pytest Documentation

Release 2.9.1

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

April 13, 2016

Download latest version as PDF

CHAPTER

ONE

GETTING STARTED BASICS

1.1 Installation and Getting Started

Pythons: Python 2.6,2.7,3.3,3.4,3.5, Jython, PyPy-2.3

Platforms: Unix/Posix and Windows

PyPI package name: pytest

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

documentation as PDF: download latest

1.1.1 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 2.9.1, imported from $PYTHON_PREFIX/lib/python3.4/site-packages/pytest.py
```

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

1.1.2 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
```

```
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 0.12 seconds ========
```

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 assert introspection will intelligently report intermediate values of the assert expression freeing you from the need to learn the many names of JUnit legacy methods.

1.1.3 Running multiple tests

`pytest` will run all files in the current directory and its subdirectories of the form test_*.py or *_test.py. More generally, it follows *standard test discovery rules*.

1.1.4 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 0.12 seconds
```

1.1.5 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 0xdeadbeef>

    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 0.12 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.

1.1.6 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 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('PYTEST_TMPDIR/test_needsfiles0')

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

test_tmpdir.py:3: AssertionError
--------------------------- Captured stdout call ---------------------------
PYTEST_TMPDIR/test_needsfiles0
1 failed in 0.12 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 fixtures exist by typing:

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

1.1.7 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 practices* for virtualenv, test layout, genscript support
- fixtures for providing a functional baseline to your tests
- apiref for documentation and examples on using `pytest`
- plugins managing and writing plugins

1.1.8 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

1.2 Usage and Invocations

1.2.1 Calling pytest through `python -m pytest`

New in version 2.0.

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.

1.2.2 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
```

1.2.3 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
```

1.2.4 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
                      # "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
py.test test_mod.py::TestClass::test_method  # run a single method in
                                             # a single class
```

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

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

1.2.5 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=auto    # (default) 'long' tracebacks for the first and last
                     # entry, but 'short' style for the other entries
py.test --tb=long    # exhaustive, informative traceback formatting
py.test --tb=short   # shorter traceback format
py.test --tb=line    # only one line per failure
py.test --tb=native  # Python standard library formatting
py.test --tb=no      # no traceback at all
```

1.2.6 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
```

Note that on any failure the exception information is stored on `sys.last_value`, `sys.last_type` and `sys.last_traceback`. In interactive use, this allows one to drop into postmortem debugging with any debug tool. One can also manually access the exception information, for example:

```
>>> import sys
>>> sys.last_traceback.tb_lineno
42
>>> sys.last_value
AssertionError('assert result == "ok"',)
```

1.2.7 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`.

1.2.8 Profiling test execution duration

To get a list of the slowest 10 test durations:

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

1.2.9 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`.

record_xml_property

New in version 2.8.

If you want to log additional information for a test, you can use the `record_xml_property` fixture:

```
def test_function(record_xml_property):
    record_xml_property("example_key", 1)
    assert 0
```

This will add an extra property `example_key="1"` to the generated `testcase` tag:

```
<testcase classname="test_function" file="test_function.py" line="0" name="test_function" time="0.000
  <properties>
    <property name="example_key" value="1" />
  </properties>
</testcase>
```

> **Warning:** This is an experimental feature, and its interface might be replaced by something more powerful and general in future versions. The functionality per-se will be kept, however.
> Currently it does not work when used with the `pytest-xdist` plugin.
> Also please note that using this feature will break any schema verification. This might be a problem when used with some CI servers.

1.2.10 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.

1.2.11 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.

1.2.12 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
```

1.2.13 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
```

1.3 Good Integration Practices

1.3.1 Conventions for Python test discovery

`pytest` implements the following standard test discovery:

- If no arguments are specified then collection starts from `testpaths` (if configured) or the current directory. Alternatively, command line arguments can be used in any combination of directories, file names or node 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.

1.3.2 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 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 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.

1.3.3 Tox

For development, we recommend to use virtualenv environments and pip for installing your application and any dependencies as well as the `pytest` package itself. This ensures your code and dependencies are isolated from the system Python installation.

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.

Continuous integration services such as Jenkins can make use of the `--junitxml=PATH` option to create a JUnitXML file and generate reports.

1.3.4 Integrating with setuptools / `python setup.py test` / `pytest-runner`

You can integrate test runs into your setuptools based project with the pytest-runner plugin.

Add this to `setup.py` file:

```
from setuptools import setup

setup(
    #...,
    setup_requires=['pytest-runner', ...],
    tests_require=['pytest', ...],
    #...,
)
```

And create an alias into `setup.cfg` file:

```
[aliases]
test=pytest
```

If you now type:

```
python setup.py test
```

this will execute your tests using `pytest-runner`. 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 py.test such as your test directory or other options using `--addopts`.

Manual Integration

If for some reason you don't want/can't use `pytest-runner`, you can write your own setuptools Test command for invoking pytest.

```
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 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`.

1.3.5 (deprecated) Create a pytest standalone script

Deprecated since version 2.8.

Note: `genscript` has been deprecated because:

- It cannot support plugins, rendering its usefulness extremely limited;
- Tooling has become much better since `genscript` was introduced;
- It is possible to build a zipped `pytest` application without the shortcomings above.

There's no planned version in which this command will be removed at the moment of this writing, but its use is discouraged for new applications.

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
```

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.

1.4 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
- Astropy and affiliated packages
- 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

1.4.1 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
- ESSS, Brazil
- many more ... (please be so kind to send a note via contact)

1.5 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.

1.5.1 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 you to return deferreds from pytest-style tests, allowing the use of fixtures 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 the use of 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: 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 --assert=plain 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.

1.5.2 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 fixtures may feel more natural. Moreover, fixtures are ideal for writing advanced test support code (like e.g. the monkeypatch, the tmpdir or capture fixtures) 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 @pytest.fixture 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. See also *Parametrizing tests* for more examples.

1.5.3 pytest interaction with other packages

Issues with pytest, multiprocess and setuptools?

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

As of mid-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).

