pytest Documentation

Release 3.2

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

Sep 19, 2017

Contents

Download latest version as PDF

CHAPTER 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), ordereddict (py26).

documentation as PDF: download latest

Installation

Installation:

```
pip install -U pytest
```

To check your installation has installed the correct version:

```
$ pytest --version
This is pytest version 3.x.y, imported from $PYTHON_PREFIX/lib/python3.5/site-
↪packages/pytest.py
```

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:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
```

```
collected 1 item

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

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*.

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:

```
$ pytest -q test_sysexit.py
.
1 passed in 0.12 seconds
```

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(object):
    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:

```
$ pytest -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       AssertionError: assert False
E        +  where False = 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.

Going functional: requesting a unique temporary directory

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

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

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

```
$ pytest -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 *pytest fixtures: explicit, modular, scalable* exist by typing:

```
pytest --fixtures   # shows builtin and custom fixtures
```

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
- *Using pytest with an existing test suite* for working with pre-existing tests
- *pytest fixtures: explicit, modular, scalable* for providing a functional baseline to your tests
- *Writing plugins* managing and writing plugins

CHAPTER 2

Usage and Invocations

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 almost equivalent to invoking the command line script `pytest [...]` directly, except that Python will also add the current directory to `sys.path`.

Possible exit codes

Running `pytest` can result in six different exit codes:

Exit code 0 All tests were collected and passed successfully

Exit code 1 Tests were collected and run but some of the tests failed

Exit code 2 Test execution was interrupted by the user

Exit code 3 Internal error happened while executing tests

Exit code 4 pytest command line usage error

Exit code 5 No tests were collected

Getting help on version, option names, environment variables

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

Stopping after the first (or N) failures

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

```
pytest -x            # stop after first failure
pytest --maxfail=2    # stop after two failures
```

Specifying tests / selecting tests

Pytest supports several ways to run and select tests from the command-line.

Run tests in a module

```
pytest test_mod.py
```

Run tests in a directory

```
pytest testing/
```

Run tests by keyword expressions

```
pytest -k "MyClass and not method"
```

This will run tests which contain names that match the given *string expression*, which can include Python operators that use filenames, class names and function names as variables. The example above will run `TestMyClass.test_something` but not `TestMyClass.test_method_simple`. **Run tests by node ids**

Each collected test is assigned a unique `nodeid` which consist of the module filename followed by specifiers like class names, function names and parameters from parametrization, separated by `::` characters.

To run a specific test within a module:

```
pytest test_mod.py::test_func
```

Another example specifying a test method in the command line:

```
pytest test_mod.py::TestClass::test_method
```

Run tests by marker expressions

```
pytest -m slow
```

Will run all tests which are decorated with the `@pytest.mark.slow` decorator.

For more information see *marks*.

Run tests from packages

```
pytest --pyargs pkg.testing
```

This will import `pkg.testing` and use its filesystem location to find and run tests from.

Modifying Python traceback printing

Examples for modifying traceback printing:

```
pytest --showlocals # show local variables in tracebacks
pytest -l           # show local variables (shortcut)

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

The `--full-trace` causes very long traces to be printed on error (longer than `--tb=long`). It also ensures that a stack trace is printed on **KeyboardInterrupt** (Ctrl+C). This is very useful if the tests are taking too long and you interrupt them with Ctrl+C to find out where the tests are *hanging*. By default no output will be shown (because KeyboardInterrupt is caught by pytest). By using this option you make sure a trace is shown.

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:

```
pytest --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:

```
pytest -x --pdb   # drop to PDB on first failure, then end test session
pytest --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"',)
```

Setting breakpoints

To set a breakpoint in your code use the native Python `import pdb;pdb.set_trace()` call in your code and pytest automatically disables its output capture for that test:

- 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.

Profiling test execution duration

To get a list of the slowest 10 test durations:

```
pytest --durations=10
```

Creating JUnitXML format files

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

```
pytest --junitxml=path
```

to create an XML file at `path`.

New in version 3.1.

To set the name of the root test suite xml item, you can configure the `junit_suite_name` option in your config file:

```
[pytest]
junit_suite_name = my_suite
```

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="0.0009">
  <properties>
    <property name="example_key" value="1" />
  </properties>
</testcase>
```

Warning: `record_xml_property` 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.

LogXML: add_global_property

New in version 3.0.

If you want to add a properties node in the testsuite level, which may contains properties that are relevant to all testcases you can use `LogXML.add_global_properties`

```
import pytest

@pytest.fixture(scope="session")
def log_global_env_facts(f):

    if pytest.config.pluginmanager.hasplugin('junitxml'):
        my_junit = getattr(pytest.config, '_xml', None)

    my_junit.add_global_property('ARCH', 'PPC')
    my_junit.add_global_property('STORAGE_TYPE', 'CEPH')

@pytest.mark.usefixtures(log_global_env_facts)
def start_and_prepare_env():
    pass

class TestMe(object):
    def test_foo(self):
        assert True
```

This will add a property node below the testsuite node to the generated xml:

```
<testsuite errors="0" failures="0" name="pytest" skips="0" tests="1" time="0.006">
  <properties>
    <property name="ARCH" value="PPC"/>
    <property name="STORAGE_TYPE" value="CEPH"/>
  </properties>
  <testcase classname="test_me.TestMe" file="test_me.py" line="16" name="test_foo"␣
↪time="0.000243663787842"/>
</testsuite>
```

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.

Creating resultlog format files

Deprecated since version 3.0: This option is rarely used and is scheduled for removal in 4.0.

An alternative for users which still need similar functionality is to use the pytest-tap plugin which provides a stream of test data.

If you have any concerns, please don't hesitate to open an issue.

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

```
pytest --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.

Sending test report to online pastebin service

Creating a URL for each test failure:

```
pytest --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:

```
pytest --pastebin=all
```

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

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 pytest like this:

```
pytest -p no:doctest
```

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 "pytest" 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'])
```

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

```
# content of myinvoke.py
import pytest
class MyPlugin(object):
    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
```

CHAPTER 3

Using pytest with an existing test suite

Pytest can be used with most existing test suites, but its behavior differs from other test runners such as *nose* or Python's default unittest framework.

Before using this section you will want to *install pytest*.

Running an existing test suite with pytest

Say you want to contribute to an existing repository somewhere. After pulling the code into your development space using some flavor of version control and (optionally) setting up a virtualenv you will want to run:

```
cd <repository>
pip install -e .  # Environment dependent alternatives include
                  # 'python setup.py develop' and 'conda develop'
```

in your project root. This will set up a symlink to your code in site-packages, allowing you to edit your code while your tests run against it as if it were installed.

Setting up your project in development mode lets you avoid having to reinstall every time you want to run your tests, and is less brittle than mucking about with sys.path to point your tests at local code.

Also consider using *tox*.

CHAPTER 4

The writing and reporting of assertions in tests

Asserting with the `assert` statement

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

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

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

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

```
$ pytest test_assert1.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_assert1.py F

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

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

test_assert1.py:5: AssertionError
======= 1 failed in 0.12 seconds ========
```

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

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

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

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

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

Assertions about expected exceptions

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

```
import pytest

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

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

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

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

Changed in version 3.0.

In the context manager form you may use the keyword argument `message` to specify a custom failure message:

```
>>> with raises(ZeroDivisionError, message="Expecting ZeroDivisionError"):
...     pass
... Failed: Expecting ZeroDivisionError
```

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

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

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

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

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

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

Also, the context manager form accepts a `match` keyword parameter to test that a regular expression matches on the string representation of an exception (like the `TestCase.assertRaisesRegexp` method from `unittest`):

```
import pytest

def myfunc():
    raise ValueError("Exception 123 raised")

def test_match():
    with pytest.raises(ValueError, match=r'.* 123 .*'):
        myfunc()
```

The regexp parameter of the `match` method is matched with the `re.search` function. So in the above example `match='123'` would have worked as well.

Assertions about expected warnings

New in version 2.8.

You can check that code raises a particular warning using *pytest.warns*.

Making use of context-sensitive comparisons

New in version 2.0.

`pytest` has rich support for providing context-sensitive information when it encounters comparisons. For example:

```
# content of test_assert2.py

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

if you run this module:

```
$ pytest test_assert2.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_assert2.py F

======= FAILURES ========
_______ test_set_comparison ________

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

```
E         '5'
E         Use -v to get the full diff

test_assert2.py:5: AssertionError
======= 1 failed in 0.12 seconds ========
```

Special comparisons are done for a number of cases:

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

See the *reporting demo* for many more examples.

Defining your own assertion comparison

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

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

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

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

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

now, given this test module:

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

    def __eq__(self, other):
        return self.val == other.val

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

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

```
$ pytest -q test_foocompare.py
F
======= FAILURES ========
_______ test_compare _________
```

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

test_foocompare.py:11: AssertionError
1 failed in 0.12 seconds
```

Advanced assertion introspection

New in version 2.1.

Reporting details about a failing assertion is achieved by rewriting assert statements before they are run. Rewritten assert statements put introspection information into the assertion failure message. `pytest` only rewrites test modules directly discovered by its test collection process, so asserts in supporting modules which are not themselves test modules will not be rewritten.

Note: `pytest` rewrites test modules on import by using an import hook to write new `pyc` files. Most of the time this works transparently. However, if you are messing with import yourself, the import hook may interfere.

If this is the case you have two options:

- Disable rewriting for a specific module by adding the string `PYTEST_DONT_REWRITE` to its docstring.
- Disable rewriting for all modules by using `--assert=plain`.

Additionally, rewriting will fail silently if it cannot write new `.pyc` files, i.e. in a read-only filesystem or a zipfile.

For further information, Benjamin Peterson wrote up Behind the scenes of pytest's new assertion rewriting.

New in version 2.1: Add assert rewriting as an alternate introspection technique.

Changed in version 2.1: Introduce the `--assert` option. Deprecate `--no-assert` and `--nomagic`.

Changed in version 3.0: Removes the `--no-assert` and `--nomagic` options. Removes the `--assert=reinterp` option.

CHAPTER 5

Pytest API and builtin fixtures

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

For information on plugin hooks and objects, see *Writing plugins*.

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

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

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

Invoking pytest interactively

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

Parameters

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

More examples at *Calling pytest from Python code*

Helpers for assertions about Exceptions/Warnings

raises(*expected_exception*, **args*, ***kwargs*)
Assert that a code block/function call raises `expected_exception` and raise a failure exception otherwise.

This helper produces a `ExceptionInfo()` object (see below).

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

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

Changed in version 2.10.

In the context manager form you may use the keyword argument `message` to specify a custom failure message:

```
>>> with raises(ZeroDivisionError, message="Expecting ZeroDivisionError"):
...     pass
Traceback (most recent call last):
  ...
Failed: Expecting ZeroDivisionError
```

Note: When using `pytest.raises` as a context manager, it's worthwhile to note that normal context manager rules apply and that the exception raised *must* be the final line in the scope of the context manager. Lines of code after that, within the scope of the context manager will not be executed. For example:

```
>>> value = 15
>>> with raises(ValueError) as exc_info:
...     if value > 10:
...         raise ValueError("value must be <= 10")
...     assert exc_info.type == ValueError  # this will not execute
```

Instead, the following approach must be taken (note the difference in scope):

```
>>> with raises(ValueError) as exc_info:
...     if value > 10:
...         raise ValueError("value must be <= 10")
...
>>> assert exc_info.type == ValueError
```

Since version `3.1` you can use the keyword argument `match` to assert that the exception matches a text or regex:

```
>>> with raises(ValueError, match='must be 0 or None'):
...     raise ValueError("value must be 0 or None")

>>> with raises(ValueError, match=r'must be \d+$'):
...     raise ValueError("value must be 42")
```

Legacy forms

The forms below are fully supported but are discouraged for new code because the context manager form is regarded as more readable and less error-prone.

It is possible to specify a callable by passing a to-be-called lambda:

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

or you can specify an arbitrary callable with arguments:

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

It is also possible to pass a string to be evaluated at runtime:

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

The string will be evaluated using the same `locals()` and `globals()` at the moment of the `raises` call.

class ExceptionInfo (*tup=None*, *exprinfo=None*)
wraps sys.exc_info() objects and offers help for navigating the traceback.

type = None
the exception class

value = None
the exception instance

tb = None
the exception raw traceback

typename = None
the exception type name

traceback = None
the exception traceback (_pytest._code.Traceback instance)

exconly (*tryshort=False*)
return the exception as a string

when 'tryshort' resolves to True, and the exception is a _pytest._code._AssertionError, only the actual exception part of the exception representation is returned (so 'AssertionError: ' is removed from the beginning)

errisinstance (*exc*)
return True if the exception is an instance of exc

getrepr (*showlocals=False*, *style='long'*, *abspath=False*, *tbfilter=True*, *funcargs=False*)
return str()able representation of this exception info. showlocals: show locals per traceback entry style: long|short|no|native traceback style tbfilter: hide entries (where __tracebackhide__ is true)

in case of style==native, tbfilter and showlocals is ignored.

match (*regexp*)
Match the regular expression 'regexp' on the string representation of the exception. If it matches then True is returned (so that it is possible to write 'assert excinfo.match()'). If it doesn't match an AssertionError is raised.

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

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

Examples at *Assertions about expected exceptions*.

deprecated_call (*func=None*, **args*, ***kwargs*)
context manager that can be used to ensure a block of code triggers a `DeprecationWarning` or `PendingDeprecationWarning`:

```
>>> import warnings
>>> def api_call_v2():
...     warnings.warn('use v3 of this api', DeprecationWarning)
...     return 200
```

```
>>> with deprecated_call():
...     assert api_call_v2() == 200
```

`deprecated_call` can also be used by passing a function and `*args` and `*kwargs`, in which case it will ensure calling `func(*args, **kwargs)` produces one of the warnings types above.

Comparing floating point numbers

approx (*expected*, *rel=None*, *abs=None*, *nan_ok=False*)

Assert that two numbers (or two sets of numbers) are equal to each other within some tolerance.

Due to the intricacies of floating-point arithmetic, numbers that we would intuitively expect to be equal are not always so:

```
>>> 0.1 + 0.2 == 0.3
False
```

This problem is commonly encountered when writing tests, e.g. when making sure that floating-point values are what you expect them to be. One way to deal with this problem is to assert that two floating-point numbers are equal to within some appropriate tolerance:

```
>>> abs((0.1 + 0.2) - 0.3) < 1e-6
True
```

However, comparisons like this are tedious to write and difficult to understand. Furthermore, absolute comparisons like the one above are usually discouraged because there's no tolerance that works well for all situations. `1e-6` is good for numbers around `1`, but too small for very big numbers and too big for very small ones. It's better to express the tolerance as a fraction of the expected value, but relative comparisons like that are even more difficult to write correctly and concisely.

The `approx` class performs floating-point comparisons using a syntax that's as intuitive as possible:

```
>>> from pytest import approx
>>> 0.1 + 0.2 == approx(0.3)
True
```

The same syntax also works for sequences of numbers:

```
>>> (0.1 + 0.2, 0.2 + 0.4) == approx((0.3, 0.6))
True
```

Dictionary *values*:

```
>>> {'a': 0.1 + 0.2, 'b': 0.2 + 0.4} == approx({'a': 0.3, 'b': 0.6})
True
```

And `numpy` arrays:

```
>>> import numpy as np
>>> np.array([0.1, 0.2]) + np.array([0.2, 0.4]) == approx(np.array([0.3, 0.6]))
True
```

By default, `approx` considers numbers within a relative tolerance of `1e-6` (i.e. one part in a million) of its expected value to be equal. This treatment would lead to surprising results if the expected value was `0.0`,

because nothing but `0.0` itself is relatively close to `0.0`. To handle this case less surprisingly, `approx` also considers numbers within an absolute tolerance of `1e-12` of its expected value to be equal. Infinity and NaN are special cases. Infinity is only considered equal to itself, regardless of the relative tolerance. NaN is not considered equal to anything by default, but you can make it be equal to itself by setting the `nan_ok` argument to True. (This is meant to facilitate comparing arrays that use NaN to mean "no data".)

