



pytest Documentation

Release 2.6.4

holger krekel, trainer and consultant, <http://merlinux.eu>

October 24, 2014

| | | |
|----------|---|-----------|
| 1 | Getting started basics | 3 |
| 1.1 | pytest: helps you write better programs | 3 |
| 1.2 | Installation and Getting Started | 4 |
| 1.3 | Usage and Invocations | 7 |
| 1.4 | Good Integration Practises | 11 |
| 1.5 | Project examples | 16 |
| 1.6 | Some Issues and Questions | 17 |
| 2 | pytest reference documentation | 21 |
| 2.1 | Pytest API and builtin fixtures | 21 |
| 2.2 | Basic test configuration | 25 |
| 2.3 | The writing and reporting of assertions in tests | 27 |
| 2.4 | pytest fixtures: explicit, modular, scalable | 31 |
| 2.5 | Fixture functions using “yield” / context manager integration | 41 |
| 2.6 | Parametrizing fixtures and test functions | 43 |
| 2.7 | classic xunit-style setup | 46 |
| 2.8 | Capturing of the stdout/stderr output | 48 |
| 2.9 | Monkeypatching/mockng modules and environments | 49 |
| 2.10 | xdist: pytest distributed testing plugin | 51 |
| 2.11 | Temporary directories and files | 54 |
| 2.12 | Marking test functions with attributes | 55 |
| 2.13 | Skip and xfail: dealing with tests that can not succeed | 56 |
| 2.14 | Asserting deprecation and other warnings | 61 |
| 2.15 | Support for unittest.TestCase / Integration of fixtures | 61 |
| 2.16 | Running tests written for nose | 64 |
| 2.17 | Doctest integration for modules and test files | 65 |
| 3 | Working with plugins and conftest files | 67 |
| 3.1 | conftest.py: local per-directory plugins | 67 |
| 3.2 | Installing External Plugins / Searching | 68 |
| 3.3 | Writing a plugin by looking at examples | 68 |
| 3.4 | Making your plugin installable by others | 68 |
| 3.5 | Plugin discovery order at tool startup | 69 |
| 3.6 | Requiring/Loading plugins in a test module or conftest file | 69 |
| 3.7 | Accessing another plugin by name | 70 |
| 3.8 | Finding out which plugins are active | 70 |
| 3.9 | Deactivating / unregistering a plugin by name | 70 |
| 4 | pytest default plugin reference | 71 |
| 5 | pytest hook reference | 73 |

| | | |
|-----------|--|------------|
| 5.1 | Hook specification and validation | 73 |
| 5.2 | Initialization, command line and configuration hooks | 73 |
| 5.3 | Generic “runtest” hooks | 74 |
| 5.4 | Collection hooks | 74 |
| 5.5 | Reporting hooks | 75 |
| 5.6 | Debugging/Interaction hooks | 75 |
| 5.7 | Declaring new hooks | 75 |
| 5.8 | Using hooks from 3rd party plugins | 76 |
| 6 | Reference of objects involved in hooks | 77 |
| 7 | List of Third-Party Plugins | 81 |
| 8 | Usages and Examples | 85 |
| 8.1 | Demo of Python failure reports with pytest | 85 |
| 8.2 | Basic patterns and examples | 95 |
| 8.3 | Parametrizing tests | 107 |
| 8.4 | Working with custom markers | 119 |
| 8.5 | A session-fixture which can look at all collected tests | 127 |
| 8.6 | Changing standard (Python) test discovery | 129 |
| 8.7 | Working with non-python tests | 131 |
| 9 | Talks and Tutorials | 135 |
| 9.1 | Talks and blog postings | 135 |
| 9.2 | Older conference talks and tutorials | 136 |
| 10 | Contributing | 137 |
| 10.1 | Types of contributions | 137 |
| 10.2 | Preparing Pull Requests on Bitbucket | 138 |
| 11 | pytest-2.3: reasoning for fixture/funcarg evolution | 141 |
| 11.1 | Shortcomings of the previous <code>pytest_funcarg__</code> mechanism | 141 |
| 11.2 | Direct scoping of fixture/funcarg factories | 142 |
| 11.3 | Direct parametrization of funcarg resource factories | 142 |
| 11.4 | No <code>pytest_funcarg__</code> prefix when using <code>@fixture</code> decorator | 142 |
| 11.5 | solving per-session setup / autouse fixtures | 143 |
| 11.6 | funcargs/fixture discovery now happens at collection time | 143 |
| 11.7 | Conclusion and compatibility notes | 143 |
| 12 | Release announcements | 145 |
| 12.1 | pytest-2.6.3: fixes and little improvements | 145 |
| 12.2 | Changes 2.6.3 | 145 |
| 12.3 | pytest-2.6.2: few fixes and <code>cx_freeze</code> support | 145 |
| 12.4 | pytest-2.6.1: fixes and new <code>xfail</code> feature | 146 |
| 12.5 | Changes 2.6.1 | 146 |
| 12.6 | pytest-2.6.0: shorter tracebacks, new warning system, test runner compat | 147 |
| 12.7 | pytest-2.5.2: fixes | 149 |
| 12.8 | pytest-2.5.1: fixes and new home page styling | 150 |
| 12.9 | pytest-2.5.0: now down to ZERO reported bugs! | 150 |
| 12.10 | pytest-2.4.2: colorama on windows, plugin/tmpdir fixes | 152 |
| 12.11 | pytest-2.4.1: fixing three regressions compared to 2.3.5 | 153 |
| 12.12 | pytest-2.4.0: new fixture features/hooks and bug fixes | 153 |
| 12.13 | pytest-2.3.5: bug fixes and little improvements | 156 |
| 12.14 | pytest-2.3.4: stabilization, more flexible selection via “-k expr” | 157 |
| 12.15 | pytest-2.3.3: integration fixes, py24 suport, <code>*/**</code> shown in traceback | 158 |

| | | |
|-----------|---|------------|
| 12.16 | pytest-2.3.2: some fixes and more traceback-printing speed | 159 |
| 12.17 | pytest-2.3.1: fix regression with factory functions | 160 |
| 12.18 | pytest-2.3: improved fixtures / better unittest integration | 160 |
| 12.19 | pytest-2.2.4: bug fixes, better junitxml/unittest/python3 compat | 162 |
| 12.20 | pytest-2.2.2: bug fixes | 163 |
| 12.21 | pytest-2.2.1: bug fixes, perfect teardowns | 163 |
| 12.22 | py.test 2.2.0: test marking++, parametrization++ and duration profiling | 164 |
| 12.23 | py.test 2.1.3: just some more fixes | 166 |
| 12.24 | py.test 2.1.2: bug fixes and fixes for jython | 166 |
| 12.25 | py.test 2.1.1: assertion fixes and improved junitxml output | 167 |
| 12.26 | py.test 2.1.0: perfected assertions and bug fixes | 167 |
| 12.27 | py.test 2.0.3: bug fixes and speed ups | 168 |
| 12.28 | py.test 2.0.2: bug fixes, improved xfail/skip expressions, speed ups | 169 |
| 12.29 | py.test 2.0.1: bug fixes | 170 |
| 12.30 | py.test 2.0.0: asserts++, unittest++, reporting++, config++, docs++ | 171 |
| 13 | Changelog history | 175 |
| 13.1 | 2.6.4 | 175 |
| 13.2 | 2.6.3 | 175 |
| 13.3 | 2.6.2 | 175 |
| 13.4 | 2.6.1 | 176 |
| 13.5 | 2.6 | 176 |
| 13.6 | 2.5.2 | 178 |
| 13.7 | 2.5.1 | 178 |
| 13.8 | 2.5.0 | 178 |
| 13.9 | v2.4.2 | 180 |
| 13.10 | v2.4.1 | 180 |
| 13.11 | v2.4 | 180 |
| 13.12 | v2.3.5 | 183 |
| 13.13 | v2.3.4 | 183 |
| 13.14 | v2.3.3 | 184 |
| 13.15 | v2.3.2 | 184 |
| 13.16 | v2.3.1 | 185 |
| 13.17 | v2.3.0 | 185 |
| 13.18 | v2.2.4 | 186 |
| 13.19 | v2.2.3 | 186 |
| 13.20 | v2.2.2 | 187 |
| 13.21 | v2.2.1 | 187 |
| 13.22 | v2.2.0 | 187 |
| 13.23 | v2.1.3 | 188 |
| 13.24 | v2.1.2 | 188 |
| 13.25 | v2.1.1 | 188 |
| 13.26 | v2.1.0 | 189 |
| 13.27 | v2.0.3 | 189 |
| 13.28 | v2.0.2 | 189 |
| 13.29 | v2.0.1 | 190 |
| 13.30 | v2.0.0 | 191 |
| 13.31 | v1.3.4 | 192 |
| 13.32 | v1.3.3 | 192 |
| 13.33 | v1.3.2 | 192 |
| 13.34 | v1.3.1 | 194 |
| 13.35 | v1.3.0 | 194 |
| 13.36 | v1.2.0 | 195 |
| 13.37 | v1.1.1 | 196 |

| | |
|--------------------------|------------|
| 13.38 v1.1.0 | 197 |
| 13.39 v1.0.2 | 197 |
| 13.40 v1.0.2 | 198 |
| 13.41 v1.0.1 | 199 |
| 13.42 v1.0.0 | 199 |
| 13.43 v1.0.0b9 | 199 |
| 13.44 v1.0.0b8 | 200 |
| 13.45 v1.0.0b7 | 200 |
| 13.46 v1.0.0b3 | 200 |
| 13.47 v1.0.0b1 | 200 |
| 13.48 v0.9.2 | 201 |
| 13.49 v0.9.1 | 201 |
| Index | 203 |

Note: [improving your automated testing with pytest](#), July 25th 2014, Berlin, Germany
[professional testing with pytest and tox](#), 24-26th November 2014, Freiburg, Germany

Download latest version as PDF

GETTING STARTED BASICS

1.1 pytest: helps you write better programs

a mature full-featured Python testing tool

- runs on Posix/Windows, Python 2.6-3.4, PyPy and (possibly still) Jython-2.5.1
- **well tested** with more than a thousand tests against itself
- **strict backward compatibility policy** for safe pytest upgrades
- *comprehensive online* and PDF documentation
- many *third party plugins* and *builtin helpers*,
- used in *many small and large projects and organisations*
- comes with many *tested examples*

provides easy no-boilerplate testing

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

scales from simple unit to complex functional testing

- *modular parametrizeable fixtures* (new in 2.3, continously improved)
- *parametrized test functions*
- *Marking test functions with attributes*
- *Skip and xfail: dealing with tests that can not succeed* (improved in 2.4)
- *distribute tests to multiple CPUs through xdist plugin*
- *continuously re-run failing tests*
- flexible *Conventions for Python test discovery*

integrates with other testing methods and tools:

- multi-paradigm: pytest can run nose, unittest and doctest 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 generating *test coverage reports*
- 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 plugin
- it is easy to add command line options or customize existing behaviour

1.2 Installation and Getting Started

Pythons: Python 2.6-3.4, Jython, PyPy-2.3

Platforms: Unix/Posix and Windows

PyPI package name: `pytest`

dependencies: `py`, `colorama` (Windows), `argparse` (py26).

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 pytest version 2.6.4, imported from /home/hpk/p/pytest/.tox/regen/lib/python3.4/site-packages
```

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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 =====
```

pytest 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
.
1 passed in 0.00 seconds
```

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 object at 0x2b9209071470>

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

test_class.py:8: AssertionError
1 failed, 1 passed in 0.01 seconds
```

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 pytest 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-108/test_needsfiles0')

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

test_tmpdir.py:3: AssertionError
----- Captured stdout call -----
/tmp/pytest-108/test_needsfiles0
1 failed in 0.02 seconds
```

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
- *pytest reference documentation* for documentation and examples on using pytest
- *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.

Install [setuptools](#) to get `easy_install` which allows to install `.egg` binary format packages in addition to source-based ones.

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 your tests with Jython using `pytest`.

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 somepath
py.test -k stringexpr # only run tests with names that match the
                    # the "string expression", e.g. "MyClass and not method"
                    # will select TestMyClass.test_something
                    # but not TestMyClass.test_method_simple
py.test test_mod.py::test_func # only run tests that match the "node ID",
                    # e.g "test_mod.py::test_func" will select
                    # only test_func in test_mod.py
```

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**. `pytest` 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 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
```

Prior to pytest version 2.0.0 you could only enter **PDB** tracing if you disabled capturing on the command line via `py.test -s`. In later versions, pytest automatically disables its output capture when you enter **PDB** tracing:

- Output capture in other tests is not affected.
- Any prior test output that has already been captured and will be processed as such.
- Any later output produced within the same test will not be captured and will instead get sent directly to `sys.stdout`. Note that this holds true even for test output occurring after you exit the interactive **PDB** tracing session and continue with the regular test run.

Since pytest version 2.4.0 you can also use the native Python `import pdb; pdb.set_trace()` call to enter **PDB** tracing without having to use the `pytest.set_trace()` wrapper or explicitly disable pytest's output capturing 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 Disabling plugins

To disable loading specific plugins at invocation time, use the `-p` option together with the prefix `no:`.

Example: to disable loading the plugin `doctest`, which is responsible for executing `doctest` tests from text files, invoke `py.test` like this:

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

1.3.13 Calling pytest from Python code

New in version 2.0.

You can invoke `pytest` 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 `pip` (or `easy_install`) for installing your application and any dependencies as well as the `pytest` package itself. This way you will get an isolated and reproducible environment. Given you have installed `virtualenv` and execute it from the command line, here is an example session for unix or windows:

```
virtualenv .    # create a virtualenv directory in the current directory

source bin/activate  # on unix

scripts/activate    # on Windows
```

We can now install `pytest`:

```
pip install pytest
```

Due to the `activate` step above the `pip` will come from the `virtualenv` directory and install any package into the isolated virtual environment.

1.4.2 Choosing a test layout / import rules

`pytest` supports two common test layouts:

- putting tests into an extra directory outside your actual application code, useful if you have many functional tests or for other reasons want to keep tests separate from actual application code (often a good idea):

```
setup.py    # your distutils/setuptools Python package metadata
mypkg/
  __init__.py
  appmodule.py
tests/
  test_app.py
...
```

- inlining test directories into your application package, useful if you have direct relation between (unit-)test and application modules and want to distribute your tests along with your application:

```
setup.py    # your distutils/setuptools Python package metadata
mypkg/
  __init__.py
  appmodule.py
  ...
  test/
    test_app.py
  ...
```

Important notes relating to both schemes:

- **make sure that “`mypkg`” is importable**, for example by typing once:

```
pip install -e .    # install package using setup.py in editable mode
```

- **avoid “`__init__.py`” files in your test directories**. This way your tests can run easily against an installed version of `mypkg`, independently from the installed package if it contains the tests or not.

- With inlined tests you might put `__init__.py` into test directories and make them installable as part of your application. Using the `py.test --pyargs mypkg` invocation pytest will discover where mypkg is installed and collect tests from there. With the “external” test you can still distribute tests but they will not be installed or become importable.

Typically you can run tests by pointing to test directories or modules:

```
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
...
```

Because of the above `editable install` mode you can change your source code (both tests and the app) and rerun tests at will. Once you are done with your work, you can [use tox](#) to make sure that the package is really correct and tests pass in all required configurations.

Note: You can use Python3 namespace packages (PEP420) for your application but pytest will still perform [test package name](#) discovery based on the presence of `__init__.py` files. If you use one of the two recommended file system layouts above but leave away the `__init__.py` files from your directories it should just work on Python3.3 and above. From “inlined tests”, however, you will need to use absolute imports for getting at your application code.

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

- determine `basedir`: this is the first “upward” (towards the root) directory not containing an `__init__.py`. If e.g. both `a` and `b` contain an `__init__.py` file then the parent directory of `a` will become the `basedir`.
- 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.

1.4.3 Use tox and Continuous Integration servers

If you frequently release code and want to make sure that your actual package passes all tests you may want to look into [tox](#), the virtualenv test automation tool and its [pytest support](#). Tox helps you to setup virtualenv environments with pre-defined dependencies and then executing a pre-configured test command with options. It will run tests against the installed package and not against your source code checkout, helping to detect packaging glitches.

If you want to use [Jenkins](#) you can use the `--junitxml=PATH` option to create a JUnitXML file that [Jenkins](#) can pick up and generate reports.

1.4.4 Create a pytest standalone script

If you are a maintainer or application developer and want people who don’t deal with python much to easily run tests you may generate a standalone `pytest` script:

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

This generates a `runtests.py` script which is a fully functional basic `pytest` 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
```

1.4.5 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 `pytest` 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 `pytest` 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.

1.4.6 Integration with setuptools test commands

Setuptools supports writing our own Test command for invoking `pytest`. Most often it is better to use `tox` instead, but here is how you can get started with setuptools integration:

```
import sys

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

```
class PyTest(TestCommand):
    user_options = [('pytest-args=', 'a', "Arguments to pass to py.test")]

    def initialize_options(self):
        TestCommand.initialize_options(self)
        self.pytest_args = []

    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.pytest_args)
        sys.exit(errno)

setup(
    #...,
    tests_require=['pytest'],
    cmdclass = {'test': PyTest},
)
```

Now if you run:

```
python setup.py test
```

this will download `pytest` if needed and then run your tests as you would expect it to. You can pass a single string of arguments using the `--pytest-args` or `-a` command-line option. For example:

```
python setup.py test -a "--durations=5"
```

is equivalent to running `py.test --durations=5`.

1.4.7 Conventions for Python test discovery

pytest 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 [test 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, `pytest` also discovers tests using the standard `unittest.TestCase` subclassing technique.



Alex Gaynor

@alex_gaynor

py.test is pretty much the best thing ever. Not entirely sure why you'd use anything else.



theuni

@theuni

Switched test runner for [#batou](#) to [#pytest](#) picked up everything correctly, no failing tests. Correct skips. Kudos to [@hpk42](#) Very impressed.



David Cramer

@zeeg

Converting all my projects to py.test. Not sure why it took me so long. /cc [@hpk42](#)



Vladimir Keleshev

@keleshev

Seriously, [#pytest](#) is among my top-5 reasons to use [#python](#).

1.5 Project examples

Here are some examples of projects using `pytest` (please send notes via *contact*):

- [PyPy](#), Python with a JIT compiler, running over 21000 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 `pytest` 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 pytest

- [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
- [Laboratory of Bioinformatics](#), Warsaw
- [merlinux](#), Germany
- many more ... (please be so kind to send a note via *contact*)

1.6 Some Issues and Questions

Note: This FAQ is here only mostly for historic reasons. Checkout [pytest Q&A at Stackoverflow](#) for many questions and answers related to pytest and/or use *contact channels* to get help.

1.6.1 On naming, nosetests, licensing and magic

How does pytest relate to nose and unittest?

`pytest` 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 `pytest`. `nose` was originally created as a clone of `pytest` when `pytest` 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 pytest relate to twisted's trial?

Since some time `pytest` 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 pytest 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 pytest? (historic notes)

Around 2007 (version 0.8) some people thought that `pytest` was using too much “magic”. It had been part of the `pylib` which contains a lot of unrelated 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 release 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, `pytest` explicitly 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 losing assertions in test modules.

`pytest` contains a second, mostly obsolete, assert debugging technique, invoked via `--assert=reinterpret`, activated by default on Python-2.5: When an `assert` statement fails, `pytest` 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. `pytest` 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 pytest fixtures, parametrized tests

Is using pytest fixtures 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 monkey-patch, the `tmpdir` or capture funcargs) because the support code can register setup/teardown functions in a managed class/module/function scope.

Can I yield multiple values from a fixture function?

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

- 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.
- 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.

However, with pytest-2.3 you can use the *Fixtures as Function arguments* 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 pytest interaction with other packages

Issues with pytest, multiprocessing and setuptools?

On windows the multiprocessing 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.

As of middle 2013, there shouldn't be a problem anymore when you use the standard setuptools (note that distribute has been merged back into setuptools which is now shipped directly with virtualenv).

PYTEST 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.

This helper produces a `py.code.ExceptionInfo()` object.

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 ...>
```

Similar to caught exception objects in Python, explicitly clearing local references to returned `py.code.ExceptionInfo` objects can help the Python interpreter speed up its garbage collection.

Clearing those references breaks a reference cycle (`ExceptionInfo` → caught exception → frame stack raising the exception → current frame stack → local variables → `ExceptionInfo`) which makes Python keep all objects referenced from that cycle (including all local variables in the current frame) alive until the next cyclic garbage collection run. See the official Python `try` statement documentation for more detailed information.