CHAPTER

TWO

USAGES AND EXAMPLES

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

For basic examples, see

- Installation and Getting Started for basic introductory examples
- assert for basic assertion examples
- fixtures for basic fixture/setup examples
- parametrize for basic test function parametrization
- `../unittest` for basic unittest integration
- `../nose` for basic nosetests integration

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

2.1 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR/assertion, inifile:
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:16: AssertionError
_______ TestFailing.test_simple _________
```

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

    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 0xdeadbeef>()
E        +  and   43 = <function TestFailing.test_simple.<locals>.g at 0xdeadbeef>()

failure_demo.py:29: AssertionError
_______ TestFailing.test_simple_multiline ________

self = <failure_demo.TestFailing object at 0xdeadbeef>

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

failure_demo.py:34:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 42, b = 54

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

failure_demo.py:12: AssertionError
_______ TestFailing.test_not _________

self = <failure_demo.TestFailing object at 0xdeadbeef>

    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 0xdeadbeef>()

failure_demo.py:39: AssertionError
_______ TestSpecialisedExplanations.test_eq_text _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

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

failure_demo.py:43: AssertionError
_______ TestSpecialisedExplanations.test_eq_similar_text _________
```

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

    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:46: AssertionError
_______ TestSpecialisedExplanations.test_eq_multiline_text ________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:49: AssertionError
_______ TestSpecialisedExplanations.test_eq_long_text _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

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

failure_demo.py:54: AssertionError
_______ TestSpecialisedExplanations.test_eq_long_text_multiline ________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

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

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

failure_demo.py:59: AssertionError
_______ TestSpecialisedExplanations.test_eq_list _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:62: AssertionError
_______ TestSpecialisedExplanations.test_eq_list_long _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_list_long(self):
        a = [0]*100 + [1] + [3]*100
        b = [0]*100 + [2] + [3]*100
>       assert a == b
E       assert [0, 0, 0, 0, 0, 0, ...] == [0, 0, 0, 0, 0, 0, ...]
E         At index 100 diff: 1 != 2
E         Use -v to get the full diff

failure_demo.py:67: AssertionError
_______ TestSpecialisedExplanations.test_eq_dict _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:70: AssertionError
_______ TestSpecialisedExplanations.test_eq_set _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:73: AssertionError
_______ TestSpecialisedExplanations.test_eq_longer_list _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:76: AssertionError
_______ TestSpecialisedExplanations.test_in_list _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:79: AssertionError
_______ TestSpecialisedExplanations.test_not_in_text_multiline _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:83: AssertionError
_______ TestSpecialisedExplanations.test_not_in_text_single _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    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:87: AssertionError
```

```
_______ TestSpecialisedExplanations.test_not_in_text_single_long ________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_not_in_text_single_long(self):
        text = 'head ' * 50 + 'foo ' + 'tail ' * 20
>       assert 'foo' not in text
E       assert 'foo' not in 'head head head head hea...ail tail tail tail '
E         'foo' is contained here:
E           head head foo tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail
E         ?           +++

failure_demo.py:91: AssertionError
______ TestSpecialisedExplanations.test_not_in_text_single_long_term _______

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

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

failure_demo.py:95: 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 0xdeadbeef>.b

failure_demo.py:102: 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 0xdeadbeef>.b
E        +    where <failure_demo.test_attribute_instance.<locals>.Foo object at 0xdeadbeef> = <class

failure_demo.py:108: 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:117:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <failure_demo.test_attribute_failure.<locals>.Foo object at 0xdeadbeef>

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

failure_demo.py:114: 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 0xdeadbeef>.b
E        +    where <failure_demo.test_attribute_multiple.<locals>.Foo object at 0xdeadbeef> = <class
E        +  and   2 = <failure_demo.test_attribute_multiple.<locals>.Bar object at 0xdeadbeef>.b
E        +    where <failure_demo.test_attribute_multiple.<locals>.Bar object at 0xdeadbeef> = <class

failure_demo.py:125: AssertionError
_______ TestRaises.test_raises _________

self = <failure_demo.TestRaises object at 0xdeadbeef>

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

failure_demo.py:134:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

<0-codegen $PYTHON_PREFIX/lib/python3.4/site-packages/_pytest/python.py:1302>:1: ValueError
_______ TestRaises.test_raises_doesnt _________

self = <failure_demo.TestRaises object at 0xdeadbeef>

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

failure_demo.py:137: Failed
_______ TestRaises.test_raise _________

self = <failure_demo.TestRaises object at 0xdeadbeef>

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

failure_demo.py:140: ValueError
```

```
_______ TestRaises.test_tupleerror ________

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_tupleerror(self):
>       a,b = [1]
E       ValueError: need more than 1 value to unpack

failure_demo.py:143: ValueError
______ TestRaises.test_reinterpret_fails_with_print_for_the_fun_of_it ______

self = <failure_demo.TestRaises object at 0xdeadbeef>

    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:148: TypeError
--------------------------- Captured stdout call ---------------------------
l is [1, 2, 3]
_______ TestRaises.test_some_error ________

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_some_error(self):
>       if namenotexi:
E       NameError: name 'namenotexi' is not defined

failure_demo.py:151: 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 = _pytest._code.compile(src, name, 'exec')
        py.builtin.exec_(code, module.__dict__)
        py.std.sys.modules[name] = module
>       module.foo()

failure_demo.py:166:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    def foo():
>    assert 1 == 0
E    assert 1 == 0

<2-codegen 'abc-123' $REGENDOC_TMPDIR/assertion/failure_demo.py:163>:2: AssertionError
_______ TestMoreErrors.test_complex_error ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_complex_error(self):
        def f():
            return 44
        def g():
```

```
            return 43
>       somefunc(f(), g())

failure_demo.py:176:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
failure_demo.py:9: in somefunc
    otherfunc(x,y)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 44, b = 43

    def otherfunc(a,b):
>       assert a==b
E       assert 44 == 43

failure_demo.py:6: AssertionError
_______ TestMoreErrors.test_z1_unpack_error ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_z1_unpack_error(self):
        l = []
>       a,b  = l
E       ValueError: need more than 0 values to unpack

failure_demo.py:180: ValueError
_______ TestMoreErrors.test_z2_type_error ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_z2_type_error(self):
        l = 3
>       a,b  = l
E       TypeError: 'int' object is not iterable

failure_demo.py:184: TypeError
_______ TestMoreErrors.test_startswith ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

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

failure_demo.py:189: AssertionError
_______ TestMoreErrors.test_startswith_nested ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

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

failure_demo.py:196: AssertionError
_______ TestMoreErrors.test_global_func ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_global_func(self):
>       assert isinstance(globf(42), float)
E       assert isinstance(43, float)
E        +  where 43 = globf(42)

failure_demo.py:199: AssertionError
_______ TestMoreErrors.test_instance ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