Both the relative and absolute tolerances can be changed by passing arguments to the `approx` constructor:

```
>>> 1.0001 == approx(1)
False
>>> 1.0001 == approx(1, rel=1e-3)
True
>>> 1.0001 == approx(1, abs=1e-3)
True
```

If you specify `abs` but not `rel`, the comparison will not consider the relative tolerance at all. In other words, two numbers that are within the default relative tolerance of `1e-6` will still be considered unequal if they exceed the specified absolute tolerance. If you specify both `abs` and `rel`, the numbers will be considered equal if either tolerance is met:

```
>>> 1 + 1e-8 == approx(1)
True
>>> 1 + 1e-8 == approx(1, abs=1e-12)
False
>>> 1 + 1e-8 == approx(1, rel=1e-6, abs=1e-12)
True
```

If you're thinking about using `approx`, then you might want to know how it compares to other good ways of comparing floating-point numbers. All of these algorithms are based on relative and absolute tolerances and should agree for the most part, but they do have meaningful differences:

- `math.isclose(a,b,rel_tol=1e-9,abs_tol=0.0)`: True if the relative tolerance is met w.r.t. either `a` or `b` or if the absolute tolerance is met. Because the relative tolerance is calculated w.r.t. both `a` and `b`, this test is symmetric (i.e. neither `a` nor `b` is a "reference value"). You have to specify an absolute tolerance if you want to compare to `0.0` because there is no tolerance by default. Only available in python>=3.5. More information...
- `numpy.isclose(a,b,rtol=1e-5,atol=1e-8)`: True if the difference between `a` and `b` is less that the sum of the relative tolerance w.r.t. `b` and the absolute tolerance. Because the relative tolerance is only calculated w.r.t. `b`, this test is asymmetric and you can think of `b` as the reference value. Support for comparing sequences is provided by `numpy.allclose`. More information...
- `unittest.TestCase.assertAlmostEqual(a,b)`: True if `a` and `b` are within an absolute tolerance of `1e-7`. No relative tolerance is considered and the absolute tolerance cannot be changed, so this function is not appropriate for very large or very small numbers. Also, it's only available in subclasses of `unittest.TestCase` and it's ugly because it doesn't follow PEP8. More information...
- `a == pytest.approx(b,rel=1e-6,abs=1e-12)`: True if the relative tolerance is met w.r.t. `b` or if the absolute tolerance is met. Because the relative tolerance is only calculated w.r.t. `b`, this test is asymmetric and you can think of `b` as the reference value. In the special case that you explicitly specify an absolute tolerance but not a relative tolerance, only the absolute tolerance is considered.

> **Warning:** Changed in version 3.2.
>
> In order to avoid inconsistent behavior, `TypeError` is raised for `>`, `>=`, `<` and `<=` comparisons. The example below illustrates the problem:

```
assert approx(0.1) > 0.1 + 1e-10  # calls approx(0.1).__gt__(0.1 + 1e-10)
assert 0.1 + 1e-10 > approx(0.1)  # calls approx(0.1).__lt__(0.1 + 1e-10)
```

In the second example one expects `approx(0.1).__le__(0.1 + 1e-10)` to be called. But instead, `approx(0.1).__lt__(0.1 + 1e-10)` is used to comparison. This is because the call hierarchy of rich comparisons follows a fixed behavior. More information...

Raising a specific test outcome

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

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

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

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

importorskip(*modname*, *minversion=None*)
return imported module if it has at least "minversion" as its __version__ attribute. If no minversion is specified the a skip is only triggered if the module can not be imported.

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

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

Fixtures and requests

To mark a fixture function:

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

This decorator can be used (with or without parameters) to define a fixture function. The name of the fixture function can later be referenced to cause its invocation ahead of running tests: test modules or classes can use the pytest.mark.usefixtures(fixturename) marker. Test functions can directly use fixture names as input arguments in which case the fixture instance returned from the fixture function will be injected.

Parameters

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

- **ids** – list of string ids each corresponding to the params so that they are part of the test id. If no ids are provided they will be generated automatically from the params.
- **name** – the name of the fixture. This defaults to the name of the decorated function. If a fixture is used in the same module in which it is defined, the function name of the fixture will be shadowed by the function arg that requests the fixture; one way to resolve this is to name the decorated function `fixture_<fixturename>` and then use `@pytest.fixture(name='<fixturename>')`.

Fixtures can optionally provide their values to test functions using a `yield` statement, instead of `return`. In this case, the code block after the `yield` statement is executed as teardown code regardless of the test outcome. A fixture function must yield exactly once.

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

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

class FixtureRequest

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

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

fixturename = None
fixture for which this request is being performed

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

node
underlying collection node (depends on current request scope)

config
the pytest config object associated with this request.

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

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

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

module
python module object where the test function was collected.

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

keywords
keywords/markers dictionary for the underlying node.

session
pytest session object.

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

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

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

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

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

Parameters

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

getfixturevalue(*argname*)
Dynamically run a named fixture function.

Declaring fixtures via function argument is recommended where possible. But if you can only decide whether to use another fixture at test setup time, you may use this function to retrieve it inside a fixture or test function body.

getfuncargvalue(*argname*)
Deprecated, use getfixturevalue.

Builtin fixtures/function arguments

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

```
$ pytest -q --fixtures
cache
    Return a cache object that can persist state between testing sessions.

    cache.get(key, default)
    cache.set(key, value)

    Keys must be a ``/`` separated value, where the first part is usually the
    name of your plugin or application to avoid clashes with other cache users.

    Values can be any object handled by the json stdlib module.
capsys
    Enable capturing of writes to sys.stdout/sys.stderr and make
    captured output available via ``capsys.readouterr()`` method calls
    which return a ``(out, err)`` tuple.
capfd
    Enable capturing of writes to file descriptors 1 and 2 and make
    captured output available via ``capfd.readouterr()`` method calls
    which return a ``(out, err)`` tuple.
doctest_namespace
    Inject names into the doctest namespace.
pytestconfig
```

```
    the pytest config object with access to command line opts.
record_xml_property
    Add extra xml properties to the tag for the calling test.
    The fixture is callable with ``(name, value)``, with value being automatically
    xml-encoded.
monkeypatch
    The returned ``monkeypatch`` fixture provides these
    helper methods to modify objects, dictionaries or os.environ::

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

    All modifications will be undone after the requesting
    test function or fixture has finished. The ``raising``
    parameter determines if a KeyError or AttributeError
    will be raised if the set/deletion operation has no target.
recwarn
    Return a WarningsRecorder instance that provides these methods:

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

    See http://docs.python.org/library/warnings.html for information
    on warning categories.
tmpdir_factory
    Return a TempdirFactory instance for the test session.
tmpdir
    Return a temporary directory path object
    which is unique to each test function invocation,
    created as a sub directory of the base temporary
    directory.  The returned object is a `py.path.local`_
    path object.

no tests ran in 0.12 seconds
```

CHAPTER 6

pytest fixtures: explicit, modular, scalable

New in version 2.0/2.3/2.4.

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

- fixtures have explicit names and are activated by declaring their use from test functions, modules, classes or whole projects.
- fixtures are implemented in a modular manner, as each fixture name triggers a *fixture function* which can itself use other fixtures.
- fixture management scales from simple unit to complex functional testing, allowing to parametrize fixtures and tests according to configuration and component options, or to re-use fixtures across class, module or whole test session scopes.

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

Fixtures as Function arguments

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

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

@pytest.fixture
def smtp():
    import smtplib
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

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

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

```
$ pytest test_smtpsimple.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_smtpsimple.py F

======= FAILURES ========
_______ test_ehlo _________

smtp = <smtplib.SMTP object at 0xdeadbeef>

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

test_smtpsimple.py:11: AssertionError
======= 1 failed in 0.12 seconds ========
```

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

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

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

Note: You can always issue:

```
pytest --fixtures test_simplefactory.py
```

to see available fixtures.

Fixtures: a prime example of dependency injection

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

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

Fixtures requiring network access depend on connectivity and are usually time-expensive to create. Extending the previous example, we can add a `scope='module'` parameter to the `@pytest.fixture` invocation to cause the decorated `smtp` fixture function to only be invoked once per test module. Multiple test functions in a test module

will thus each receive the same `smtp` fixture instance. The next example puts the fixture function into a separate `conftest.py` file so that tests from multiple test modules in the directory can access the fixture function:

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

@pytest.fixture(scope="module")
def smtp():
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
```

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

```
# content of test_module.py

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

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

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

```
$ pytest test_module.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py FF

======= FAILURES ========
_______ test_ehlo _________

smtp = <smtplib.SMTP object at 0xdeadbeef>

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

test_module.py:6: AssertionError
_______ test_noop _________

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_noop(smtp):
        response, msg = smtp.noop()
        assert response == 250
>       assert 0  # for demo purposes
```

```
E       assert 0

test_module.py:11: AssertionError
======= 2 failed in 0.12 seconds ========
```

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

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

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

Fixture finalization / executing teardown code

pytest supports execution of fixture specific finalization code when the fixture goes out of scope. By using a `yield` statement instead of `return`, all the code after the *yield* statement serves as the teardown code:

```
# content of conftest.py

import smtplib
import pytest

@pytest.fixture(scope="module")
def smtp():
    smtp = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    yield smtp  # provide the fixture value
    print("teardown smtp")
    smtp.close()
```

The `print` and `smtp.close()` statements will execute when the last test in the module has finished execution, regardless of the exception status of the tests.

Let's execute it:

```
$ pytest -s -q --tb=no
FFteardown smtp

2 failed in 0.12 seconds
```

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

Note that we can also seamlessly use the `yield` syntax with `with` statements:

```
# content of test_yield2.py

import smtplib
import pytest

@pytest.fixture(scope="module")
def smtp():
```

```
    with smtplib.SMTP("smtp.gmail.com", 587, timeout=5) as smtp:
        yield smtp  # provide the fixture value
```

The `smtp` connection will be closed after the test finished execution because the `smtp` object automatically closes when the `with` statement ends.

Note that if an exception happens during the *setup* code (before the `yield` keyword), the *teardown* code (after the `yield`) will not be called.

An alternative option for executing *teardown* code is to make use of the `addfinalizer` method of the *request-context* object to register finalization functions.

Here's the `smtp` fixture changed to use `addfinalizer` for cleanup:

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

@pytest.fixture(scope="module")
def smtp(request):
    smtp = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    def fin():
        print ("teardown smtp")
        smtp.close()
    request.addfinalizer(fin)
    return smtp  # provide the fixture value
```

Both `yield` and `addfinalizer` methods work similarly by calling their code after the test ends, but `addfinalizer` has two key differences over `yield`:

1. It is possible to register multiple finalizer functions.
2. Finalizers will always be called regardless if the fixture *setup* code raises an exception. This is handy to properly close all resources created by a fixture even if one of them fails to be created/acquired:

   ```
   @pytest.fixture
   def equipments(request):
       r = []
       for port in ('C1', 'C3', 'C28'):
           equip = connect(port)
           request.addfinalizer(equip.disconnect)
           r.append(equip)
       return r
   ```

   In the example above, if `"C28"` fails with an exception, `"C1"` and `"C3"` will still be properly closed. Of course, if an exception happens before the finalize function is registered then it will not be executed.

Fixtures can introspect the requesting test context

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

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

```
@pytest.fixture(scope="module")
def smtp(request):
    server = getattr(request.module, "smtpserver", "smtp.gmail.com")
    smtp = smtplib.SMTP(server, 587, timeout=5)
    yield smtp
    print ("finalizing %s (%s)" % (smtp, server))
    smtp.close()
```

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

```
$ pytest -s -q --tb=no
FFfinalizing <smtplib.SMTP object at 0xdeadbeef> (smtp.gmail.com)

2 failed in 0.12 seconds
```

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

```
# content of test_anothersmtp.py

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

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

Running it:

```
$ pytest -qq --tb=short test_anothersmtp.py
F
======= FAILURES ========
_______ test_showhelo _________
test_anothersmtp.py:5: in test_showhelo
    assert 0, smtp.helo()
E   AssertionError: (250, b'mail.python.org')
E   assert 0
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef> (mail.python.org)
```

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

Parametrizing fixtures

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

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

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

@pytest.fixture(scope="module",
                params=["smtp.gmail.com", "mail.python.org"])
```

```
def smtp(request):
    smtp = smtplib.SMTP(request.param, 587, timeout=5)
    yield smtp
    print ("finalizing %s" % smtp)
    smtp.close()
```

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

```
$ pytest -q test_module.py
FFFF
======= FAILURES ========
________ test_ehlo[smtp.gmail.com] _________

smtp = <smtplib.SMTP object at 0xdeadbeef>

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

test_module.py:6: AssertionError
________ test_noop[smtp.gmail.com] _________

smtp = <smtplib.SMTP object at 0xdeadbeef>

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

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

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp):
        response, msg = smtp.ehlo()
        assert response == 250
>       assert b"smtp.gmail.com" in msg
E       AssertionError: assert b'smtp.gmail.com' in b'mail.python.
 ↪org\nPIPELINING\nSIZE 51200000\nETRN\nSTARTTLS\nAUTH DIGEST-MD5 NTLM CRAM-
 ↪MD5\nENHANCEDSTATUSCODES\n8BITMIME\nDSN\nSMTPUTF8'

test_module.py:5: AssertionError
-------------------------- Captured stdout setup ---------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
________ test_noop[mail.python.org] _________

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_noop(smtp):
        response, msg = smtp.noop()
        assert response == 250
```

```
>       assert 0  # for demo purposes
E       assert 0

test_module.py:11: AssertionError
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
4 failed in 0.12 seconds
```

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

pytest will build a string that is the test ID for each fixture value in a parametrized fixture, e.g. `test_ehlo[smtp.gmail.com]` and `test_ehlo[mail.python.org]` in the above examples. 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. It is possible to customise the string used in a test ID for a certain fixture value by using the `ids` keyword argument:

```
# content of test_ids.py
import pytest

@pytest.fixture(params=[0, 1], ids=["spam", "ham"])
def a(request):
    return request.param

def test_a(a):
    pass

def idfn(fixture_value):
    if fixture_value == 0:
        return "eggs"
    else:
        return None

@pytest.fixture(params=[0, 1], ids=idfn)
def b(request):
    return request.param

def test_b(b):
    pass
```

The above shows how `ids` can be either a list of strings to use or a function which will be called with the fixture value and then has to return a string to use. In the latter case if the function return `None` then pytest's auto-generated ID will be used.

Running the above tests results in the following test IDs being used:

```
$ pytest --collect-only
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 10 items
<Module 'test_anothersmtp.py'>
  <Function 'test_showhelo[smtp.gmail.com]'>
  <Function 'test_showhelo[mail.python.org]'>
```

```
<Module 'test_ids.py'>
  <Function 'test_a[spam]'>
  <Function 'test_a[ham]'>
  <Function 'test_b[eggs]'>
  <Function 'test_b[1]'>
<Module 'test_module.py'>
  <Function 'test_ehlo[smtp.gmail.com]'>
  <Function 'test_noop[smtp.gmail.com]'>
  <Function 'test_ehlo[mail.python.org]'>
  <Function 'test_noop[mail.python.org]'>

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

Modularity: using fixtures from a fixture function

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

```
# content of test_appsetup.py

import pytest

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

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

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

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

```
$ pytest -v test_appsetup.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 2 items

test_appsetup.py::test_smtp_exists[smtp.gmail.com] PASSED
test_appsetup.py::test_smtp_exists[mail.python.org] PASSED

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

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

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

Automatic grouping of tests by fixture instances

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

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

```
# content of test_module.py
import pytest

@pytest.fixture(scope="module", params=["mod1", "mod2"])
def modarg(request):
    param = request.param
    print ("  SETUP modarg %s" % param)
    yield param
    print ("  TEARDOWN modarg %s" % param)

@pytest.fixture(scope="function", params=[1,2])
def otherarg(request):
    param = request.param
    print ("  SETUP otherarg %s" % param)
    yield param
    print ("  TEARDOWN otherarg %s" % param)

def test_0(otherarg):
    print ("  RUN test0 with otherarg %s" % otherarg)
def test_1(modarg):
    print ("  RUN test1 with modarg %s" % modarg)
def test_2(otherarg, modarg):
    print ("  RUN test2 with otherarg %s and modarg %s" % (otherarg, modarg))
```

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

```
$ pytest -v -s test_module.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 8 items

test_module.py::test_0[1]   SETUP otherarg 1
  RUN test0 with otherarg 1
PASSED  TEARDOWN otherarg 1

test_module.py::test_0[2]   SETUP otherarg 2
  RUN test0 with otherarg 2
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod1]   SETUP modarg mod1
```

```
  RUN test1 with modarg mod1
PASSED
test_module.py::test_2[1-mod1]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod1
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[2-mod1]   SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod1
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod2]   TEARDOWN modarg mod1
  SETUP modarg mod2
  RUN test1 with modarg mod2
PASSED
test_module.py::test_2[1-mod2]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod2
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[2-mod2]   SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod2
PASSED  TEARDOWN otherarg 2
  TEARDOWN modarg mod2


======= 8 passed in 0.12 seconds ========
```

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

In particular notice that test_0 is completely independent and finishes first. Then test_1 is executed with `mod1`, then test_2 with `mod1`, then test_1 with `mod2` and finally test_2 with `mod2`.

The `otherarg` parametrized resource (having function scope) was set up before and teared down after every test that used it.

