config" option to show conftest.py / ENV-var names for all longopt cmdline options, and some special conftest.py variables. renamed 'conf_capture' conftest setting to 'option_capture' accordingly.
- fix issue #27: better reporting on non-collectable items given on commandline (e.g. pyc files)
- fix issue #33: added –version flag (thanks Benjamin Peterson)
- fix issue #32: adding support for "incomplete" paths to wcpath.status()
- "Test" prefixed classes are *not* collected by default anymore if they have an __init__ method
- monkeypatch setenv() now accepts a "prepend" parameter
- improved reporting of collection error tracebacks
- simplified multicall mechanism and plugin architecture, renamed some internal methods and argnames

12.31 Changes between 1.0.0b9 and 1.0.0

- more terse reporting try to show filesystem path relatively to current dir
- improve xfail output a bit

12.32 Changes between 1.0.0b8 and 1.0.0b9

- cleanly handle and report final teardown of test setup
- fix svn-1.6 compat issue with py.path.svnwc().versioned() (thanks Wouter Vanden Hove)
- setup/teardown or collection problems now show as ERRORs or with big "E"'s in the progress lines. they are reported and counted separately.
- dist-testing: properly handle test items that get locally collected but cannot be collected on the remote side - often due to platform/dependency reasons
- simplified py.test.mark API - see keyword plugin documentation
- integrate better with logging: capturing now by default captures test functions and their immediate setup/teardown in a single stream

- capsys and capfd funcargs now have a readouterr() and a close() method (underlyingly py.io.StdCapture/FD objects are used which grew a readouterr() method as well to return snapshots of captured out/err)
- make assert-reinterpretation work better with comparisons not returning bools (reported with numpy from thanks maciej fijalkowski)
- reworked per-test output capturing into the pytest_iocapture.py plugin and thus removed capturing code from config object
- item.repr_failure(excinfo) instead of item.repr_failure(excinfo, outerr)

12.33 Changes between 1.0.0b7 and 1.0.0b8

- pytest_unittest-plugin is now enabled by default
- introduced pytest_keyboardinterrupt hook and refined pytest_sessionfinish hooked, added tests.
- workaround a buggy logging module interaction ("closing already closed files"). Thanks to Sridhar Ratnakumar for triggering.
- if plugins use "py.test.importorskip" for importing a dependency only a warning will be issued instead of exiting the testing process.
- many improvements to docs: - refined funcargs doc , use the term "factory" instead of "provider" - added a new talk/tutorial doc page - better download page - better plugin docstrings - added new plugins page and automatic doc generation script
- fixed teardown problem related to partially failing funcarg setups (thanks MrTopf for reporting), "pytest_runtest_teardown" is now always invoked even if the "pytest_runtest_setup" failed.
- tweaked doctest output for docstrings in py modules, thanks Radomir.

12.34 Changes between 1.0.0b3 and 1.0.0b7

- renamed py.test.xfail back to py.test.mark.xfail to avoid two ways to decorate for xfail
- re-added py.test.mark decorator for setting keywords on functions (it was actually documented so removing it was not nice)
- remove scope-argument from request.addfinalizer() because request.cached_setup has the scope arg. TOOWTDI.
- perform setup finalization before reporting failures
- apply modified patches from Andreas Kloeckner to allow test functions to have no func_code (#22) and to make "-k" and function keywords work (#20)
- apply patch from Daniel Peolzleithner (issue #23)
- resolve issue #18, multiprocessing.Manager() and redirection clash
- make __name__ == "__channelexec__" for remote_exec code

12.35 Changes between 1.0.0b1 and 1.0.0b3

- plugin classes are removed: one now defines hooks directly in conftest.py or global pytest_*.py files.

- added new pytest_namespace(config) hook that allows to inject helpers directly to the py.test.* namespace.
- documented and refined many hooks
- added new style of generative tests via pytest_generate_tests hook that integrates well with function arguments.