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

failure_demo.py:203: AssertionError
_______ TestMoreErrors.test_compare ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_compare(self):
>       assert globf(10) < 5
E       assert 11 < 5
E        +  where 11 = globf(10)

failure_demo.py:206: AssertionError
_______ TestMoreErrors.test_try_finally ________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

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

failure_demo.py:211: AssertionError
_______ TestCustomAssertMsg.test_single_line ________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    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:222: AssertionError
_______ TestCustomAssertMsg.test_multiline ________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    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:228: AssertionError
_______ TestCustomAssertMsg.test_custom_repr ________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    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:238: AssertionError
======= 42 failed in 0.12 seconds ========
```

2.2 Basic patterns and examples

2.2.1 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 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 0.12 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 0.12 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.

2.2.2 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 0 items

======= no tests ran in 0.12 seconds ========
```

2.2.3 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")
```

We can now write a test module like this:

```
# content of test_module.py

import pytest


slow = pytest.mark.skipif(
    not pytest.config.getoption("--runslow"),
    reason="need --runslow option to run"
)


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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py .s
======= short test summary info ========
SKIP [1] test_module.py:14: need --runslow option to run

======= 1 passed, 1 skipped in 0.12 seconds ========
```

Or run it including the `slow` marked test:

```
$ py.test --runslow
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py ..

======= 2 passed in 0.12 seconds ========
```

2.2.4 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 0.12 seconds
```

2.2.5 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.

2.2.6 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
project deps: mylib-1.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 0 items

======= no tests ran in 0.12 seconds ========
```

You can also return a list of strings which will be considered as several lines of information. You can of course also make the amount of reporting information on e.g. the value of `config.option.verbose` so that you present more information appropriately:

```
# content of conftest.py

def pytest_report_header(config):
    if config.option.verbose > 0:
        return ["info1: did you know that ...", "did you?"]
```

which will add info only when run with "–v":

```
$ py.test -v
======= test session starts ========
```

```
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
info1: did you know that ...
did you?
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 0 items

======= no tests ran in 0.12 seconds ========
```

and nothing when run plainly:

```
$ py.test
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 0 items

======= no tests ran in 0.12 seconds ========
```

2.2.7 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 3 items

test_some_are_slow.py ...

======= slowest 3 test durations ========
0.20s call     test_some_are_slow.py::test_funcslow2
0.10s call     test_some_are_slow.py::test_funcslow1
0.00s setup    test_some_are_slow.py::test_funcfast
======= 3 passed in 0.12 seconds ========
```

2.2.8 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 4 items

test_step.py .Fx.
======= short test summary info ========
XFAIL test_step.py::TestUserHandling::()::test_deletion
  reason: previous test failed (test_modification)

======= FAILURES ========
_______ TestUserHandling.test_modification _________

self = <test_step.TestUserHandling object at 0xdeadbeef>
```

```
    def test_modification(self):
>       assert 0
E       assert 0

test_step.py:9: AssertionError
======= 1 failed, 2 passed, 1 xfailed in 0.12 seconds ========
```

We'll see that `test_deletion` was not executed because `test_modification` failed. It is reported as an "expected failure".

2.2.9 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 a an example for making a `db` fixture available in a directory:

```
# content of a/conftest.py
import pytest

class DB:
    pass

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

and then a test module in that directory:

```
# content of a/test_db.py
def test_a1(db):
    assert 0, db  # to show value
```

another test module:

```
# content of a/test_db2.py
def test_a2(db):
    assert 0, db  # to show value
```

and then a module in a sister directory which will not see the `db` fixture:

```
# content of b/test_error.py
def test_root(db):  # no db here, will error out
    pass
```

We can run this:

```
$ py.test
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
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 $REGENDOC_TMPDIR/b/test_error.py, line 1
  def test_root(db):  # no db here, will error out
        fixture 'db' not found
        available fixtures: record_xml_property, recwarn, cache, capsys, pytestconfig, tmpdir_factory
        use 'py.test --fixtures [testpath]' for help on them.

$REGENDOC_TMPDIR/b/test_error.py:1
======= FAILURES ========
_______ TestUserHandling.test_modification _________

self = <test_step.TestUserHandling object at 0xdeadbeef>

    def test_modification(self):
>       assert 0
E       assert 0

test_step.py:9: AssertionError
_______ test_a1 _________

db = <conftest.DB object at 0xdeadbeef>

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

a/test_db.py:2: AssertionError
_______ test_a2 _________

db = <conftest.DB object at 0xdeadbeef>

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

a/test_db2.py:2: AssertionError
======= 3 failed, 2 passed, 1 xfailed, 1 error in 0.12 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).

2.2.10 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.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    # execute all other hooks to obtain the report object
    outcome = yield
    rep = outcome.get_result()

    # 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.fixturenames:
                extra = " (%s)" % item.funcargs["tmpdir"]
            else:
                extra = ""

            f.write(rep.nodeid + extra + "\n")
```

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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py FF

======= FAILURES ========
_______ test_fail1 _________

tmpdir = local('PYTEST_TMPDIR/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 0.12 seconds ========
```

you will have a "failures" file which contains the failing test ids:

```
$ cat failures
test_module.py::test_fail1 (PYTEST_TMPDIR/test_fail10)
test_module.py::test_fail2
```

2.2.11 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.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    # execute all other hooks to obtain the report object
    outcome = yield
    rep = outcome.get_result()

    # set an report attribute for each phase of a call, which can
    # be "setup", "call", "teardown"

    setattr(item, "rep_" + rep.when, 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
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 0.12 seconds ========
```

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

2.2.12 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 --junitxml=results.xml test-suite/
```

2.3 Parametrizing tests

`pytest` allows to easily parametrize test functions. For basic docs, see parametrize-basics.

In the following we provide some examples using the builtin mechanisms.

2.3.1 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 0.12 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 0.12 seconds
```

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

2.3.2 Different options for test IDs

pytest will build a string that is the test ID for each set of values in a parametrized test. These IDs can be used with `-k` to select specific cases to run, and they will also identify the specific case when one is failing. Running pytest with `--collect-only` will show the generated IDs.