Using fixtures from classes, modules or projects

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

```
# content of conftest.py

import pytest
import tempfile
import os

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

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

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

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

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

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

```
$ pytest -q
..
2 passed in 0.12 seconds
```

You can specify multiple fixtures like this:

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

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

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

Note that the assigned variable *must* be called `pytestmark`, assigning e.g. `foomark` will not activate the fixtures.

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

```
# content of pytest.ini
[pytest]
usefixtures = cleandir
```

Autouse fixtures (xUnit setup on steroids)

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

```
# content of test_db_transact.py

import pytest

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

```
        self.intransaction.pop()

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

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

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

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

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

If we run it, we get two passing tests:

```
$ pytest -q
..
2 passed in 0.12 seconds
```

Here is how autouse fixtures work in other scopes:

- autouse fixtures obey the `scope=` keyword-argument: if an autouse fixture has `scope='session'` it will only be run once, no matter where it is defined. `scope='class'` means it will be run once per class, etc.
- if an autouse fixture is defined in a test module, all its test functions automatically use it.
- if an autouse fixture is defined in a conftest.py file then all tests in all test modules below its directory will invoke the fixture.
- lastly, and **please use that with care**: if you define an autouse fixture in a plugin, it will be invoked for all tests in all projects where the plugin is installed. This can be useful if a fixture only anyway works in the presence of certain settings e. g. in the ini-file. Such a global fixture should always quickly determine if it should do any work and avoid otherwise expensive imports or computation.

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

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

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

```
@pytest.mark.usefixtures("transact")
class TestClass(object):
```

```
    def test_method1(self):
        ...
```

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

Shifting (visibility of) fixture functions

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

Overriding fixtures on various levels

In relatively large test suite, you most likely need to `override` a `global` or `root` fixture with a `locally` defined one, keeping the test code readable and maintainable.

Override a fixture on a folder (conftest) level

Given the tests file structure is:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'

    test_something.py
        # content of tests/test_something.py
        def test_username(username):
            assert username == 'username'

    subfolder/
        __init__.py

        conftest.py
            # content of tests/subfolder/conftest.py
            import pytest

            @pytest.fixture
            def username(username):
                return 'overridden-' + username

        test_something.py
            # content of tests/subfolder/test_something.py
            def test_username(username):
                assert username == 'overridden-username'
```

As you can see, a fixture with the same name can be overridden for certain test folder level. Note that the `base` or `super` fixture can be accessed from the `overriding` fixture easily - used in the example above.

Override a fixture on a test module level

Given the tests file structure is:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        @pytest.fixture
        def username():
            return 'username'

    test_something.py
        # content of tests/test_something.py
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-' + username

        def test_username(username):
            assert username == 'overridden-username'

    test_something_else.py
        # content of tests/test_something_else.py
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-else-' + username

        def test_username(username):
            assert username == 'overridden-else-username'
```

In the example above, a fixture with the same name can be overridden for certain test module.

Override a fixture with direct test parametrization

Given the tests file structure is:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'

        @pytest.fixture
```

```
        def other_username(username):
            return 'other-' + username

    test_something.py
        # content of tests/test_something.py
        import pytest

        @pytest.mark.parametrize('username', ['directly-overridden-username'])
        def test_username(username):
            assert username == 'directly-overridden-username'

        @pytest.mark.parametrize('username', ['directly-overridden-username-other'])
        def test_username_other(other_username):
            assert other_username == 'other-directly-overridden-username-other'
```

In the example above, a fixture value is overridden by the test parameter value. Note that the value of the fixture can be overridden this way even if the test doesn't use it directly (doesn't mention it in the function prototype).

Override a parametrized fixture with non-parametrized one and vice versa

Given the tests file structure is:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture(params=['one', 'two', 'three'])
        def parametrized_username(request):
            return request.param

        @pytest.fixture
        def non_parametrized_username(request):
            return 'username'

    test_something.py
        # content of tests/test_something.py
        import pytest

        @pytest.fixture
        def parametrized_username():
            return 'overridden-username'

        @pytest.fixture(params=['one', 'two', 'three'])
        def non_parametrized_username(request):
            return request.param

        def test_username(parametrized_username):
            assert parametrized_username == 'overridden-username'

        def test_parametrized_username(non_parametrized_username):
            assert non_parametrized_username in ['one', 'two', 'three']

    test_something_else.py
        # content of tests/test_something_else.py
```

```
        def test_username(parametrized_username):
            assert parametrized_username in ['one', 'two', 'three']

        def test_username(non_parametrized_username):
            assert non_parametrized_username == 'username'
```

In the example above, a parametrized fixture is overridden with a non-parametrized version, and a non-parametrized fixture is overridden with a parametrized version for certain test module. The same applies for the test folder level obviously.

CHAPTER 7

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` fixture 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.

Simple example: monkeypatching functions

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

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

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

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

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.

Note: Be advised that it is not recommended to patch builtin functions such as `open`, `compile`, etc., because it might break pytest's internals. If that's unavoidable, passing `--tb=native`, `--assert=plain` and `--capture=no` might help although there's no guarantee.

Method reference of the monkeypatch fixture

class `MonkeyPatch`

Object returned by the `monkeypatch` fixture 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 8

Temporary directories and files

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:

```
$ pytest test_tmpdir.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

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 ========
```

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.

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:

```
pytest --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 9

Capturing of the stdout/stderr output

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.

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:

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

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:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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 ========
```

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).

New in version 3.0.

To temporarily disable capture within a test, both `capsys` and `capfd` have a `disabled()` method that can be used as a context manager, disabling capture inside the `with` block:

```
def test_disabling_capturing(capsys):
    print('this output is captured')
```

```
    with capsys.disabled():
        print('output not captured, going directly to sys.stdout')
    print('this output is also captured')
```

CHAPTER 10

Warnings Capture

New in version 3.1.

Starting from version `3.1`, pytest now automatically catches warnings during test execution and displays them at the end of the session:

```
# content of test_show_warnings.py
import warnings

def api_v1():
    warnings.warn(UserWarning("api v1, should use functions from v2"))
    return 1

def test_one():
    assert api_v1() == 1
```

Running pytest now produces this output:

```
$ pytest test_show_warnings.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_show_warnings.py .

======= warnings summary ========
test_show_warnings.py::test_one
  $REGENDOC_TMPDIR/test_show_warnings.py:4: UserWarning: api v1, should use functions␣
 ↪from v2
    warnings.warn(UserWarning("api v1, should use functions from v2"))

-- Docs: http://doc.pytest.org/en/latest/warnings.html
======= 1 passed, 1 warnings in 0.12 seconds ========
```

Pytest by default catches all warnings except for `DeprecationWarning` and `PendingDeprecationWarning`.

The `-W` flag can be passed to control which warnings will be displayed or even turn them into errors:

```
$ pytest -q test_show_warnings.py -W error::UserWarning
F
======= FAILURES ========
_______ test_one _________
```

```
    def test_one():
>       assert api_v1() == 1

test_show_warnings.py:8:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    def api_v1():
>       warnings.warn(UserWarning("api v1, should use functions from v2"))
E       UserWarning: api v1, should use functions from v2

test_show_warnings.py:4: UserWarning
1 failed in 0.12 seconds
```

The same option can be set in the `pytest.ini` file using the `filterwarnings` ini option. For example, the configuration below will ignore all user warnings, but will transform all other warnings into errors.

```
[pytest]
filterwarnings =
    error
    ignore::UserWarning
```

When a warning matches more than one option in the list, the action for the last matching option is performed.

Both `-W` command-line option and `filterwarnings` ini option are based on Python's own -W option and warnings.simplefilter, so please refer to those sections in the Python documentation for other examples and advanced usage.

@pytest.mark.filterwarnings

New in version 3.2.

You can use the `@pytest.mark.filterwarnings` to add warning filters to specific test items, allowing you to have finer control of which warnings should be captured at test, class or even module level:

```
import warnings

def api_v1():
    warnings.warn(UserWarning("api v1, should use functions from v2"))
    return 1

@pytest.mark.filterwarnings('ignore:api v1')
def test_one():
    assert api_v1() == 1
```

Filters applied using a mark take precedence over filters passed on the command line or configured by the `filterwarnings` ini option.

You may apply a filter to all tests of a class by using the `filterwarnings` mark as a class decorator or to all tests in a module by setting the `pytestmark` variable:

```
# turns all warnings into errors for this module
pytestmark = @pytest.mark.filterwarnings('error')
```

Note: `DeprecationWarning` and `PendingDeprecationWarning` are hidden by the standard library by default so you have to explicitly configure them to be displayed in your `pytest.ini`:

```
[pytest]
filterwarnings =
    once::DeprecationWarning
    once::PendingDeprecationWarning
```

Credits go to Florian Schulze for the reference implementation in the pytest-warnings *plugin.*

Disabling warning capture

This feature is enabled by default but can be disabled entirely in your `pytest.ini` file with:

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

Or passing `-p no:warnings` in the command-line.

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*.

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=<type 'exceptions.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: `RecordedWarning` was changed from a plain class to a namedtuple in pytest 3.1

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

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 11

Doctest integration for modules and test files

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

```
pytest --doctest-glob='*.rst'
```

on the command line. Since version `2.9`, `--doctest-glob` can be given multiple times in the command-line.

New in version 3.1: You can specify the encoding that will be used for those doctest files using the `doctest_encoding` ini option:

```
# content of pytest.ini
[pytest]
doctest_encoding = latin1
```

The default encoding is UTF-8.

You can also trigger running of doctests from docstrings in all python modules (including regular python test modules):

```
pytest --doctest-modules
```

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

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

If you then have a text file like this:

```
# content of example.rst

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

and another like this:

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

```
    """
    return 42
```

then you can just invoke `pytest` without command line options:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 1 item

mymodule.py .

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

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

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

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

The standard `doctest` module provides some setting flags to configure the strictness of doctest tests. In pytest You can enable those flags those flags using the configuration file. To make pytest ignore trailing whitespaces and ignore lengthy exception stack traces you can just write:

```
[pytest]
doctest_optionflags= NORMALIZE_WHITESPACE IGNORE_EXCEPTION_DETAIL
```

pytest also introduces new options to allow doctests to run in Python 2 and Python 3 unchanged:

- `ALLOW_UNICODE`: when enabled, the `u` prefix is stripped from unicode strings in expected doctest output.
- `ALLOW_BYTES`: when enabled, the `b` prefix is stripped from byte strings in expected doctest output.

As with any other option flag, these flags can be enabled in `pytest.ini` using the `doctest_optionflags` ini option:

```
[pytest]
doctest_optionflags = ALLOW_UNICODE ALLOW_BYTES
```

Alternatively, it can be enabled by an inline comment in the doc test itself:

```
# content of example.rst
>>> get_unicode_greeting()  # doctest: +ALLOW_UNICODE
'Hello'
```

The 'doctest_namespace' fixture

New in version 3.0.

The `doctest_namespace` fixture can be used to inject items into the namespace in which your doctests run. It is intended to be used within your own fixtures to provide the tests that use them with context.

`doctest_namespace` is a standard `dict` object into which you place the objects you want to appear in the doctest namespace:

```
# content of conftest.py
import numpy
@pytest.fixture(autouse=True)
def add_np(doctest_namespace):
    doctest_namespace['np'] = numpy
```

which can then be used in your doctests directly:

```
# content of numpy.py
def arange():
    """
    >>> a = np.arange(10)
    >>> len(a)
    10
    """
    pass
```

Output format

New in version 3.0.

You can change the diff output format on failure for your doctests by using one of standard doctest modules format in options (see `doctest.REPORT_UDIFF`, `doctest.REPORT_CDIFF`, `doctest.REPORT_NDIFF`, `doctest.REPORT_ONLY_FIRST_FAILURE`):

```
pytest --doctest-modules --doctest-report none
pytest --doctest-modules --doctest-report udiff
pytest --doctest-modules --doctest-report cdiff
pytest --doctest-modules --doctest-report ndiff
pytest --doctest-modules --doctest-report only_first_failure
```

CHAPTER 12

Marking test functions with attributes

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

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

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

Note: Marks can only be applied to tests, having no effect on *fixtures*.

API reference for mark related objects

class MarkGenerator

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

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

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

class MarkDecorator(*mark*)

A decorator for test functions and test classes. When applied it will create *MarkInfo* objects which may be *retrieved by hooks as item keywords*. MarkDecorator instances are often created like this:

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

and can then be applied as decorators to test functions:

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

When a MarkDecorator instance is called it does the following:

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

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

name
alias for mark.name

args
alias for mark.args

kwargs
alias for mark.kwargs

with_args(**args*, ***kwargs*)
return a MarkDecorator with extra arguments added

unlike call this can be used even if the sole argument is a callable/class

Returns MarkDecorator

class MarkInfo(*mark*)
Marking object created by *MarkDecorator* instances.

name
alias for combined.name

args
alias for combined.args

kwargs
alias for combined.kwargs

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

Skip and xfail: dealing with tests that cannot succeed

You can mark test functions that cannot be run on certain platforms or that you expect to fail so pytest can deal with them accordingly and present a summary of the test session, while keeping the test suite *green*.

A **skip** means that you expect your test to pass only if some conditions are met, otherwise pytest should skip running the test altogether. Common examples are skipping windows-only tests on non-windows platforms, or skipping tests that depend on an external resource which is not available at the moment (for example a database).

A **xfail** means that you expect a test to fail for some reason. A common example is a test for a feature not yet implemented, or a bug not yet fixed.

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

```
pytest -rxs  # show extra info on skips and xfails
```

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

Skipping test functions

New in version 2.9.

The simplest way to skip a test function is to mark it with the `skip` decorator which may be passed an optional `reason`:

```
@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
    ...
```

Alternatively, it is also possible to skip imperatively during test execution or setup by calling the `pytest.skip(reason)` function:

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

The imperative method is useful when it is not possible to evaluate the skip condition during import time.

skipif

New in version 2.0.

If you wish to skip something conditionally then you can use `skipif` instead. Here is an example of marking a test function to be skipped when run on a Python3.3 interpreter:

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

If the condition evaluates to `True` during collection, the test function will be skipped, with the specified reason appearing in the summary when using `-rs`.

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

```
# content of test_mymodule.py
import mymodule
minversion = pytest.mark.skipif(mymodule.__versioninfo__ < (1,1),
                                reason="at least mymodule-1.1 required")
@minversion
def test_function():
    ...
```

You can import the marker and reuse it in another test module:

```
# test_myothermodule.py
from test_mymodule import minversion

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

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

Alternatively, you can use *condition strings* instead of booleans, but they can't be shared between modules easily so they are supported mainly for backward compatibility reasons.

Skip all test functions of a class or module

You can use the `skipif` marker (as any other marker) on classes:

```
@pytest.mark.skipif(sys.platform == 'win32',
                    reason="does not run on windows")
class TestPosixCalls(object):

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

If the condition is `True`, this marker will produce a skip result for each of the test methods of that class.

Warning: The use of `skipif` on classes that use inheritance is strongly discouraged. A Known bug in pytest's markers may cause unexpected behavior in super classes.

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

```
# test_module.py
pytestmark = pytest.mark.skipif(...)
```

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

Skipping on a missing import dependency

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

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

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

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

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

Summary

Here's a quick guide on how to skip tests in a module in different situations:

1. Skip all tests in a module unconditionally:

   ```
   pytestmark = pytest.mark.skip('all tests still WIP')
   ```

2. Skip all tests in a module based on some condition:

   ```
   pytestmark = pytest.mark.skipif(sys.platform == 'win32', 'tests for linux␣
    ↪only')
   ```

3. Skip all tests in a module if some import is missing:

   ```
   pexpect = pytest.importorskip('pexpect')
   ```

XFail: mark test functions as expected to fail

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

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

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

Alternatively, you can also mark a test as `XFAIL` from within a test or setup function imperatively:

```
def test_function():
    if not valid_config():
        pytest.xfail("failing configuration (but should work)")
```

This will unconditionally make `test_function` `XFAIL`. Note that no other code is executed after `pytest.xfail` call, differently from the marker. That's because it is implemented internally by raising a known exception.

Here's the signature of the `xfail` **marker** (not the function), using Python 3 keyword-only arguments syntax:

```
def xfail(condition=None, *, reason=None, raises=None, run=True, strict=False):
```

`strict` parameter

New in version 2.9.

Both `XFAIL` and `XPASS` don't fail the test suite, unless the `strict` keyword-only parameter is passed as `True`:

```
@pytest.mark.xfail(strict=True)
def test_function():
    ...
```

This will make `XPASS` ("unexpectedly passing") results from this test to fail the test suite.

You can change the default value of the `strict` parameter using the `xfail_strict` ini option:

```
[pytest]
xfail_strict=true
```

`reason` parameter

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

```
@pytest.mark.xfail(sys.version_info >= (3,3),
                   reason="python3.3 api changes")
def test_function():
    ...
```

`raises` parameter

If you want to be more specific as to why the test is failing, you can specify a single exception, or a list of exceptions, in the `raises` argument.

```
@pytest.mark.xfail(raises=RuntimeError)
def test_function():
    ...
```

Then the test will be reported as a regular failure if it fails with an exception not mentioned in `raises`.

`run` parameter

If a test should be marked as xfail and reported as such but should not be even executed, use the `run` parameter as `False`:

```
@pytest.mark.xfail(run=False)
def test_function():
    ...
```

This is specially useful for xfailing tests that are crashing the interpreter and should be investigated later.