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 `pytest.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 at least “minversion” as its `__version__` attribute. If no `minversion` is specified the a skip is only triggered if the module can not be imported. Note that version comparison only works with simple version strings like “1.2.3” but not “1.2.3.dev1” or others.

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, ids=None*)
(return a) decorator to mark a fixture factory function.

This decorator can be used (with 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.
- **ids** – list of string ids each corresponding to the params so that they are part of the test id. If no ids are provided they will be generated automatically from the params.

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 `pytest.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 `capfd.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.

in 0.00 seconds

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

pytest 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 `pytest` 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 `pytest`. 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 `pytest` 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'`, `'_darcs'`, `'{arch}'`, `'*.egg'`. Setting a `norecursedirs` 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 `pytest` 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. Note that this has no effect on methods that live on a `unittest.TestCase` derived class.

See [Changing naming conventions](#) for examples.

2.3 The writing and reporting of assertions in tests

2.3.1 Asserting with the `assert` statement

`pytest` 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 =====
```

`pytest` 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 pytest](#)). 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

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

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

```
def test_recursion_depth():
    with pytest.raises(RuntimeError) as excinfo:
        def f():
            f()
        f()
    assert 'maximum recursion' in str(excinfo.value)
```

`excinfo` is a `py.code.ExceptionInfo` instance, which is a wrapper around the actual exception raised. The main attributes of interest are `.type`, `.value` and `.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*.

Note that it is also possible to specify a “raises” argument to `pytest.mark.xfail`, which checks that the test is failing in a more specific way than just having any exception raised:

```
@pytest.mark.xfail(raises=IndexError)
def test_f():
    f()
```

Using `pytest.raises` is likely to be better for cases where you are testing exceptions your own code is deliberately raising, whereas using `@pytest.mark.xfail` with a check function is probably better for something like documenting unfixed bugs (where the test describes what “should” happen) or bugs in dependencies.

2.3.3 Making use of context-sensitive comparisons

New in version 2.0.

`pytest` 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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'
E           Use -v to get the full diff

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 slightly, 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
1 failed in 0.01 seconds
```

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, `pytest` configuration, and Python version being used to run `pytest`. 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, `pytest` rewrites assert statements in test modules. Rewritten assert statements put introspection information into the assertion failure message. `pytest` 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: `pytest` 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, `pytest` falls back on assert reinterpretation. In assert reinterpretation, `pytest` walks the frame of the function containing the assert statement to discover sub-expression results of the failing assert statement. You can force `pytest` 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 pytest'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 2.0/2.3/2.4.

The [purpose of test fixtures](#) is to provide a fixed baseline upon which tests can reliably and repeatedly execute. pytest 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 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.

Note: pytest-2.4 introduced an additional experimental [yield fixture mechanism](#) for easier context manager integration and more linear writing of teardown code.

2.4.1 Fixtures as Function arguments

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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 1 items

test_smtpsimple.py F

===== FAILURES =====
_____ test_ehlo _____

smtp = <smtpplib.SMTP object at 0x2b88f2d1b0b8>

    def test_ehlo(smtp):
        response, msg = smtp.ehlo()
        assert response == 250
>         assert "merlinux" in msg
E         TypeError: Type str doesn't support the buffer API

test_smtpsimple.py:11: TypeError
===== 1 failed in 0.28 seconds =====
```

In the failure traceback we see that the test function was called with a `smtp` argument, the `smtpplib.SMTP()` instance created by the fixture function. The test function fails on our deliberate `assert 0`. Here is the exact protocol used by pytest 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 docs today. 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” remain as the main way as they allow to directly state the dependencies of a test function.

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 Sharing a fixture across tests in a module (or class/session)

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 puts the fixture function into a separate `conftest.py` file so that tests from multiple 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 2 items

test_module.py FF

===== FAILURES =====
_____ test_ehlo _____

smtp = <smtplib.SMTP object at 0x2b29b71bd8d0>

    def test_ehlo(smtp):
        response = smtp.ehlo()
        assert response[0] == 250
>         assert "merlinux" in response[1]
```

```
E      TypeError: Type str doesn't support the buffer API

test_module.py:5: TypeError
_____ test_noop _____

smtp = <smtpplib.SMTP object at 0x2b29b71bd8d0>

    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 0.28 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 fixture finalization / executing teardown code

pytest supports execution of fixture specific finalization code when the fixture goes out of scope. By accepting a `request` object into your fixture function you can call its `request.addfinalizer` one or multiple times:

```
# content of conftest.py

import smtpplib
import pytest

@pytest.fixture(scope="module")
def smtp(request):
    smtp = smtpplib.SMTP("merlinux.eu")
    def fin():
        print("teardown smtp")
        smtp.close()
    request.addfinalizer(fin)
    return smtp # provide the fixture value
```

The `fin` function will execute when the last test using the fixture in the module has finished execution.

Let's execute it:

```
$ py.test -s -q --tb=no
FFteardown smtp

2 failed in 0.21 seconds
```

We see that the `smtp` instance is finalized after the two tests finished execution. Note that if we decorated our fixture function with `scope='function'` then fixture setup and cleanup would occur around each single test. In either case the test module itself does not need to change or know about these details of fixture setup.

2.4.5 Fixtures can introspect the requesting test context

Fixture function can accept the `request` object to introspect the “requesting” test function, class or module context. Further extending the previous `smtp` fixture example, let’s read an optional server URL from the test module which uses our fixture:

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

@pytest.fixture(scope="module")
def smtp(request):
    server = getattr(request.module, "smtpserver", "merlinlinux.eu")
    smtp = smtplib.SMTP(server)

    def fin():
        print("finalizing %s (%s)" % (smtp, server))
        smtp.close()

    return smtp
```

We use the `request.module` attribute to optionally obtain an `smtpserver` attribute from the test module. If we just execute again, nothing much has changed:

```
$ py.test -s -q --tb=no
FF
2 failed in 0.19 seconds
```

Let’s quickly create another test module that actually sets the server URL in its module namespace:

```
# 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, b'mail.python.org')
E   assert 0
```

voila! The `smtp` fixture function picked up our mail server name from the module namespace.

2.4.6 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 object at 0x2b6b796568d0>

    def test_ehlo(smtp):
        response = smtp.ehlo()
        assert response[0] == 250
>         assert "merlinux" in response[1]
E         TypeError: Type str doesn't support the buffer API

test_module.py:5: TypeError
_____ test_noop[merlinux.eu] _____

smtp = <smtplib.SMTP object at 0x2b6b796568d0>

    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 object at 0x2b6b79656780>

    def test_ehlo(smtp):
        response = smtp.ehlo()
        assert response[0] == 250
>         assert "merlinux" in response[1]
E         TypeError: Type str doesn't support the buffer API

test_module.py:5: TypeError
----- Captured stdout setup -----
finalizing <smtplib.SMTP object at 0x2b6b796568d0>
_____ test_noop[mail.python.org] _____
```

```
smtp = <smtpplib.SMTP object at 0x2b6b79656780>
```

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

```
test_module.py:11: AssertionError
4 failed in 7.02 seconds
```

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.

2.4.7 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 2 items

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

===== 2 passed in 6.63 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.

2.4.8 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 %s" % param)
    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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 8 items

test_module.py::test_0[1]    test0 1
PASSED
test_module.py::test_0[2]    test0 2
PASSED
test_module.py::test_1[mod1] create mod1
    test1 mod1
PASSED
test_module.py::test_2[1-mod1] test2 1 mod1
PASSED
test_module.py::test_2[2-mod1] test2 2 mod1
PASSED
test_module.py::test_1[mod2] create mod2
    test1 mod2
PASSED
test_module.py::test_2[1-mod2] test2 1 mod2
PASSED
test_module.py::test_2[2-mod2] test2 2 mod2
PASSED

===== 8 passed in 0.01 seconds =====
```

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.

2.4.9 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 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
..
2 passed in 0.01 seconds
```

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
```

2.4.10 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
..
2 passed in 0.01 seconds
```

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.

2.4.11 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 Fixture functions using “yield” / context manager integration

New in version 2.4.

pytest-2.4 allows fixture functions to seamlessly use a `yield` instead of a `return` statement to provide a fixture value while otherwise fully supporting all other fixture features.

Note: “yielding” fixture values is an experimental feature and its exact declaration may change later but earliest in a 2.5 release. You can thus safely use this feature in the 2.4 series but may need to adapt later. Test functions themselves will not need to change (as a general feature, they are ignorant of how fixtures are setup).

Let’s look at a simple standalone-example using the new `yield` syntax:

```
# content of test_yield.py

import pytest

@pytest.yield_fixture
def passwd():
    print ("\nsetup before yield")
    f = open("/etc/passwd")
    yield f.readlines()
    print ("teardown after yield")
    f.close()

def test_has_lines(passwd):
    print ("test called")
    assert passwd
```

In contrast to *finalization through registering callbacks*, our fixture function used a `yield` statement to provide the lines of the `/etc/passwd` file. The code after the `yield` statement serves as the teardown code, avoiding the indirection of registering a teardown callback function.

Let’s run it with output capturing disabled:

```
$ py.test -q -s test_yield.py

setup before yield
test called
.teardown after yield

1 passed in 0.01 seconds
```

We can also seamlessly use the new syntax with `with` statements. Let’s simplify the above `passwd` fixture:

```
# content of test_yield2.py

import pytest

@pytest.yield_fixture
def passwd():
    with open("/etc/passwd") as f:
        yield f.readlines()

def test_has_lines(passwd):
    assert len(passwd) >= 1
```

The file `f` will be closed after the test finished execution because the Python `file` object supports finalization when the `with` statement ends.

Note that the new syntax is fully integrated with using `scope`, `params` and other fixture features. Changing existing fixture functions to use `yield` is thus straight forward.

2.5.1 Discussion and future considerations / feedback

The `yield`-syntax has been discussed by pytest users extensively. In general, the advantages of the using a `yield` fixture syntax are:

- easy provision of fixtures in conjunction with context managers.
- no need to register a callback, providing for more synchronous control flow in the fixture function. Also there is no need to accept the `request` object into the fixture function just for providing finalization code.

However, there are also limitations or foreseeable irritations:

- usually `yield` is used for producing multiple values. But fixture functions can only yield exactly one value. Yielding a second fixture value will get you an error. It’s possible we can evolve pytest to allow for producing multiple values as an alternative to current parametrization. For now, you can just use the normal *fixture parametrization* mechanisms together with `yield`-style fixtures.
- the `yield` syntax is similar to what `contextlib.contextmanager()` decorated functions provide. With pytest fixture functions, the “after yield” part will always be invoked, independently from the exception status of the test function which uses the fixture. The pytest behaviour makes sense if you consider that many different test functions might use a module or session scoped fixture. Some test functions might raise exceptions and others not, so how could pytest re-raise a single exception at the `yield` point in the fixture function?
- lastly `yield` introduces more than one way to write fixture functions, so what’s the obvious way to a newcomer? Newcomers reading the docs will see feature examples using the `return` style so should use that, if in doubt. Others can start experimenting with writing `yield`-style fixtures and possibly help evolving them further.

If you want to feedback or participate in the ongoing discussion, please join our *contact channels*. you are most welcome.

2.6 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.

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

New in version 2.2,: improved in 2.4

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 `(input, expected)` tuples so that the `test_eval` function will run three times using them in turn:

```
$ py.test
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 designed in this example, 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 you could also use the `parametrize` marker on a class or a module (see [Marking test functions with attributes](#)) which would invoke several functions with the argument sets.

It is also possible to mark individual test instances within `parametrize`, for example with the builtin `mark.xfail`:

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

Let's run this:

```
$ py.test
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 3 items

test_expectation.py ..x

===== 2 passed, 1 xfailed in 0.01 seconds =====
```

The one parameter set which caused a failure previously now shows up as an “xfailed (expected to fail)” test.

Note: In versions prior to 2.4 one needed to specify the argument names as a tuple. This remains valid but the simpler “name1, name2, ...” comma-separated-string syntax is now advertised first because it's easier to write and produces less line noise.

2.6.2 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 `pytest` 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
..
2 passed in 0.01 seconds
```

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 0x2ae3eb376c00>()
E         + where <built-in method isalpha of str object at 0x2ae3eb376c00> = '!'.isalpha

test_strings.py:3: AssertionError
1 failed in 0.01 seconds
```

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 list:

```
$ py.test -q -rs test_strings.py
s
===== short test summary info =====
SKIP [1] /home/hpk/p/pytest/.tox/regen/lib/python3.4/site-packages/_pytest/python.py:1139: got empty
1 skipped in 0.01 seconds
```

For further examples, you might want to look at [more parametrization examples](#).

2.6.3 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** – a comma-separated string denoting one or more argument names, or a list/tuple of argument strings.
- **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.

2.7 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. You can mix both fixture mechanisms in the same file but unittest-based test methods cannot receive fixture arguments.

Note: As of `pytest-2.4`, `teardownX` functions are not called if `setupX` existed and failed/was skipped. This harmonizes behaviour across all major python testing tools.

2.7.1 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.
    """
```

2.7.2 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.
    """
```

2.7.3 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.

2.8 Capturing of the stdout/stderr output

2.8.1 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.

2.8.2 Setting capturing methods or disabling capturing

There are two ways in which `pytest` 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
```

2.8.3 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 setup -----
setting up <function test_func2 at 0x2af94beald08>
===== 1 failed, 1 passed in 0.01 seconds =====

```

2.8.4 Accessing captured output from a test function

The `capsys` and `capfd` fixtures allow to access stdout/stderr output created during test execution. 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 pytest's own per-test capturing.

If you want to capture on filedescriptor level you can use the `capfd` function argument which offers the exact same interface but allows to also capture output from libraries or subprocesses that directly write to operating system level output streams (FD1 and FD2).

2.9 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.

2.9.1 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 a function that calls it. After the test function finishes the `os.path.expanduser` modification will be undone.

2.9.2 example: preventing “requests” from remote operations

If you want to prevent the “requests” library from performing http requests in all your tests, you can do:

```
# content of conftest.py

import pytest
@pytest.fixture(autouse=True)
def no_requests(monkeypatch):
    monkeypatch.delattr("requests.sessions.Session.request")
```

This `autouse` fixture will be executed for each test function and it will delete the method `request.session.Session.request` so that any attempts within tests to create http requests will fail.

2.9.3 example: setting an attribute on some class

If you need to patch out `os.getcwd()` to return an artificial value:

```
def test_some_interaction(monkeypatch):
    monkeypatch.setattr("os.getcwd", lambda: "/")
```

which is equivalent to the long form:

```
def test_some_interaction(monkeypatch):
    import os
    monkeypatch.setattr(os, "getcwd", lambda: "/")
```

2.9.4 Method reference of the monkeypatch function argument

class `monkeypatch`

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

`setattr` (*target*, *name*, *value*=<notset>, *raising*=*True*)

set attribute value on target, memorizing the old value. By default raise `AttributeError` if the attribute did not exist.

For convenience you can specify a string as *target* which will be interpreted as a dotted import path, with the last part being the attribute name. Example: `monkeypatch.setattr("os.getcwd", lambda x: "/")` would set the `getcwd` function of the `os` module.

The *raising* value determines if the `setattr` should fail if the attribute is not already present (defaults to `True` which means it will raise).

`delattr` (*target*, *name*=<notset>, *raising*=*True*)

delete attribute name from target, by default raise `AttributeError` if the attribute did not previously exist.

If no *name* is specified and *target* is a string it will be interpreted as a dotted import path with the last part being the attribute name.

If *raising* is set to `false`, the attribute is allowed to not pre-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` if 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.

2.10 xdist: pytest distributed testing plugin

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

- **Looponfail:** run your tests repeatedly in a subprocess. After each run, `pytest` 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, `pytest` 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.

2.10.1 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
```

2.10.2 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 simultaneously.

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 `pytest` 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.

2.11 Temporary directories and files

2.11.1 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 1 items

test_tmpdir.py F

===== FAILURES =====
_____ test_create_file _____

tmpdir = local('/tmp/pytest-109/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 =====
```

2.11.2 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, `pytest` 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.

2.12 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.

2.12.1 API reference for mark related objects

class `MarkGenerator`

Factory for `MarkDecorator` objects - exposed as a `pytest.mark` singleton instance. Example:

```
import py
@pytest.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 = pytest.mark.NAME # simple MarkDecorator
mark2 = pytest.mark.NAME(name1=value) # parametrized MarkDecorator
```

and can then be applied as decorators to test functions:

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

When a `MarkDecorator` instance is called it does the following:

1. If called with a single class as its only positional argument and no additional keyword arguments, it attaches itself to the class so it gets applied automatically to all test cases found in that class.
2. If called with a single function as its only positional argument and no additional keyword arguments, it attaches a `MarkInfo` object to the function, containing all the arguments already stored internally in the `MarkDecorator`.
3. When called in any other case, it performs a ‘fake construction’ call, i.e. it returns a new `MarkDecorator` instance with the original `MarkDecorator`’s content updated with the arguments passed to this call.

Note: The rules above prevent `MarkDecorator` objects from storing only a single function or class reference as their positional argument with no additional keyword or positional arguments.

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.

2.13 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 the environment (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.

pytest counts and lists *skip* and *xfail* tests separately. 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*)

2.13.1 Marking a test function to be skipped

New in version 2.0,: 2.4

Here is an example of marking a test function to be skipped when run on a Python3.3 interpreter:

```
import sys
@pytest.mark.skipif(sys.version_info < (3,3),
                    reason="requires python3.3")
def test_function():
    ...
```

During test function setup the condition (“`sys.version_info >= (3,3)`”) is checked. If it evaluates to True, the test function will be skipped with the specified reason. Note that pytest enforces specifying a reason in order to report meaningful “skip reasons” (e.g. when using `-rs`). If the condition is a string, it will be evaluated as python expression.

You can share skipif markers between modules. Consider this test module:

```
# content of test_mymodule.py

import mymodule
minversion = pytest.mark.skipif(mymodule.__versioninfo__ < (1,1),
                                reason="at least mymodule-1.1 required")

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

You can import it from another test module:

```
# test_myothermodule.py
from test_mymodule import minversion

@minversion
def test_anotherfunction():
    ...
```

For larger test suites it's usually a good idea to have one file where you define the markers which you then consistently apply throughout your test suite.

Alternatively, the pre pytest-2.4 way to specify *condition strings* instead of booleans will remain fully supported in future versions of pytest. It couldn't be easily used for importing markers between test modules so it's no longer advertised as the primary method.

2.13.2 Skip all test functions of a class or module

As with all function *marking* you can skip test functions at the whole class- or module level. If your code targets python2.6 or above you use the skipif decorator (and any other marker) on classes:

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

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

If the condition is true, this marker will produce a skip result for each of the test methods.

If your code targets python2.5 where class-decorators are not available, you can set the `pytestmark` attribute of a class:

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

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

As with the class-decorator, the `pytestmark` special name tells pytest to apply it to each test function in the class.

If you want to skip all test functions of a module, you must use the `pytestmark` name on the global level:

```
# test_module.py

pytestmark = pytest.mark.skipif(...)
```

If multiple “skipif” decorators are applied to a test function, it will be skipped if any of the skip conditions is true.

2.13.3 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 >= (3,3),
                    reason="python3.3 api changes")
def test_function():
    ...
```

If you want to be more specific as to why the test is failing, you can specify a single exception, or a list of exceptions, in the `raises` argument. Then the test will be reported as a regular failure if it fails with an exception not mentioned in `raises`.