Numbers, strings, booleans and None will have their usual string representation used in the test ID. For other objects, pytest will make a string based on the argument name:

```
# content of test_time.py

import pytest

from datetime import datetime, timedelta

testdata = [
    (datetime(2001, 12, 12), datetime(2001, 12, 11), timedelta(1)),
    (datetime(2001, 12, 11), datetime(2001, 12, 12), timedelta(-1)),
]


@pytest.mark.parametrize("a,b,expected", testdata)
def test_timedistance_v0(a, b, expected):
    diff = a - b
    assert diff == expected


@pytest.mark.parametrize("a,b,expected", testdata, ids=["forward", "backward"])
def test_timedistance_v1(a, b, expected):
```

```
    diff = a - b
    assert diff == expected


def idfn(val):
    if isinstance(val, (datetime,)):
        # note this wouldn't show any hours/minutes/seconds
        return val.strftime('%Y%m%d')


@pytest.mark.parametrize("a,b,expected", testdata, ids=idfn)
def test_timedistance_v2(a, b, expected):
    diff = a - b
    assert diff == expected
```

In `test_timedistance_v0`, we let pytest generate the test IDs.

In `test_timedistance_v1`, we specified `ids` as a list of strings which were used as the test IDs. These are succinct, but can be a pain to maintain.

In `test_timedistance_v2`, we specified `ids` as a function that can generate a string representation to make part of the test ID. So our `datetime` values use the label generated by `idfn`, but because we didn't generate a label for `timedelta` objects, they are still using the default pytest representation:

```
$ py.test test_time.py --collect-only
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 6 items
<Module 'test_time.py'>
  <Function 'test_timedistance_v0[a0-b0-expected0]'>
  <Function 'test_timedistance_v0[a1-b1-expected1]'>
  <Function 'test_timedistance_v1[forward]'>
  <Function 'test_timedistance_v1[backward]'>
  <Function 'test_timedistance_v2[20011212-20011211-expected0]'>
  <Function 'test_timedistance_v2[20011211-20011212-expected1]'>

======= no tests ran in 0.12 seconds ========
```

2.3.3 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 4 items

test_scenarios.py ....

======= 4 passed in 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
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]'>

======= no tests ran in 0.12 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.

2.3.4 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items
<Module 'test_backends.py'>
  <Function 'test_db_initialized[d1]'>
  <Function 'test_db_initialized[d2]'>

======= no tests ran in 0.12 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 0xdeadbeef>

    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 0.12 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.

2.3.5 Apply indirect on particular arguments

Very often parametrization uses more than one argument name. There is opportunity to apply `indirect` parameter on particular arguments. It can be done by passing list or tuple of arguments' names to `indirect`. In the example below there is a function `test_indirect` which uses two fixtures: `x` and `y`. Here we give to indirect the list, which contains the name of the fixture `x`. The indirect parameter will be applied to this argument only, and the value `a` will be passed to respective fixture function:

```
# content of test_indirect_list.py

import pytest
@pytest.fixture(scope='function')
def x(request):
    return request.param * 3

@pytest.fixture(scope='function')
def y(request):
    return request.param * 2

@pytest.mark.parametrize('x, y', [('a', 'b')], indirect=['x'])
def test_indirect(x,y):
    assert x == 'aaa'
    assert y == 'b'
```

The result of this test will be successful:

```
$ py.test test_indirect_list.py --collect-only
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 items
<Module 'test_indirect_list.py'>
  <Function 'test_indirect[a-b]'>

======= no tests ran in 0.12 seconds ========
```

2.3.6 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 parametrizer but in a lot less code:

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

def pytest_generate_tests(metafunc):
    # called once per each test function
    funcarglist = metafunc.cls.params[metafunc.function.__name__]
    argnames = list(funcarglist[0])
    metafunc.parametrize(argnames, [[funcargs[name] for name in argnames]
            for funcargs in funcarglist])

class TestClass:
    # a map specifying multiple argument sets for a test method
    params = {
        'test_equals': [dict(a=1, b=2), dict(a=3, b=3), ],
        'test_zerodivision': [dict(a=1, b=0), ],
    }
```

```
    def test_equals(self, a, b):
        assert a == b

    def test_zerodivision(self, a, b):
        pytest.raises(ZeroDivisionError, "a/b")
```

Our test generator looks up a class-level definition which specifies which argument sets to use for each test function. Let's run it:

```
$ py.test -q
F..
======= FAILURES ========
_______ TestClass.test_equals[1-2] _________

self = <test_parametrize.TestClass object at 0xdeadbeef>, 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 0.12 seconds
```

2.3.7 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
import _pytest._code

pythonlist = ['python2.6', 'python2.7', 'python3.3']
@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(_pytest._code.Source("""
            import pickle
            f = open(%r, 'wb')
            s = pickle.dump(%r, f, protocol=2)
            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(_pytest._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
ssssssssssss...ssssssssssss
======= short test summary info ========
SKIP [12] $REGENDOC_TMPDIR/CWD/multipython.py:23: 'python3.3' not found
SKIP [12] $REGENDOC_TMPDIR/CWD/multipython.py:23: 'python2.6' not found
3 passed, 24 skipped in 0.12 seconds
```

2.3.8 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py .s
======= short test summary info ========
SKIP [1] $REGENDOC_TMPDIR/conftest.py:10: could not import 'opt2'

======= 1 passed, 1 skipped in 0.12 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.

2.4 Working with custom markers

Here are some example using the mark mechanism.

2.4.1 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items

test_server.py::test_send_http PASSED

======= 3 tests deselected by "-m 'webtest'" ========
======= 1 passed, 3 deselected in 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
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 0.12 seconds ========
```

2.4.2 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 5 items

```

```
test_server.py::TestClass::test_method PASSED

======= 1 passed in 0.12 seconds ========
```

You can also select on the class:

```
$ py.test -v test_server.py::TestClass
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items

test_server.py::TestClass::test_method PASSED

======= 1 passed in 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 8 items

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

======= 2 passed in 0.12 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`.

2.4.3 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items

test_server.py::test_send_http PASSED

======= 3 tests deselected by '-khttp' ========
======= 1 passed, 3 deselected in 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
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 0.12 seconds ========
```

Or to select "http" and "quick" tests:

```
$ py.test -k "http or quick" -v
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
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 0.12 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".

2.4.4 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 val

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

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

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

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

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

For an example on how to add and work with markers from a plugin, see *Custom marker and command line option to control test runs*.

Note: It is recommended to 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.

2.4.5 Marking whole classes or modules

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.

2.4.6 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.

Note: If the data you are parametrizing happen to be single callables, you need to be careful when marking these items. *pytest.mark.xfail(my_func)* won't work because it's also the signature of a function being decorated. To resolve this ambiguity, you need to pass a reason argument: *pytest.mark.xfail(func_bar, reason="Issue#7")*.

2.4.7 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.addinivalue_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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 items

test_someenv.py s

======= 1 skipped in 0.12 seconds ========
```

and here is one that specifies exactly the environment needed:

```
$ py.test -E stage1
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 items

test_someenv.py .

======= 1 passed in 0.12 seconds ========
```