Ignoring xfail

By specifying on the commandline:

```
pytest --runxfail
```

you can force the running and reporting of an `xfail` marked test as if it weren't marked at all. This also causes `pytest.xfail` to produce no effect.

Examples

Here is a simple test file with the several usages:

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

@xfail
def test_hello():
    assert 0

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

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

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

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

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

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

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

```
example $ pytest -rx xfail_demo.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR/example, inifile:
collected 7 items

xfail_demo.py xxxxxxx
======= short test summary info ========
```

```
XFAIL xfail_demo.py::test_hello
XFAIL xfail_demo.py::test_hello2
  reason: [NOTRUN]
XFAIL xfail_demo.py::test_hello3
  condition: hasattr(os, 'sep')
XFAIL xfail_demo.py::test_hello4
  bug 110
XFAIL xfail_demo.py::test_hello5
  condition: pytest.__version__[0] != "17"
XFAIL xfail_demo.py::test_hello6
  reason: reason
XFAIL xfail_demo.py::test_hello7

======= 7 xfailed in 0.12 seconds ========
```

Skip/xfail with parametrize

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

```
import pytest

@pytest.mark.parametrize(("n", "expected"), [
    (1, 2),
pytest.param(1, 0, marks=pytest.mark.xfail),
    pytest.param(1, 3, marks=pytest.mark.xfail(reason="some bug")),
    (2, 3),
    (3, 4),
    (4, 5),
pytest.param(10, 11, marks=pytest.mark.skipif(sys.version_info >= (3, 0), reason="py2k
↪")),
])
def test_increment(n, expected):
    assert n + 1 == expected
```

CHAPTER 14

Parametrizing fixtures and test functions

pytest enables test parametrization at several levels:

- `pytest.fixture()` allows one to *parametrize fixture functions*.
- *@pytest.mark.parametrize* allows one to define multiple sets of arguments and fixtures at the test function or class.
- *pytest_generate_tests* allows one to define custom parametrization schemes or extensions.

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

New in version 2.2.

Changed in version 2.4: Several improvements.

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

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

Here, the `@parametrize` decorator defines three different `(test_input,expected)` tuples so that the `test_eval` function will run three times using them in turn:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 3 items

test_expectation.py ..F

======= FAILURES ========
_______ test_eval[6*9-42] _________
```

```
test_input = '6*9', expected = 42

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

test_expectation.py:8: AssertionError
======= 1 failed, 2 passed in 0.12 seconds ========
```

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

Note that you could also use the parametrize marker on a class or a module (see *Marking test functions with attributes*) which would invoke several functions with the argument sets.

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

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

Let's run this:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 3 items

test_expectation.py ..x

======= 2 passed, 1 xfailed in 0.12 seconds ========
```

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

To get all combinations of multiple parametrized arguments you can stack `parametrize` decorators:

```
import pytest
@pytest.mark.parametrize("x", [0, 1])
@pytest.mark.parametrize("y", [2, 3])
def test_foo(x, y):
    pass
```

This will run the test with the arguments set to `x=0/y=2`, `x=0/y=3`, `x=1/y=2` and `x=1/y=3`.

Basic `pytest_generate_tests` example

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

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

```
# content of test_strings.py

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

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

```
# content of conftest.py

def pytest_addoption(parser):
    parser.addoption("--stringinput", action="append", default=[],
        help="list of stringinputs to pass to test functions")

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

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

```
$ pytest -q --stringinput="hello" --stringinput="world" test_strings.py
..
2 passed in 0.12 seconds
```

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

```
$ pytest -q --stringinput="!" test_strings.py
F
======= FAILURES ========
_______ test_valid_string[!] _________

stringinput = '!'

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

test_strings.py:3: AssertionError
1 failed in 0.12 seconds
```

As expected our test function fails.

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

```
$ pytest -q -rs test_strings.py
s
======= short test summary info ========
SKIP [1] test_strings.py:2: got empty parameter set ['stringinput'], function test_
↪valid_string at $REGENDOC_TMPDIR/test_strings.py:1
1 skipped in 0.12 seconds
```

Note that when calling `metafunc.parametrize` multiple times with different parameter sets, all parameter names across those sets cannot be duplicated, otherwise an error will be raised.

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

The metafunc object

class `Metafunc` (*function*, *fixtureinfo*, *config*, *cls=None*, *module=None*)

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

`config` = None

access to the *`_pytest.config.Config`* object for the test session

`module` = None

the module object where the test function is defined in.

`function` = None

underlying python test function

`fixturenames` = None

set of fixture names required by the test function

`cls` = None

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

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

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

Parameters

- **`argnames`** – a comma-separated string denoting one or more argument names, or a list/tuple of argument strings.
- **`argvalues`** – The list of argvalues determines how often a test is invoked with different argument values. If only one argname was specified argvalues is a list of values. If N argnames were specified, argvalues must be a list of N-tuples, where each tuple-element specifies a value for its respective argname.
- **`indirect`** – The list of argnames or boolean. A list of arguments' names (subset of argnames). If True the list contains all names from the argnames. Each argvalue corresponding to an argname in this list will be passed as request.param to its respective argname fixture function so that it can perform more expensive setups during the setup phase of a test rather than at collection time.
- **`ids`** – list of string ids, or a callable. If strings, each is corresponding to the argvalues so that they are part of the test id. If None is given as id of specific test, the automatically generated id for that argument will be used. If callable, it should take one argument (a

single argvalue) and return a string or return None. If None, the automatically generated id for that argument will be used. If no ids are provided they will be generated automatically from the argvalues.

- **scope** – if specified it denotes the scope of the parameters. The scope is used for grouping tests by parameter instances. It will also override any fixture-function defined scope, allowing to set a dynamic scope using test context or configuration.

addcall (*funcargs=None*, *id=<object object>*, *param=<object object>*)

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

Parameters

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

CHAPTER 15

Cache: working with cross-testrun state

New in version 2.8.

Usage

The plugin provides two command line options to rerun failures from the last `pytest` 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 `pytest` 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`).

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:

```
$ pytest -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`:

```
$ pytest --lf
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 50 items
run-last-failure: rerun previous 2 failures

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
======= 48 tests deselected ========
======= 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):

```
$ pytest --ff
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 50 items
run-last-failure: rerun previous 2 failures first

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 ========
```

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 *pytest fixtures: explicit, modular, scalable* which re-uses previously created state across pytest 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:

```
$ pytest -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:

```
$ pytest -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.

Inspecting Cache content

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

```
$ py.test --cache-show
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
cachedir: $REGENDOC_TMPDIR/.cache
------------------------------- cache values -------------------------------
cache/lastfailed contains:
  {'test_caching.py::test_function': True}
example/value contains:
  42

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

Clearing Cache content

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

```
pytest --cache-clear
```

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

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 16

unittest.TestCase Support

`pytest` supports running Python `unittest`-based tests out of the box. It's meant for leveraging existing `unittest`-based test suites to use pytest as a test runner and also allow to incrementally adapt the test suite to take full advantage of pytest's features.

To run an existing `unittest`-style test suite using `pytest`, type:

```
pytest tests
```

pytest will automatically collect `unittest.TestCase` subclasses and their `test` methods in `test_*.py` or `*_test.py` files.

Almost all `unittest` features are supported:

- `@unittest.skip` style decorators;
- `setUp/tearDown`;
- `setUpClass/tearDownClass()`;

Up to this point pytest does not have support for the following features:

- load_tests protocol;
- setUpModule/tearDownModule;
- subtests;

Benefits out of the box

By running your test suite with pytest you can make use of several features, in most cases without having to modify existing code:

- Obtain *more informative tracebacks*;
- *stdout and stderr* capturing;
- *Test selection options* using `-k` and `-m` flags;
- *Stopping after the first (or N) failures*;
- *–pdb* command-line option for debugging on test failures (see *note* below);
- Distribute tests to multiple CPUs using the pytest-xdist plugin;
- Use *plain assert-statements* instead of `self.assert*` functions (unittest2pytest is immensely helpful in this);

pytest features in `unittest.TestCase` subclasses

The following pytest features work in `unittest.TestCase` subclasses:

- *Marks*: *skip*, *skipif*, *xfail*;
- *Auto-use fixtures*;

The following pytest features **do not** work, and probably never will due to different design philosophies:

- *Fixtures* (except for `autouse` fixtures, see *below*);
- *Parametrization*;
- *Custom hooks*;

Third party plugins may or may not work well, depending on the plugin and the test suite.

Mixing pytest fixtures into `unittest.TestCase` subclasses using marks

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

```
# content of conftest.py

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

import pytest

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

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

```
# content of test_unittest_db.py

import unittest
import pytest

@pytest.mark.usefixtures("db_class")
class MyTest(unittest.TestCase):
    def test_method1(self):
        assert hasattr(self, "db")
        assert 0, self.db   # fail for demo purposes
```

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

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

```
$ pytest test_unittest_db.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_unittest_db.py FF

======= FAILURES ========
_______ MyTest.test_method1 _________

self = <test_unittest_db.MyTest testMethod=test_method1>

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

test_unittest_db.py:9: AssertionError
_______ MyTest.test_method2 _________

self = <test_unittest_db.MyTest testMethod=test_method2>

    def test_method2(self):
>       assert 0, self.db   # fail for demo purposes
E       AssertionError: <conftest.db_class.<locals>.DummyDB object at 0xdeadbeef>
E       assert 0

test_unittest_db.py:12: AssertionError
======= 2 failed in 0.12 seconds ========
```

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

Using autouse fixtures and accessing other fixtures

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

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

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

```
import unittest

class MyTest(unittest.TestCase):

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

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

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

Running this test module ...:

```
$ pytest -q test_unittest_cleandir.py
.
1 passed in 0.12 seconds
```

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

Note: `unittest.TestCase` methods cannot directly receive fixture arguments as implementing that is likely to inflict on the ability to run general unittest.TestCase test suites.

The above `usefixtures` and `autouse` examples should help to mix in pytest fixtures into unittest suites.

You can also gradually move away from subclassing from `unittest.TestCase` to *plain asserts* and then start to benefit from the full pytest feature set step by step.

Note: Running tests from `unittest.TestCase` subclasses with `--pdb` will disable tearDown and cleanup methods for the case that an Exception occurs. This allows proper post mortem debugging for all applications which have significant logic in their tearDown machinery. However, supporting this feature has the following side effect: If people overwrite `unittest.TestCase __call__` or `run`, they need to to overwrite `debug` in the same way (this is also true for standard unittest).

CHAPTER 17

Running tests written for nose

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

Usage

After *Installation* type:

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

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

Supported nose Idioms

- setup and teardown at module/class/method level
- SkipTest exceptions and markers
- setup/teardown decorators
- `yield`-based tests and their setup (considered deprecated as of pytest 3.0)
- `__test__` attribute on modules/classes/functions
- general usage of nose utilities

Unsupported idioms / known issues

- unittest-style `setUp,tearDown,setUpClass,tearDownClass` are recognized only on `unittest.TestCase` classes but not on plain classes. `nose` supports these methods also on plain classes but pytest deliberately does not. As nose and pytest already both support `setup_class,teardown_class,setup_method,teardown_method` it doesn't seem useful to duplicate the unittest-API like nose does. If you however rather think pytest should support the unittest-spelling on plain classes please post to this issue.
- nose imports test modules with the same import path (e.g. `tests.test_mod`) but different file system paths (e.g. `tests/test_mode.py` and `other/tests/test_mode.py`) by extending sys.path/import semantics. pytest does not do that but there is discussion in #268 for adding some support. Note that nose2 choose to avoid this sys.path/import hackery.

If you place a conftest.py file in the root directory of your project (as determined by pytest) pytest will run tests "nose style" against the code below that directory by adding it to your `sys.path` instead of running against your installed code.

You may find yourself wanting to do this if you ran `python setup.py install` to set up your project, as opposed to `python setup.py develop` or any of the package manager equivalents. Installing with develop in a virtual environment like Tox is recommended over this pattern.

- nose-style doctests are not collected and executed correctly, also doctest fixtures don't work.
- no nose-configuration is recognized.
- `yield`-based methods don't support `setup` properly because the `setup` method is always called in the same class instance. There are no plans to fix this currently because `yield`-tests are deprecated in pytest 3.0, with `pytest.mark.parametrize` being the recommended alternative.

CHAPTER 18

classic xunit-style setup

This section describes a classic and popular way how you can implement fixtures (setup and teardown test state) on a per-module/class/function basis.

Note: While these setup/teardown methods are simple and familiar to those coming from a `unittest` or nose `background`, you may also consider using pytest's more powerful *fixture mechanism* which leverages the concept of dependency injection, allowing for a more modular and more scalable approach for managing test state, especially for larger projects and for functional testing. You can mix both fixture mechanisms in the same file but test methods of `unittest.TestCase` subclasses cannot receive fixture arguments.

Module level setup/teardown

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

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

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

As of pytest-3.0, the `module` parameter is optional.

Class level setup/teardown

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

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

@classmethod
def teardown_class(cls):
```

```
    """ teardown any state that was previously setup with a call to
    setup_class.
    """
```

Method and function level setup/teardown

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

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

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

As of pytest-3.0, the `method` parameter is optional.

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

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

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

As of pytest-3.0, the `function` parameter is optional.

Remarks:

- It is possible for setup/teardown pairs to be invoked multiple times per testing process.
- teardown functions are not called if the corresponding setup function existed and failed/was skipped.

CHAPTER 19

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.
- 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 pytest and Python versions, please visit plugincompat.

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

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.

Finding out which plugins are active

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

```
pytest --trace-config
```

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.

Deactivating / unregistering a plugin by name

You can prevent plugins from loading or unregister them:

```
pytest -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.

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.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.

Continued on next page

Table 19.1 – continued from previous page

_pytest.pastebin	submit failure or test session information to a pastebin service.
_pytest.debugging	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.
_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 20

Writing plugins

It is easy to implement *local conftest plugins* for your own project or *pip-installable plugins* that can be used throughout many projects, including third party projects. Please refer to *Installing and Using plugins* if you only want to use but not write plugins.

A plugin contains one or multiple hook functions. *Writing hooks* explains the basics and details of how you can write a hook function yourself. `pytest` implements all aspects of configuration, collection, running and reporting by calling *well specified hooks* of the following plugins:

- *Pytest default plugin reference*: loaded from pytest's internal `_pytest` directory.
- *external plugins*: modules discovered through *setuptools entry points*
- *conftest.py plugins*: modules auto-discovered in test directories

In principle, each hook call is a `1:N` Python function call where `N` is the number of registered implementation functions for a given specification. All specifications and implementations follow the `pytest_` prefix naming convention, making them easy to distinguish and find.

Plugin discovery order at tool startup

`pytest` loads plugin modules at tool startup in the following way:

- by loading all builtin plugins
- by loading all plugins registered through *setuptools entry points*.
- by pre-scanning the command line for the `-p name` option and loading the specified plugin before actual command line parsing.
- by loading all `conftest.py` files as inferred by the command line invocation:
  - if no test paths are specified use current dir as a test path
  - if exists, load `conftest.py` and `test*/conftest.py` relative to the directory part of the first test path.

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

conftest.py: local per-directory plugins

Local `conftest.py` plugins contain directory-specific hook implementations. Hook Session and test running activities will invoke all hooks defined in `conftest.py` files closer to the root of the filesystem. Example of implementing the `pytest_runtest_setup` hook so that is called for tests in the `a` sub directory but not for other directories:

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

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

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

Here is how you might run it:

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

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

See also: *pytest import mechanisms and sys.path/PYTHONPATH*.

Writing your own plugin

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

- a custom collection example plugin: *A basic example for specifying tests in Yaml files*
- around 20 *Pytest default plugin reference* which provide pytest's own functionality
- many external plugins providing additional features

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

Note: Make sure to check out the excellent cookiecutter-pytest-plugin project, which is a cookiecutter template for authoring plugins.

The template provides an excellent starting point with a working plugin, tests running with tox, a comprehensive README file as well as a pre-configured entry-point.

Also consider *contributing your plugin to pytest-dev* once it has some happy users other than yourself.

Making your plugin installable by others

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

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

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

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

    # custom PyPI classifier for pytest plugins
    classifiers=[
        "Framework :: Pytest",
    ],
)
```

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

Note: Make sure to include `Framework :: Pytest` in your list of PyPI classifiers to make it easy for users to find your plugin.

Assertion Rewriting

One of the main features of `pytest` is the use of plain assert statements and the detailed introspection of expressions upon assertion failures. This is provided by "assertion rewriting" which modifies the parsed AST before it gets compiled to bytecode. This is done via a **PEP 302** import hook which gets installed early on when `pytest` starts up and will perform this re-writing when modules get imported. However since we do not want to test different bytecode then you will run in production this hook only re-writes test modules themselves as well as any modules which are part of plugins. Any other imported module will not be re-written and normal assertion behaviour will happen.