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")

@xfail(raises=IndexError)
def test_hello7():
    x = []
    x[1] = 1
```

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

```
example $ py.test -rx xfail_demo.py
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 7 items

xfail_demo.py xxxxxxxx
===== 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
XFAIL xfail_demo.py::test_hello7

===== 7 xfailed in 0.04 seconds =====

```

2.13.4 Skip/xfail with parametrize

It is possible to apply markers like skip and xfail to individual test instances when using parametrize:

```

import pytest

@pytest.mark.parametrize(("n", "expected"), [
    (1, 2),
    pytest.mark.xfail((1, 0)),
    pytest.mark.xfail(reason="some bug")((1, 3)),
    (2, 3),
    (3, 4),
    (4, 5),
    pytest.mark.skipif("sys.version_info >= (3,0)")((10, 11)),
])
def test_increment(n, expected):
    assert n + 1 == expected

```

2.13.5 Imperative xfail from within a test or setup function

If you cannot declare xfail- or skipif conditions at import time you can also imperatively produce an according outcome imperatively, in test or setup code:

```

def test_function():
    if not valid_config():
        pytest.xfail("failing configuration (but should work)")
        # or
        pytest.skip("unsupported configuration")

```

2.13.6 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.

2.13.7 specifying conditions as strings versus booleans

Prior to pytest-2.4 the only way to specify skipif/xfail conditions was to use strings:

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

During test function setup the skipif condition is evaluated by calling `eval('sys.version_info >= (3,0)', namespace)`. The namespace contains all the module globals, and `os` and `sys` as a minimum.

Since pytest-2.4 [condition booleans](#) are considered preferable because markers can then be freely imported between test modules. With strings you need to import not only the marker but all variables everything used by the marker, which violates encapsulation.

The reason for specifying the condition as a string was that `pytest` can report a summary of skip conditions based purely on the condition string. With conditions as booleans you are required to specify a `reason` string.

Note that string conditions will remain fully supported and you are free to use them if you have no need for cross-importing markers.

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(...):
    ...
```

The equivalent with “boolean conditions” is:

```
@pytest.mark.skipif(not pytest.config.getvalue("db"),
                    reason="--db was not specified")
def test_function(...):
    pass
```

Note: You cannot use `pytest.config.getvalue()` in code imported before `py.test`’s argument parsing takes place. For example, `conftest.py` files are imported before command line parsing and thus `config.getvalue()` will not execute correctly.

2.14 Asserting deprecation and other warnings

2.14.1 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 isinstance(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

2.14.2 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)
```

2.15 Support for unittest.TestCase / Integration of fixtures

pytest 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 pytest. We assume here that you are familiar with writing `unittest.TestCase` style tests and rather focus on integration aspects.

2.15.1 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.

2.15.2 Mixing pytest fixtures into unittest.TestCase style tests

Running your unittest with `pytest` 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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.db_class.<locals>.DummyDB object at 0x2b98cc5a2e80>
E       assert 0

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.db_class.<locals>.DummyDB object at 0x2b98cc5a2e80>
E       assert 0

test_unittest_db.py:12: AssertionError
===== 2 failed in 0.04 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.

2.15.3 autouse fixtures and accessing other fixtures

Although it's usually better to explicitly 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
.
1 passed in 0.05 seconds
    
```

... 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.

2.16 Running tests written for nose

`pytest` has basic support for running tests written for `nose`.

2.16.1 Usage

After *Installation* type:

```
python setup.py develop # make sure tests can import our package
py.test # instead of 'nosetests'
```

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

2.16.2 Supported nose Idioms

- setup and teardown at module/class/method level
- `SkipTest` exceptions and markers
- setup/teardown decorators
- yield-based tests and their setup
- `__test__` attribute on modules/classes/functions
- general usage of `nose` utilities

2.16.3 Unsupported idioms / known issues

- `unittest`-style `setUp`, `tearDown`, `setUpClass`, `tearDownClass` are recognized only on `unittest.TestCase` classes but not on plain classes. `nose` supports these methods also on plain classes but `pytest` deliberately does not. As `nose` and `pytest` already both support `setup_class`, `teardown_class`, `setup_method`, `teardown_method` it doesn't seem useful to duplicate the `unittest`-API like `nose` does. If you however rather think `pytest` should support the `unittest`-spelling on plain classes please post to [this issue](#).
- `nose` imports test modules with the same import path (e.g. `tests.test_mod`) but different file system paths (e.g. `tests/test_mode.py` and `other/tests/test_mode.py`) by extending `sys.path/import` semantics. `pytest` does not do that but there is discussion in [issue268](#) for adding some support. Note that `nose2` choose to avoid this `sys.path/import` hackery.
- `nose`-style doctests are not collected and executed correctly, also doctest fixtures don't work.
- no `nose`-configuration is recognized

2.17 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 1 items

mymodule.py .

===== 1 passed in 0.06 seconds =====
```

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

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

Also, *using fixtures from classes, modules or projects* and *autouse fixtures (xUnit setup on steroids)* fixtures are supported when executing text doctest files.

WORKING WITH PLUGINS AND CONFTEST FILES

`pytest` 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 `pytest`'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_sub.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, `pytest` automatically finds and integrates it, there is no need to activate it. We have a [page listing all 3rd party plugins and their status against the latest `py.test` version](#) and here is a little annotated list for some popular 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-cov`: coverage reporting, compatible with distributed testing
- `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-instafail`: to report failures while the test run is happening.
- `pytest-bdd` and `pytest-konira` to write tests using behaviour-driven testing.
- `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-pep8`: a `--pep8` option to enable PEP8 compliance checking.
- `oejskit`: a plugin to run javascript unittests in life browsers

To see a complete list of all plugins with their latest testing status against different `py.test` and Python versions, please visit [pytest-plugins](#).

You may also 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 `pytest`'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 `pytest` finds your plugin module. Entry points are a feature that is provided by `setuptools` or `Distribute`. `pytest` looks up the `pytest11` entypoint to discover its plugins and you can thus make your plugin available by defining 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 pytest
    entry_points = {
        'pytest11': [
            'name_of_plugin = myproject.pluginmodule',
        ]
    },
)
```

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

3.5 Plugin discovery order at tool startup

`pytest` 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:
 - if no test paths are specified use current dir as a test path
 - if exists, load `conftest.py` and `test*/conftest.py` relative to the directory part of the first test path.

Note that `pytest` does not find `conftest.py` files in deeper nested sub directories at tool startup. It is usually a good idea to keep your `conftest.py` file in the top level test or project root directory.

- 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 `pytest` 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.

PYTEST DEFAULT PLUGIN REFERENCE

You can find the source code for the following plugins in the [pytest repository](#).

| | |
|----------------------------------|--|
| <code>_pytest.assertion</code> | support for presenting detailed information in failing assertions. |
| <code>_pytest.capture</code> | per-test stdout/stderr capturing mechanism. |
| <code>_pytest.config</code> | |
| <code>_pytest.doctest</code> | Module doctest – a framework for running examples in docstrings. |
| <code>_pytest.genscript</code> | |
| <code>_pytest.helpconfig</code> | |
| <code>_pytest.junitxml</code> | |
| <code>_pytest.mark</code> | |
| <code>_pytest.monkeypatch</code> | |
| <code>_pytest.nose</code> | |
| <code>_pytest.pastebin</code> | |
| <code>_pytest.pdb</code> | pdb++, a drop-in replacement for pdb |
| <code>_pytest.pytester</code> | (disabled by default) support for testing pytest and pytest plugins. |
| <code>_pytest.python</code> | |
| <code>_pytest.recwarn</code> | |
| <code>_pytest.resultlog</code> | |
| <code>_pytest.runner</code> | |
| <code>_pytest.main</code> | |
| <code>_pytest.skipping</code> | |
| <code>_pytest.terminal</code> | |
| <code>_pytest.tmpdir</code> | |
| <code>_pytest.unittest</code> | Python unit testing framework, based on Erich Gamma’s JUnit and Kent Beck’s Smalltalk testing fram |

PYTEST HOOK REFERENCE

5.1 Hook specification and validation

`pytest` calls hook functions to implement initialization, running, test execution and reporting. When `pytest` 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_load_initial_conftests` (*args, early_config, parser*)

implements the loading of initial conftest files ahead of command line option parsing.

`pytest_cmdline_preparse` (*config, args*)

(deprecated) 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 `pytest` namespace. This hook is called before command line options are parsed.

`pytest_addoption` (*parser*)

register argparse-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 conf test files been loaded.

pytest_unconfigure (*config*)

called before test process is exited.

5.3 Generic “runtest” hooks

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.
- **nextitem** – 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 **nextitem** – 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

pytest 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.

After collection is complete, you can modify the order of items, delete or otherwise amend the test items:

pytest_collection_modifyitems (*session, config, items*)
 called after collection has been performed, may filter or re-order the items in-place.

5.5 Reporting hooks

Session related reporting hooks:

pytest_collectstart (*collector*)
 collector starts collecting.

pytest_itemcollected (*item*)
 we just collected a test item.

pytest_collectreport (*report*)
 collector finished collecting.

pytest_deselected (*items*)
 called for test items deselected by keyword.

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

pytest_runtest_logreport (*report*)
 process a test setup/call/teardown report relating to the respective phase of executing a test.

5.6 Debugging/Interaction hooks

There are few hooks which can be used for special reporting or interaction with exceptions:

pytest_internalerror (*excrepr, excinfo*)
 called for internal errors.

pytest_keyboard_interrupt (*excinfo*)
 called for keyboard interrupt.

pytest_exception_interact (*node, call, report*)
 (experimental, new in 2.4) called when an exception was raised which can potentially be interactively handled.

This hook is only called if an exception was raised that is not an internal exception like “`skip.Exception`”.

5.7 Declaring new hooks

Plugins and `conftest.py` files may declare new hooks that can then be implemented by other plugins in order to alter behaviour or interact with the new plugin:

pytest_addhooks (*pluginmanager*)

called at plugin load time to allow adding new hooks via a call to `pluginmanager.registerhooks(module)`.

Hooks are usually declared as do-nothing functions that contain only documentation describing when the hook will be called and what return values are expected.

For an example, see `newhooks.py` from *xdist: pytest distributed testing plugin*.

5.8 Using hooks from 3rd party plugins

Using new hooks from plugins as explained above might be a little tricky because the standard [Hook specification and validation](#) mechanism: if you depend on a plugin that is not installed, validation will fail and the error message will not make much sense to your users.

One approach is to defer the hook implementation to a new plugin instead of declaring the hook functions directly in your plugin module, for example:

```
# contents of myplugin.py

class DeferPlugin(object):
    """Simple plugin to defer pytest-xdist hook functions."""

    def pytest_testnodedown(self, node, error):
        """standard xdist hook function.
        """

def pytest_configure(config):
    if config.pluginmanager.hasplugin('xdist'):
        config.pluginmanager.register(DeferPlugin())
```

This has the added benefit of allowing you to conditionally install hooks depending on which plugins are installed.

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

warn (*code, message*)

generate a warning for this test session.

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, default=<NOTSET>, skip=False*)

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.
- **default** – default value if no option of that name exists.
- **skip** – if True raise `pytest.skip` if option does not exists or has a None value.

getvalue (*name, path=None*)

(deprecated, use `getoption()`)

getvalueorskip (*name, path=None*)

(deprecated, use `getoption(skip=True)`)

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 `argparse` 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

extra_keyword_matches = None
allow adding of extra keywords to use for matching

ihook
`fspath` sensitive hook proxy used to call `pytest` hooks

warn (*code*, *message*)
generate a warning with the given code and message for this item.

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.

add_marker (*marker*)

dynamically add a marker object to the node.

marker can be a string or `pytest.mark.*` instance.

get_marker (*name*)

get a marker object from this node or `None` if the node doesn't have a marker with that name.

listextrakeywords ()

Return a set of all extra keywords in self and any parents.

addfinalizer (*fin*)

register a function to be called when this node is finalized.

This method can only be called when this node is active in a setup chain, for example during `self.setup()`.

getparent (*cls*)

get the next parent node (including ourself) which is an instance of the given class

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.python.FuncargnamesCompatAttr`, `_pytest.main.Item`

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 be marshallable

duration = None

time it took to run just the test

LIST OF THIRD-PARTY PLUGINS

The table below contains a listing of plugins found in PyPI and their status when tested using `py.test 2.6.4.dev1` and python 2.7 and 3.3.

A complete listing can also be found at [pytest-plugs](#), which contains tests status against other `py.test` releases.




















| Name | Py27 | Py34 | Home | Summary |
|--|------|------|---|--|
| pytest-allure-adaptor | | |  | Plugin for <code>py.test</code> to generate allure xml reports |
| pytest-bdd | | |  | BDD for <code>pytest</code> |
| pytest-beds | | |  | Fixtures for testing Google Appengine (GAE) apps |
| pytest-bench | | |  | Benchmark utility that plugs into <code>pytest</code> . |
| pytest-blockage | | |  | Disable network requests during a test run. |
| pytest-browsermob-proxy | | |  | BrowserMob proxy plugin for <code>py.test</code> . |
| pytest-bugzilla | | |  | <code>py.test</code> bugzilla integration plugin |
| pytest-cache | | |  | <code>pytest</code> plugin with mechanisms for caching across test runs |
| pytest-capturelog | | |  | <code>py.test</code> plugin to capture log messages |
| pytest-codecheckers | | |  | <code>pytest</code> plugin to add source code sanity checks (pep8 and friends) |
| pytest-config | | |  | Base configurations and utilities for developing your Python project test suite |
| pytest-contextfixture | | |  | Define <code>pytest</code> fixtures as context managers. |
| pytest-couchdbkit | | |  | <code>py.test</code> extension for per-test couchdb databases using <code>couchdbkit</code> |
| pytest-cov | | |  | <code>py.test</code> plugin for coverage reporting with support for both centralised and distributed |
| pytest-cpp | | |  | Use <code>pytest</code> 's runner to discover and execute C++ tests |
| pytest-dbfixtures | | |  | Databases fixtures plugin for <code>py.test</code> . |
| pytest-dbus-notification | | |  | D-BUS notifications for <code>pytest</code> results. |
| pytest-describe | | |  | Describe-style plugin for <code>pytest</code> |
| pytest-diffeo | | |  | Common <code>py.test</code> support for Diffeo packages |

Table 7.1 – continued from previous page

















































| Name | Py27 | Py34 | Home | Summary |
|--|------|------|---|---|
| pytest-django | | | link | A Django plugin for py.test. |
| pytest-django-haystack | | |  | Cleanup your Haystack indexes between tests |
| pytest-django-lite | | |  | The bare minimum to integrate py.test with Django. |
| pytest-echo | | | link | pytest plugin with mechanisms for echoing environment variables, package |
| pytest-eradicate | | |  | pytest plugin to check for commented out code |
| pytest-fgleaf | | |  | py.test fgleaf coverage plugin |
| pytest-fixture-tools | | | ? | Plugin for pytest which provides tools for fixtures |
| pytest-flakes | | |  | pytest plugin to check source code with pyflakes |
| pytest-flask | | |  | A set of py.test fixtures to test Flask applications. |
| pytest-greendots | | | ? | Green progress dots |
| pytest-growl | | | ? | Growl notifications for pytest results. |
| pytest-httpbin | | |  | Easily test your HTTP library against a local copy of httpbin |
| pytest-httpretty | | |  | A thin wrapper of HTTPretty for pytest |
| pytest-incremental | | |  | an incremental test runner (pytest plugin) |
| pytest-instafail | | |  | py.test plugin to show failures instantly |
| pytest-ipdb | | |  | A py.test plug-in to enable drop to ipdb debugger on test failure. |
| pytest-jira | | |  | py.test JIRA integration plugin, using markers |
| pytest-knows | | |  | A pytest plugin that can automatically skip test case based on dependence inf |
| pytest-konira | | |  | Run Konira DSL tests with py.test |
| pytest-localserver | | |  | py.test plugin to test server connections locally. |
| pytest-marker-bugzilla | | |  | py.test bugzilla integration plugin, using markers |
| pytest-markfiltration | | |  | UNKNOWN |
| pytest-marks | | |  | UNKNOWN |
| pytest-mock | | |  | Thin-wrapper around the mock package for easier use with py.test |
| pytest-monkeyplus | | |  | pytest's monkeypatch subclass with extra functionalities |
| pytest-mozwebqa | | |  | Mozilla WebQA plugin for py.test. |
| pytest-oerp | | |  | pytest plugin to test OpenERP modules |
| pytest-ordering | | |  | pytest plugin to run your tests in a specific order |
| pytest-osxnotify | | |  | OS X notifications for py.test results. |
| pytest-paste-config | | | ? | Allow setting the path to a paste config file |
| pytest-pep8 | | |  | pytest plugin to check PEP8 requirements |

Table 7.1 – continued from previous page

| Name | Py27 | Py34 | Home | Summary |
|--------------------------------------|------|------|---|---|
| pytest-pipeline | | |  | Pytest plugin for functional testing of data analysis pipelines |
| pytest-poo | | |  | Visualize your crappy tests |
| pytest-pycharm | | |  | Plugin for py.test to enter PyCharm debugger on uncaught exceptions |
| pytest-pydev | | |  | py.test plugin to connect to a remote debug server with PyDev or PyCharm. |
| pytest-pythonpath | | |  | pytest plugin for adding to the PYTHONPATH from command line or config |
| pytest-qt | | |  | pytest support for PyQt and PySide applications |
| pytest-quickcheck | | |  | pytest plugin to generate random data inspired by QuickCheck |
| pytest-rage | | |  | pytest plugin to implement PEP712 |
| pytest-raisesregexp | | |  | Simple pytest plugin to look for regex in Exceptions |
| pytest-random | | |  | py.test plugin to randomize tests |
| pytest-regtest | | | link | py.test plugin for regression tests |
| pytest-rerunfailures | | |  | py.test plugin to re-run tests to eliminate flakey failures |
| pytest-runfailed | | |  | implement a –failed option for pytest |
| pytest-runner | | |  | Invoke py.test as distutils command with dependency resolution. |
| pytest-sftpserver | | |  | py.test plugin to locally test sftp server connections. |
| pytest-spec | | |  | pytest plugin to display test execution output like a SPECIFICATION |
| pytest-splinter | | |  | Splinter plugin for pytest testing framework |
| pytest-stepwise | | |  | Run a test suite one failing test at a time. |
| pytest-sugar | | |  | py.test is a plugin for py.test that changes the default look and feel of py.test |
| pytest-timeout | | |  | py.test plugin to abort hanging tests |
| pytest-twisted | | |  | A twisted plugin for py.test. |
| pytest-xdist | | |  | py.test xdist plugin for distributed testing and loop-on-failing modes |
| pytest-xprocess | | |  | pytest plugin to manage external processes across test runs |
| pytest-yamlwsgi | | | ? | Run tests against wsgi apps defined in yaml |
| pytest-zap | | |  | OWASP ZAP plugin for py.test. |

(Updated on 2014-09-27)

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](https://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.