The `--markers` option always gives you a list of available markers:

```
$ py.test --markers
@pytest.mark.env(name): mark test to run only on named environment

@pytest.mark.skipif(condition): skip the given test function if eval(condition) results in a True val

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

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

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

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

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

2.4.8 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:

```python
# 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 0.12 seconds
```

2.4.9 marking platform specific tests with pytest

Consider you have a test suite which marks tests for particular platforms, namely `pytest.mark.darwin`, `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("darwin 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.darwin
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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 4 items

test_plat.py sss.
======= short test summary info ========
SKIP [3] $REGENDOC_TMPDIR/conftest.py:12: cannot run on platform linux

======= 1 passed, 3 skipped in 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 4 items

test_plat.py s

======= 3 tests deselected by "-m 'linux2'" ========
======= 1 skipped, 3 deselected in 0.12 seconds ========
```

then the unmarked-tests will not be run. It is thus a way to restrict the run to the specific tests.

2.4.10 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
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 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
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 0.12 seconds ========
```

2.5 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 0.12 seconds
```

2.6 Changing standard (Python) test discovery

2.6.1 Ignore paths during test collection

You can easily ignore certain test directories and modules during collection by passing the `--ignore=path` option on the cli. `pytest` allows multiple `--ignore` options. Example:

```
tests/
|-- example
|   |-- test_example_01.py
|   |-- test_example_02.py
|   '-- test_example_03.py
|-- foobar
|   |-- test_foobar_01.py
|   |-- test_foobar_02.py
|   '-- test_foobar_03.py
'-- hello
    '-- world
        |-- test_world_01.py
        |-- test_world_02.py
        '-- test_world_03.py
```

Now if you invoke `pytest` with `--ignore=tests/foobar/test_foobar_03.py --ignore=tests/hello/`, you will see that `pytest` only collects test-modules, which do not match the patterns specified:

```
========= test session starts ==========
platform darwin -- Python 2.7.10, pytest-2.8.2, py-1.4.30, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 5 items

tests/example/test_example_01.py .
tests/example/test_example_02.py .
tests/example/test_example_03.py .
tests/foobar/test_foobar_01.py .
tests/foobar/test_foobar_02.py .

======= 5 passed in 0.02 seconds =======
```

2.6.2 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.

2.6.3 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 tests in files that match the `check_* .py` glob-pattern, `Check` prefixes in classes, and functions and methods that match `*_check`. For example, if we have:

```
# content of check_myapp.py
class CheckMyApp:
    def simple_check(self):
        pass
    def complex_check(self):
        pass
```

then the test collection looks like this:

```
$ py.test --collect-only
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile: setup.cfg
collected 2 items
<Module 'check_myapp.py'>
  <Class 'CheckMyApp'>
    <Instance '()'>
      <Function 'simple_check'>
      <Function 'complex_check'>

======= no tests ran in 0.12 seconds ========
```

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

2.6.4 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.

2.6.5 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 3 items
<Module 'CWD/pythoncollection.py'>
  <Function 'test_function'>
  <Class 'TestClass'>
    <Instance '()'>
      <Function 'test_method'>
      <Function 'test_anothermethod'>

======= no tests ran in 0.12 seconds ========
```

2.6.6 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
```

```
collected 0 items

======= no tests ran in 0.12 seconds ========
```

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

2.7 Working with non-python tests

2.7.1 A basic example for specifying tests in Yaml files

Here is an example `conftest.py` (extracted from Ali Afshnars special purpose pytest-yamlwsgi plugin). This `conftest.py` will collect `test*.yml` files and will execute the yaml-formatted content as custom tests:

```
# content of conftest.py

import pytest

def pytest_collect_file(parent, path):
    if path.ext == ".yml" and path.basename.startswith("test"):
        return YamlFile(path, parent)

class YamlFile(pytest.File):
    def collect(self):
        import yaml # we need a yaml parser, e.g. PyYAML
        raw = yaml.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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR/nonpython, inifile:
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 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- $PYTHON_PREFIX/bin/python3.4
cachedir: .cache
rootdir: $REGENDOC_TMPDIR/nonpython, inifile:
collecting ... collected 2 items

test_simple.yml::hello FAILED
test_simple.yml::ok PASSED

======= FAILURES ========
_______ usecase: hello _________
usecase execution failed
   spec failed: 'some': 'other'
   no further details known at this point.
======= 1 failed, 1 passed in 0.12 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR/nonpython, inifile:
```

```
collected 2 items
<YamlFile 'test_simple.yml'>
  <YamlItem 'hello'>
  <YamlItem 'ok'>

======= no tests ran in 0.12 seconds ========
```

CHAPTER

THREE

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.

3.1 Simple example: monkeypatching functions

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

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

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

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

3.2 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.

3.3 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: "/")
```

3.4 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 it 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, no exception will be raised if the attribute is missing.

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.

If `raising` is set to False, no exception will be raised if the key is missing.

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 the environment. Raise KeyError it does not exist.

If `raising` is set to False, no exception will be raised if the environment variable is missing.

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.

There is generally no need to call *undo()*, since it is called automatically during tear-down.

Note that the same *monkeypatch* fixture is used across a single test function invocation. If *monkeypatch* is used both by the test function itself and one of the test fixtures, calling *undo()* will undo all of the changes made in both functions.

`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.

CHAPTER

FOUR

TEMPORARY DIRECTORIES AND FILES

4.1 The 'tmpdir' fixture

You can use the `tmpdir` fixture 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 items

test_tmpdir.py F

======= FAILURES ========
_______ test_create_file _________

tmpdir = local('PYTEST_TMPDIR/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 0.12 seconds ========
```

4.2 The 'tmpdir_factory' fixture

New in version 2.8.

The `tmpdir_factory` is a session-scoped fixture which can be used to create arbitrary temporary directories from any other fixture or test.