If you have assertion helpers in other modules where you would need assertion rewriting to be enabled you need to ask `pytest` explicitly to re-write this module before it gets imported.

register_assert_rewrite(**names*)

Register one or more module names to be rewritten on import.

This function will make sure that this module or all modules inside the package will get their assert statements rewritten. Thus you should make sure to call this before the module is actually imported, usually in your __init__.py if you are a plugin using a package.

Raises **TypeError** – if the given module names are not strings.

This is especially important when you write a pytest plugin which is created using a package. The import hook only treats `conftest.py` files and any modules which are listed in the `pytest11` entrypoint as plugins. As an example consider the following package:

```
pytest_foo/__init__.py
pytest_foo/plugin.py
pytest_foo/helper.py
```

With the following typical `setup.py` extract:

```
setup(
    ...
    entry_points={'pytest11': ['foo = pytest_foo.plugin']},
    ...
)
```

In this case only `pytest_foo/plugin.py` will be re-written. If the helper module also contains assert statements which need to be re-written it needs to be marked as such, before it gets imported. This is easiest by marking it for re-writing inside the `__init__.py` module, which will always be imported first when a module inside a package is imported. This way `plugin.py` can still import `helper.py` normally. The contents of `pytest_foo/__init__.py` will then need to look like this:

```
import pytest

pytest.register_assert_rewrite('pytest_foo.helper')
```

Requiring/Loading plugins in a test module or conftest file

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

```
pytest_plugins = ["name1", "name2"]
```

When the test module or conftest plugin is loaded the specified plugins will be loaded as well. Any module can be blessed as a plugin, including internal application modules:

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

`pytest_plugins` variables are processed recursively, so note that in the example above if `myapp.testsupport.myplugin` also declares `pytest_plugins`, the contents of the variable will also be loaded as plugins, and so on.

This mechanism makes it easy to share fixtures within applications or even external applications without the need to create external plugins using the `setuptools`'s entry point technique.

Plugins imported by `pytest_plugins` will also automatically be marked for assertion rewriting (see *pytest.register_assert_rewrite()*). However for this to have any effect the module must not be imported already; if it was already imported at the time the `pytest_plugins` statement is processed, a warning will result and assertions inside the plugin will not be re-written. To fix this you can either call *pytest.register_assert_rewrite()* yourself before the module is imported, or you can arrange the code to delay the importing until after the plugin is registered.

Accessing another plugin by name

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

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

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

Testing plugins

pytest comes with a plugin named `pytester` that helps you write tests for your plugin code. The plugin is disabled by default, so you will have to enable it before you can use it.

You can do so by adding the following line to a `conftest.py` file in your testing directory:

```
# content of conftest.py

pytest_plugins = ["pytester"]
```

Alternatively you can invoke pytest with the `-p pytester` command line option.

This will allow you to use the *`testdir`* fixture for testing your plugin code.

Let's demonstrate what you can do with the plugin with an example. Imagine we developed a plugin that provides a fixture `hello` which yields a function and we can invoke this function with one optional parameter. It will return a string value of `Hello World!` if we do not supply a value or `Hello {value}!` if we do supply a string value.

```
# -*- coding: utf-8 -*-

import pytest

def pytest_addoption(parser):
    group = parser.getgroup('helloworld')
    group.addoption(
        '--name',
        action='store',
        dest='name',
        default='World',
        help='Default "name" for hello().'
    )

@pytest.fixture
def hello(request):
    name = request.config.getoption('name')

    def _hello(name=None):
        if not name:
            name = request.config.getoption('name')
        return "Hello {name}!".format(name=name)

    return _hello
```

Now the `testdir` fixture provides a convenient API for creating temporary `conftest.py` files and test files. It also allows us to run the tests and return a result object, with which we can assert the tests' outcomes.

```
def test_hello(testdir):
    """Make sure that our plugin works."""

    # create a temporary conftest.py file
    testdir.makeconftest("""
        import pytest

        @pytest.fixture(params=[
            "Brianna",
            "Andreas",
            "Floris",
        ])
        def name(request):
            return request.param
    """)

    # create a temporary pytest test file
    testdir.makepyfile("""
        def test_hello_default(hello):
            assert hello() == "Hello World!"

        def test_hello_name(hello, name):
            assert hello(name) == "Hello {0}!".format(name)
    """)

    # run all tests with pytest
    result = testdir.runpytest()

    # check that all 4 tests passed
    result.assert_outcomes(passed=4)
```

For more information about the result object that `runpytest()` returns, and the methods that it provides please check out the *`RunResult`* documentation.

CHAPTER 21

Writing hook functions

hook function validation and execution

pytest calls hook functions from registered plugins for any given hook specification. Let's look at a typical hook function for the `pytest_collection_modifyitems(session, config, items)` hook which pytest calls after collection of all test items is completed.

When we implement a `pytest_collection_modifyitems` function in our plugin pytest will during registration verify that you use argument names which match the specification and bail out if not.

Let's look at a possible implementation:

```
def pytest_collection_modifyitems(config, items):
    # called after collection is completed
    # you can modify the ``items`` list
```

Here, `pytest` will pass in `config` (the pytest config object) and `items` (the list of collected test items) but will not pass in the `session` argument because we didn't list it in the function signature. This dynamic "pruning" of arguments allows `pytest` to be "future-compatible": we can introduce new hook named parameters without breaking the signatures of existing hook implementations. It is one of the reasons for the general long-lived compatibility of pytest plugins.

Note that hook functions other than `pytest_runtest_*` are not allowed to raise exceptions. Doing so will break the pytest run.

firstresult: stop at first non-None result

Most calls to `pytest` hooks result in a **list of results** which contains all non-None results of the called hook functions.

Some hook specifications use the `firstresult=True` option so that the hook call only executes until the first of N registered functions returns a non-None result which is then taken as result of the overall hook call. The remaining hook functions will not be called in this case.

hookwrapper: executing around other hooks

New in version 2.7.

pytest plugins can implement hook wrappers which wrap the execution of other hook implementations. A hook wrapper is a generator function which yields exactly once. When pytest invokes hooks it first executes hook wrappers and passes the same arguments as to the regular hooks.

At the yield point of the hook wrapper pytest will execute the next hook implementations and return their result to the yield point in the form of a *CallOutcome* instance which encapsulates a result or exception info. The yield point itself will thus typically not raise exceptions (unless there are bugs).

Here is an example definition of a hook wrapper:

```
import pytest

@pytest.hookimpl(hookwrapper=True)
def pytest_pyfunc_call(pyfuncitem):
    # do whatever you want before the next hook executes

    outcome = yield
    # outcome.excinfo may be None or a (cls, val, tb) tuple

    res = outcome.get_result()  # will raise if outcome was exception
    # postprocess result
```

Note that hook wrappers don't return results themselves, they merely perform tracing or other side effects around the actual hook implementations. If the result of the underlying hook is a mutable object, they may modify that result but it's probably better to avoid it.

Hook function ordering / call example

For any given hook specification there may be more than one implementation and we thus generally view `hook` execution as a `1:N` function call where `N` is the number of registered functions. There are ways to influence if a hook implementation comes before or after others, i.e. the position in the `N`-sized list of functions:

```
# Plugin 1
@pytest.hookimpl(tryfirst=True)
def pytest_collection_modifyitems(items):
    # will execute as early as possible

# Plugin 2
@pytest.hookimpl(trylast=True)
def pytest_collection_modifyitems(items):
    # will execute as late as possible

# Plugin 3
@pytest.hookimpl(hookwrapper=True)
def pytest_collection_modifyitems(items):
    # will execute even before the tryfirst one above!
    outcome = yield
    # will execute after all non-hookwrappers executed
```

Here is the order of execution:

1. Plugin3's pytest_collection_modifyitems called until the yield point because it is a hook wrapper.
2. Plugin1's pytest_collection_modifyitems is called because it is marked with `tryfirst=True`.
3. Plugin2's pytest_collection_modifyitems is called because it is marked with `trylast=True` (but even without this mark it would come after Plugin1).

4. Plugin3's pytest_collection_modifyitems then executing the code after the yield point. The yield receives a *CallOutcome* instance which encapsulates the result from calling the non-wrappers. Wrappers shall not modify the result.

It's possible to use `tryfirst` and `trylast` also in conjunction with `hookwrapper=True` in which case it will influence the ordering of hookwrappers among each other.

Declaring new hooks

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

pytest_addhooks (*pluginmanager*)
called at plugin registration time to allow adding new hooks via a call to pluginmanager.add_hookspecs(module_or_class, prefix).

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

For an example, see newhooks.py from xdist.

Optionally using hooks from 3rd party plugins

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

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

```
# contents of myplugin.py

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

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

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

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

CHAPTER 22

pytest hook reference

Initialization, command line and configuration hooks

pytest_load_initial_conftests(*early_config*, *parser*, *args*)
implements the loading of initial conftest files ahead of command line option parsing.

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

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

Stops at first non-None result, see *firstresult: stop at first non-None result*

pytest_addoption(*parser*)
register argparse-style options and ini-style config values, called once at the beginning of a test run.

Note: This function should be implemented only in plugins or `conftest.py` files situated at the tests root directory due to how pytest *discovers plugins during startup*.

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

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

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

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

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

Stops at first non-None result, see *firstresult: stop at first non-None result*

pytest_configure(*config*)
Allows plugins and conftest files to perform initial configuration.

This hook is called for every plugin and initial conftest file after command line options have been parsed.

After that, the hook is called for other conftest files as they are imported.

Parameters **config** (*_pytest.config.Config*) – pytest config object

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

Generic "runtest" hooks

All runtest related hooks receive a *pytest.Item* object.

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

Parameters

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

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

Stops at first non-None result, see *firstresult: stop at first non-None result*

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

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

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

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

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

Stops at first non-None result, see *firstresult: stop at first non-None result*

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

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

Collection hooks

`pytest` calls the following hooks for collecting files and directories:

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

Stops at first non-None result, see *firstresult: stop at first non-None result*

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

Stops at first non-None result, see *firstresult: stop at first non-None result*

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

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

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

Stops at first non-None result, see *firstresult: stop at first non-None result*

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

pytest_make_parametrize_id(*config*, *val*, *argname*)
Return a user-friendly string representation of the given `val` that will be used by @pytest.mark.parametrize calls. Return None if the hook doesn't know about `val`. The parameter name is available as `argname`, if required.

Stops at first non-None result, see *firstresult: stop at first non-None result*

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

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

Reporting hooks

Session related reporting hooks:

pytest_collectstart(*collector*)
collector starts collecting.

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

pytest_collectreport(*report*)
collector finished collecting.

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

pytest_report_header(*config*, *startdir*)
return a string or list of strings to be displayed as header info for terminal reporting.

Parameters

- **config** – the pytest config object.
- **startdir** – py.path object with the starting dir

Note: This function should be implemented only in plugins or `conftest.py` files situated at the tests root directory due to how pytest *discovers plugins during startup*.

---

pytest_report_collectionfinish(*config*, *startdir*, *items*)
New in version 3.2.

return a string or list of strings to be displayed after collection has finished successfully.

This strings will be displayed after the standard "collected X items" message.

Parameters

- **config** – the pytest config object.
- **startdir** – py.path object with the starting dir
- **items** – list of pytest items that are going to be executed; this list should not be modified.

pytest_report_teststatus(*report*)
return result-category, shortletter and verbose word for reporting.

Stops at first non-None result, see *firstresult: stop at first non-None result*

pytest_terminal_summary(*terminalreporter*, *exitstatus*)
add additional section in terminal summary reporting.

pytest_fixture_setup(*fixturedef*, *request*)
performs fixture setup execution.

Stops at first non-None result, see *firstresult: stop at first non-None result*

pytest_fixture_post_finalizer(*fixturedef*)
called after fixture teardown, but before the cache is cleared so the fixture result cache `fixturedef.cached_result` can still be accessed.

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

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

You can also use this hook to customize assertion representation for some types:

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

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

Debugging/Interaction hooks

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

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

pytest_keyboard_interrupt(*excinfo*)
called for keyboard interrupt.

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

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

pytest_enter_pdb(*config*)
called upon pdb.set_trace(), can be used by plugins to take special action just before the python debugger enters in interactive mode.

Parameters **config** (_pytest.config.Config) – pytest config object

CHAPTER 23

Reference of objects involved in hooks

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

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

pluginmanager = None
a pluginmanager instance

add_cleanup(*func*)
Add a function to be called when the config object gets out of use (usually coninciding with pytest_unconfigure).

warn(*code*, *message*, *fslocation=None*, *nodeid=None*)
generate a warning for this test session.

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

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

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

getoption(*name*, *default=<NOTSET>*, *skip=False*)
return command line option value.

Parameters

- **name** – name of the option. You may also specify the literal `--OPT` option instead of the "dest" option name.
- **default** – default value if no option of that name exists.
- **skip** – if True raise pytest.skip if option does not exists or has a None value.

getvalue(*name*, *path=None*)
(deprecated, use getoption())

getvalueorskip(*name*, *path=None*)
(deprecated, use getoption(skip=True))

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

Variables **extra_info** – dict of generic param -> value to display in case there's an error processing the command line arguments.

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

Name name of the option group.

Description long description for –help output.

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

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

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

Opts option names, can be short or long options.

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

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

parse_known_args(*args*, *namespace=None*)
parses and returns a namespace object with known arguments at this point.

parse_known_and_unknown_args(*args*, *namespace=None*)
parses and returns a namespace object with known arguments, and the remaining arguments unknown at this point.

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

Name name of the ini-variable

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

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

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

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

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

parent = None
the parent collector node.

config = None
the pytest config object

session = None
the session this node is part of

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

keywords = None
keywords/markers collected from all scopes

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

ihook
fspath sensitive hook proxy used to call pytest hooks

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

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

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

add_marker(*marker*)
dynamically add a marker object to the node.

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

get_marker(*name*)
get a marker object from this node or None if the node doesn't have a marker with that name.

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

addfinalizer(*fin*)
register a function to be called when this node is finalized.

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

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

class Collector
Bases: *_pytest.main.Node*

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

exception CollectError
Bases: exceptions.Exception

an error during collection, contains a custom message.

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

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

class Item
Bases: *_pytest.main.Node*

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

add_report_section(*when*, *key*, *content*)
Adds a new report section, similar to what's done internally to add stdout and stderr captured output:

```
item.add_report_section("call", "stdout", "report section contents")
```

Parameters

- **when** (str) – One of the possible capture states, `"setup"`, `"call"`, `"teardown"`.

- **key** (str) – Name of the section, can be customized at will. Pytest uses "stdout" and "stderr" internally.
- **content** (str) – The full contents as a string.

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

Collector for test classes and functions.

class Class
Bases: _pytest.python.PyCollector

Collector for test methods.

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

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

originalname = None
original function name, without any decorations (for example parametrization adds a "[...]" suffix to function names).

New in version 3.0.

function
underlying python 'function' object

runtest()
execute the underlying test function.

class FixtureDef
A container for a factory definition.

class CallInfo
Result/Exception info a function invocation.

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

excinfo = None
None or ExceptionInfo object.

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

nodeid = None
normalized collection node id

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

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

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

longrepr = None
None or a failure representation.

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

sections = None
list of pairs `(str, str)` of extra information which needs to marshallable. Used by pytest to add captured text from `stdout` and `stderr`, but may be used by other plugins to add arbitrary information to reports.

duration = None
time it took to run just the test

capstderr
Return captured text from stderr, if capturing is enabled

New in version 3.0.

capstdout
Return captured text from stdout, if capturing is enabled

New in version 3.0.

longreprtext
Read-only property that returns the full string representation of `longrepr`.

New in version 3.0.

class _CallOutcome
Outcome of a function call, either an exception or a proper result. Calling the `get_result` method will return the result or reraise the exception raised when the function was called.

get_plugin_manager()
Obtain a new instance of the *_pytest.config.PytestPluginManager*, with default plugins already loaded.

This function can be used by integration with other tools, like hooking into pytest to run tests into an IDE.

class PytestPluginManager
Bases: *_pytest.vendored_packages.pluggy.PluginManager*

Overwrites *pluggy.PluginManager* to add pytest-specific functionality:

- loading plugins from the command line, `PYTEST_PLUGIN` env variable and `pytest_plugins` global variables found in plugins being loaded;
- `conftest.py` loading during start-up;

addhooks(*module_or_class*)
Deprecated since version 2.8.

Use *pluggy.PluginManager.add_hookspecs* instead.

parse_hookimpl_opts(*plugin*, *name*)

parse_hookspec_opts(*module_or_class*, *name*)

register(*plugin*, *name=None*)

getplugin(*name*)

hasplugin(*name*)
Return True if the plugin with the given name is registered.

pytest_configure(*config*)

consider_preparse(*args*)

consider_pluginarg(*arg*)

consider_conftest(*conftestmodule*)

consider_env()

consider_module(*mod*)

import_plugin(*modname*)

class PluginManager

Core Pluginmanager class which manages registration of plugin objects and 1:N hook calling.

You can register new hooks by calling `add_hookspec(module_or_class)`. You can register plugin objects (which contain hooks) by calling `register(plugin)`. The Pluginmanager is initialized with a prefix that is searched for in the names of the dict of registered plugin objects. An optional excludefunc allows to blacklist names which are not considered as hooks despite a matching prefix.

For debugging purposes you can call `enable_tracing()` which will subsequently send debug information to the trace helper.

register(*plugin*, *name=None*)
Register a plugin and return its canonical name or None if the name is blocked from registering. Raise a ValueError if the plugin is already registered.

unregister(*plugin=None*, *name=None*)
unregister a plugin object and all its contained hook implementations from internal data structures.

set_blocked(*name*)
block registrations of the given name, unregister if already registered.

is_blocked(*name*)
return True if the name blogs registering plugins of that name.

add_hookspecs(*module_or_class*)
add new hook specifications defined in the given module_or_class. Functions are recognized if they have been decorated accordingly.

get_plugins()
return the set of registered plugins.

is_registered(*plugin*)
Return True if the plugin is already registered.

get_canonical_name(*plugin*)
Return canonical name for a plugin object. Note that a plugin may be registered under a different name which was specified by the caller of register(plugin, name). To obtain the name of an registered plugin use `get_name(plugin)` instead.

get_plugin(*name*)
Return a plugin or None for the given name.

has_plugin(*name*)
Return True if a plugin with the given name is registered.

get_name(*plugin*)
Return name for registered plugin or None if not registered.

check_pending()
Verify that all hooks which have not been verified against a hook specification are optional, otherwise raise PluginValidationError

load_setuptools_entrypoints(*entrypoint_name*)
Load modules from querying the specified setuptools entrypoint name. Return the number of loaded plugins.

list_plugin_distinfo()
return list of distinfo/plugin tuples for all setuptools registered plugins.

list_name_plugin()
return list of name/plugin pairs.

get_hookcallers(*plugin*)
get all hook callers for the specified plugin.

add_hookcall_monitoring(*before*, *after*)
add before/after tracing functions for all hooks and return an undo function which, when called, will remove the added tracers.

`before(hook_name,hook_impls,kwargs)` will be called ahead of all hook calls and receive a hookcaller instance, a list of HookImpl instances and the keyword arguments for the hook call.

`after(outcome,hook_name,hook_impls,kwargs)` receives the same arguments as `before` but also a *_CallOutcome* object which represents the result of the overall hook call.

enable_tracing()
enable tracing of hook calls and return an undo function.

subset_hook_caller(*name*, *remove_plugins*)
Return a new _HookCaller instance for the named method which manages calls to all registered plugins except the ones from remove_plugins.

class Testdir
Temporary test directory with tools to test/run pytest itself.

This is based on the `tmpdir` fixture but provides a number of methods which aid with testing pytest itself. Unless `chdir()` is used all methods will use `tmpdir` as current working directory.

Attributes:

Tmpdir The `py.path.local` instance of the temporary directory.

Plugins A list of plugins to use with `parseconfig()` and *runpytest()*. Initially this is an empty list but plugins can be added to the list. The type of items to add to the list depend on the method which uses them so refer to them for details.

makeconftest(*source*)
Write a contest.py file with 'source' as contents.

makepyfile(**args*, ***kwargs*)
Shortcut for .makefile() with a .py extension.

runpytest(**args*, ***kwargs*)
Run pytest inline or in a subprocess, depending on the command line option "–runpytest" and return a *RunResult*.

runpytest_inprocess(**args*, ***kwargs*)
Return result of running pytest in-process, providing a similar interface to what self.runpytest() provides.

runpytest_subprocess(**args*, ***kwargs*)
Run pytest as a subprocess with given arguments.

Any plugins added to the `plugins` list will added using the `-p` command line option. Addtionally `--basetemp` is used put any temporary files and directories in a numbered directory prefixed with "runpytest-" so they do not conflict with the normal numberd pytest location for temporary files and directories.

Returns a *RunResult*.

class RunResult
The result of running a command.

Attributes:

Ret The return value.

Outlines List of lines captured from stdout.

Errlines List of lines captures from stderr.

Stdout *LineMatcher* of stdout, use `stdout.str()` to reconstruct stdout or the commonly used `stdout.fnmatch_lines()` method.

Stderrr *LineMatcher* of stderr.

Duration Duration in seconds.

assert_outcomes(*passed=0*, *skipped=0*, *failed=0*, *error=0*)
assert that the specified outcomes appear with the respective numbers (0 means it didn't occur) in the text output from a test run.

parseoutcomes()
Return a dictionary of outcomestring->num from parsing the terminal output that the test process produced.

class LineMatcher
Flexible matching of text.

This is a convenience class to test large texts like the output of commands.

The constructor takes a list of lines without their trailing newlines, i.e. `text.splitlines()`.

fnmatch_lines(*lines2*)
Search the text for matching lines.

The argument is a list of lines which have to match and can use glob wildcards. If they do not match an pytest.fail() is called. The matches and non-matches are also printed on stdout.

fnmatch_lines_random(*lines2*)
Check lines exist in the output.

The argument is a list of lines which have to occur in the output, in any order. Each line can contain glob whildcards.

get_lines_after(*fnline*)
Return all lines following the given line in the text.

The given line can contain glob wildcards.

str()
Return the entire original text.

CHAPTER 24

Good Integration Practices

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`*.
- In those directories, search for `test_*.py` or `*_test.py` files, imported by their *test package name*.
- From those files, collect test items:
  - `test_` prefixed test functions or methods outside of class
  - `test_` prefixed test functions or methods inside `Test` prefixed test classes (without an `__init__` method)

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.

Choosing a test layout / import rules

`pytest` supports two common test layouts:

Tests outside application code

Putting tests into an extra directory outside your actual application code might be 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
mypkg/
    __init__.py
    app.py
    view.py
tests/
    test_app.py
    test_view.py
    ...
```

This way your tests can run easily against an installed version of `mypkg`.

Note that using this scheme your test files must have **unique names**, because `pytest` will import them as *top-level* modules since there are no packages to derive a full package name from. In other words, the test files in the example above will be imported as `test_app` and `test_view` top-level modules by adding `tests/` to `sys.path`.

If you need to have test modules with the same name, you might add `__init__.py` files to your `tests` folder and subfolders, changing them to packages:

```
setup.py
mypkg/
    ...
tests/
    __init__.py
    foo/
        __init__.py
        test_view.py
    bar/
        __init__.py
        test_view.py
```

Now pytest will load the modules as `tests.foo.test_view` and `tests.bar.test_view`, allowing you to have modules with the same name. But now this introduces a subtle problem: in order to load the test modules from the `tests` directory, pytest prepends the root of the repository to `sys.path`, which adds the side-effect that now `mypkg` is also importable. This is problematic if you are using a tool like tox to test your package in a virtual environment, because you want to test the *installed* version of your package, not the local code from the repository.

In this situation, it is **strongly** suggested to use a `src` layout where application root package resides in a sub-directory of your root:

```
setup.py
src/
    mypkg/
        __init__.py
        app.py
        view.py
tests/
    __init__.py
    foo/
        __init__.py
        test_view.py
    bar/
        __init__.py
        test_view.py
```

This layout prevents a lot of common pitfalls and has many benefits, which are better explained in this excellent blog post by Ionel Cristian Mărieș.

Tests as part of application code

Inlining test directories into your application package is useful if you have direct relation between tests and application modules and want to distribute them along with your application:

```
setup.py
mypkg/
    __init__.py
    app.py
```

```
    view.py
    test/
        __init__.py
        test_app.py
        test_view.py
        ...
```

In this scheme, it is easy to run your tests using the `--pyargs` option:

```
pytest --pyargs mypkg
```

`pytest` will discover where `mypkg` is installed and collect tests from there.

Note that this layout also works in conjunction with the `src` layout mentioned in the previous section.

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 module getting imported twice.

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.

You can then install your package in "editable" mode:

```
pip install -e .
```

which lets you change your source code (both tests and application) and rerun tests at will. This is similar to running *python setup.py develop* or *conda develop* in that it installs your package using a symlink to your development code.

Once you are done with your work 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.

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 pytest such as your test directory or other options using `--addopts`.

You can also specify other pytest-ini options in your `setup.cfg` file by putting them into a `[tool:pytest]` section:

```
[tool:pytest]
addopts = --verbose
python_files = testing/*/*.py
```

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 pytest")]

    def initialize_options(self):
        TestCommand.initialize_options(self)
```

```
        self.pytest_args = ''

    def run_tests(self):
        import shlex
        #import here, cause outside the eggs aren't loaded
        import pytest
        errno = pytest.main(shlex.split(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 `pytest --durations=5`.

CHAPTER 25

pytest import mechanisms and `sys.path`/`PYTHONPATH`

Here's a list of scenarios where pytest may need to change `sys.path` in order to import test modules or `conftest.py` files.

Test modules / `conftest.py` files inside packages

Consider this file and directory layout:

```
root/
|- foo/
   |- __init__.py
   |- conftest.py
   |- bar/
      |- __init__.py
      |- tests/
         |- __init__.py
         |- test_foo.py
```

When executing:

```
pytest root/
```

pytest will find `foo/bar/tests/test_foo.py` and realize it is part of a package given that there's an `__init__.py` file in the same folder. It will then search upwards until it can find the last folder which still contains an `__init__.py` file in order to find the package *root* (in this case `foo/`). To load the module, it will insert `root/` to the front of `sys.path` (if not there already) in order to load `test_foo.py` as the *module* `foo.bar.tests.test_foo`.

The same logic applies to the `conftest.py` file: it will be imported as `foo.conftest` module.

Preserving the full package name is important when tests live in a package to avoid problems and allow test modules to have duplicated names. This is also discussed in details in *Conventions for Python test discovery*.

Standalone test modules / `conftest.py` files

Consider this file and directory layout:

```
root/
|- foo/
```

```
   |- conftest.py
   |- bar/
      |- tests/
         |- test_foo.py
```

When executing:

```
pytest root/
```

pytest will find `foo/bar/tests/test_foo.py` and realize it is NOT part of a package given that there's no `__init__.py` file in the same folder. It will then add `root/foo/bar/tests` to `sys.path` in order to import `test_foo.py` as the *module* `test_foo`. The same is done with the `conftest.py` file by adding `root/foo` to `sys.path` to import it as `conftest`.

For this reason this layout cannot have test modules with the same name, as they all will be imported in the global import namespace.

This is also discussed in details in *Conventions for Python test discovery*.

CHAPTER 26

Configuration

Command line options and configuration file settings

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

```
pytest -h   # prints options _and_ config file settings
```

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

Initialization: determining rootdir and inifile

New in version 2.7.

pytest determines a `rootdir` for each test run which depends on the command line arguments (specified test files, paths) and on the existence of *ini-files*. The determined `rootdir` and *ini-file* are printed as part of the pytest header during startup.

Here's a summary what `pytest` uses `rootdir` for:

- Construct *nodeids* during collection; each test is assigned a unique *nodeid* which is rooted at the `rootdir` and takes in account full path, class name, function name and parametrization (if any).
- Is used by plugins as a stable location to store project/test run specific information; for example, the internal *cache* plugin creates a `.cache` subdirectory in `rootdir` to store its cross-test run state.

Important to emphasize that `rootdir` is **NOT** used to modify `sys.path`/`PYTHONPATH` or influence how modules are imported. See *pytest import mechanisms and sys.path/PYTHONPATH* for more details.

Finding the `rootdir`

Here is the algorithm which finds the rootdir from `args`:

- determine the common ancestor directory for the specified `args` that are recognised as paths that exist in the file system. If no such paths are found, the common ancestor directory is set to the current working directory.
- look for `pytest.ini`, `tox.ini` and `setup.cfg` files in the ancestor directory and upwards. If one is matched, it becomes the ini-file and its directory becomes the rootdir.
- if no ini-file was found, look for `setup.py` upwards from the common ancestor directory to determine the `rootdir`.

- if no `setup.py` was found, look for `pytest.ini`, `tox.ini` and `setup.cfg` in each of the specified `args` and upwards. If one is matched, it becomes the ini-file and its directory becomes the rootdir.
- if no ini-file was found, use the already determined common ancestor as root directory. This allows the use of pytest in structures that are not part of a package and don't have any particular ini-file configuration.

If no `args` are given, pytest collects test below the current working directory and also starts determining the rootdir from there.

> **warning** custom pytest plugin commandline arguments may include a path, as in `pytest --log-output ../../test.log args`. Then `args` is mandatory, otherwise pytest uses the folder of test.log for rootdir determination (see also issue 1435). A dot `.` for referencing to the current working directory is also possible.

Note that an existing `pytest.ini` file will always be considered a match, whereas `tox.ini` and `setup.cfg` will only match if they contain a `[pytest]` or `[tool:pytest]` section, respectively. Options from multiple ini-files candidates are never merged - the first one wins (`pytest.ini` always wins, even if it does not contain a `[pytest]` section).

The `config` object will subsequently carry these attributes:

- `config.rootdir`: the determined root directory, guaranteed to exist.
- `config.inifile`: the determined ini-file, may be `None`.

The rootdir is used a reference directory for constructing test addresses ("nodeids") and can be used also by plugins for storing per-testrun information.

Example:

```
pytest path/to/testdir path/other/
```

will determine the common ancestor as `path` and then check for ini-files as follows:

```
# first look for pytest.ini files
path/pytest.ini
path/setup.cfg  # must also contain [tool:pytest] section to match
path/tox.ini    # must also contain [pytest] section to match
pytest.ini
... # all the way down to the root

# now look for setup.py
path/setup.py
setup.py
... # all the way down to the root
```

How to change command line options defaults

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

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

Alternatively, you can set a `PYTEST_ADDOPTS` environment variable to add command line options while the environment is in use:

```
export PYTEST_ADDOPTS="-v"
```

Here's how the command-line is built in the presence of `addopts` or the environment variable:

```
<pytest.ini:addopts> $PYTEST_ADDOTPS <extra command-line arguments>
```

So if the user executes in the command-line:

```
pytest -m slow
```

The actual command line executed is:

```
pytest -ra -q -v -m slow
```

Note that as usual for other command-line applications, in case of conflicting options the last one wins, so the example above will show verbose output because `-v` overwrites `-q`.

Builtin configuration file options

minversion
Specifies a minimal pytest version required for running tests.

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

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

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

issuing `pytest test_hello.py` actually means:

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

Default is to add no options.

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

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

Default patterns are `'.*','build','dist','CVS','_darcs','{arch}','*.egg','venv'`. Setting a `norecursedirs` replaces the default. Here is an example of how to avoid certain directories:

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

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

Additionally, `pytest` will attempt to intelligently identify and ignore a virtualenv by the presence of an activation script. Any directory deemed to be the root of a virtual environment will not be considered during test collection unless `collectinvirtualenv` is given. Note also that `norecursedirs` takes precedence over `collectinvirtualenv`; e.g. if you intend to run tests in a virtualenv with a base directory that matches `'.*'` you *must* override `norecursedirs` in addition to using the `collectinvirtualenv` flag.

testpaths

New in version 2.8.

Sets list of directories that should be searched for tests when no specific directories, files or test ids are given in the command line when executing pytest from the *rootdir* directory. Useful when all project tests are in a known location to speed up test collection and to avoid picking up undesired tests by accident.

```
# content of pytest.ini
[pytest]
testpaths = testing doc
```

This tells pytest to only look for tests in `testing` and `doc` directories when executing from the root directory.

python_files

One or more Glob-style file patterns determining which python files are considered as test modules. By default, pytest will consider any file matching with `test_*.py` and `*_test.py` globs as a test module.

python_classes

One or more name prefixes or glob-style patterns determining which classes are considered for test collection. By default, pytest will consider any class prefixed with `Test` as a test collection. Here is an example of how to collect tests from classes that end in `Suite`:

```
# content of pytest.ini
[pytest]
python_classes = *Suite
```

Note that `unittest.TestCase` derived classes are always collected regardless of this option, as `unittest`'s own collection framework is used to collect those tests.

python_functions

One or more name prefixes or glob-patterns determining which test functions and methods are considered tests. By default, pytest will consider any function prefixed with `test` as a test. Here is an example of how to collect test functions and methods that end in `_test`:

```
# content of pytest.ini
[pytest]
python_functions = *_test
```

Note that this has no effect on methods that live on a `unittest .TestCase` derived class, as `unittest`'s own collection framework is used to collect those tests.

See *Changing naming conventions* for more detailed examples.

doctest_optionflags

One or more doctest flag names from the standard `doctest` module. *See how pytest handles doctests*.

confcutdir

Sets a directory where search upwards for `conftest.py` files stops. By default, pytest will stop searching for `conftest.py` files upwards from `pytest.ini`/`tox.ini`/`setup.cfg` of the project if any, or up to the file-system root.

filterwarnings

New in version 3.1.