8.1 Demo of Python failure reports with pytest

Here is a nice run of several tens of failures and how `pytest` 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 42 items

failure_demo.py FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

===== 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 0x2b4436e4d390>
```

```

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

>     assert f() == g()
E     assert 42 == 43
E     + where 42 = <function TestFailing.test_simple.<locals>.f at 0x2b4436f1e6a8>()
E     + and 43 = <function TestFailing.test_simple.<locals>.g at 0x2b4436f1e7b8>()

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

self = <failure_demo.TestFailing object at 0x2b4436f167b8>

    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 0x2b4436f12668>

    def test_not(self):
        def f():
            return 42
>     assert not f()
E     assert not 42
E     + where 42 = <function TestFailing.test_not.<locals>.f at 0x2b4436f1ebf8>()

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

self = <failure_demo.TestSpecialisedExplanations object at 0x2b4436efaba8>

    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 0x2b4436ee02e8>

```

```

    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 0x2b4436ed5e48>

    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 0x2b4436ed2240>

    def test_eq_long_text(self):
        a = '1'*100 + 'a' + '2'*100
        b = '1'*100 + 'b' + '2'*100
>     assert a == b
E     assert '11111111111...22222222222' == '11111111111...22222222222'
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 0x2b4436f18780>

    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\n1\n...n2\n2\n2\n2\n' == '1\n1\n1\n1\n1\n...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 0x2b4436e42b00>

    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
E             Use -v to get the full diff

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

self = <failure_demo.TestSpecialisedExplanations object at 0x2b4436f164e0>

    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, 0, 0, ...]
E             At index 100 diff: 1 != 2
E             Use -v to get the full diff

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

self = <failure_demo.TestSpecialisedExplanations object at 0x2b4436f12c50>

    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             Omitting 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}
E             Use -v to get the full diff

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

self = <failure_demo.TestSpecialisedExplanations object at 0x2b4436ec1208>

    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
E           Use -v to get the full diff

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

self = <failure_demo.TestSpecialisedExplanations object at 0x2b4436efab70>

    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
E         Use -v to get the full diff

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

self = <failure_demo.TestSpecialisedExplanations object at 0x2b4436d73278>

    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 0x2b4436ebfac8>

    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 0x2b4436ee0898>

    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 0x2b4436ed5748>
```

```
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 tail '
E      'foo' is contained here:
E      head head foo 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 0x2b4436e4db70>
```

```
def test_not_in_text_single_long_term(self):
    text = 'head ' * 50 + 'f'*70 + 'tail ' * 20
>    assert 'f'*70 not in text
E    assert 'ffffffffffff...ffffffffffff' not in 'head head he...l tail tail '
E      'ffffffffffffffffffff...ffffffffffffffffffff' is contained here:
E      head head fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail tai
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.test_attribute.<locals>.Foo object at 0x2b4436e42ac8>.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.test_attribute_instance.<locals>.Foo object at 0x2b4436f185c0>.b
E      + where <failure_demo.test_attribute_instance.<locals>.Foo object at 0x2b4436f185c0> = <
```

```
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.test_attribute_failure.<locals>.Foo object at 0x2b4436f16a58>
```

```
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.test_attribute_multiple.<locals>.Foo object at 0x2b4436f12a90>.b
E     + where <failure_demo.test_attribute_multiple.<locals>.Foo object at 0x2b4436f12a90> = <
E     + and 2 = <failure_demo.test_attribute_multiple.<locals>.Bar object at 0x2b4436f12ac8>.b
E     + where <failure_demo.test_attribute_multiple.<locals>.Bar object at 0x2b4436f12ac8> = <
```

```
failure_demo.py:124: AssertionError
```

```
TestRaises.test_raises
```

```
self = <failure_demo.TestRaises object at 0x2b4436ec1d68>
```

```
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/lib/python3.4/site-packages/_pytest/python.py:1028>:1: Value
```

```
TestRaises.test_raises_doesnt
```

```
self = <failure_demo.TestRaises object at 0x2b4436ee9860>
```

```
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 object at 0x2b4436ed5198>
```

```
def test_raise(self):
>     raise ValueError("demo error")
E     ValueError: demo error
```

```
failure_demo.py:139: ValueError
```

```
TestRaises.test_tupleerror
```

```
self = <failure_demo.TestRaises object at 0x2b4436ebf320>
```

```

    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 object at 0x2b4436f1c6d8>

    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 call -----
l is [1, 2, 3]
_____ TestRaises.test_some_error _____

self = <failure_demo.TestRaises object at 0x2b4436e6abe0>

    def test_some_error(self):
>         if namenotexi:
E         NameError: 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: AssertionError
_____ TestMoreErrors.test_complex_error _____

self = <failure_demo.TestMoreErrors object at 0x2b4436ee59e8>

    def test_complex_error(self):
        def f():
            return 44
        def g():
            return 43
>         somefunc(f(), g())

failure_demo.py:175:

```

```

-----
failure_demo.py:8: in somefunc
    otherfunc(x,y)
-----

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 object at 0x2b4436f1a940>

    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 object at 0x2b4436ef1ef0>

    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 object at 0x2b4436f16710>

    def test_startswith(self):
        s = "123"
        g = "456"
>         assert s.startswith(g)
E         assert <built-in method startswith of str object at 0x2b4436e42ea0>('456')
E         + where <built-in method startswith of str object at 0x2b4436e42ea0> = '123'.startswith

failure_demo.py:188: AssertionError
_____ TestMoreErrors.test_startswith_nested _____

self = <failure_demo.TestMoreErrors object at 0x2b4436f18c88>

    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 0x2b4436e42ea0>('456')
E         + where <built-in method startswith of str object at 0x2b4436e42ea0> = '123'.startswith
E         +   where '123' = <function TestMoreErrors.test_startswith_nested.<locals>.f at 0x2b4436f1e8...
E         + and '456' = <function TestMoreErrors.test_startswith_nested.<locals>.g at 0x2b4436f1e8...

```

```

failure_demo.py:195: AssertionError
_____ TestMoreErrors.test_global_func _____

self = <failure_demo.TestMoreErrors object at 0x2b4436eed0f0>

    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 object at 0x2b4436e67c50>

    def test_instance(self):
        self.x = 6*7
>     assert self.x != 42
E     assert 42 != 42
E     + where 42 = <failure_demo.TestMoreErrors object at 0x2b4436e67c50>.x

failure_demo.py:202: AssertionError
_____ TestMoreErrors.test_compare _____

self = <failure_demo.TestMoreErrors object at 0x2b4436ebf668>

    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 object at 0x2b4436edf588>

    def test_try_finally(self):
        x = 1
        try:
>             assert x == 0
E             assert 1 == 0

failure_demo.py:210: AssertionError
_____ TestCustomAssertMsg.test_single_line _____

self = <failure_demo.TestCustomAssertMsg object at 0x2b4436ed5128>

    def test_single_line(self):
        class A:
            a = 1
            b = 2
>     assert A.a == b, "A.a appears not to be b"
E     AssertionError: A.a appears not to be b
E     assert 1 == 2
E     + where 1 = <class 'failure_demo.TestCustomAssertMsg.test_single_line.<locals>.A'>.a

failure_demo.py:221: AssertionError
_____ TestCustomAssertMsg.test_multiline _____

```

```

self = <failure_demo.TestCustomAssertMsg object at 0x2b4436e6ad30>

    def test_multiline(self):
        class A:
            a = 1
            b = 2
>         assert A.a == b, "A.a appears not to be b\n" \
            "or does not appear to be b\none of those"
E         AssertionError: A.a appears not to be b
E         or does not appear to be b
E         one of those
E         assert 1 == 2
E         + where 1 = <class 'failure_demo.TestCustomAssertMsg.test_multiline.<locals>.A'>.a

failure_demo.py:227: AssertionError
_____ TestCustomAssertMsg.test_custom_repr _____

self = <failure_demo.TestCustomAssertMsg object at 0x2b4436ee92b0>

    def test_custom_repr(self):
        class JSON:
            a = 1
            def __repr__(self):
                return "This is JSON\n{\n  'foo': 'bar'\n}"
            a = JSON()
            b = 2
>         assert a.a == b, a
E         AssertionError: This is JSON
E         {
E         'foo': 'bar'
E         }
E         assert 1 == 2
E         + where 1 = This is JSON\n{\n  'foo': 'bar'\n}.a

failure_demo.py:237: AssertionError
===== 42 failed in 0.23 seconds =====

```

8.2 Basic patterns and examples

8.2.1 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 call -----
first
1 failed in 0.01 seconds
```

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 call -----
second
1 failed in 0.01 seconds
```

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.

8.2.2 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 0 items

===== in 0.00 seconds =====
```

8.2.3 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 2 items
```

```
test_module.py .s
===== short test summary info =====
SKIP [1] /tmp/doc-exec-73/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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 2 items

test_module.py ..

===== 2 passed in 0.01 seconds =====
```

8.2.4 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 pytest 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
1 failed in 0.01 seconds
```

8.2.5 Detect if running from within a pytest 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.

8.2.6 Adding info to test report header

It's easy to present extra information in a pytest 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 ["info!: did you know that ...", "did you?"]
```

which will add info only when run with “-v”:

```
$ py.test -v
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
info!: 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 0 items

===== in 0.00 seconds =====
```

8.2.7 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 artificial 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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_funcslow2
===== 3 passed in 0.31 seconds =====
```

8.2.8 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 4 items

test_step.py .Fx.

===== FAILURES =====
_____ TestUserHandling.test_modification _____

self = <test_step.TestUserHandling object at 0x2aad15c6d048>

    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”.

8.2.9 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 implicitly executing setup/teardown functions, especially if they are far away from the actual tests.

Here is 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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-73/b/test_error.py, line 1
  def test_root(db): # no db here, will error out
      fixture 'db' not found
      available fixtures: monkeypatch, pytestconfig, tmpdir, capfd, capsys, recwarn
      use 'py.test --fixtures [testpath]' for help on them.

/tmp/doc-exec-73/b/test_error.py:1
===== FAILURES =====
_____ TestUserHandling.test_modification _____

self = <test_step.TestUserHandling object at 0x2b058dc29e10>

    def test_modification(self):
```

```
>         assert 0
E         assert 0

test_step.py:9: AssertionError
_____ test_a1 _____

db = <conftest.DB object at 0x2b058db494a8>

    def test_a1(db):
>         assert 0, db # to show value
E         AssertionError: <conftest.DB object at 0x2b058db494a8>
E         assert 0

a/test_db.py:2: AssertionError
_____ test_a2 _____

db = <conftest.DB object at 0x2b058db494a8>

    def test_a2(db):
>         assert 0, db # to show value
E         AssertionError: <conftest.DB object at 0x2b058db494a8>
E         assert 0

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).

8.2.10 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 2 items

test_module.py FF

===== FAILURES =====
_____ test_fail1 _____

tmpdir = local('/tmp/pytest-112/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-112/test_fail10)
test_module.py::test_fail2
```

8.2.11 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 3 items

test_module.py Esetting up a test failed! test_module.py::test_setup_fails
Fexecuting test failed test_module.py::test_call_fails
F

===== 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 =====
```

You'll see that the fixture finalizers could use the precise reporting information.

8.2.12 Integrating pytest runner and cx_freeze

If you freeze your application using a tool like `cx_freeze` in order to distribute it to your end-users, it is a good idea to also package your test runner and run your tests using the frozen application.

This way packaging errors such as dependencies not being included into the executable can be detected early while also allowing you to send test files to users so they can run them in their machines, which can be invaluable to obtain more information about a hard to reproduce bug.

Unfortunately `cx_freeze` can't discover them automatically because of `pytest`'s use of dynamic module loading, so you must declare them explicitly by using `pytest.freeze_includes()`:

```
# contents of setup.py
from cx_Freeze import setup, Executable
import pytest

setup(
    name="app_main",
    executables=[Executable("app_main.py")],
    options={"build_exe":
        {
            'includes': pytest.freeze_includes()
        }},
    # ... other options
)
```

If you don't want to ship a different executable just in order to run your tests, you can make your program check for a certain flag and pass control over to `pytest` instead. For example:

```
# contents of app_main.py
import sys

if len(sys.argv) > 1 and sys.argv[1] == '--pytest':
    import pytest
    sys.exit(pytest.main(sys.argv[2:]))
else:
```



```
# normal application execution: at this point argv can be parsed
# by your argument-parsing library of choice as usual
...
```

This makes it convenient to execute your tests from within your frozen application, using standard `py.test` command-line options:

```
$ ./app_main --pytest --verbose --tb=long --junit-xml=results.xml test-suite/
/bin/sh: 1: ./app_main: not found
```

8.3 Parametrizing tests

`pytest` 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.

8.3.1 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
..
2 passed in 0.01 seconds
```

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
1 failed, 4 passed in 0.01 seconds

```

As expected when running the full range of `param1` values we'll get an error on the last one.

8.3.2 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 --collect-only test_scenarios.py
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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.

8.3.3 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 --collect-only
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 object at 0x2b04d7936be0>

    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
1 failed, 1 passed in 0.01 seconds
```

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.

8.3.4 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[2-1] _____

self = <test_parametrize.TestClass object at 0x2af4cdee0da0>, a = 1, b = 2

    def test_equals(self, a, b):
>     assert a == b
E     assert 1 == 2

test_parametrize.py:18: AssertionError
1 failed, 2 passed in 0.01 seconds
```

8.3.5 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
import pytest

pythonlist = ['python2.6', 'python2.7', 'python3.4']
@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:
            pytest.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
.....FFFFF...
===== FAILURES =====
_____ test_basic_objects[python3.4-python2.6-42] _____

python1 = <multipython.Python object at 0x2afc7d8c2828>
python2 = <multipython.Python object at 0x2afc7d8c2588>, obj = 42

    @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)

multipython.py:51:
-----
multipython.py:46: in load_and_is_true
    py.process.cmdexec("%s %s" % (self.pythonpath, loadfile))
-----

cmd = '/home/hpk/bin/python2.6 /tmp/pytest-111/test_basic_objects_python3_4_p0/load.py'

def cmdexec(cmd):
    """ return unicode output of executing 'cmd' in a separate process.

    raise cmdexec.Error exeception if the command failed.
    the exception will provide an 'err' attribute containing
    the error-output from the command.
    if the subprocess module does not provide a proper encoding/unicode strings
    sys.getdefaultencoding() will be used, if that does not exist, 'UTF-8'.
    """
    process = subprocess.Popen(cmd, shell=True,
                               universal_newlines=True,
                               stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = process.communicate()
    if sys.version_info[0] < 3: # on py3 we get unicode strings, on py2 not
        try:
```

```

        default_encoding = sys.getdefaultencoding() # jython may not have it
    except AttributeError:
        default_encoding = sys.stdout.encoding or 'UTF-8'
    out = unicode(out, process.stdout.encoding or default_encoding)
    err = unicode(err, process.stderr.encoding or default_encoding)
    status = process.poll()
    if status:
>         raise ExecutionFailed(status, status, cmd, out, err)
E         py.process.cmdexec.Error: ExecutionFailed: 1 /home/hpk/bin/python2.6 /tmp/pytest-111/tes
E         Traceback (most recent call last):
E             File "/tmp/pytest-111/test_basic_objects_python3_4_p0/load.py", line 4, in <module>
E                 obj = pickle.load(f)
E             File "/usr/lib/python2.6/pickle.py", line 1370, in load
E                 return Unpickler(file).load()
E             File "/usr/lib/python2.6/pickle.py", line 858, in load
E                 dispatch[key](self)
E             File "/usr/lib/python2.6/pickle.py", line 886, in load_proto
E                 raise ValueError, "unsupported pickle protocol: %d" % proto
E         ValueError: unsupported pickle protocol: 3

../../../../toxin/lib/python3.4/site-packages/py/_process/cmdexec.py:28: Error
----- Captured stdout call -----
/tmp/pytest-111/test_basic_objects_python3_4_p0/load.py
_____ test_basic_objects[python3.4-python2.6-obj1] _____

python1 = <multipython.Python object at 0x2afc7d8b8c88>
python2 = <multipython.Python object at 0x2afc7d8b8e48>, obj = {}

    @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)

multipython.py:51:
-----
multipython.py:46: in load_and_is_true
    py.process.cmdexec("%s %s" %(self.pythonpath, loadfile))
-----

cmd = '/home/hpk/bin/python2.6 /tmp/pytest-111/test_basic_objects_python3_4_p1/load.py'

def cmdexec(cmd):
    """ return unicode output of executing 'cmd' in a separate process.

    raise cmdexec.Error exeception if the command failed.
    the exception will provide an 'err' attribute containing
    the error-output from the command.
    if the subprocess module does not provide a proper encoding/unicode strings
    sys.getdefaultencoding() will be used, if that does not exist, 'UTF-8'.
    """
    process = subprocess.Popen(cmd, shell=True,
                               universal_newlines=True,
                               stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = process.communicate()
    if sys.version_info[0] < 3: # on py3 we get unicode strings, on py2 not
        try:
            default_encoding = sys.getdefaultencoding() # jython may not have it
        except AttributeError:

```

```

        default_encoding = sys.stdout.encoding or 'UTF-8'
        out = unicode(out, process.stdout.encoding or default_encoding)
        err = unicode(err, process.stderr.encoding or default_encoding)
    status = process.poll()
    if status:
>         raise ExecutionFailed(status, status, cmd, out, err)
E         py.process.cmdexec.Error: ExecutionFailed: 1 /home/hpk/bin/python2.6 /tmp/pytest-111/tes
E         Traceback (most recent call last):
E             File "/tmp/pytest-111/test_basic_objects_python3_4_p1/load.py", line 4, in <module>
E                 obj = pickle.load(f)
E             File "/usr/lib/python2.6/pickle.py", line 1370, in load
E                 return Unpickler(file).load()
E             File "/usr/lib/python2.6/pickle.py", line 858, in load
E                 dispatch[key](self)
E             File "/usr/lib/python2.6/pickle.py", line 886, in load_proto
E                 raise ValueError, "unsupported pickle protocol: %d" % proto
E         ValueError: unsupported pickle protocol: 3

../../../../toxin/lib/python3.4/site-packages/py/_process/cmdexec.py:28: Error
----- Captured stdout call -----
/tmp/pytest-111/test_basic_objects_python3_4_p1/load.py
_____ test_basic_objects[python3.4-python2.6-obj2] _____

python1 = <multipython.Python object at 0x2afc7d8bf6d8>
python2 = <multipython.Python object at 0x2afc7d8bf860>, obj = {1: 3}

    @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)

multipython.py:51:
-----
multipython.py:46: in load_and_is_true
    py.process.cmdexec("%s %s" %(self.pythonpath, loadfile))
-----

cmd = '/home/hpk/bin/python2.6 /tmp/pytest-111/test_basic_objects_python3_4_p2/load.py'

def cmdexec(cmd):
    """ return unicode output of executing 'cmd' in a separate process.

    raise cmdexec.Error exception if the command failed.
    the exception will provide an 'err' attribute containing
    the error-output from the command.
    if the subprocess module does not provide a proper encoding/unicode strings
    sys.getdefaultencoding() will be used, if that does not exist, 'UTF-8'.
    """
    process = subprocess.Popen(cmd, shell=True,
                               universal_newlines=True,
                               stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = process.communicate()
    if sys.version_info[0] < 3: # on py3 we get unicode strings, on py2 not
        try:
            default_encoding = sys.getdefaultencoding() # jython may not have it
        except AttributeError:
            default_encoding = sys.stdout.encoding or 'UTF-8'
        out = unicode(out, process.stdout.encoding or default_encoding)

```



```

        err = unicode(err, process.stderr.encoding or default_encoding)
    status = process.poll()
    if status:
>         raise ExecutionFailed(status, status, cmd, out, err)
E         py.process.cmdexec.Error: ExecutionFailed: 1 /home/hpk/bin/python2.6 /tmp/pytest-111/te
E         Traceback (most recent call last):
E             File "/tmp/pytest-111/test_basic_objects_python3_4_p2/load.py", line 4, in <module>
E                 obj = pickle.load(f)
E             File "/usr/lib/python2.6/pickle.py", line 1370, in load
E                 return Unpickler(file).load()
E             File "/usr/lib/python2.6/pickle.py", line 858, in load
E                 dispatch[key](self)
E             File "/usr/lib/python2.6/pickle.py", line 886, in load_proto
E                 raise ValueError, "unsupported pickle protocol: %d" % proto
E         ValueError: unsupported pickle protocol: 3

../../../../toxin/regen/lib/python3.4/site-packages/py/_process/cmdexec.py:28: Error
----- Captured stdout call -----
/tmp/pytest-111/test_basic_objects_python3_4_p2/load.py
_____ test_basic_objects[python3.4-python2.7-42] _____

python1 = <multipython.Python object at 0x2afc7d8b8710>
python2 = <multipython.Python object at 0x2afc7d8b8748>, obj = 42

    @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)

multipython.py:51:
-----
multipython.py:46: in load_and_is_true
    py.process.cmdexec("%s %s" % (self.pythonpath, loadfile))
-----

cmd = '/home/hpk/venv/0/bin/python2.7 /tmp/pytest-111/test_basic_objects_python3_4_p3/load.py'

def cmdexec(cmd):
    """ return unicode output of executing 'cmd' in a separate process.

    raise cmdexec.Error exception if the command failed.
    the exception will provide an 'err' attribute containing
    the error-output from the command.
    if the subprocess module does not provide a proper encoding/unicode strings
    sys.getdefaultencoding() will be used, if that does not exist, 'UTF-8'.
    """
    process = subprocess.Popen(cmd, shell=True,
                               universal_newlines=True,
                               stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = process.communicate()
    if sys.version_info[0] < 3: # on py3 we get unicode strings, on py2 not
        try:
            default_encoding = sys.getdefaultencoding() # jython may not have it
        except AttributeError:
            default_encoding = sys.stdout.encoding or 'UTF-8'
        out = unicode(out, process.stdout.encoding or default_encoding)
        err = unicode(err, process.stderr.encoding or default_encoding)
    status = process.poll()

```

```

        if status:
>         raise ExecutionFailed(status, status, cmd, out, err)
E         py.process.cmdexec.Error: ExecutionFailed: 1 /home/hpk/venv/0/bin/python2.7 /tmp/pytest-
E         Traceback (most recent call last):
E           File "/tmp/pytest-111/test_basic_objects_python3_4_p3/load.py", line 4, in <module>
E             obj = pickle.load(f)
E           File "/usr/lib/python2.7/pickle.py", line 1378, in load
E             return Unpickler(file).load()
E           File "/usr/lib/python2.7/pickle.py", line 858, in load
E             dispatch[key](self)
E           File "/usr/lib/python2.7/pickle.py", line 886, in load_proto
E             raise ValueError, "unsupported pickle protocol: %d" % proto
E         ValueError: unsupported pickle protocol: 3

../../../../toxin/lib/python3.4/site-packages/py/_process/cmdexec.py:28: Error
----- Captured stdout call -----
/tmp/pytest-111/test_basic_objects_python3_4_p3/load.py
_____ test_basic_objects[python3.4-python2.7-obj1] _____

python1 = <multipython.Python object at 0x2afc7d8bfb38>
python2 = <multipython.Python object at 0x2afc7d8bf3c8>, obj = {}