For example, suppose your test suite needs a large image on disk, which is generated procedurally. Instead of computing the same image for each test that uses it into its own `tmpdir`, you can generate it once per-session to save time:

```
# contents of conftest.py
import pytest

@pytest.fixture(scope='session')
def image_file(tmpdir_factory):
    img = compute_expensive_image()
    fn = tmpdir_factory.mktemp('data').join('img.png')
    img.save(str(fn))
    return fn

# contents of test_image.py
def test_histogram(image_file):
    img = load_image(image_file)
    # compute and test histogram
```

`tmpdir_factory` instances have the following methods:

`TempdirFactory.`**`mktemp`**`(`*basename*, *numbered=True*`)`
Create a subdirectory of the base temporary directory and return it. If `numbered`, ensure the directory is unique by adding a number prefix greater than any existing one.

`TempdirFactory.`**`getbasetemp`**`()`
return base temporary directory.

4.3 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.

CHAPTER

FIVE

CAPTURING OF THE STDOUT/STDERR OUTPUT

5.1 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.

5.2 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
```

5.3 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, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
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 0xdeadbeef>
======= 1 failed, 1 passed in 0.12 seconds ========
```

5.4 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).

CHAPTER

SIX

ASSERTING WARNINGS

6.1 Asserting warnings with the warns function

New in version 2.8.

You can check that code raises a particular warning using `pytest.warns`, which works in a similar manner to raises:

```
import warnings
import pytest

def test_warning():
    with pytest.warns(UserWarning):
        warnings.warn("my warning", UserWarning)
```

The test will fail if the warning in question is not raised.

You can also call `pytest.warns` on a function or code string:

```
pytest.warns(expected_warning, func, *args, **kwargs)
pytest.warns(expected_warning, "func(*args, **kwargs)")
```

The function also returns a list of all raised warnings (as `warnings.WarningMessage` objects), which you can query for additional information:

```
with pytest.warns(RuntimeWarning) as record:
    warnings.warn("another warning", RuntimeWarning)

# check that only one warning was raised
assert len(record) == 1
# check that the message matches
assert record[0].message.args[0] == "another warning"
```

Alternatively, you can examine raised warnings in detail using the *recwarn* fixture (see below).

Note: `DeprecationWarning` and `PendingDeprecationWarning` are treated differently; see *Ensuring a function triggers a deprecation warning*.

6.2 Recording warnings

You can record raised warnings either using `pytest.warns` or with the `recwarn` fixture.

To record with `pytest.warns` without asserting anything about the warnings, pass `None` as the expected warning type:

```
with pytest.warns(None) as record:
    warnings.warn("user", UserWarning)
    warnings.warn("runtime", RuntimeWarning)

assert len(record) == 2
assert str(record[0].message) == "user"
assert str(record[1].message) == "runtime"
```

The `recwarn` fixture will record warnings for the whole function:

```
import warnings

def test_hello(recwarn):
    warnings.warn("hello", UserWarning)
    assert len(recwarn) == 1
    w = recwarn.pop(UserWarning)
    assert issubclass(w.category, UserWarning)
    assert str(w.message) == "hello"
    assert w.filename
    assert w.lineno
```

Both `recwarn` and `pytest.warns` return the same interface for recorded warnings: a WarningsRecorder instance. To view the recorded warnings, you can iterate over this instance, call `len` on it to get the number of recorded warnings, or index into it to get a particular recorded warning. It also provides these methods:

class `WarningsRecorder`

A context manager to record raised warnings.

Adapted from *warnings.catch_warnings*.

`list`

The list of recorded warnings.

`pop`(*cls=<class 'Warning'>*)

Pop the first recorded warning, raise exception if not exists.

`clear`()

Clear the list of recorded warnings.

Each recorded warning has the attributes `message`, `category`, `filename`, `lineno`, `file`, and `line`. The `category` is the class of the warning. The `message` is the warning itself; calling `str(message)` will return the actual message of the warning.

Note: `DeprecationWarning` and `PendingDeprecationWarning` are treated differently; see *Ensuring a function triggers a deprecation warning*.

6.3 Ensuring a function triggers a deprecation warning

You can also call a global helper for checking that a certain function call triggers a `DeprecationWarning` or `PendingDeprecationWarning`:

```
import pytest
```

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

By default, `DeprecationWarning` and `PendingDeprecationWarning` will not be caught when using `pytest.warns` or `recwarn` because default Python warnings filters hide them. If you wish to record them in your own code, use the command `warnings.simplefilter('always')`:

```
import warnings
import pytest

def test_deprecation(recwarn):
    warnings.simplefilter('always')
    warnings.warn("deprecated", DeprecationWarning)
    assert len(recwarn) == 1
    assert recwarn.pop(DeprecationWarning)
```

You can also use it as a contextmanager:

```
def test_global():
    with pytest.deprecated_call():
        myobject.deprecated_method()
```

CHAPTER

SEVEN

CACHE: WORKING WITH CROSS-TESTRUN STATE

New in version 2.8.

Warning: The functionality of this core plugin was previously distributed as a third party plugin named `pytest-cache`. The core plugin is compatible regarding command line options and API usage except that you can only store/receive data between test runs that is json-serializable.

7.1 Usage

The plugin provides two command line options to rerun failures from the last `py.test` invocation:

- `--lf`, `--last-failed` - to only re-run the failures.
- `--ff`, `--failed-first` - to run the failures first and then the rest of the tests.

For cleanup (usually not needed), a `--cache-clear` option allows to remove all cross-session cache contents ahead of a test run.

Other plugins may access the *config.cache* object to set/get **json encodable** values between `py.test` invocations.

Note: This plugin is enabled by default, but can be disabled if needed: see *Deactivating / unregistering a plugin by name* (the internal name for this plugin is `cacheprovider`).

7.2 Rerunning only failures or failures first

First, let's create 50 test invocation of which only 2 fail:

```
# content of test_50.py
import pytest

@pytest.mark.parametrize("i", range(50))
def test_num(i):
    if i in (17, 25):
       pytest.fail("bad luck")
```

If you run this for the first time you will see two failures:

```
$ py.test -q
.................F.......F........................
======= FAILURES ========
```

```
_______ test_num[17] _________

i = 17

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>           pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
_______ test_num[25] _________

i = 25

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>           pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
2 failed, 48 passed in 0.12 seconds
```

If you then run it with `--lf`:

```
$ py.test --lf
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
run-last-failure: rerun last 2 failures
rootdir: $REGENDOC_TMPDIR, inifile:
collected 50 items

test_50.py FF

======= FAILURES ========
_______ test_num[17] _________

i = 17

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>           pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
_______ test_num[25] _________

i = 25

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>           pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
```

```
======= 2 failed, 48 deselected in 0.12 seconds ========
```

You have run only the two failing test from the last run, while 48 tests have not been run ("deselected").