Sets a list of filters and actions that should be taken for matched warnings. By default all warnings emitted during the test session will be displayed in a summary at the end of the test session.

```
# content of pytest.ini
[pytest]
filterwarnings =
    error
    ignore::DeprecationWarning
```

This tells pytest to ignore deprecation warnings and turn all other warnings into errors. For more information please refer to *Warnings Capture*.

cache_dir

New in version 3.2.

Sets a directory where stores content of cache plugin. Default directory is `.cache` which is created in *rootdir*. Directory may be relative or absolute path. If setting relative path, then directory is created relative to *rootdir*. Additionally path may contain environment variables, that will be expanded. For more information about cache plugin please refer to *Cache: working with cross-testrun state*.

CHAPTER 27

Examples and customization tricks

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

For basic examples, see

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

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

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 $ pytest failure_demo.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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       AssertionError: 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       AssertionError: 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       AssertionError: 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       AssertionError: 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       AssertionError: 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
E         ...Full output truncated (7 lines hidden), use '-vv' to show

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       AssertionError: assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
E         Omitting 1 identical items, use -vv 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
E         ...Full output truncated (2 lines hidden), use '-vv' to show

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       AssertionError: assert {0, 10, 11, 12} == {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
E         ...Full output truncated (2 lines hidden), use '-vv' to show

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       AssertionError: 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
E         ...Full output truncated (2 lines hidden), use '-vv' to show

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       AssertionError: 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       AssertionError: assert 'foo' not in 'head head head head hea...ail tail tail␣
 ↪tail tail '
E         'foo' is contained here:
E           head head foo tail tail tail tail tail tail tail tail tail tail tail tail␣
 ↪tail tail 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       AssertionError: assert 'fffffffffff...fffffffffff' not in 'head head he...l␣
 ↪tail tail '
E         'fffffffffffffffff...fffffffffffffffff' is contained here:
E           head head␣
 ↪fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail tail␣
 ↪tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail␣
 ↪tail 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       AssertionError: 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.test_attribute_instance.<locals>.Foo'>()

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       AssertionError: 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 'failure_demo.test_attribute_multiple.<locals>.Foo'>()
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.test_attribute_multiple.<locals>.Bar'>()

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/python3.5/site-packages/_pytest/python_api.py:579>: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: not enough values to unpack (expected 2, got 1)

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    AssertionError

<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: not enough values to unpack (expected 2, got 0)

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       AssertionError: assert False
E        +  where False = <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       AssertionError: assert False
E        +  where False = <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 False
E        +  where False = 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(object):
            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(object):
            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(object):
            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 ========
```

Basic patterns and examples

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:

```
$ pytest -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:

```
$ pytest -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.

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:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 0 items

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

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 `pytest.mark.slow` marked tests:

```
# content of conftest.py

import pytest
def pytest_addoption(parser):
    parser.addoption("--runslow", action="store_true",
                     default=False, help="run slow tests")

def pytest_collection_modifyitems(config, items):
    if config.getoption("--runslow"):
        # --runslow given in cli: do not skip slow tests
        return
    skip_slow = pytest.mark.skip(reason="need --runslow option to run")
    for item in items:
        if "slow" in item.keywords:
            item.add_marker(skip_slow)
```

We can now write a test module like this:

```
# content of test_module.py
import pytest


def test_func_fast():
    pass


@pytest.mark.slow
def test_func_slow():
    pass
```

and when running it will see a skipped "slow" test:

```
$ pytest -rs    # "-rs" means report details on the little 's'
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py .s
======= short test summary info ========
SKIP [1] test_module.py:8: need --runslow option to run

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

Or run it including the `slow` marked test:

```
$ pytest --runslow
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py ..
```

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

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 `--full-trace` command line option is specified. Let's run our little function:

```
$ pytest -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
```

If you only want to hide certain exceptions, you can set `__tracebackhide__` to a callable which gets the `ExceptionInfo` object. You can for example use this to make sure unexpected exception types aren't hidden:

```
import operator
import pytest

class ConfigException(Exception):
    pass

def checkconfig(x):
    __tracebackhide__ = operator.methodcaller('errisinstance', ConfigException)
    if not hasattr(x, "config"):
        raise ConfigException("not configured: %s" %(x,))

def test_something():
    checkconfig(42)
```

This will avoid hiding the exception traceback on unrelated exceptions (i.e. bugs in assertion helpers).

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):
    import sys
    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.

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:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
project deps: mylib-1.1
rootdir: $REGENDOC_TMPDIR, inifile:
collected 0 items

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

It is also possible to return a list of strings which will be considered as several lines of information. You may consider `config.getoption('verbose')` in order to display more information if applicable:

```
# content of conftest.py

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

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

```
$ pytest -v
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
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:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 0 items

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

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():
    time.sleep(0.1)

def test_funcslow1():
    time.sleep(0.2)

def test_funcslow2():
    time.sleep(0.3)
```

Now we can profile which test functions execute the slowest:

```
$ pytest --durations=3
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 3 items

test_some_are_slow.py ...

======= slowest 3 test durations ========
0.30s call     test_some_are_slow.py::test_funcslow2
0.20s call     test_some_are_slow.py::test_funcslow1
0.10s call     test_some_are_slow.py::test_funcfast
======= 3 passed in 0.12 seconds ========
```

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(object):
    def test_login(self):
        pass
    def test_modification(self):
        assert 0
    def test_deletion(self):
        pass

def test_normal():
    pass
```

If we run this:

```
$ pytest -rx
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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".

Package/Directory-level fixtures (setups)

If you have nested test directories, you can have per-directory fixture scopes by placing fixture functions in a `conftest.py` file in that directory You can use all types of fixtures including *autouse fixtures* which are the equivalent of xUnit's setup/teardown concept. It's however recommended to have explicit fixture references in your tests or test classes rather than relying on implicitly executing setup/teardown functions, especially if they are far away from the actual tests.

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

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

class DB(object):
    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:

```
$ pytest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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
E       fixture 'db' not found
>       available fixtures: cache, capfd, capsys, doctest_namespace, monkeypatch,␣
↪pytestconfig, record_xml_property, recwarn, tmpdir, tmpdir_factory
>       use 'pytest --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).

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 information 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:

```
$ pytest test_module.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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
```

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 a 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):
    yield
    # 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)
```

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:

```
$ pytest -s test_module.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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.

`PYTEST_CURRENT_TEST` environment variable

New in version 3.2.

Sometimes a test session might get stuck and there might be no easy way to figure out which test got stuck, for example

if pytest was run in quiet mode (`-q`) or you don't have access to the console output. This is particularly a problem if the problem helps only sporadically, the famous "flaky" kind of tests.

`pytest` sets a `PYTEST_CURRENT_TEST` environment variable when running tests, which can be inspected by process monitoring utilities or libraries like psutil to discover which test got stuck if necessary:

```
import psutil

for pid in psutil.pids():
    environ = psutil.Process(pid).environ()
    if 'PYTEST_CURRENT_TEST' in environ:
        print(f'pytest process {pid} running: {environ["PYTEST_CURRENT_TEST"]}')
```

During the test session pytest will set `PYTEST_CURRENT_TEST` to the current test *nodeid* and the current stage, which can be `setup`, `call` and `teardown`.

For example, when running a single test function named `test_foo` from `foo_module.py`, `PYTEST_CURRENT_TEST` will be set to:

1. `foo_module.py::test_foo (setup)`
2. `foo_module.py::test_foo (call)`
3. `foo_module.py::test_foo (teardown)`

In that order.

Note: The contents of `PYTEST_CURRENT_TEST` is meant to be human readable and the actual format can be changed between releases (even bug fixes) so it shouldn't be relied on for scripting or automation.

Freezing pytest

If you freeze your application using a tool like PyInstaller 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 useful to obtain more information about a hard to reproduce bug.

Fortunately recent `PyInstaller` releases already have a custom hook for pytest, but if you are using another tool to freeze executables such as `cx_freeze` or `py2exe`, you can use `pytest.freeze_includes()` to obtain the full list of internal pytest modules. How to configure the tools to find the internal modules varies from tool to tool, however.

Instead of freezing the pytest runner as a separate executable, you can make your frozen program work as the pytest runner by some clever argument handling during program startup. This allows you to have a single executable, which is usually more convenient.

```
# 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 allows you to execute tests using the frozen application with standard `pytest` command-line options:

```
./app_main --pytest --verbose --tb=long --junitxml=results.xml test-suite/
```

Parametrizing tests

`pytest` allows to easily parametrize test functions. For basic docs, see *Parametrizing fixtures and test functions*.

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

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.getoption('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`:

```
$ pytest -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:

```
$ pytest -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.

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

@pytest.mark.parametrize("a,b,expected", [
    pytest.param(datetime(2001, 12, 12), datetime(2001, 12, 11),
                 timedelta(1), id='forward'),
    pytest.param(datetime(2001, 12, 11), datetime(2001, 12, 12),
                 timedelta(-1), id='backward'),
])
def test_timedistance_v3(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:

```
$ pytest test_time.py --collect-only
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 8 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]'>
  <Function 'test_timedistance_v3[forward]'>
  <Function 'test_timedistance_v3[backward]'>

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

In `test_timedistance_v3`, we used `pytest.param` to specify the test IDs together with the actual data, instead of listing them separately.

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(object):
    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:

```
$ pytest test_scenarios.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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:

```
$ pytest --collect-only test_scenarios.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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.

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(object):
    "one database object"
class DB2(object):
    "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:

```
$ pytest test_backends.py --collect-only
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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:

```
$ pytest -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.

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:

```
$ pytest test_indirect_list.py --collect-only
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item
<Module 'test_indirect_list.py'>
  <Function 'test_indirect[a-b]'>

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

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 = sorted(funcarglist[0])
    metafunc.parametrize(argnames, [[funcargs[name] for name in argnames]
            for funcargs in funcarglist])

class TestClass(object):
    # 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:

```
$ pytest -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
```

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', 'python3.4', 'python3.5']
@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(object):
    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):

```
. $ pytest -rs -q multipython.py
sssssssssssssss.........sss.........sss.........
======= short test summary info ========
SKIP [21] $REGENDOC_TMPDIR/CWD/multipython.py:24: 'python2.6' not found
27 passed, 21 skipped in 0.12 seconds
```

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:

```
$ pytest -rs test_module.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py .s
======= short test summary info ========
SKIP [1] $REGENDOC_TMPDIR/conftest.py:11: 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.

Working with custom markers

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

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(object):
    def test_method(self):
        pass
```

New in version 2.2.

You can then restrict a test run to only run tests marked with `webtest`:

```
$ pytest -v -m webtest
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items

test_server.py::test_send_http PASSED

======= 3 tests deselected ========
======= 1 passed, 3 deselected in 0.12 seconds ========
```

Or the inverse, running all tests except the webtest ones:

```
$ pytest -v -m "not webtest"
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
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 ========
======= 3 passed, 1 deselected in 0.12 seconds ========
```

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:

```
$ pytest -v test_server.py::TestClass::test_method
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
```

```
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED

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

You can also select on the class:

```
$ pytest -v test_server.py::TestClass
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED

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

Or select multiple nodes:

```
$ pytest -v test_server.py::TestClass test_server.py::test_send_http
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 2 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 pytest with the `-rf` option. You can also construct Node IDs from the output of `pytest --collectonly`.

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:

```
$ pytest -v -k http  # running with the above defined example module
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
```

```
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items

test_server.py::test_send_http PASSED

======= 3 tests deselected ========
======= 1 passed, 3 deselected in 0.12 seconds ========
```

And you can also run all tests except the ones that match the keyword:

```
$ pytest -k "not send_http" -v
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
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 ========
======= 3 passed, 1 deselected in 0.12 seconds ========
```

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

```
$ pytest -k "http or quick" -v
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
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 ========
======= 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 using Python's eval function.

> 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"`.

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:

```
$ pytest --markers
@pytest.mark.webtest: mark a test as a webtest.

@pytest.mark.skip(reason=None): skip the given test function with an optional reason.␣
↪Example: skip(reason="no way of currently testing this") skips the test.

@pytest.mark.skipif(condition): skip the given test function if eval(condition)␣
↪results in a True value.  Evaluation happens within the module global context.␣
↪Example: skipif('sys.platform == "win32"') skips the test if we are on the win32␣
↪platform. see http://pytest.org/latest/skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, strict=False): mark␣
↪the test function as an expected failure if eval(condition) has a True value.␣
↪Optionally specify a reason for better reporting and run=False if you don't even␣
↪want to execute the test function. If only specific exception(s) are expected, you␣
↪can list them in raises, and if the test fails in other ways, it will be reported␣
↪as a true failure. See http://pytest.org/latest/skipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times␣
↪passing in different arguments in turn. argvalues generally needs to be a list of␣
↪values if argnames specifies only one name or a list of tuples of values if␣
↪argnames specifies multiple names. Example: @parametrize('arg1', [1,2]) would lead␣
↪to two calls of the decorated test function, one with arg1=1 and another with␣
↪arg1=2.see http://pytest.org/latest/parametrize.html for more info and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all␣
↪of the specified fixtures. see http://pytest.org/latest/fixture.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin␣
↪machinery will try to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin␣
↪machinery will try to call it last/as late as possible.
```

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 `pytest --markers` gives good output
- Typos in function markers are treated as an error if you use the `--strict` option.

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(object):
    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(object):
    pytestmark = pytest.mark.webtest
```

or if you need to use multiple markers you can use a list:

```
import pytest

class TestClass(object):
    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.

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")*.

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:

```
$ pytest -E stage2
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_someenv.py s

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

and here is one that specifies exactly the environment needed:

```
$ pytest -E stage1
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_someenv.py .

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

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

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

@pytest.mark.skip(reason=None): skip the given test function with an optional reason.␣
↪Example: skip(reason="no way of currently testing this") skips the test.

@pytest.mark.skipif(condition): skip the given test function if eval(condition)␣
↪results in a True value.  Evaluation happens within the module global context.␣
↪Example: skipif('sys.platform == "win32"') skips the test if we are on the win32␣
↪platform. see http://pytest.org/latest/skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, strict=False): mark␣
↪the test function as an expected failure if eval(condition) has a True value.␣
↪Optionally specify a reason for better reporting and run=False if you don't even␣
↪want to execute the test function. If only specific exception(s) are expected, you␣
↪can list them in raises, and if the test fails in other ways, it will be reported␣
↪as a true failure. See http://pytest.org/latest/skipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times␣
↪passing in different arguments in turn. argvalues generally needs to be a list of␣
↪values if argnames specifies only one name or a list of tuples of values if␣
↪argnames specifies multiple names. Example: @parametrize('arg1', [1,2]) would lead␣
↪to two calls of the decorated test function, one with arg1=1 and another with␣
↪arg1=2.see http://pytest.org/latest/parametrize.html for more info and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all␣
↪of the specified fixtures. see http://pytest.org/latest/fixture.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin␣
↪machinery will try to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin␣
↪machinery will try to call it last/as late as possible.
```

Passing a callable to custom markers

Below is the config file that will be used in the next examples:

```
# content of conftest.py
import sys

def pytest_runtest_setup(item):
    marker = item.get_marker('my_marker')
    if marker is not None:
        for info in marker:
            print('Marker info name={} args={} kwars={}'.format(info.name, info.args,␣
↪info.kwargs))
            sys.stdout.flush()
```

A custom marker can have its argument set, i.e. `args` and `kwargs` properties, defined by either invoking it as a callable or using `pytest.mark.MARKER_NAME.with_args`. These two methods achieve the same effect most of the time.

However, if there is a callable as the single positional argument with no keyword arguments, using the `pytest.mark.MARKER_NAME(c)` will not pass `c` as a positional argument but decorate `c` with the custom marker

(see *MarkDecorator*). Fortunately, `pytest.mark.MARKER_NAME.with_args` comes to the rescue:

```
# content of test_custom_marker.py
import pytest

def hello_world(*args, **kwargs):
    return 'Hello World'

@pytest.mark.my_marker.with_args(hello_world)
def test_with_args():
    pass
```

The output is as follows:

```
$ pytest -q -s
Marker info name=my_marker args=(<function hello_world at 0xdeadbeef>,) kwars={}
.
1 passed in 0.12 seconds
```

We can see that the custom marker has its argument set extended with the function `hello_world`. This is the key difference between creating a custom marker as a callable, which invokes `__call__` behind the scenes, and using `with_args`.

Reading markers which were set from multiple places

If you are heavily using markers in your test suite you may encounter the case where a marker is applied several times to a test function. From plugin code you can read over all such settings. Example:

```
# content of test_mark_three_times.py
import pytest
pytestmark = pytest.mark.glob("module", x=1)

@pytest.mark.glob("class", x=2)
class TestClass(object):
    @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:

```
$ pytest -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
```

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 linux 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.linux
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 tests skipped and two executed tests as expected:

```
$ pytest -rs # this option reports skip reasons
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 4 items

test_plat.py s.s.
======= short test summary info ========
SKIP [2] $REGENDOC_TMPDIR/conftest.py:13: cannot run on platform linux
```

```
======= 2 passed, 2 skipped in 0.12 seconds ========
```

Note that if you specify a platform via the marker-command line option like this:

```
$ pytest -m linux
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 4 items

test_plat.py .