12.36 Changes between 0.9.2 and 1.0.0b1

- introduced new "funcarg" setup method, see doc/test/funcarg.txt
- introduced plugin architecuture and many new py.test plugins, see doc/test/plugins.txt
- teardown_method is now guaranteed to get called after a test method has run.
- new method: py.test.importorskip(mod,minversion) will either import or call py.test.skip()
- completely revised internal py.test architecture
- new py.process.ForkedFunc object allowing to fork execution of a function to a sub process and getting a result back.

XXX lots of things missing here XXX

12.37 Changes between 0.9.1 and 0.9.2

- refined installation and metadata, created new setup.py, now based on setuptools/ez_setup (thanks to Ralf Schmitt for his support).
- improved the way of making py.* scripts available in windows environments, they are now added to the Scripts directory as ".cmd" files.
- py.path.svnwc.status() now is more complete and uses xml output from the 'svn' command if available (Guido Wesdorp)
- fix for py.path.svn* to work with svn 1.5 (Chris Lamb)
- fix path.relto(otherpath) method on windows to use normcase for checking if a path is relative.
- py.test's traceback is better parseable from editors (follows the filenames:LINENO: MSG convention) (thanks to Osmo Salomaa)
- fix to javascript-generation, "py.test –runbrowser" should work more reliably now
- removed previously accidentally added py.test.broken and py.test.notimplemented helpers.
- there now is a py.__version__ attribute

12.38 Changes between 0.9.0 and 0.9.1

This is a fairly complete list of changes between 0.9 and 0.9.1, which can serve as a reference for developers.

- allowing + signs in py.path.svn urls [39106]
- fixed support for Failed exceptions without excinfo in py.test [39340]
- added support for killing processes for Windows (as well as platforms that support os.kill) in py.misc.killproc [39655]
- added setup/teardown for generative tests to py.test [40702]

- added detection of FAILED TO LOAD MODULE to py.test [40703, 40738, 40739]
- fixed problem with calling .remove() on wcpaths of non-versioned files in py.path [44248]
- fixed some import and inheritance issues in py.test [41480, 44648, 44655]
- fail to run greenlet tests when pypy is available, but without stackless [45294]
- small fixes in rsession tests [45295]
- fixed issue with 2.5 type representations in py.test [45483, 45484]
- made that internal reporting issues displaying is done atomically in py.test [45518]
- made that non-existing files are igored by the py.lookup script [45519]
- improved exception name creation in py.test [45535]
- made that less threads are used in execnet [merge in 45539]
- removed lock required for atomical reporting issue displaying in py.test [45545]
- removed globals from execnet [45541, 45547]
- refactored cleanup mechanics, made that setDaemon is set to 1 to make atexit get called in 2.5 (py.execnet) [45548]
- fixed bug in joining threads in py.execnet's servemain [45549]
- refactored py.test.rsession tests to not rely on exact output format anymore [45646]
- using repr() on test outcome [45647]
- added 'Reason' classes for py.test.skip() [45648, 45649]
- killed some unnecessary sanity check in py.test.collect [45655]
- avoid using os.tmpfile() in py.io.fdcapture because on Windows it's only usable by Administrators [45901]
- added support for locking and non-recursive commits to py.path.svnwc [45994]
- locking files in py.execnet to prevent CPython from segfaulting [46010]
- added export() method to py.path.svnurl
- fixed -d -x in py.test [47277]
- fixed argument concatenation problem in py.path.svnwc [49423]
- restore py.test behaviour that it exits with code 1 when there are failures [49974]
- don't fail on html files that don't have an accompanying .txt file [50606]
- fixed 'utestconvert.py < input' [50645]
- small fix for code indentation in py.code.source [50755]
- fix _docgen.py documentation building [51285]
- improved checks for source representation of code blocks in py.test [51292]
- added support for passing authentication to py.path.svn* objects [52000, 52001]
- removed sorted() call for py.apigen tests in favour of [].sort() to support Python 2.3 [52481]

INDEX

N

O

P

R

S

T

U

W

X