Now, if you run with the `--ff` option, all tests will be run but the first previous failures will be executed first (as can be seen from the series of `FF` and dots):

```
$ py.test --ff
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
run-last-failure: rerun last 2 failures first
rootdir: $REGENDOC_TMPDIR, inifile:
collected 50 items

test_50.py FF................................................

======= FAILURES ========
________ test_num[17] _________

i = 17

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>           pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
________ test_num[25] _________

i = 25

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>           pytest.fail("bad luck")
E           Failed: bad luck

test_50.py:6: Failed
======= 2 failed, 48 passed in 0.12 seconds ========
```

7.3 The new config.cache object

Plugins or conftest.py support code can get a cached value using the pytest `config` object. Here is a basic example plugin which implements a fixture which re-uses previously created state across py.test invocations:

```
# content of test_caching.py
import pytest
import time

@pytest.fixture
def mydata(request):
    val = request.config.cache.get("example/value", None)
    if val is None:
        time.sleep(9*0.6) # expensive computation :)
        val = 42
```

```
        request.config.cache.set("example/value", val)
    return val

def test_function(mydata):
    assert mydata == 23
```

If you run this command once, it will take a while because of the sleep:

```
$ py.test -q
F
======= FAILURES ========
_______ test_function _________

mydata = 42

    def test_function(mydata):
>       assert mydata == 23
E       assert 42 == 23

test_caching.py:14: AssertionError
1 failed in 0.12 seconds
```

If you run it a second time the value will be retrieved from the cache and this will be quick:

```
$ py.test -q
F
======= FAILURES ========
_______ test_function _________

mydata = 42

    def test_function(mydata):
>       assert mydata == 23
E       assert 42 == 23

test_caching.py:14: AssertionError
1 failed in 0.12 seconds
```

See the *cache-api* for more details.

7.4 Inspecting Cache content

You can always peek at the content of the cache using the `--cache-clear` command line option:

```
$ py.test --cache-clear
======= test session starts ========
platform linux -- Python 3.4.0, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 items

test_caching.py F

======= FAILURES ========
_______ test_function _________

mydata = 42
```

```
    def test_function(mydata):
>       assert mydata == 23
E       assert 42 == 23

test_caching.py:14: AssertionError
======= 1 failed in 0.12 seconds ========
```

7.5 Clearing Cache content

You can instruct pytest to clear all cache files and values by adding the `--cache-clear` option like this:

```
py.test --cache-clear
```

This is recommended for invocations from Continous Integration servers where isolation and correctness is more important than speed.

7.6 config.cache API

The `config.cache` object allows other plugins, including `conftest.py` files, to safely and flexibly store and retrieve values across test runs because the `config` object is available in many places.

Under the hood, the cache plugin uses the simple dumps/loads API of the json stdlib module

`Cache.`**`get`**(*key*, *default*)
return cached value for the given key. If no value was yet cached or the value cannot be read, the specified default is returned.

Parameters

- **`key`** – must be a `/` separated value. Usually the first name is the name of your plugin or your application.
- **`default`** – must be provided in case of a cache-miss or invalid cache values.

`Cache.`**`set`**(*key*, *value*)
save value for the given key.

Parameters

- **`key`** – must be a `/` separated value. Usually the first name is the name of your plugin or your application.
- **`value`** – must be of any combination of basic python types, including nested types like e. g. lists of dictionaries.

`Cache.`**`makedir`**(*name*)
return a directory path object with the given name. If the directory does not yet exist, it will be created. You can use it to manage files likes e. g. store/retrieve database dumps across test sessions.

Parameters **`name`** – must be a string not containing a `/` separator. Make sure the name contains your plugin or application identifiers to prevent clashes with other cache users.

CHAPTER

EIGHT

INSTALLING AND USING PLUGINS

This section talks about installing and using third party plugins. For writing your own plugins, please refer to writing-plugins.

Installing a third party plugin can be easily done with `pip`:

```
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.

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-catchlog: 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-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-pep8: a `--pep8` option to enable PEP8 compliance checking.
- pytest-flakes: check source code with pyflakes.
- oejskit: a plugin to run javascript unittests in live browsers.

To see a complete list of all plugins with their latest testing status against different py.test and Python versions, please visit plugincompat.

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

8.1 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 = "myapp.testsupport.myplugin",
```

When the test module or conftest plugin is loaded the specified plugins will be loaded as well.

pytest_plugins = "myapp.testsupport.myplugin"

which will import the specified module as a `pytest` plugin.

8.2 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.

8.3 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 not work.

If you want to unconditionally disable a plugin for a project, you can add this option to your `pytest.ini` file:

```
[pytest]
addopts = -p no:NAME
```

Alternatively to disable it only in certain environments (for example in a CI server), you can set `PYTEST_ADDOPTS` environment variable to `-p no:name`.

See *Finding out which plugins are active* for how to obtain the name of a plugin.

8.4 Pytest default plugin reference

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

`_pytest.assertion`	support for presenting detailed information in failing assertions.
`_pytest.cacheprovider`	merged implementation of the cache provider
`_pytest.capture`	per-test stdout/stderr capturing mechanism.
`_pytest.config`	command line options, ini-file and conftest.py processing.
`_pytest.doctest`	discover and run doctests in modules and test files.
`_pytest.genscript`	(deprecated) generate a single-file self-contained version of pytest
`_pytest.helpconfig`	version info, help messages, tracing configuration.
`_pytest.junitxml`	report test results in JUnit-XML format,
`_pytest.mark`	generic mechanism for marking and selecting python functions.
`_pytest.monkeypatch`	monkeypatching and mocking functionality.
`_pytest.nose`	run test suites written for nose.
`_pytest.pastebin`	submit failure or test session information to a pastebin service.
`_pytest.pdb`	interactive debugging with PDB, the Python Debugger.
`_pytest.pytester`	(disabled by default) support for testing pytest and pytest plugins.
`_pytest.python`	Python test discovery, setup and run of test functions.
`_pytest.recwarn`	recording warnings during test function execution.

Continued on next page

Table 8.1 – continued from previous page

_pytest.resultlog	log machine-parseable test session result information in a plain
_pytest.runner	basic collect and runtest protocol implementations
_pytest.main	core implementation of testing process: init, session, runtest loop.
_pytest.skipping	support for skip/xfail functions and markers.
_pytest.terminal	terminal reporting of the full testing process.
_pytest.tmpdir	support for providing temporary directories to test functions.
_pytest.unittest	discovery and running of std-library "unittest" style tests.