======= 3 tests deselected ========
======= 1 passed, 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.

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:

```
$ pytest -m interface --tb=short
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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 ========
======= 2 failed, 2 deselected in 0.12 seconds ========
```

or to select both "event" and "interface" tests:

```
$ pytest -m "interface or event" --tb=short
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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 ========
======= 3 failed, 1 deselected in 0.12 seconds ========
```

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(object):
    @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(object):
    @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:

```
$ pytest -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
```

Changing standard (Python) test discovery

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 =======
```

Keeping duplicate paths specified from command line

Default behavior of `pytest` is to ignore duplicate paths specified from the command line. Example:

```
py.test path_a path_a

...
collected 1 item
...
```

Just collect tests once.

To collect duplicate tests, use the `--keep-duplicates` option on the cli. Example:

```
py.test --keep-duplicates path_a path_a

...
```

```
collected 2 items
...
```

As the collector just works on directories, if you specify twice a single test file, `pytest` will still collect it twice, no matter if the `--keep-duplicates` is not specified. Example:

```
py.test test_a.py test_a.py

...
collected 2 items
...
```

Changing directory recursion

You can set the *norecursedirs* option in an ini-file, for example your `pytest.ini` in the project root directory:

```
# content of pytest.ini
[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.

Changing naming conventions

You can configure different naming conventions by setting the *python_files*, *python_classes* and *python_functions* configuration options. Example:

```
# content of pytest.ini
# can also be defined in tox.ini or setup.cfg file, although the section
# name in setup.cfg files should be "tool:pytest"
[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(object):
    def simple_check(self):
        pass
    def complex_check(self):
        pass
```

then the test collection looks like this:

```
$ pytest --collect-only
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
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.

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:

```
pytest --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 `pytest NAME` will check if NAME exists as an importable package/module and otherwise treat it as a filesystem path.

Finding out what is collected

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

```
. $ pytest --collect-only pythoncollection.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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 ========
```

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 and will leave out the setup.py file:

```
#$ pytest --collect-only
====== test session starts ======
platform linux2 -- Python 2.7.10, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 1 items
<Module 'pkg/module_py2.py'>
  <Function 'test_only_on_python2'>

====== no tests ran in 0.04 seconds ======
```

If you run with a Python3 interpreter both the one test and the setup.py file will be left out:

```
$ pytest --collect-only
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 0 items

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

Working with non-python tests

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 sorted(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 sorted(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 $ pytest test_simple.yml
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
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:` 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 $ pytest -v
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python3.5
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 $ pytest --collect-only
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR/nonpython, inifile:
collected 2 items
<YamlFile 'test_simple.yml'>
  <YamlItem 'hello'>
  <YamlItem 'ok'>

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

CHAPTER 28

Setting up bash completion

When using bash as your shell, `pytest` can use argcomplete (https://argcomplete.readthedocs.io/) for auto-completion. For this `argcomplete` needs to be installed **and** enabled.

Install argcomplete using:

```
sudo pip install 'argcomplete>=0.5.7'
```

For global activation of all argcomplete enabled python applications run:

```
sudo activate-global-python-argcomplete
```

For permanent (but not global) `pytest` activation, use:

```
register-python-argcomplete pytest >> ~/.bashrc
```

For one-time activation of argcomplete for `pytest` only, use:

```
eval "$(register-python-argcomplete pytest)"
```

CHAPTER 29

Backwards Compatibility Policy

Keeping backwards compatibility has a very high priority in the pytest project. Although we have deprecated functionality over the years, most of it is still supported. All deprecations in pytest were done because simpler or more efficient ways of accomplishing the same tasks have emerged, making the old way of doing things unnecessary.

With the pytest 3.0 release we introduced a clear communication scheme for when we will actually remove the old busted joint and politely ask you to use the new hotness instead, while giving you enough time to adjust your tests or raise concerns if there are valid reasons to keep deprecated functionality around.

To communicate changes we are already issuing deprecation warnings, but they are not displayed by default. In pytest 3.0 we changed the default setting so that pytest deprecation warnings are displayed if not explicitly silenced (with `--disable-pytest-warnings`).

We will only remove deprecated functionality in major releases (e.g. if we deprecate something in 3.0 we will remove it in 4.0), and keep it around for at least two minor releases (e.g. if we deprecate something in 3.9 and 4.0 is the next release, we will not remove it in 4.0 but in 5.0).

CHAPTER 30

Historical Notes

This page lists features or behavior from previous versions of pytest which have changed over the years. They are kept here as a historical note so users looking at old code can find documentation related to them.

cache plugin integrated into the core

New in version 2.8.

The functionality of the *core cache* 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.

funcargs and `pytest_funcarg__`

Changed in version 2.3.

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

`@pytest.yield_fixture` decorator

Changed in version 2.10.

Prior to version 2.10, in order to use a `yield` statement to execute teardown code one had to mark a fixture using the `yield_fixture` marker. From 2.10 onward, normal fixtures can use `yield` directly so the `yield_fixture` decorator is no longer needed and considered deprecated.

`[pytest]` header in `setup.cfg`

Changed in version 3.0.

Prior to 3.0, the supported section name was `[pytest]`. Due to how this may collide with some distutils commands, the recommended section name for `setup.cfg` files is now `[tool:pytest]`.

Note that for `pytest.ini` and `tox.ini` files the section name is `[pytest]`.

Applying marks to `@pytest.mark.parametrize` parameters

Changed in version 3.1.

Prior to version 3.1 the supported mechanism for marking values used the syntax:

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

This was an initial hack to support the feature but soon was demonstrated to be incomplete, broken for passing functions or applying multiple marks with the same name but different parameters.

The old syntax is planned to be removed in pytest-4.0.

`@pytest.mark.parametrize` argument names as a tuple

Changed in version 2.4.

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

setup: is now an "autouse fixture"

Changed in version 2.3.

During development prior to the pytest-2.3 release the name `pytest.setup` was used but before the release it was renamed and moved to become part of the general fixture mechanism, namely *Autouse fixtures (xUnit setup on steroids)*

Conditions as strings instead of booleans

Changed in version 2.4.

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

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

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

Since pytest-2.4 *boolean conditions* are considered preferable because markers can then be freely imported between test modules. With strings you need to import not only the marker but all variables used by the marker, which violates encapsulation.

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

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

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

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

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

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

The equivalent with "boolean conditions" is:

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

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

pytest.set_trace()

Changed in version 2.4.

Previous to version 2.4 to set a break point in code one needed to use `pytest.set_trace()`:

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

This is no longer needed and one can use the native `import pdb;pdb.set_trace()` call directly.

For more details see *Setting breakpoints*.

CHAPTER 31

License

Distributed under the terms of the MIT license, pytest is free and open source software.

```
The MIT License (MIT)

Copyright (c) 2004-2017 Holger Krekel and others

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 32

Contribution getting started

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

Contribution links

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.

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 the 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 cannot fix the bug itself.

Fix bugs

Look through the GitHub issues for bugs. Here is a 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!

Implement features

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

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

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 using a local copy. This can be convenient for small fixes.

Note: Build the documentation locally with the following command:

```
$ tox -e docs
```

The built documentation should be available in the `doc/en/_build/`.

Where 'en' refers to the documentation language.

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 wishes 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.
- a changelog

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):

- `joedoe` transfers repository ownership to `pytest-dev` administrator `calvin`.
- `calvin` creates `pytest-xyz-admin` and `pytest-xyz-developers` teams, inviting `joedoe` to both as **maintainer**.
- `calvin` transfers repository to `pytest-dev` and configures team access:
  - `pytest-xyz-admin` **admin** access;
  - `pytest-xyz-developers` **write** access;

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 rest 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".

Preparing Pull Requests

Short version

1. Fork the repository;
2. Target `master` for bugfixes and doc changes;
3. Target `features` for new features or functionality changes.
4. Follow **PEP-8**. There's a `tox` command to help fixing it: `tox -e fix-lint`.
5. Tests are run using `tox`:

```
tox -e linting,py27,py36
```

The test environments above are usually enough to cover most cases locally.

6. Write a `changelog` entry: `changelog/2574.bugfix`, use issue id number and one of `bugfix`, `removal`, `feature`, `vendor`, `doc` or `trivial` for the issue type.
7. Add yourself to `AUTHORS` file if not there yet, in alphabetical order.

Long version

What is a "pull request"? It informs the project's core developers about the changes you want to review and merge. Pull requests are stored on GitHub servers. Once you send a pull request, we can discuss its 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.

Here is a simple overview, with pytest-specific bits:

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 3.6 available in your system. Now running tests is as simple as issuing this command:

```
$ tox -e linting,py27,py36
```

This command will run tests via the "tox" tool against Python 2.7 and 3.6 and also perform "lint" coding-style checks.

5. You can now edit your local working copy. Please follow PEP-8.

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

If you have too much linting errors, try running:

```
$ tox -e fix-lint
```

To fix pep8 related errors.

You can pass different options to `tox`. For example, to run tests on Python 2.7 and pass options to pytest (e.g. enter pdb on failure) to pytest you can do:

```
$ tox -e py27 -- --pdb
```

Or to only run tests in a particular test module on Python 3.6:

```
$ tox -e py36 -- 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
```

7. Create a new changelog entry in `changelog`. The file should be named `<issueid>.<type>`, where *issueid* is the number of the issue related to the change and *type* is one of `bugfix`, `removal`, `feature`, `vendor`, `doc` or `trivial`.
8. Add yourself to `AUTHORS` file if not there yet, in alphabetical order.
9. 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: features        # if it's a feature
```

CHAPTER 33

Development Guide

Some general guidelines regarding development in pytest for core maintainers and general contributors. Nothing here is set in stone and can't be changed, feel free to suggest improvements or changes in the workflow.

Code Style

- PEP-8
- flake8 for quality checks
- invoke to automate development tasks

Branches

We have two long term branches:

- `master`: contains the code for the next bugfix release.
- `features`: contains the code with new features for the next minor release.

The official repository usually does not contain topic branches, developers and contributors should create topic branches in their own forks.

Exceptions can be made for cases where more than one contributor is working on the same topic or where it makes sense to use some automatic capability of the main repository, such as automatic docs from readthedocs for a branch dealing with documentation refactoring.

Issues

Any question, feature, bug or proposal is welcome as an issue. Users are encouraged to use them whenever they need.

GitHub issues should use labels to categorize them. Labels should be created sporadically, to fill a niche; we should avoid creating labels just for the sake of creating them.

Here is a list of labels and a brief description mentioning their intent.

Type

- `type: backward compatibility`: issue that will cause problems with old pytest versions.
- `type: bug`: problem that needs to be addressed.

- `type: deprecation`: feature that will be deprecated in the future.
- `type: docs`: documentation missing or needing clarification.
- `type: enhancement`: new feature or API change, should be merged into `features`.
- `type: feature-branch`: new feature or API change, should be merged into `features`.
- `type: infrastructure`: improvement to development/releases/CI structure.
- `type: performance`: performance or memory problem/improvement.
- `type: proposal`: proposal for a new feature, often to gather opinions or design the API around the new feature.
- `type: question`: question regarding usage, installation, internals or how to test something.
- `type: refactoring`: internal improvements to the code.
- `type: regression`: indicates a problem that was introduced in a release which was working previously.

Status

- `status: critical`: grave problem or usability issue that affects lots of users.
- `status: easy`: easy issue that is friendly to new contributors.
- `status: help wanted`: core developers need help from experts on this topic.
- `status: needs information`: reporter needs to provide more information; can be closed after 2 or more weeks of inactivity.

Topic

- `topic: collection`
- `topic: fixtures`
- `topic: parametrize`
- `topic: reporting`
- `topic: selection`
- `topic: tracebacks`

Plugin (internal or external)

- `plugin: cache`
- `plugin: capture`
- `plugin: doctests`
- `plugin: junitxml`
- `plugin: monkeypatch`
- `plugin: nose`
- `plugin: pastebin`
- `plugin: pytester`
- `plugin: tmpdir`
- `plugin: unittest`
- `plugin: warnings`
- `plugin: xdist`

OS

Issues specific to a single operating system. Do not use as a means to indicate where an issue originated from, only for problems that happen **only** in that system.

- `os: linux`
- `os: mac`
- `os: windows`

Temporary

Used to classify issues for limited time, to help find issues related in events for example. They should be removed after they are no longer relevant.

- `temporary: EP2017 sprint:`
- `temporary: sprint-candidate:`

Release Procedure

Our current policy for releasing is to aim for a bugfix every few weeks and a minor release every 2-3 months. The idea is to get fixes and new features out instead of trying to cram a ton of features into a release and by consequence taking a lot of time to make a new one.

Important: pytest releases must be prepared on **Linux** because the docs and examples expect to be executed in that platform.

---

1. Install development dependencies in a virtual environment with:

   ```
   pip3 install -r tasks/requirements.txt
   ```

2. Create a branch `release-X.Y.Z` with the version for the release.

   - **patch releases**: from the latest `master`;
   - **minor releases**: from the latest `features`; then merge with the latest `master`;

   Ensure your are in a clean work tree.

3. Generate docs, changelog, announcements and upload a package to your `devpi` staging server:

   ```
   invoke generate.pre-release <VERSION> <DEVPI USER> --password <DEVPI PASSWORD>
   ```

   If `--password` is not given, it is assumed the user is already logged in `devpi`. If you don't have an account, please ask for one.

4. Open a PR for this branch targeting `master`.

5. Test the package

   - **Manual method**

     Run from multiple machines:

     ```
     devpi use https://devpi.net/USER/dev
     devpi test pytest==VERSION
     ```

     Check that tests pass for relevant combinations with:

```
devpi list pytest
```

- **CI servers**

  Configure a repository as per-instructions on devpi-cloud-test to test the package on Travis and AppVeyor. All test environments should pass.

6. Publish to PyPI:

```
invoke generate.publish-release <VERSION> <DEVPI USER> <PYPI_NAME>
```

   where PYPI_NAME is the name of pypi.python.org as configured in your `~/.pypirc` file for devpi.

7. After a minor/major release, merge `release-X.Y.Z` into `master` and push (or open a PR).

CHAPTER 34

Talks and Tutorials

Books

- Python Testing with pytest, by Brian Okken (2017).

Talks and blog postings

- Pythonic testing, Igor Starikov (Russian, PyNsk, November 2016).
- 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.
- *pytest fixtures: explicit, modular, scalable*
- Testing Django Applications with pytest, Andreas Pelme, EuroPython 2013.
- Testes pythonics com py.test, Vinicius Belchior Assef Neto, Plone Conf 2013, Brazil.
- Introduction to py.test fixtures, FOSDEM 2013, Floris Bruynooghe.
- pytest feature and release highlights, Holger Krekel (GERMAN, October 2013)
- pytest introduction from Brian Okken (January 2013)
- pycon australia 2012 pytest talk from Brianna Laugher (video, slides, code)
- pycon 2012 US talk video from Holger Krekel
- monkey patching done right (blog post, consult monkeypatch plugin for up-to-date API)

Test parametrization:

- generating parametrized tests with fixtures.
- 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

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.

CHAPTER 35

Project examples

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

- 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 an 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

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 channels*)

CHAPTER 36

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.

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 *pytest fixtures: explicit, modular, scalable* and other features.

how does pytest work with Django?

In 2012, some work is going into the pytest-django plugin. It substitutes the usage of Django's `manage.py test` and allows 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.

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

Why can I use both `pytest` and `py.test` commands?

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. Once `pytest` became a separate package, the `py.test` name was retained due to avoid a naming conflict with another tool. This conflict was eventually resolved, and the `pytest` command was therefore introduced. In future versions of pytest, we may deprecate and later remove the `py.test` command to avoid perpetuating the confusion.

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 *Fixtures as Function arguments* decorator and specify `params` so that all tests depending on the factory-created resource will run multiple times with different parameters.

You can also use the `pytest_generate_tests` hook to implement the parametrization scheme of your choice. See also *Parametrizing tests* for more examples.

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 37

Contact channels

- pytest issue tracker to report bugs or suggest features (for version 2.0 and above).
- pytest on stackoverflow.com to post questions with the tag `pytest`. New Questions will usually be seen by pytest users or developers and answered quickly.
- Testing In Python: a mailing list for Python testing tools and discussion.
- pytest-dev at python.org (mailing list) pytest specific announcements and discussions.
- pytest-commit at python.org (mailing list): for commits and new issues
- *contribution guide* for help on submitting pull requests to GitHub.
- `#pylib` on irc.freenode.net IRC channel for random questions.
- private mail to Holger.Krekel at gmail com if you want to communicate sensitive issues
- merlinux.eu offers pytest and tox-related professional teaching and consulting.

Symbols

A

C

D

E

T

U

V

W

X