    @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)

multipython.py:51:
-----
multipython.py:46: in load_and_is_true
    py.process.cmdexec("%s %s" %(self.pythonpath, loadfile))
-----

cmd = '/home/hpk/venv/0/bin/python2.7 /tmp/pytest-111/test_basic_objects_python3_4_p4/load.py'

def cmdexec(cmd):
    """ return unicode output of executing 'cmd' in a separate process.

    raise cmdexec.Error exeception if the command failed.
    the exception will provide an 'err' attribute containing
    the error-output from the command.
    if the subprocess module does not provide a proper encoding/unicode strings
    sys.getdefaultencoding() will be used, if that does not exist, 'UTF-8'.
    """
    process = subprocess.Popen(cmd, shell=True,
                               universal_newlines=True,
                               stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = process.communicate()
    if sys.version_info[0] < 3: # on py3 we get unicode strings, on py2 not
        try:
            default_encoding = sys.getdefaultencoding() # jython may not have it
        except AttributeError:
            default_encoding = sys.stdout.encoding or 'UTF-8'
        out = unicode(out, process.stdout.encoding or default_encoding)
        err = unicode(err, process.stderr.encoding or default_encoding)
    status = process.poll()
    if status:
>         raise ExecutionFailed(status, status, cmd, out, err)

```

```

E       py.process.cmdexec.Error: ExecutionFailed: 1 /home/hpk/venv/0/bin/python2.7 /tmp/pytest-
E       Traceback (most recent call last):
E       File "/tmp/pytest-111/test_basic_objects_python3_4_p4/load.py", line 4, in <module>
E       obj = pickle.load(f)
E       File "/usr/lib/python2.7/pickle.py", line 1378, in load
E       return Unpickler(file).load()
E       File "/usr/lib/python2.7/pickle.py", line 858, in load
E       dispatch[key](self)
E       File "/usr/lib/python2.7/pickle.py", line 886, in load_proto
E       raise ValueError, "unsupported pickle protocol: %d" % proto
E       ValueError: unsupported pickle protocol: 3

.../.../...tox/regen/lib/python3.4/site-packages/py/_process/cmdexec.py:28: Error
----- Captured stdout call -----
/tmp/pytest-111/test_basic_objects_python3_4_p4/load.py
_____ test_basic_objects[python3.4-python2.7-obj2] _____

python1 = <multipython.Python object at 0x2afc7d8b86a0>
python2 = <multipython.Python object at 0x2afc7d8c2a90>, obj = {1: 3}

    @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)

multipython.py:51:
-----
multipython.py:46: in load_and_is_true
    py.process.cmdexec("%s %s" % (self.pythonpath, loadfile))
-----

cmd = '/home/hpk/venv/0/bin/python2.7 /tmp/pytest-111/test_basic_objects_python3_4_p5/load.py'

def cmdexec(cmd):
    """ return unicode output of executing 'cmd' in a separate process.

    raise cmdexec.Error exeception if the command failed.
    the exception will provide an 'err' attribute containing
    the error-output from the command.
    if the subprocess module does not provide a proper encoding/unicode strings
    sys.getdefaultencoding() will be used, if that does not exist, 'UTF-8'.
    """
    process = subprocess.Popen(cmd, shell=True,
                               universal_newlines=True,
                               stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = process.communicate()
    if sys.version_info[0] < 3: # on py3 we get unicode strings, on py2 not
        try:
            default_encoding = sys.getdefaultencoding() # jython may not have it
        except AttributeError:
            default_encoding = sys.stdout.encoding or 'UTF-8'
        out = unicode(out, process.stdout.encoding or default_encoding)
        err = unicode(err, process.stderr.encoding or default_encoding)
    status = process.poll()
    if status:
>       raise ExecutionFailed(status, status, cmd, out, err)
E       py.process.cmdexec.Error: ExecutionFailed: 1 /home/hpk/venv/0/bin/python2.7 /tmp/pytest-
E       Traceback (most recent call last):

```

```

E           File "/tmp/pytest-111/test_basic_objects_python3_4_p5/load.py", line 4, in <module>
E           obj = pickle.load(f)
E           File "/usr/lib/python2.7/pickle.py", line 1378, in load
E           return Unpickler(file).load()
E           File "/usr/lib/python2.7/pickle.py", line 858, in load
E           dispatch[key](self)
E           File "/usr/lib/python2.7/pickle.py", line 886, in load_proto
E           raise ValueError, "unsupported pickle protocol: %d" % proto
E           ValueError: unsupported pickle protocol: 3

../../../../../tox/regen/lib/python3.4/site-packages/py/_process/cmdexec.py:28: Error
----- Captured stdout call -----
/tmp/pytest-111/test_basic_objects_python3_4_p5/load.py
6 failed, 21 passed in 1.66 seconds

```

8.3.6 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 2 items

```

```
test_module.py .s
===== short test summary info =====
SKIP [1] /tmp/doc-exec-70/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.

8.4 Working with custom markers

Here are some example using the *Marking test functions with attributes* mechanism.

8.4.1 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
class TestClass:
    def test_method(self):
        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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/python
collecting ... collected 4 items

test_server.py::test_send_http PASSED

===== 3 tests deselected by "-m 'webtest'" =====
===== 1 passed, 3 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 4 items

test_server.py::test_something_quick PASSED
test_server.py::test_another PASSED
test_server.py::TestClass::test_method PASSED

===== 1 tests deselected by "-m 'not webtest'" =====
===== 3 passed, 1 deselected in 0.01 seconds =====
```

8.4.2 Selecting tests based on their node ID

You can provide one or more *node IDs* as positional arguments to select only specified tests. This makes it easy to select tests based on their module, class, method, or function name:

```
$ py.test -v test_server.py::TestClass::test_method
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 5 items

test_server.py::TestClass::test_method PASSED

===== 1 passed in 0.01 seconds =====
```

You can also select on the class:

```
$ py.test -v test_server.py::TestClass
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 4 items

test_server.py::TestClass::test_method PASSED

===== 1 passed in 0.01 seconds =====
```

Or select multiple nodes:

```
$ py.test -v test_server.py::TestClass test_server.py::test_send_http
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 8 items

test_server.py::TestClass::test_method PASSED
test_server.py::test_send_http PASSED

===== 2 passed in 0.01 seconds =====
```

Note: Node IDs are of the form `module.py::class::method` or `module.py::function`. Node IDs control which tests are collected, so `module.py::class` will select all test methods on the class. Nodes are also created for each parameter of a parametrized fixture or test, so selecting a parametrized test must include the parameter value, e.g. `module.py::function[param]`.

Node IDs for failing tests are displayed in the test summary info when running `py.test` with the `-rf` option. You can also construct Node IDs from the output of `py.test --collectonly`.

8.4.3 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 4 items

test_server.py::test_send_http PASSED

===== 3 tests deselected by '-khttp' =====
===== 1 passed, 3 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 4 items

test_server.py::test_something_quick PASSED
test_server.py::test_another PASSED
test_server.py::TestClass::test_method PASSED

===== 1 tests deselected by '-knot send_http' =====
===== 3 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 4 items

test_server.py::test_send_http PASSED
test_server.py::test_something_quick PASSED

===== 2 tests deselected by '-khttp or quick' =====
===== 2 passed, 2 deselected in 0.01 seconds =====
```

Note: If you are using expressions such as “X and Y” then both X and Y need to be simple non-keyword names. For example, “pass” or “from” will result in `SyntaxErrors` because “-k” evaluates the expression.

However, if the “-k” argument is a simple string, no such restrictions apply. Also “-k ‘not STRING’” has no restrictions. You can also specify numbers like “-k 1.3” to match tests which are parametrized with the float “1.3”.

8.4.4 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 value

@pytest.mark.xfail(condition, reason=None, run=True, raises=None): mark the the test function as an expected failure

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times passing in different arguments

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all of the specified fixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin machinery will try to call it first

@pytest.mark.trylast: mark a hook implementation function such that the plugin machinery will try to call it last
```

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 explicitly 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. Future versions of `pytest` are probably going to start treating non-registered markers as errors at some point.
-

8.4.5 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.

8.4.6 Marking individual tests when using parametrize

When using parametrize, applying a mark will make it apply to each individual test. However it is also possible to apply a marker to an individual test instance:

```
import pytest

@pytest.mark.foo
@pytest.mark.parametrize(("n", "expected"), [
    (1, 2),
    pytest.mark.bar((1, 3)),
    (2, 3),
])
def test_increment(n, expected):
    assert n + 1 == expected
```

In this example the mark “foo” will apply to each of the three tests, whereas the “bar” mark is only applied to the second test. Skip and xfail marks can also be applied in this way, see [Skip/xfail with parametrize](#).

8.4.7 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.addinvalue_line("markers",
        "env(name): mark test to run only on named environment")

def pytest_runtest_setup(item):
    envmarker = item.get_marker("env")
    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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 value

@pytest.mark.xfail(condition, reason=None, run=True, raises=None): mark the the test function as an expected failure

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times passing in different arguments

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all of the specified fixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin machinery will try to call it first

@pytest.mark.trylast: mark a hook implementation function such that the plugin machinery will try to call it last
```

8.4.8 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.get_marker("glob")
    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}
.
1 passed in 0.01 seconds
```

8.4.9 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 not item.get_marker(plat):
            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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 4 items

test_plat.py sss.
===== short test summary info =====
SKIP [3] /tmp/doc-exec-68/conftest.py:12: cannot run on platform linux

===== 1 passed, 3 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 4 items

test_plat.py s

===== 3 tests deselected by "-m 'linux2'" =====
===== 1 skipped, 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.

8.4.10 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.add_marker(pytest.mark.interface)
    elif "event" in item.nodeid:
        item.add_marker(pytest.mark.event)

```

We can now use the `-m` option to select one set:

```

$ py.test -m interface --tb=short
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 0.01 seconds =====

```

8.5 A session-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
.
4 passed in 0.04 seconds
```

8.6 Changing standard (Python) test discovery

8.6.1 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 `pytest` to not recurse into typical subversion or sphinx-build directories or into any `tmp` prefixed directory.

8.6.2 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 `pytest` 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 --collect-only
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 2 items
<Module 'check_myapp.py'>
  <Class 'CheckMyApp'>
    <Instance '()'>
      <Function 'check_simple'>
      <Function 'check_complex'>

===== in 0.01 seconds =====
```

Note: the `python_functions` and `python_classes` has no effect for `unittest.TestCase` test discovery because `pytest` delegates detection of test case methods to `unittest` code.

8.6.3 Interpreting cmdline arguments as Python packages

You can use the `--pyargs` option to make `pytest` 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.

8.6.4 Finding out what is collected

You can always peek at the collection tree without running tests like this:

```
. $ py.test --collect-only pythoncollection.py
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 3 items
<Module 'pythoncollection.py'>
  <Function 'test_function'>
  <Class 'TestClass'>
    <Instance '()'>
      <Function 'test_method'>
      <Function 'test_anothermethod'>

===== in 0.01 seconds =====
```

8.6.5 customizing test collection to find all .py files

You can easily instruct `pytest` 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 exception 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 --collect-only
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 0 items

===== in 0.00 seconds =====
```

If you run with a Python3 interpreter the moduled added through the conftest.py file will not be considered for test collection.

8.7 Working with non-python tests

8.7.1 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*.yaml 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 == ".yaml" 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.safe_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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
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 0.03 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 linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4 -- /home/hpk/p/pytest/.tox/regen/bin/pyth
collecting ... collected 2 items

test_simple.yml::usecase: hello FAILED

```

```
test_simple.yml::usecase: ok PASSED
```

```
===== FAILURES =====
_____ usecase: hello _____
usecase execution failed
  spec failed: 'some': 'other'
  no further details known at this point.
===== 1 failed, 1 passed in 0.03 seconds =====
```

While developing your custom test collection and execution it's also interesting to just look at the collection tree:

```
nonpython $ py.test --collect-only
===== test session starts =====
platform linux -- Python 3.4.0 -- py-1.4.26 -- pytest-2.6.4
collected 2 items
<YamlFile 'test_simple.yml'>
  <YamlItem 'hello'>
  <YamlItem 'ok'>

===== in 0.03 seconds =====
```


TALKS AND TUTORIALS

Next Open Trainings

professional testing with pytest and tox, 24-26th November 2014, Freiburg, Germany

9.1 Talks and blog postings

- Introduction to pytest, Andreas Pelme, EuroPython 2014.
- Advanced Uses of py.test Fixtures, Floris Bruynooghe, EuroPython 2014.
- Why i use py.test and maybe you should too, Andy Todd, Pycon AU 2013
- 3-part blog series about pytest from @pydanny alias Daniel Greenfeld (January 2014)
- pytest: helps you write better Django apps, Andreas Pelme, DjangoCon Europe 2014.
- *pytest fixtures: explicit, modular, scalable*
- Testing Django Applications with pytest, Andreas Pelme, EuroPython 2013.
- Testes pythonics com py.test, Vinicius Belchior Assef Neto, Plone Conf 2013, Brazil.
- Introduction to py.test fixtures, FOSDEM 2013, Floris Bruynooghe.
- pytest feature and release highlights, Holger Krekel (GERMAN, October 2013)
- pytest introduction from Brian Okken (January 2013)
- 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 pytest's new assertion rewriting

Distributed testing:

- simultaneously test your code on all platforms (blog entry)

Plugin specific examples:

- [skipping slow tests by default in pytest](#) (blog entry)
- many examples in the docs for plugins

9.2 Older conference talks and tutorials

- [pycon australia 2012 pytest talk](#) from Brianna Laughner (video, slides, code)
- [pycon 2012 US talk](#) video from Holger Krekel
- [pycon 2010 tutorial PDF](#) and [tutorial1 repository](#)
- [ep2009-rapidtesting.pdf](#) tutorial slides (July 2009):
 - testing terminology
 - basic pytest usage, file system layout
 - test function arguments (funcargs) and test fixtures
 - existing plugins
 - distributed testing
- [ep2009-pytest.pdf](#) 60 minute pytest talk, highlighting unique features and a roadmap (July 2009)
- [pycon2009-pytest-introduction.zip](#) slides and files, extended version of pytest 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.

CONTRIBUTING

Contributions are highly welcomed and appreciated. Every little help counts, so do not hesitate!

10.1 Types of contributions

10.1.1 Report bugs

Report bugs at <https://bitbucket.org/hpk42/pytest/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting, specifically Python interpreter version, installed libraries and pytest version.
- Detailed steps to reproduce the bug.

10.1.2 Submit feedback for developers

Do you like pytest? Share some love on Twitter or in your blog posts!

We'd also like to hear about your propositions and suggestions. Feel free to [submit them as issues](#) and:

- Set the “kind” to “enhancement” or “proposal” so that we can quickly find about them.
- Explain in detail how they should work.
- Keep the scope as narrow as possible. This will make it easier to implement.
- If you have required skills and/or knowledge, we are very happy for *pull requests*.

10.1.3 Fix bugs

Look through the BitBucket issues for bugs. Here is sample filter you can use:
<https://bitbucket.org/hpk42/pytest/issues?status=new&status=open&kind=bug>

Talk to developers to find out how you can fix specific bugs.

10.1.4 Implement features

Look through the BitBucket issues for enhancements. Here is sample filter you can use: <https://bitbucket.org/hpk42/pytest/issues?status=new&status=open&kind=enhancement>

Talk to developers to find out how you can implement specific features.

10.1.5 Write documentation

pytest could always use more documentation. What exactly is needed?

- More complementary documentation. Have you perhaps found something unclear?
- Documentation translations. We currently have English and Japanese versions.
- Docstrings. There's never too much of them.
- Blog posts, articles and such – they're all very appreciated.

10.2 Preparing Pull Requests on Bitbucket

Note: What is a “pull request”? It informs project's core developers about the changes you want to review and merge. Pull requests are stored on [BitBucket servers](#). Once you send pull request, we can discuss it's potential modifications and even add more commits to it later on.

The primary development platform for pytest is BitBucket. You can find all the issues there and submit your pull requests.

1. Fork the [pytest BitBucket repository](#). It's fine to use `pytest` as your fork repository name because it will live under your user.
2. Create and activate a fork-specific virtualenv (<http://www.virtualenv.org/en/latest/>):

```
$ virtualenv pytest-venv
$ source pytest-venv/bin/activate
```

3. Clone your fork locally using [Mercurial](#) (hg) and create a branch:

```
$ hg clone ssh://hg@bitbucket.org/YOUR_BITBUCKET_USERNAME/pytest
$ cd pytest
$ hg branch your-branch-name
```

If you need some help with Mercurial, follow this quick start guide: <http://mercurial.selenic.com/wiki/QuickStart>

4. You can now edit your local working copy. To test you need to install the “tox” tool into your virtualenv:

```
$ pip install tox
```

You need to have Python 2.7 and 3.3 available in your system. Now running tests is as simple as issuing this command:

```
$ python runtox.py -e py27,py33,flakes
```

This command will run tests via the “tox” tool against Python 2.7 and 3.3 and also perform “flakes” coding-style checks. `runtox.py` is a thin wrapper around `tox` which installs from a development

package index where newer (not yet released to pypi) versions of dependencies (especially `py`) might be present.

To run tests on `py27` and pass options (e.g. enter `pdb` on failure) to `pytest` you can do:

```
$ python runtox.py -e py27 -- --pdb
```

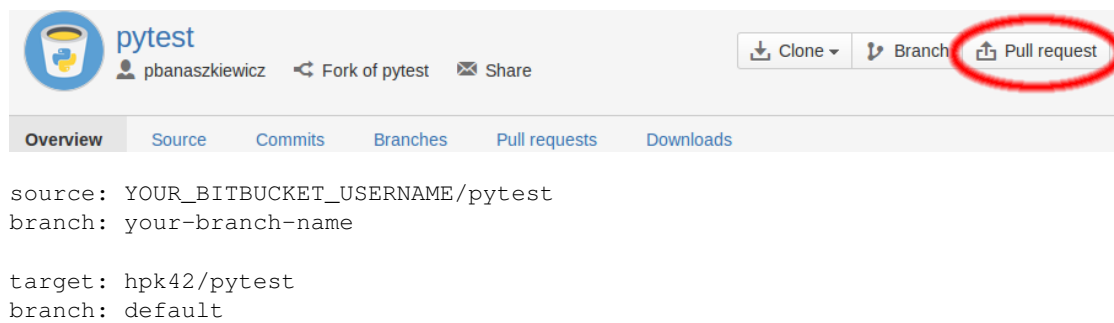
or to only run tests in a particular test module on `py33`:

```
$ python runtox.py -e py33 -- testing/test_config.py
```

5. Commit and push once your tests pass and you are happy with your change(s):

```
$ hg commit -m"<commit message>"
$ hg push -b .
```

6. Finally, submit a pull request through the BitBucket website:



10.2.1 What about git (and so GitHub)?

There used to be the `pytest` GitHub mirror. It was removed in favor of the Mercurial one, to remove confusion of people not knowing where it's better to put their issues and pull requests. Also it wasn't easily possible to automate the mirroring process.

However, it's still possible to use `git` to contribute to `pytest` using tools like `gitifyhg` which allows you to clone and work with Mercurial repo still using `git`.

Warning: Remember that `git` is **not** a default version control system for `pytest` and you need to be careful using it.

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.

11.1 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*.

11.2 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.

11.3 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 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.

11.4 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.

11.5 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 coordinates the test run activities of the slave processes.
2. if you only perform a collection (with “`--collect-only`”) 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.

11.6 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 --collect-only`” 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.

11.7 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 recommended to invest a few minutes and simplify your fixture function code to use the *Fixtures as Function arguments* decorator instead. This will also allow to take advantage of the automatic per-resource grouping of tests.

RELEASE ANNOUNCEMENTS

12.1 pytest-2.6.3: fixes and little improvements

pytest is a mature Python testing tool with more than a 1100 tests against itself, passing on many different interpreters and platforms. This release is drop-in compatible to 2.5.2 and 2.6.X. See below for the changes and see docs at:

<http://pytest.org>

As usual, you can upgrade from pypi via:

```
pip install -U pytest
```

Thanks to all who contributed, among them:

Floris Bruynooghe Oleg Sinyavskiy Uwe Schmitt Charles Cloud Wolfgang Schnerring
have fun, holger krekel

12.2 Changes 2.6.3

- fix issue575: xunit-xml was reporting collection errors as failures instead of errors, thanks Oleg Sinyavskiy.
- fix issue582: fix setuptools example, thanks Laszlo Papp and Ronny Pfannschmidt.
- Fix infinite recursion bug when pickling capture.EncodedFile, thanks Uwe Schmitt.
- fix issue589: fix bad interaction with numpy and others when showing exceptions. Check for precise “maximum recursion depth exceed” exception instead of presuming any RuntimeError is that one (implemented in py dep). Thanks Charles Cloud for analysing the issue.
- fix confest related fixture visibility issue: when running with a CWD outside a test package pytest would get fixture discovery wrong. Thanks to Wolfgang Schnerring for figuring out a reproducible example.
- Introduce `pytest_enter_pdb` hook (needed e.g. by `pytest_timeout` to cancel the timeout when interactively entering `pdb`). Thanks Wolfgang Schnerring.
- check `xfail/skip` also with non-python function test items. Thanks Floris Bruynooghe.

12.3 pytest-2.6.2: few fixes and `cx_freeze` support

pytest is a mature Python testing tool with more than a 1100 tests against itself, passing on many different interpreters and platforms. This release is drop-in compatible to 2.5.2 and 2.6.X. It also brings support for including pytest with `cx_freeze` or similar freezing tools into your single-file app distribution. For details see the CHANGELOG below.

See docs at:

<http://pytest.org>

As usual, you can upgrade from pypi via:

```
pip install -U pytest
```

Thanks to all who contributed, among them:

Floris Bruynooghe Benjamin Peterson Bruno Oliveira

have fun, holger krekel

12.3.1 2.6.2

- Added function `pytest.freeze_includes()`, which makes it easy to embed pytest into executables using tools like `cx_freeze`. See docs for examples and rationale. Thanks Bruno Oliveira.
- Improve assertion rewriting cache invalidation precision.
- fixed issue561: adapt autouse fixture example for python3.
- fixed issue453: assertion rewriting issue with `__repr__` containing “n{”, “n}” and “n~”.
- fix issue560: correctly display code if an “else:” or “finally:” is followed by statements on the same line.
- Fix example in monkeypatch documentation, thanks t-8ch.
- fix issue572: correct tmpdir doc example for python3.
- Do not mark as universal wheel because Python 2.6 is different from other builds due to the extra `argparse` dependency. Fixes issue566. Thanks sontek.

12.4 pytest-2.6.1: fixes and new xfail feature

pytest is a mature Python testing tool with more than a 1100 tests against itself, passing on many different interpreters and platforms. The 2.6.1 release is drop-in compatible to 2.5.2 and actually fixes some regressions introduced with 2.6.0. It also brings a little feature to the xfail marker which now recognizes expected exceptions, see the CHANGELOG below.

See docs at:

<http://pytest.org>

As usual, you can upgrade from pypi via:

```
pip install -U pytest
```

Thanks to all who contributed, among them:

Floris Bruynooghe Bruno Oliveira Nicolas Delaby

have fun, holger krekel

12.5 Changes 2.6.1

- No longer show line numbers in the `-verbose` output, the output is now purely the nodeid. The line number is still shown in failure reports. Thanks Floris Bruynooghe.

- fix issue437 where assertion rewriting could cause pytest-xdist slaves to collect different tests. Thanks Bruno Oliveira.
- fix issue555: add “errors” attribute to capture-streams to satisfy some distutils and possibly other code accessing `sys.stdout.errors`.
- fix issue547 capsys/capfd also work when output capturing (“-s”) is disabled.
- address issue170: allow `pytest.mark.xfail(...)` to specify expected exceptions via an optional “raises=EXC” argument where EXC can be a single exception or a tuple of exception classes. Thanks David Mohr for the complete PR.
- fix integration of pytest with `unittest.mock.patch` decorator when it uses the “new” argument. Thanks Nicolas Delaby for test and PR.
- fix issue with detecting conftest files if the arguments contain “::” node id specifications (copy pasted from “-v” output)
- fix issue544 by only removing “@NUM” at the end of “::” separated parts and if the part has an “.py” extension
- don’t use `py.std` import helper, rather import things directly. Thanks Bruno Oliveira.

12.6 pytest-2.6.0: shorter tracebacks, new warning system, test runner compat

pytest is a mature Python testing tool with more than a 1000 tests against itself, passing on many different interpreters and platforms.

The 2.6.0 release should be drop-in backward compatible to 2.5.2 and fixes a number of bugs and brings some new features, mainly:

- shorter tracebacks by default: only the first (test function) entry and the last (failure location) entry are shown, the ones between only in “short” format. Use `--tb=long` to get back the old behaviour of showing “long” entries everywhere.
- a new warning system which reports oddities during collection and execution. For example, ignoring collecting `Test*` classes with an `__init__` now produces a warning.
- various improvements to nose/mock/unittest integration

Note also that 2.6.0 departs with the “zero reported bugs” policy because it has been too hard to keep up with it, unfortunately. Instead we are for now rather bound to work on “upvoted” issues in the <https://bitbucket.org/hpk42/pytest/issues?status=new&status=open&sort=-votes> issue tracker.

See docs at:

<http://pytest.org>

As usual, you can upgrade from pypi via:

```
pip install -U pytest
```

Thanks to all who contributed, among them:

Benjamin Peterson Jurko Gospodnetić Floris Bruynooghe Marc Abramowitz Marc Schlaich Trevor Bekolay Bruno Oliveira Alex Groenholm

have fun, holger krekel

12.6.1 2.6.0

- fix issue537: Avoid importing old assertion reinterpretation code by default. Thanks Benjamin Peterson.
- fix issue364: shorten and enhance tracebacks representation by default. The new “-tb=auto” option (default) will only display long tracebacks for the first and last entry. You can get the old behaviour of printing all entries as long entries with “-tb=long”. Also short entries by default are now printed very similarly to “-tb=native” ones.
- fix issue514: teach assertion reinterpretation about private class attributes Thanks Benjamin Peterson.
- change -v output to include full node IDs of tests. Users can copy a node ID from a test run, including line number, and use it as a positional argument in order to run only a single test.
- fix issue 475: fail early and comprehensible if calling `pytest.raises` with wrong exception type.
- fix issue516: tell in getting-started about current dependencies.
- cleanup `setup.py` a bit and specify supported versions. Thanks Jurko Gospodnetic for the PR.
- change XPASS colour to yellow rather than red when tests are run with -v.
- fix issue473: work around mock putting an unbound method into a class dict when double-patching.
- fix issue498: if a fixture finalizer fails, make sure that the fixture is still invalidated.
- fix issue453: the result of the `pytest_assertrepr_compare` hook now gets its newlines escaped so that `format_exception` does not blow up.
- internal new warning system: pytest will now produce warnings when it detects oddities in your test collection or execution. Warnings are ultimately sent to a new `pytest_logwarning` hook which is currently only implemented by the terminal plugin which displays warnings in the summary line and shows more details when -rw (report on warnings) is specified.
- change skips into warnings for test classes with an `__init__` and callables in test modules which look like a test but are not functions.
- fix issue436: improved finding of initial conftest files from command line arguments by using the result of `parse_known_args` rather than the previous flaky heuristics. Thanks Marc Abramowitz for tests and initial fixing approaches in this area.
- fix issue #479: properly handle nose/unittest(2) `SkipTest` exceptions during collection/loading of test modules. Thanks to Marc Schlaich for the complete PR.
- fix issue490: include `pytest_load_initial_conftests` in documentation and improve docstring.
- fix issue472: clarify that `pytest.config.getvalue()` cannot work if it's triggered ahead of command line parsing.
- merge PR123: improved integration with `mock.patch` decorator on tests.
- fix issue412: messing with stdout/stderr FD-level streams is now captured without crashes.
- fix issue483: `trial/py33` works now properly. Thanks Daniel Grana for PR.
- improve example for pytest integration with “python setup.py test” which now has a generic “-a” or “-pytest-args” option where you can pass additional options as a quoted string. Thanks Trevor Bekolay.
- simplified internal capturing mechanism and made it more robust against tests or setups changing FD1/FD2, also better integrated now with `pytest.pdb()` in single tests.
- improvements to pytest's own test-suite leakage detection, courtesy of PRs from Marc Abramowitz
- fix issue492: avoid leak in `test_writeorg`. Thanks Marc Abramowitz.
- fix issue493: don't run tests in doc directory with `python setup.py test` (use `tox -e doctesting` for that)

- fix issue486: better reporting and handling of early conftest loading failures
- some cleanup and simplification of internal conftest handling.
- work a bit harder to break reference cycles when catching exceptions. Thanks Jurko Gospodnetic.
- fix issue443: fix skip examples to use proper comparison. Thanks Alex Groenholm.
- support nose-style `__test__` attribute on modules, classes and functions, including unittest-style Classes. If set to False, the test will not be collected.
- fix issue512: show “<notset>” for arguments which might not be set in monkeypatch plugin. Improves output in documentation.
- avoid importing “py.test” (an old alias module for “pytest”)

12.7 pytest-2.5.2: fixes

pytest is a mature Python testing tool with more than a 1000 tests against itself, passing on many different interpreters and platforms.

The 2.5.2 release fixes a few bugs with two maybe-bugs remaining and actively being worked on (and waiting for the bug reporter’s input). We also have a new contribution guide thanks to Piotr Banaszkiewicz and others.

See docs at:

<http://pytest.org>

As usual, you can upgrade from pypi via:

```
pip install -U pytest
```

Thanks to the following people who contributed to this release:

Anatoly Bubenvkov Ronny Pfannschmidt Floris Bruynooghe Bruno Oliveira Andreas Pelme Jurko Gospodnetić Piotr Banaszkiewicz Simon Liedtke Iakka Lukasz Balcerzak Philippe Muller Daniel Hahler

have fun, holger krekel

12.7.1 2.5.2

- fix issue409 – better interoperate with `cx_freeze` by not trying to import from `collections.abc` which causes problems for `py27/cx_freeze`. Thanks Wolfgang L. for reporting and tracking it down.
- fixed docs and code to use “pytest” instead of “py.test” almost everywhere. Thanks Jurko Gospodnetic for the complete PR.
- fix issue425: mention at end of “py.test -h” that `--markers` and `--fixtures` work according to specified test path (or current dir)
- fix issue413: exceptions with unicode attributes are now printed correctly also on python2 and with `pytest-xdist` runs. (the fix requires `py-1.4.20`)
- copy, cleanup and integrate `py.io` capture from `pylib 1.4.20.dev2` (rev 13d9af95547e)
- address issue416: clarify docs as to `conftest.py` loading semantics
- fix issue429: comparing byte strings with non-ascii chars in assert expressions now work better. Thanks Floris Bruynooghe.
- make `capfd/capsys.capture` private, its unused and shouldnt be exposed

12.8 pytest-2.5.1: fixes and new home page styling

pytest is a mature Python testing tool with more than a 1000 tests against itself, passing on many different interpreters and platforms.

The 2.5.1 release maintains the “zero-reported-bugs” promise by fixing the three bugs reported since the last release a few days ago. It also features a new home page styling implemented by Tobias Bieniek, based on the flask theme from Armin Ronacher:

<http://pytest.org>

If you have anything more to improve styling and docs, we’d be very happy to merge further pull requests.

On the coding side, the release also contains a little enhancement to fixture decorators allowing to directly influence generation of test ids, thanks to Floris Bruynooghe. Other thanks for helping with this release go to Anatoly Bubenkoff and Ronny Pfannschmidt.

As usual, you can upgrade from pypi via:

```
pip install -U pytest
```

have fun and a nice remaining “bug-free” time of the year :) holger krekel

12.8.1 2.5.1

- merge new documentation styling PR from Tobias Bieniek.
- fix issue403: allow parametrize of multiple same-name functions within a collection node. Thanks Andreas Kloeckner and Alex Gaynor for reporting and analysis.
- Allow parameterized fixtures to specify the ID of the parameters by adding an ids argument to `pytest.fixture()` and `pytest.yield_fixture()`. Thanks Floris Bruynooghe.
- fix issue404 by always using the binary xml escape in the junitxml plugin. Thanks Ronny Pfannschmidt.
- fix issue407: fix adoption docstring to point to argparse instead of optparse. Thanks Daniel D. Wright.

12.9 pytest-2.5.0: now down to ZERO reported bugs!

pytest-2.5.0 is a big fixing release, the result of two community bug fixing days plus numerous additional works from many people and reporters. The release should be fully compatible to 2.4.2, existing plugins and test suites. We aim at maintaining this level of ZERO reported bugs because it’s no fun if your testing tool has bugs, is it? Under a condition, though: when submitting a bug report please provide clear information about the circumstances and a simple example which reproduces the problem.

The issue tracker is of course not empty now. We have many remaining “enhancement” issues which we’ll hopefully can tackle in 2014 with your help.

For those who use older Python versions, please note that pytest is not automatically tested on python2.5 due to virtualenv, setuptools and tox not supporting it anymore. Manual verification shows that it mostly works fine but it’s not going to be part of the automated release process and thus likely to break in the future.

As usual, current docs are at

<http://pytest.org>

and you can upgrade from pypi via:

```
pip install -U pytest
```

Particular thanks for helping with this release go to Anatoly Bubenkoff, Floris Bruynooghe, Marc Abramowitz, Ralph Schmitt, Ronny Pfannschmidt, Donald Stufft, James Lan, Rob Dennis, Jason R. Coombs, Mathieu Agopian, Virgil Dupras, Bruno Oliveira, Alex Gaynor and others.

have fun, holger krekel

12.9.1 2.5.0

- dropped python2.5 from automated release testing of pytest itself which means it's probably going to break soon (but still works with this release we believe).
- simplified and fixed implementation for calling finalizers when parametrized fixtures or function arguments are involved. finalization is now performed lazily at setup time instead of in the “teardown phase”. While this might sound odd at first, it helps to ensure that we are correctly handling setup/teardown even in complex code. User-level code should not be affected unless it's implementing the `pytest_runtest_teardown` hook and expecting certain fixture instances are torn down within (very unlikely and would have been unreliable anyway).
- PR90: add `--color=yes|no|auto` option to force terminal coloring mode (“auto” is default). Thanks Marc Abramowitz.
- fix issue319 - correctly show unicode in assertion errors. Many thanks to Floris Bruynooghe for the complete PR. Also means we depend on `py>=1.4.19` now.
- fix issue396 - correctly sort and finalize class-scoped parametrized tests independently from number of methods on the class.
- refix issue323 in a better way – parametrization should now never cause Runtime Recursion errors because the underlying algorithm for re-ordering tests per-scope/per-fixture is not recursive anymore (it was tail-call recursive before which could lead to problems for more than >966 non-function scoped parameters).
- fix issue290 - there is preliminary support now for parametrizing with repeated same values (sometimes useful to test if calling a second time works as with the first time).
- close issue240 - document precisely how pytest module importing works, discuss the two common test directory layouts, and how it interacts with PEP420-namespaces packages.
- fix issue246 fix finalizer order to be LIFO on independent fixtures depending on a parametrized higher-than-function scoped fixture. (was quite some effort so please bear with the complexity of this sentence :) Thanks Ralph Schmitt for the precise failure example.
- fix issue244 by implementing special index for parameters to only use indices for parametrized test ids
- fix issue287 by running all finalizers but saving the exception from the first failing finalizer and re-raising it so teardown will still have failed. We reraise the first failing exception because it might be the cause for other finalizers to fail.
- fix ordering when `mock.patch` or other standard decorator-wrappings are used with test methods. This fixes issue346 and should help with random “xdist” collection failures. Thanks to Ronny Pfannschmidt and Donald Stufft for helping to isolate it.
- fix issue357 - special case “-k” expressions to allow for filtering with simple strings that are not valid python expressions. Examples: “-k 1.3” matches all tests parametrized with 1.3. “-k None” filters all tests that have “None” in their name and conversely “-k 'not None’”. Previously these examples would raise syntax errors.
- fix issue384 by removing the trial support code since the unittest compat enhancements allow trial to handle it on its own
- don't hide an ImportError when importing a plugin produces one. fixes issue375.

- fix issue275 - allow usefixtures and autouse fixtures for running doctest text files.
- fix issue380 by making `--resultlog` only rely on `longrepr` instead of the “`reprcrash`” attribute which only exists sometimes.
- address issue122: allow `@pytest.fixture(params=iterator)` by exploding into a list early on.
- fix pexpect-3.0 compatibility for pytest’s own tests. (fixes issue386)
- allow nested parametrize-value markers, thanks James Lan for the PR.
- fix unicode handling with new `monkeypatch.setattr(import_path, value)` API. Thanks Rob Dennis. Fixes issue371.
- fix unicode handling with `junitxml`, fixes issue368.
- In assertion rewriting mode on Python 2, fix the detection of coding cookies. See issue #330.
- make “`--runxfail`” turn imperative `pytest.xfail` calls into no ops (it already did neutralize `pytest.mark.xfail` markers)
- refine `pytest / pkg_resources` interactions: The `AssertionRewritingHook` PEP302 compliant loader now registers itself with `setuptools/pkg_resources` properly so that the `pkg_resources.resource_stream` method works properly. Fixes issue366. Thanks for the investigations and full PR to Jason R. Coombs.
- `pytestconfig` fixture is now session-scoped as it is the same object during the whole test run. Fixes issue370.
- avoid one surprising case of marker malfunction/confusion:

```
@pytest.mark.some(lambda arg: ...)
def test_function():
```

would not work correctly because `pytest` assumes `@pytest.mark.some` gets a function to be decorated already. We now at least detect if this arg is an `lambda` and thus the example will work. Thanks Alex Gaynor for bringing it up.

- `xfail` a test on `pypy` that checks wrong encoding/ascii (`pypy` does not error out). fixes issue385.
- internally make `varnames()` deal with classes’s `__init__`, although it’s not needed by `pytest` itself atm. Also fix caching. Fixes issue376.
- fix issue221 - handle importing of namespace-package with no `__init__.py` properly.
- refactor internal `FixtureRequest` handling to avoid monkeypatching. One of the positive user-facing effects is that the “request” object can now be used in closures.
- fixed version comparison in `pytest.importskip(modname, minverstring)`
- fix issue377 by clarifying in the `nose-compat` docs that `pytest` does not duplicate the `unittest-API` into the “plain” namespace.
- fix verbose reporting for `@mock`’d test functions

12.10 pytest-2.4.2: colorama on windows, plugin/tmpdir fixes

pytest-2.4.2 is another bug-fixing release:

- on Windows require `colorama` and a newer `py` lib so that `py.io.TerminalWriter()` now uses `colorama` instead of its own ctypes hacks. (fixes issue365) thanks Paul Moore for bringing it up.
- fix “-k” matching of tests where “repr” and “attr” and other names would cause wrong matches because of an internal implementation quirk (don’t ask) which is now properly implemented. fixes issue345.

- avoid tmpdir fixture to create too long filenames especially when parametrization is used (issue354)
- fix pytest-pep8 and pytest-flakes / pytest interactions (collection names in mark plugin was assuming an item always has a function which is not true for those plugins etc.) Thanks Andi Zeidler.
- introduce node.get_marker/node.add_marker API for plugins like pytest-pep8 and pytest-flakes to avoid the messy details of the node.keywords pseudo-dicts. Adapted docs.
- remove attempt to “dup” stdout at startup as it’s icky. the normal capturing should catch enough possibilities of tests messing up standard FDs.
- add pluginmanager.do_configure(config) as a link to config.do_configure() for plugin-compatibility

as usual, docs at <http://pytest.org> and upgrades via:

```
pip install -U pytest
```

have fun, holger krekel

12.11 pytest-2.4.1: fixing three regressions compared to 2.3.5

pytest-2.4.1 is a quick follow up release to fix three regressions compared to 2.3.5 before they hit more people:

- When using parser.adoption() unicode arguments to the “type” keyword should also be converted to the respective types. thanks Floris Bruynooghe, @dnozay. (fixes issue360 and issue362)
- fix dotted filename completion when using argcomplete thanks Anthon van der Neuth. (fixes issue361)
- fix regression when a 1-tuple (“arg”,) is used for specifying parametrization (the values of the parametrization were passed nested in a tuple). Thanks Donald Stuft.
- also merge doc typo fixes, thanks Andy Dirnberger

as usual, docs at <http://pytest.org> and upgrades via:

```
pip install -U pytest
```

have fun, holger krekel

12.12 pytest-2.4.0: new fixture features/hooks and bug fixes

The just released pytest-2.4.0 brings many improvements and numerous bug fixes while remaining plugin- and test-suite compatible apart from a few supposedly very minor incompatibilities. See below for a full list of details. A few feature highlights:

- new yield-style fixtures `pytest.yield_fixture`, allowing to use existing with-style context managers in fixture functions.
- improved pdb support: `import pdb ; pdb.set_trace()` now works without requiring prior disabling of stdout/stderr capturing. Also the `--pdb` options works now on collection and internal errors and we introduced a new experimental hook for IDEs/plugins to intercept debugging: `pytest_exception_interact(node, call, report)`.
- shorter monkeypatch variant to allow specifying an import path as a target, for example: `monkeypatch.setattr("requests.get", myfunc)`
- better unittest/nose compatibility: all teardown methods are now only called if the corresponding setup method succeeded.

- integrate tab-completion on command line options if you have `argcomplete` configured.
- allow boolean expression directly with `skipif/xfail` if a “reason” is also specified.
- a new hook `pytest_load_initial_conftests` allows plugins like `pytest-django` to influence the environment before conftest files import `django`.
- reporting: color the last line red or green depending if failures/errors occurred or everything passed.

The documentation has been updated to accommodate the changes, see <http://pytest.org>

To install or upgrade pytest:

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

Many thanks to all who helped, including Floris Bruynooghe, Brianna Laughner, Andreas Pelme, Anthon van der Neut, Anatoly Bubenkoff, Vladimir Keleshev, Mathieu Agopian, Ronny Pfannschmidt, Christian Theunert and many others.

may passing tests be with you,

holger krekel

12.12.1 Changes between 2.3.5 and 2.4

known incompatibilities:

- if calling `-genscript` from python2.7 or above, you only get a standalone script which works on python2.7 or above. Use Python2.6 to also get a python2.5 compatible version.
- all xunit-style teardown methods (nose-style, pytest-style, unittest-style) will not be called if the corresponding setup method failed, see issue322 below.
- the `pytest_plugin_unregister` hook wasn’t ever properly called and there is no known implementation of the hook - so it got removed.
- `pytest.fixture`-decorated functions cannot be generators (i.e. use `yield`) anymore. This change might be reversed in 2.4.1 if it causes unforeseen real-life issues. However, you can always write and return an inner function/generator and change the fixture consumer to iterate over the returned generator. This change was done in lieu of the new `pytest.yield_fixture` decorator, see below.

new features:

- experimentally introduce a new `pytest.yield_fixture` decorator which accepts exactly the same parameters as `pytest.fixture` but mandates a `yield` statement instead of a `return` statement from fixture functions. This allows direct integration with “with-style” context managers in fixture functions and generally avoids registering of finalization callbacks in favour of treating the “after-`yield`” as teardown code. Thanks Andreas Pelme, Vladimir Keleshev, Floris Bruynooghe, Ronny Pfannschmidt and many others for discussions.
- allow boolean expression directly with `skipif/xfail` if a “reason” is also specified. Rework skipping documentation to recommend “condition as booleans” because it prevents surprises when importing markers between modules. Specifying conditions as strings will remain fully supported.
- reporting: color the last line red or green depending if failures/errors occurred or everything passed. thanks Christian Theunert.
- make “`import pdb ; pdb.set_trace()`” work natively wrt capturing (no “-s” needed anymore), making `pytest.set_trace()` a mere shortcut.

- fix issue181: `-pdb` now also works on collect errors (and on internal errors) . This was implemented by a slight internal refactoring and the introduction of a new hook `pytest_exception_interact` hook (see next item).
- fix issue341: introduce new experimental hook for IDEs/terminals to intercept debugging: `pytest_exception_interact(node, call, report)`.
- new `monkeypatch.setattr()` variant to provide a shorter invocation for patching out classes/functions from modules:

```
monkeypatch.setattr("requests.get", myfunc)
```

will replace the “get” function of the “requests” module with `myfunc`.
- fix issue322: `tearDownClass` is not run if `setUpClass` failed. Thanks Mathieu Agopian for the initial fix. Also make all of `pytest/nose` finalizer mimic the same generic behaviour: if a `setupX` exists and fails, don’t run `tearDownX`. This internally introduces a new method “`node.addfinalizer()`” helper which can only be called during the setup phase of a node.
- simplify `pytest.mark.parametrize()` signature: allow to pass a CSV-separated string to specify argnames. For example: `pytest.mark.parametrize("input,expected", [(1,2), (2,3)])` works as well as the previous: `pytest.mark.parametrize(("input", "expected"), ...)`.
- add support for `setUpModule/tearDownModule` detection, thanks Brian Okken.
- integrate tab-completion on options through use of “`argcomplete`”. Thanks Anthon van der Neut for the PR.
- change option names to be hyphen-separated long options but keep the old spelling backward compatible. `py.test -h` will only show the hyphenated version, for example “`-collect-only`” but “`-collectonly`” will remain valid as well (for backward-compat reasons). Many thanks to Anthon van der Neut for the implementation and to Hynek Schlawack for pushing us.
- fix issue 308 - allow to mark/xfail/skip individual parameter sets when parametrizing. Thanks Brianna Laughner.
- call new experimental `pytest_load_initial_conftests` hook to allow 3rd party plugins to do something before a conftest is loaded.

Bug fixes:

- fix issue358 - capturing options are now parsed more properly by using a new `parser.parse_known_args` method.
- `pytest` now uses `argparse` instead of `optparse` (thanks Anthon) which means that “`argparse`” is added as a dependency if installing into python2.6 environments or below.
- fix issue333: fix a case of bad `unittest/pytest` hook interaction.
- PR27: correctly handle `nose.SkipTest` during collection. Thanks Antonio Cuni, Ronny Pfannschmidt.
- fix issue355: `junitxml` puts `name="pytest"` attribute to `testsuite` tag.
- fix issue336: autouse fixture in plugins should work again.
- fix issue279: improve object comparisons on assertion failure for standard datatypes and recognise collections.abc. Thanks to Brianna Laughner and Mathieu Agopian.
- fix issue317: assertion rewriter support for the `is_package` method
- fix issue335: document `py.code.ExceptionInfo()` object returned from `pytest.raises()`, thanks Mathieu Agopian.
- remove implicit `distributed_setup` support from `setup.py`.
- fix issue305: ignore any problems when writing pyc files.
- SO-17664702: call fixture finalizers even if the fixture function partially failed (finalizers would not always be called before)

- fix issue320 - fix class scope for fixtures when mixed with module-level functions. Thanks Anatloy Bubenkoff.
- you can specify “-q” or “-qq” to get different levels of “quieter” reporting (thanks Katarzyna Jachim)
- fix issue300 - Fix order of conf test loading when starting py.test in a subdirectory.
- fix issue323 - sorting of many module-scoped arg parametrizations
- make sessionfinish hooks execute with the same cwd-context as at session start (helps fix plugin behaviour which write output files with relative path such as pytest-cov)
- fix issue316 - properly reference collection hooks in docs
- fix issue 306 - cleanup of -k/-m options to only match markers/test names/keywords respectively. Thanks Wouter van Ackooij.
- improved doctest counting for doctests in python modules – files without any doctest items will not show up anymore and doctest examples are counted as separate test items. thanks Danilo Bellini.
- fix issue245 by depending on the released py-1.4.14 which fixes py.io.dupfile to work with files with no mode. Thanks Jason R. Coombs.
- fix junitxml generation when test output contains control characters, addressing issue267, thanks Jaap Broekhuizen
- fix issue338: honor -tb style for setup/teardown errors as well. Thanks Maho.
- fix issue307 - use yaml.safe_load in example, thanks Mark Eichin.
- better parametrize error messages, thanks Brianna Laugher
- pytest_terminal_summary(terminalreporter) hooks can now use “.section(title)” and “.line(msg)” methods to print extra information at the end of a test run.

12.13 pytest-2.3.5: bug fixes and little improvements

pytest-2.3.5 is a maintenance release with many bug fixes and little improvements. See the changelog below for details. No backward compatibility issues are foreseen and all plugins which worked with the prior version are expected to work unmodified. Speaking of which, a few interesting new plugins saw the light last month:

- pytest-instafail: show failure information while tests are running
- pytest-qt: testing of GUI applications written with QT/Pyside
- pytest-xprocess: managing external processes across test runs
- pytest-random: randomize test ordering

And several others like pytest-django saw maintenance releases. For a more complete list, check out <https://pypi.python.org/pypi/?%3Aaction=search&term=pytest&submit=search>.

For general information see:

<http://pytest.org/>

To install or upgrade pytest:

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

Particular thanks to Floris, Ronny, Benjamin and the many bug reporters and fix providers.

may the fixtures be with you, holger krekel

12.13.1 Changes between 2.3.4 and 2.3.5

- 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 doesn't try to teardown if it did not setup
- issue 271 - don't write junitxml on slave nodes
- Issue 274 - don't 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 putting 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

12.14 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

12.15 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 Taumoeolau 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

12.15.1 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 Taumoeolau
- 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.

12.16 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

12.16.1 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

12.17 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

12.17.1 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.

12.18 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

12.18.1 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 - conf test 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-decorated 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: properly 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 comparisons instead of truncating

12.19 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

12.19.1 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: properly 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

12.20 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

12.20.1 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)

12.21 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 plugin make 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

12.21.1 Changes between 2.2.0 and 2.2.1

- fix issue99 (in pytest and py) internalerrors 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)

12.22 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

12.22.1 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.

12.22.2 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

12.23 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

12.23.1 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

12.24 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>

12.24.1 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

12.25 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>

12.25.1 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”

12.26 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>

12.26.1 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 releaslevel 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

12.27 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 seamless distributed and “looponfail” testing for Python.

best, holger krekel

12.27.1 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 representation internally)
- fix issue37: avoid invalid characters in junitxml’s output

12.28 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

12.28.1 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 Laughner) and particular thanks to Laura Creighton who also reviewed parts of the documentation.
- fix slightly 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)

12.29 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

12.29.1 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.

12.30 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

12.30.1 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.TestCase 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

12.30.2 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 any-more 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

12.30.3 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.

12.30.4 (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.

CHANGELOG HISTORY

13.1 2.6.4

- Improve assertion failure reporting on iterables, by using `ndiff` and `pprint`.
- removed outdated japanese docs from source tree.
- docs for “`pytest_addhooks`” hook. Thanks Bruno Oliveira.
- updated plugin index docs. Thanks Bruno Oliveira.
- fix issue557: with “`-k`” we only allow the old style “`-`” for negation at the beginning of strings and even that is deprecated. Use “`not`” instead. This should allow to pick parametrized tests where “`-`” appeared in the parameter.
- fix issue604: Escape `%` character in the assertion message.
- fix issue620: add explanation in the `–genscript` target about what the binary blob means. Thanks Dinu Gherman.
- fix issue614: fixed pastebin support.

13.2 2.6.3

- fix issue575: `xunit-xml` was reporting collection errors as failures instead of errors, thanks Oleg Sinyavskiy.
- fix issue582: fix `setuptools` example, thanks Laszlo Papp and Ronny Pfannschmidt.
- Fix infinite recursion bug when pickling `capture.EncodedFile`, thanks Uwe Schmitt.
- fix issue589: fix bad interaction with `numpy` and others when showing exceptions. Check for precise “maximum recursion depth exceed” exception instead of presuming any `RuntimeError` is that one (implemented in `py dep`). Thanks Charles Cloud for analysing the issue.
- fix `conftest` related fixture visibility issue: when running with a CWD outside a test package `pytest` would get fixture discovery wrong. Thanks to Wolfgang Schnerring for figuring out a reproducible example.
- Introduce `pytest_enter_pdb` hook (needed e.g. by `pytest_timeout` to cancel the timeout when interactively entering `pdb`). Thanks Wolfgang Schnerring.
- check `xfail/skip` also with non-python function test items. Thanks Floris Bruynooghe.

13.3 2.6.2

- Added function `pytest.freeze_includes()`, which makes it easy to embed `pytest` into executables using tools like `cx_freeze`. See docs for examples and rationale. Thanks Bruno Oliveira.

- Improve assertion rewriting cache invalidation precision.
- fixed issue561: adapt autouse fixture example for python3.
- fixed issue453: assertion rewriting issue with `__repr__` containing “n{”, “n}” and “n~”.
- fix issue560: correctly display code if an “else:” or “finally:” is followed by statements on the same line.
- Fix example in monkeypatch documentation, thanks t-8ch.
- fix issue572: correct tmpdir doc example for python3.
- Do not mark as universal wheel because Python 2.6 is different from other builds due to the extra `argparse` dependency. Fixes issue566. Thanks sontek.
- Implement issue549: user-provided assertion messages now no longer replace the `py.test` introspection message but are shown in addition to them.

13.4 2.6.1

- No longer show line numbers in the `-verbose` output, the output is now purely the nodeid. The line number is still shown in failure reports. Thanks Floris Bruynooghe.
- fix issue437 where assertion rewriting could cause `pytest-xdist` slaves to collect different tests. Thanks Bruno Oliveira.
- fix issue555: add “errors” attribute to `capture-streams` to satisfy some distutils and possibly other code accessing `sys.stdout.errors`.
- fix issue547 `capsys/capfd` also work when output capturing (“-s”) is disabled.
- address issue170: allow `pytest.mark.xfail(...)` to specify expected exceptions via an optional “`raises=EXC`” argument where `EXC` can be a single exception or a tuple of exception classes. Thanks David Mohr for the complete PR.
- fix integration of `pytest` with `unittest.mock.patch` decorator when it uses the “new” argument. Thanks Nicolas Delaby for test and PR.
- fix issue with detecting `conftest` files if the arguments contain “::” node id specifications (copy pasted from “-v” output)
- fix issue544 by only removing “@NUM” at the end of “::” separated parts and if the part has an “.py” extension
- don’t use `py.std` import helper, rather import things directly. Thanks Bruno Oliveira.

13.5 2.6

- Cache exceptions from fixtures according to their scope (issue 467).
- fix issue537: Avoid importing old assertion reinterpretation code by default.
- fix issue364: shorten and enhance tracebacks representation by default. The new “`-tb=auto`” option (default) will only display long tracebacks for the first and last entry. You can get the old behaviour of printing all entries as long entries with “`-tb=long`”. Also short entries by default are now printed very similarly to “`-tb=native`” ones.
- fix issue514: teach assertion reinterpretation about private class attributes
- change `-v` output to include full node IDs of tests. Users can copy a node ID from a test run, including line number, and use it as a positional argument in order to run only a single test.

- fix issue 475: fail early and comprehensible if calling `pytest.raises` with wrong exception type.
- fix issue516: tell in getting-started about current dependencies.
- cleanup `setup.py` a bit and specify supported versions. Thanks Jurko Gospodnetic for the PR.
- change XPASS colour to yellow rather than red when tests are run with `-v`.
- fix issue473: work around mock putting an unbound method into a class dict when double-patching.
- fix issue498: if a fixture finalizer fails, make sure that the fixture is still invalidated.
- fix issue453: the result of the `pytest_assertrepr_compare` hook now gets its newlines escaped so that `format_exception` does not blow up.
- internal new warning system: pytest will now produce warnings when it detects oddities in your test collection or execution. Warnings are ultimately sent to a new `pytest_logwarning` hook which is currently only implemented by the terminal plugin which displays warnings in the summary line and shows more details when `-rw` (report on warnings) is specified.
- change skips into warnings for test classes with an `__init__` and callables in test modules which look like a test but are not functions.
- fix issue436: improved finding of initial conftest files from command line arguments by using the result of `parse_known_args` rather than the previous flaky heuristics. Thanks Marc Abramowitz for tests and initial fixing approaches in this area.
- fix issue #479: properly handle nose/unittest(2) `SkipTest` exceptions during collection/loading of test modules. Thanks to Marc Schlaich for the complete PR.
- fix issue490: include `pytest_load_initial_conftests` in documentation and improve docstring.
- fix issue472: clarify that `pytest.config.getvalue()` cannot work if it's triggered ahead of command line parsing.
- merge PR123: improved integration with `mock.patch` decorator on tests.
- fix issue412: messing with `stdout/stderr` FD-level streams is now captured without crashes.
- fix issue483: `trial/py33` works now properly. Thanks Daniel Grana for PR.
- improve example for pytest integration with “python setup.py test” which now has a generic “-a” or “-pytest-args” option where you can pass additional options as a quoted string. Thanks Trevor Bekolay.
- simplified internal capturing mechanism and made it more robust against tests or setups changing `FD1/FD2`, also better integrated now with `pytest.pdb()` in single tests.
- improvements to pytest's own test-suite leakage detection, courtesy of PRs from Marc Abramowitz
- fix issue492: avoid leak in `test_writeorg`. Thanks Marc Abramowitz.
- fix issue493: don't run tests in doc directory with `python setup.py test` (use `tox -e doctesting` for that)
- fix issue486: better reporting and handling of early conftest loading failures
- some cleanup and simplification of internal conftest handling.
- work a bit harder to break reference cycles when catching exceptions. Thanks Jurko Gospodnetic.
- fix issue443: fix skip examples to use proper comparison. Thanks Alex Groenholm.
- support nose-style `__test__` attribute on modules, classes and functions, including unittest-style Classes. If set to `False`, the test will not be collected.
- fix issue512: show “<notset>” for arguments which might not be set in monkeypatch plugin. Improves output in documentation.

13.6 2.5.2

- fix issue409 – better interoperate with `cx_freeze` by not trying to import from `collections.abc` which causes problems for `py27/cx_freeze`. Thanks Wolfgang L. for reporting and tracking it down.
- fixed docs and code to use “`pytest`” instead of “`py.test`” almost everywhere. Thanks Jurko Gospodnetic for the complete PR.
- fix issue425: mention at end of “`py.test -h`” that `--markers` and `--fixtures` work according to specified test path (or current dir)
- fix issue413: exceptions with unicode attributes are now printed correctly also on python2 and with `pytest-xdist` runs. (the fix requires `py-1.4.20`)
- copy, cleanup and integrate `py.io` capture from `pylib 1.4.20.dev2` (rev 13d9af95547e)
- address issue416: clarify docs as to `conftest.py` loading semantics
- fix issue429: comparing byte strings with non-ascii chars in assert expressions now work better. Thanks Floris Bruynooghe.
- make `capfd/capsys.capture` private, its unused and shouldnt be exposed

13.7 2.5.1

- merge new documentation styling PR from Tobias Bieniek.
- fix issue403: allow parametrize of multiple same-name functions within a collection node. Thanks Andreas Kloeckner and Alex Gaynor for reporting and analysis.
- Allow parameterized fixtures to specify the ID of the parameters by adding an `ids` argument to `pytest.fixture()` and `pytest.yield_fixture()`. Thanks Floris Bruynooghe.
- fix issue404 by always using the binary xml escape in the `junitxml` plugin. Thanks Ronny Pfannschmidt.
- fix issue407: fix adoption docstring to point to `argparse` instead of `optparse`. Thanks Daniel D. Wright.

13.8 2.5.0

- dropped python2.5 from automated release testing of pytest itself which means it’s probably going to break soon (but still works with this release we believe).
- simplified and fixed implementation for calling finalizers when parametrized fixtures or function arguments are involved. finalization is now performed lazily at setup time instead of in the “teardown phase”. While this might sound odd at first, it helps to ensure that we are correctly handling setup/teardown even in complex code. User-level code should not be affected unless it’s implementing the `pytest_runtest_teardown` hook and expecting certain fixture instances are torn down within (very unlikely and would have been unreliable anyway).
- PR90: add `--color=yes|no|auto` option to force terminal coloring mode (“auto” is default). Thanks Marc Abramowitz.
- fix issue319 - correctly show unicode in assertion errors. Many thanks to Floris Bruynooghe for the complete PR. Also means we depend on `py>=1.4.19` now.
- fix issue396 - correctly sort and finalize class-scoped parametrized tests independently from number of methods on the class.

- refix issue323 in a better way – parametrization should now never cause Runtime Recursion errors because the underlying algorithm for re-ordering tests per-scope/per-fixture is not recursive anymore (it was tail-call recursive before which could lead to problems for more than >966 non-function scoped parameters).
- fix issue290 - there is preliminary support now for parametrizing with repeated same values (sometimes useful to to test if calling a second time works as with the first time).
- close issue240 - document precisely how pytest module importing works, discuss the two common test directory layouts, and how it interacts with PEP420-namespaces packages.
- fix issue246 fix finalizer order to be LIFO on independent fixtures depending on a parametrized higher-than-function scoped fixture. (was quite some effort so please bear with the complexity of this sentence :) Thanks Ralph Schmitt for the precise failure example.
- fix issue244 by implementing special index for parameters to only use indices for parametrized test ids
- fix issue287 by running all finalizers but saving the exception from the first failing finalizer and re-raising it so teardown will still have failed. We reraise the first failing exception because it might be the cause for other finalizers to fail.
- fix ordering when mock.patch or other standard decorator-wrappings are used with test methods. This fixes issue346 and should help with random “xdist” collection failures. Thanks to Ronny Pfannschmidt and Donald Stufft for helping to isolate it.
- fix issue357 - special case “-k” expressions to allow for filtering with simple strings that are not valid python expressions. Examples: “-k 1.3” matches all tests parametrized with 1.3. “-k None” filters all tests that have “None” in their name and conversely “-k ‘not None’”. Previously these examples would raise syntax errors.
- fix issue384 by removing the trial support code since the unittest compat enhancements allow trial to handle it on its own
- don’t hide an ImportError when importing a plugin produces one. fixes issue375.
- fix issue275 - allow usefixtures and autouse fixtures for running doctest text files.
- fix issue380 by making –resultlog only rely on longrepr instead of the “reprcrash” attribute which only exists sometimes.
- address issue122: allow @pytest.fixture(params=iterator) by exploding into a list early on.
- fix pexpect-3.0 compatibility for pytest’s own tests. (fixes issue386)
- allow nested parametrize-value markers, thanks James Lan for the PR.
- fix unicode handling with new monkeypatch.setattr(import_path, value) API. Thanks Rob Dennis. Fixes issue371.
- fix unicode handling with junitxml, fixes issue368.
- In assertion rewriting mode on Python 2, fix the detection of coding cookies. See issue #330.
- make “–runxfail” turn imperative pytest.xfail calls into no ops (it already did neutralize pytest.mark.xfail markers)
- refine pytest / pkg_resources interactions: The AssertionRewritingHook PEP302 compliant loader now registers itself with setuptools/pkg_resources properly so that the pkg_resources.resource_stream method works properly. Fixes issue366. Thanks for the investigations and full PR to Jason R. Coombs.
- pytestconfig fixture is now session-scoped as it is the same object during the whole test run. Fixes issue370.
- avoid one surprising case of marker malfunction/confusion:

```
@pytest.mark.some(lambda arg: ...)
def test_function():
```

would not work correctly because pytest assumes `@pytest.mark.some` gets a function to be decorated already. We now at least detect if this arg is an lambda and thus the example will work. Thanks Alex Gaynor for bringing it up.

- xfail a test on pypy that checks wrong encoding/ascii (pypy does not error out). fixes issue385.
- internally make `varnames()` deal with classes's `__init__`, although it's not needed by pytest itself atm. Also fix caching. Fixes issue376.
- fix issue221 - handle importing of namespace-package with no `__init__.py` properly.
- refactor internal `FixtureRequest` handling to avoid monkeypatching. One of the positive user-facing effects is that the "request" object can now be used in closures.
- fixed version comparison in `pytest.importskip(modname, minverstring)`
- fix issue377 by clarifying in the nose-compat docs that pytest does not duplicate the unittest-API into the "plain" namespace.
- fix verbose reporting for `@mock`'d test functions

13.9 v2.4.2

- on Windows require `colorama` and a newer `py` lib so that `py.io.TerminalWriter()` now uses `colorama` instead of its own `ctypes` hacks. (fixes issue365) thanks Paul Moore for bringing it up.
- fix "-k" matching of tests where "repr" and "attr" and other names would cause wrong matches because of an internal implementation quirk (don't ask) which is now properly implemented. fixes issue345.
- avoid `tmpdir` fixture to create too long filenames especially when parametrization is used (issue354)
- fix `pytest-pep8` and `pytest-flakes` / `pytest` interactions (collection names in mark plugin was assuming an item always has a function which is not true for those plugins etc.) Thanks Andi Zeidler.
- introduce `node.get_marker/node.add_marker` API for plugins like `pytest-pep8` and `pytest-flakes` to avoid the messy details of the `node.keywords` pseudo-dicts. Adapated docs.
- remove attempt to "dup" stdout at startup as it's icky. the normal capturing should catch enough possibilities of tests messing up standard FDs.
- add `pluginmanager.do_configure(config)` as a link to `config.do_configure()` for plugin-compatibility

13.10 v2.4.1

- When using `parser.addoption()` unicode arguments to the "type" keyword should also be converted to the respective types. thanks Floris Bruynooghe, @dnozay. (fixes issue360 and issue362)
- fix dotted filename completion when using `argcomplete` thanks Anthon van der Neuth. (fixes issue361)
- fix regression when a 1-tuple ("arg",) is used for specifying parametrization (the values of the parametrization were passed nested in a tuple). Thanks Donald Stufft.
- merge doc typo fixes, thanks Andy Dirnberger

13.11 v2.4

known incompatibilities:

- if calling `–genscript` from python2.7 or above, you only get a standalone script which works on python2.7 or above. Use Python2.6 to also get a python2.5 compatible version.
- all xunit-style teardown methods (nose-style, pytest-style, unittest-style) will not be called if the corresponding setup method failed, see issue322 below.
- the `pytest_plugin_unregister` hook wasn't ever properly called and there is no known implementation of the hook - so it got removed.
- `pytest.fixture`-decorated functions cannot be generators (i.e. use `yield`) anymore. This change might be reversed in 2.4.1 if it causes unforeseen real-life issues. However, you can always write and return an inner function/generator and change the fixture consumer to iterate over the returned generator. This change was done in lieu of the new `pytest.yield_fixture` decorator, see below.

new features:

- experimentally introduce a new `pytest.yield_fixture` decorator which accepts exactly the same parameters as `pytest.fixture` but mandates a `yield` statement instead of a `return` statement from fixture functions. This allows direct integration with “with-style” context managers in fixture functions and generally avoids registering of finalization callbacks in favour of treating the “after-`yield`” as teardown code. Thanks Andreas Pelme, Vladimir Keleshev, Floris Bruynooghe, Ronny Pfannschmidt and many others for discussions.
- allow boolean expression directly with `skipif`/`xfail` if a “reason” is also specified. Rework skipping documentation to recommend “condition as booleans” because it prevents surprises when importing markers between modules. Specifying conditions as strings will remain fully supported.
- reporting: color the last line red or green depending if failures/errors occurred or everything passed. thanks Christian Theunert.
- make “`import pdb ; pdb.set_trace()`” work natively wrt capturing (no “-s” needed anymore), making `pytest.set_trace()` a mere shortcut.
- fix issue181: `–pdb` now also works on collect errors (and on internal errors) . This was implemented by a slight internal refactoring and the introduction of a new hook `pytest_exception_interact` hook (see next item).
- fix issue341: introduce new experimental hook for IDEs/terminals to intercept debugging: `pytest_exception_interact(node, call, report)`.
- new `monkeypatch.setattr()` variant to provide a shorter invocation for patching out classes/functions from modules:

```
monkeypatch.setattr("requests.get", myfunc)
```

will replace the “`get`” function of the “`requests`” module with `myfunc`.

- fix issue322: `tearDownClass` is not run if `setUpClass` failed. Thanks Mathieu Agopian for the initial fix. Also make all of `pytest/nose` finalizer mimick the same generic behaviour: if a `setupX` exists and fails, don't run `tearDownX`. This internally introduces a new method “`node.addfinalizer()`” helper which can only be called during the setup phase of a node.
- simplify `pytest.mark.parametrize()` signature: allow to pass a CSV-separated string to specify argnames. For example: `pytest.mark.parametrize("input,expected", [(1,2), (2,3)])` works as well as the previous: `pytest.mark.parametrize(("input", "expected"), ...)`.
- add support for `setUpModule`/`tearDownModule` detection, thanks Brian Okken.
- integrate tab-completion on options through use of “`argcomplete`”. Thanks Anthon van der Neut for the PR.
- change option names to be hyphen-separated long options but keep the old spelling backward compatible. `pytest -h` will only show the hyphenated version, for example “`–collect-only`” but “`–collectonly`” will remain valid as well (for backward-compat reasons). Many thanks to Anthon van der Neut for the implementation and to Hynek Schlawack for pushing us.

- fix issue 308 - allow to mark/xfail/skip individual parameter sets when parametrizing. Thanks Brianna Laughher.
- call new experimental `pytest_load_initial_conftests` hook to allow 3rd party plugins to do something before a conftest is loaded.

Bug fixes:

- fix issue358 - capturing options are now parsed more properly by using a new `parser.parse_known_args` method.
- pytest now uses `argparse` instead of `optparse` (thanks Anthon) which means that “`argparse`” is added as a dependency if installing into python2.6 environments or below.
- fix issue333: fix a case of bad unittest/pytest hook interaction.
- PR27: correctly handle `nose.SkipTest` during collection. Thanks Antonio Cuni, Ronny Pfannschmidt.
- fix issue355: junitxml puts `name="pytest"` attribute to testsuite tag.
- fix issue336: autouse fixture in plugins should work again.
- fix issue279: improve object comparisons on assertion failure for standard datatypes and recognise collections.abc. Thanks to Brianna Laughher and Mathieu Agopian.
- fix issue317: assertion rewriter support for the `is_package` method
- fix issue335: document `py.code.ExceptionInfo()` object returned from `pytest.raises()`, thanks Mathieu Agopian.
- remove implicit `distribute_setup` support from `setup.py`.
- fix issue305: ignore any problems when writing pyc files.
- SO-17664702: call fixture finalizers even if the fixture function partially failed (finalizers would not always be called before)
- fix issue320 - fix class scope for fixtures when mixed with module-level functions. Thanks Anatloy Bubenkoff.
- you can specify “-q” or “-qq” to get different levels of “quieter” reporting (thanks Katarzyna Jachim)
- fix issue300 - Fix order of conftest loading when starting `py.test` in a subdirectory.
- fix issue323 - sorting of many module-scoped arg parametrizations
- make sessionfinish hooks execute with the same cwd-context as at session start (helps fix plugin behaviour which write output files with relative path such as `pytest-cov`)
- fix issue316 - properly reference collection hooks in docs
- fix issue 306 - cleanup of -k/-m options to only match markers/test names/keywords respectively. Thanks Wouter van Ackooy.
- improved doctest counting for doctests in python modules – files without any doctest items will not show up anymore and doctest examples are counted as separate test items. thanks Danilo Bellini.
- fix issue245 by depending on the released py-1.4.14 which fixes `py.io.dupfile` to work with files with no mode. Thanks Jason R. Coombs.
- fix junitxml generation when test output contains control characters, addressing issue267, thanks Jaap Broekhuizen
- fix issue338: honor `-tb` style for setup/teardown errors as well. Thanks Maho.
- fix issue307 - use `yaml.safe_load` in example, thanks Mark Eichin.
- better parametrize error messages, thanks Brianna Laughher
- `pytest_terminal_summary(terminalreporter)` hooks can now use `”.section(title)”` and `”.line(msg)”` methods to print extra information at the end of a test run.

13.12 v2.3.5

- fix issue169: respect `-tb=style` with setup/teardown errors as well.
- 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 doesn't try to teardown if it did not setup
- issue 271 - don't write junitxml on slave nodes
- Issue 274 - don't 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 putting 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

13.13 v2.3.4

- 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.

13.14 v2.3.3

- 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 Taumoe folau
- 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.

13.15 v2.3.2

- 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

13.16 v2.3.1

- 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.

13.17 v2.3.0

- 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 decorated 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: properly 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 comparisons instead of truncating

13.18 v2.2.4

- 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: properly 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

13.19 v2.2.3

- fix uploaded package to only include necessary files

13.20 v2.2.2

- 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)

13.21 v2.2.1

- fix issue99 (in pytest and py) internalerrors 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)

13.22 v2.2.0

- 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 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

13.23 v2.1.3

- 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

13.24 v2.1.2

- 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`

13.25 v2.1.1

- 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”

13.26 v2.1.0

- 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 releaslevel 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

13.27 v2.0.3

- 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 representation internally)
- fix issue37: avoid invalid characters in junitxml's output

13.28 v2.0.2

- 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 Laughner) and particular thanks to Laura Creighton who also reviewed parts of the documentation.
- fix slightly wrong output of verbose progress reporting for classes (thanks Amaury)
- more precise (avoiding of) deprecation warnings for `node.ClassFunction` accesses
- avoid std unittest assertion helper code in tracebacks (thanks Ronny)

13.29 v2.0.1

- 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 parametrization remains the “pytest_generate_tests” mechanism, see the docs.

13.30 v2.0.0

- 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 invocation: 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 any-more 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

13.31 v1.3.4

- 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

13.32 v1.3.3

- 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

13.33 v1.3.2

13.33.1 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.

13.33.2 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

13.34 v1.3.1

13.34.1 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.

13.34.2 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 seamlessly 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)

13.35 v1.3.0

- 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 specifying `--runxfail` on command line virtually ignores xfail markers
- expose (previously internal) commonly useful methods: `py.io.get_terminal_width()` -> 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

13.36 v1.2.0

- 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-recorded 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

13.37 v1.1.1

- moved `dist/loopenfailing` 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 reinterpretation 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

13.38 v1.1.0

- 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

13.39 v1.0.2

- 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.doctestsubprocess``textwraploptparse`
- 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

13.40 v1.0.2

- 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.

13.41 v1.0.1

- 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

13.42 v1.0.0

- more terse reporting try to show filesystem path relatively to current dir
- improve xfail output a bit

13.43 v1.0.0b9

- 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)

13.44 v1.0.0b8

- `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.

13.45 v1.0.0b7

- 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 Peolzeithner (issue #23)
- resolve issue #18, `multiprocessing.Manager()` and redirection clash
- make `__name__` == “`__channelexec__`” for `remote_exec` code

13.46 v1.0.0b3

- 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.

13.47 v1.0.0b1

- introduced new “`funcarg`” setup method, see `doc/test/funcarg.txt`
- introduced plugin architecture 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

13.48 v0.9.2

- 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

13.49 v0.9.1

This is a fairly complete list of v0.9.1, 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 ignored 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 `list.sort()` to support Python 2.3 [52481]

A

add() (MarkInfo method), 56
 add_marker() (Node method), 78
 addcall() (Metafunc method), 46
 addfinalizer() (FixtureRequest method), 23
 addfinalizer() (Node method), 79
 addini() (Parser method), 78
 addinivalue_line() (Config method), 77
 adoption() (Parser method), 78
 addopts
 configuration value, 26
 applymarker() (FixtureRequest method), 24
 args (MarkInfo attribute), 56

C

cached_setup() (FixtureRequest method), 24
 CallInfo (class in `_pytest.runner`), 79
 chdir() (monkeypatch method), 51
 Class (class in `_pytest.python`), 79
 cls (FixtureRequest attribute), 23
 collect() (Collector method), 79
 Collector (class in `_pytest.main`), 79
 Collector.CollectError, 79
 Config (class in `_pytest.config`), 77
 config (FixtureRequest attribute), 23
 config (Node attribute), 78
 configuration value
 addopts, 26
 minversion, 26
 norecursedirs, 26
 python_classes, 27
 python_files, 27
 python_functions, 27

D

delattr() (monkeypatch method), 50
 delenv() (monkeypatch method), 51
 delitem() (monkeypatch method), 51
 deprecated_call() (in module `pytest`), 22
 duration (TestReport attribute), 80

E

excinfo (CallInfo attribute), 80
 exit() (in module `_pytest.runner`), 22
 extra_keyword_matches (Node attribute), 78

F

fail() (in module `_pytest.runner`), 22
 fixture() (in module `_pytest.python`), 22
 fixturename (FixtureRequest attribute), 23
 FixtureRequest (class in `_pytest.python`), 23
 fromdictargs() (`_pytest.config.Config` class method), 77
 fspath (FixtureRequest attribute), 23
 fspath (Node attribute), 78
 Function (class in `_pytest.python`), 79
 function (FixtureRequest attribute), 23
 function (Function attribute), 79

G

get_marker() (Node method), 79
 getfuncargvalue() (FixtureRequest method), 24
 getgroup() (Parser method), 77
 getini() (Config method), 77
 getoption() (Config method), 77
 getparent() (Node method), 79
 getvalue() (Config method), 77
 getvalueorskip() (Config method), 77

I

ihook (Node attribute), 78
 importorskip() (in module `_pytest.runner`), 22
 instance (FixtureRequest attribute), 23
 Item (class in `_pytest.main`), 79

K

keywords (FixtureRequest attribute), 23
 keywords (Node attribute), 78
 keywords (TestReport attribute), 80
 kwargs (MarkInfo attribute), 56

L

listchain() (Node method), 78
 listextrakeywords() (Node method), 79

location (TestReport attribute), 80
 longrepr (TestReport attribute), 80

M

main() (in module pytest), 21
 MarkDecorator (class in `_pytest.mark`), 55
 MarkGenerator (class in `_pytest.mark`), 55
 MarkInfo (class in `_pytest.mark`), 55
 minversion
 configuration value, 26
 Module (class in `_pytest.python`), 79
 module (FixtureRequest attribute), 23
 monkeypatch (class in `_pytest.monkeypatch`), 50

N

name (MarkInfo attribute), 55
 name (Node attribute), 78
 Node (class in `_pytest.main`), 78
 node (FixtureRequest attribute), 23
 nodeid (Node attribute), 78
 nodeid (TestReport attribute), 80
 norecursedirs
 configuration value, 26

O

option (Config attribute), 77
 outcome (TestReport attribute), 80

P

parametrize() (Metafunc method), 46
 parent (Node attribute), 78
 Parser (class in `_pytest.config`), 77
 pluginmanager (Config attribute), 77
 pytest_addhooks() (in module `_pytest.hookspec`), 75
 pytest_addoption() (in module `_pytest.hookspec`), 73
 pytest_assertrepr_compare() (in module `_pytest.hookspec`), 29
 pytest_cmdline_main() (in module `_pytest.hookspec`), 73
 pytest_cmdline_parse() (in module `_pytest.hookspec`), 73
 pytest_cmdline_preparse() (in module `_pytest.hookspec`), 73
 pytest_collect_directory() (in module `_pytest.hookspec`), 74
 pytest_collect_file() (in module `_pytest.hookspec`), 74
 pytest_collection_modifyitems() (in module `_pytest.hookspec`), 75
 pytest_collectreport() (in module `_pytest.hookspec`), 75
 pytest_collectstart() (in module `_pytest.hookspec`), 75
 pytest_configure() (in module `_pytest.hookspec`), 73
 pytest_deselected() (in module `_pytest.hookspec`), 75
 pytest_exception_interact() (in module `_pytest.hookspec`), 75
 pytest_generate_tests() (in module `_pytest.hookspec`), 75

pytest_ignore_collect() (in module `_pytest.hookspec`), 74
 pytest_internalerror() (in module `_pytest.hookspec`), 75
 pytest_itemcollected() (in module `_pytest.hookspec`), 75
 pytest_keyboard_interrupt() (in module `_pytest.hookspec`), 75
 pytest_load_initial_conftests() (in module `_pytest.hookspec`), 73
 pytest_namespace() (in module `_pytest.hookspec`), 73
 pytest_pycollect_makeitem() (in module `_pytest.hookspec`), 75
 pytest_runtest_call() (in module `_pytest.hookspec`), 74
 pytest_runtest_logreport() (in module `_pytest.hookspec`), 75
 pytest_runtest_makereport() (in module `_pytest.hookspec`), 74
 pytest_runtest_protocol() (in module `_pytest.hookspec`), 74
 pytest_runtest_setup() (in module `_pytest.hookspec`), 74
 pytest_runtest_teardown() (in module `_pytest.hookspec`), 74
 pytest_unconfigure() (in module `_pytest.hookspec`), 74
 Python Enhancement Proposals
 PEP 8, 4
 python_classes
 configuration value, 27
 python_files
 configuration value, 27
 python_functions
 configuration value, 27

R

raiseerror() (FixtureRequest method), 24
 raises() (in module `pytest`), 21
 repr_failure() (Collector method), 79
 runtest() (Function method), 79

S

scope (FixtureRequest attribute), 23
 sections (TestReport attribute), 80
 session (FixtureRequest attribute), 23
 session (Node attribute), 78
 setattr() (monkeypatch method), 50
 setenv() (monkeypatch method), 51
 setitem() (monkeypatch method), 50
 skip() (in module `_pytest.runner`), 22
 syspath_prepend() (monkeypatch method), 51

T

TestReport (class in `_pytest.runner`), 80

U

undo() (monkeypatch method), 51

W

warn() (Config method), [77](#)
warn() (Node method), [78](#)
when (CallInfo attribute), [79](#)
when (TestReport attribute), [80](#)

X

xfail() (in module `_pytest.skipping`), [22](#)