CHAPTER

NINE

CONTRIBUTION GETTING STARTED

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

Contribution links

9.1 Feature requests and feedback

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:

- Explain in detail how they should work.
- Keep the scope as narrow as possible. This will make it easier to implement.

9.2 Report bugs

Report bugs for pytest in the issue tracker.

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.

If you can write a demonstration test that currently fails but should pass (xfail), that is a very useful commit to make as well, even if you can't find how to fix the bug yet.

9.3 Fix bugs

Look through the GitHub issues for bugs. Here is sample filter you can use: https://github.com/pytest-dev/pytest/labels/bug

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

Don't forget to check the issue trackers of your favourite plugins, too!

9.4 Implement features

Look through the GitHub issues for enhancements. Here is sample filter you can use: https://github.com/pytest-dev/pytest/labels/enhancement

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

9.5 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 only English.
- Docstrings. There can never be too many of them.
- Blog posts, articles and such – they're all very appreciated.

You can also edit documentation files directly in the Github web interface without needing to make a fork and local copy. This can be convenient for small fixes.

9.6 Submitting Plugins to pytest-dev

Pytest development of the core, some plugins and support code happens in repositories living under the `pytest-dev` organisations:

- pytest-dev on GitHub
- pytest-dev on Bitbucket

All pytest-dev Contributors team members have write access to all contained repositories. pytest core and plugins are generally developed using *pull requests* to respective repositories.

The objectives of the `pytest-dev` organisation are:

- Having a central location for popular pytest plugins
- Sharing some of the maintenance responsibility (in case a maintainer no longer whishes to maintain a plugin)

You can submit your plugin by subscribing to the pytest-dev mail list and writing a mail pointing to your existing pytest plugin repository which must have the following:

- PyPI presence with a `setup.py` that contains a license, `pytest-` prefixed name, version number, authors, short and long description.
- a `tox.ini` for running tests using tox.

- a `README.txt` describing how to use the plugin and on which platforms it runs.
- a `LICENSE.txt` file or equivalent containing the licensing information, with matching info in `setup.py`.
- an issue tracker for bug reports and enhancement requests.

If no contributor strongly objects and two agree, the repository can then be transferred to the `pytest-dev` organisation.

Here's a rundown of how a repository transfer usually proceeds (using a repository named `joedoe/pytest-xyz` as example):

- One of the `pytest-dev` administrators creates:
  - `pytest-xyz-admin` team, with full administration rights to `pytest-dev/pytest-xyz`.
  - `pytest-xyz-developers` team, with write access to `pytest-dev/pytest-xyz`.
- `joedoe` is invited to the `pytest-xyz-admin` team;
- After accepting the invitation, `joedoe` transfers the repository from its original location to `pytest-dev/pytest-xyz` (A nice feature is that GitHub handles URL redirection from the old to the new location automatically).
- `joedoe` is free to add any other collaborators to the `pytest-xyz-admin` or `pytest-xyz-developers` team as desired.

The `pytest-dev/Contributors` team has write access to all projects, and every project administrator is in it. We recommend that each plugin has at least three people who have the right to release to PyPI.

Repository owners can be assured that no `pytest-dev` administrator will ever make releases of your repository or take ownership in any way, except in rare cases where someone becomes unresponsive after months of contact attempts. As stated, the objective is to share maintenance and avoid "plugin-abandon".

9.7 Preparing Pull Requests on GitHub

There's an excellent tutorial on how Pull Requests work in the GitHub Help Center

---

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 GitHub servers. Once you send pull request, we can discuss it's potential modifications and even add more commits to it later on.

There's an excellent tutorial on how Pull Requests work in the GitHub Help Center, but here is a simple overview:

1. Fork the pytest GitHub repository. It's fine to use `pytest` as your fork repository name because it will live under your user.
2. Clone your fork locally using git and create a branch:

```
$ git clone git@github.com:YOUR_GITHUB_USERNAME/pytest.git
$ cd pytest
# now, to fix a bug create your own branch off "master":

    $ git checkout -b your-bugfix-branch-name master

# or to instead add a feature create your own branch off "features":

    $ git checkout -b your-feature-branch-name features
```

Given we have "major.minor.micro" version numbers, bugfixes will usually be released in micro releases whereas features will be released in minor releases and incompatible changes in major releases.

If you need some help with Git, follow this quick start guide: https://git.wiki.kernel.org/index.php/QuickStart

3. Install tox

   Tox is used to run all the tests and will automatically setup virtualenvs to run the tests in. (will implicitly use http://www.virtualenv.org/en/latest/):

```
$ pip install tox
```

4. Run all the tests

   You need to have Python 2.7 and 3.5 available in your system. Now running tests is as simple as issuing this command:

```
$ python runtox.py -e linting,py27,py35
```

   This command will run tests via the "tox" tool against Python 2.7 and 3.5 and also perform "lint" 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.

5. You can now edit your local working copy.

   You can now make the changes you want and run the tests again as necessary.

   To run tests on py27 and pass options to pytest (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 py35:

```
$ python runtox.py -e py35 -- testing/test_config.py
```

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

```
$ git commit -a -m "<commit message>"
$ git push -u
```

   Make sure you add a CHANGELOG message, and add yourself to AUTHORS. If you are unsure about either of these steps, submit your pull request and we'll help you fix it up.

7. Finally, submit a pull request through the GitHub website using this data:

```
head-fork: YOUR_GITHUB_USERNAME/pytest
compare: your-branch-name

base-fork: pytest-dev/pytest
base: master          # if it's a bugfix
base: feature         # if it's a feature
```

CHAPTER

TEN

TALKS AND TUTORIALS

Next Open Trainings

professional testing with pytest and tox, 27-29th June 2016, Freiburg, Germany

10.1 Talks and blog postings

- pytest - Rapid Simple Testing, Florian Bruhin, Swiss Python Summit 2016.
- Improve your testing with Pytest and Mock, Gabe Hollombe, PyCon SG 2015.
- 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.
- fixtures
- 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

10.2 Older conference talks and tutorials

- pycon australia 2012 pytest talk from Brianna Laugher (video, slides, code)
- pycon 2012 US talk video from Holger Krekel
- pycon 2010 tutorial PDF and tutorial1 repository
- 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.

C

D

G

L

M

P

S

U

W