pytest Documentation

Release 6.3

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

Dec 31, 2020

CONTENTS

Download latest version as PDF

INSTALLATION AND GETTING STARTED

Pythons: Python 3.6, 3.7, 3.8, 3.9, PyPy3

Platforms: Linux and Windows

PyPI package name: pytest

Documentation as PDF: download latest

`pytest` is a framework that makes building simple and scalable tests easy. Tests are expressive and readable—no boilerplate code required. Get started in minutes with a small unit test or complex functional test for your application or library.

1.1 Install `pytest`

1. Run the following command in your command line:

```
pip install -U pytest
```

2. Check that you installed the correct version:

```
$ pytest --version
pytest 6.2.1
```

1.2 Create your first test

Create a simple test function with just four lines of code:

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


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

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

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
```

(continues on next page)

```
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F                                                         [100%]

================================ FAILURES ================================
_____ test_answer _____

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

test_sample.py:6: AssertionError
========================= short test summary info =========================
FAILED test_sample.py::test_answer - assert 4 == 5
========================== 1 failed in 0.12s ==========================
```

The [100%] refers to the overall progress of running all test cases. After it finishes, pytest then shows a failure report because func(3) does not return 5.

Note: You can use the assert statement to verify test expectations. pytest's Advanced assertion introspection will intelligently report intermediate values of the assert expression so you can avoid the many names of JUnit legacy methods.

1.3 Run multiple tests

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

1.4 Assert that a certain exception is raised

Use the *raises* helper to assert that some code raises an exception:

```python
# content of test_sysexit.py
import pytest


def f():
    raise SystemExit(1)


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

Execute the test function with "quiet" reporting mode:

```
$ pytest -q test_sysexit.py
.                                                                        [100%]
1 passed in 0.12s
```

Note: The `-q/--quiet` flag keeps the output brief in this and following examples.

1.5 Group multiple tests in a class

Once you develop multiple tests, you may want to group them into a class. pytest makes it easy to create a class containing more than one test:

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

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

pytest discovers all tests following its *Conventions for Python test discovery*, so it finds both `test_` prefixed functions. There is no need to subclass anything, but make sure to prefix your class with `Test` otherwise the class will be skipped. We can simply run the module by passing its filename:

```
$ pytest -q test_class.py
.F                                                                       [100%]
================================= 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
========================= short test summary info =========================
FAILED test_class.py::TestClass::test_two - AssertionError: assert False
1 failed, 1 passed in 0.12s
```

The first test passed and the second failed. You can easily see the intermediate values in the assertion to help you understand the reason for the failure.

Grouping tests in classes can be beneficial for the following reasons:

- Test organization
- Sharing fixtures for tests only in that particular class
- Applying marks at the class level and having them implicitly apply to all tests

Something to be aware of when grouping tests inside classes is that each test has a unique instance of the class. Having each test share the same class instance would be very detrimental to test isolation and would promote poor test practices. This is outlined below:

```python
# content of test_class_demo.py
class TestClassDemoInstance:
    def test_one(self):
        assert 0

    def test_two(self):
        assert 0
```

```
$ pytest -k TestClassDemoInstance -q
FF                                                                       [100%]
================================= FAILURES =================================
_____ TestClassDemoInstance.test_one _____

self = <test_class_demo.TestClassDemoInstance object at 0xdeadbeef>

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

test_class_demo.py:3: AssertionError
_____ TestClassDemoInstance.test_two _____

self = <test_class_demo.TestClassDemoInstance object at 0xdeadbeef>

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

test_class_demo.py:6: AssertionError
========================= short test summary info ==========================
FAILED test_class_demo.py::TestClassDemoInstance::test_one - assert 0
FAILED test_class_demo.py::TestClassDemoInstance::test_two - assert 0
2 failed in 0.12s
```

1.6 Request a unique temporary directory for functional tests

pytest provides Builtin fixtures/function arguments to request arbitrary resources, like a unique temporary directory:

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

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. Before the test runs, pytest creates a unique-per-test-invocation temporary directory:

```
$ pytest -q test_tmpdir.py
F                                                                        [100%]
================================= FAILURES =================================
_____ test_needsfiles _____
```

(continues on next page)

```
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
========================= short test summary info =========================
FAILED test_tmpdir.py::test_needsfiles - assert 0
1 failed in 0.12s
```

More info on tmpdir handling is available at *Temporary directories and files*.

Find out what kind of builtin *pytest fixtures* exist with the command:

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

Note that this command omits fixtures with leading _ unless the -v option is added.

1.7 Continue reading

Check out additional pytest resources to help you customize tests for your unique workflow:

- "*Calling pytest through python -m pytest*" for command line invocation examples
- "*Using pytest with an existing test suite*" for working with pre-existing tests
- "*Marking test functions with attributes*" for information on the pytest.mark mechanism
- "*pytest fixtures: explicit, modular, scalable*" for providing a functional baseline to your tests
- "*Writing plugins*" for managing and writing plugins
- "*Good Integration Practices*" for virtualenv and test layouts

USAGE AND INVOCATIONS

2.1 Calling pytest through `python -m pytest`

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 calling via `python` will also add the current directory to `sys.path`.

2.2 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

They are represented by the *pytest.ExitCode* enum. The exit codes being a part of the public API can be imported and accessed directly using:

```python
from pytest import ExitCode
```

Note: If you would like to customize the exit code in some scenarios, specially when no tests are collected, consider using the pytest-custom_exit_code plugin.

2.3 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
```

The full command-line flags can be found in the *reference*.

2.4 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
```

2.5 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* (case-insensitive), 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.

2.6 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.

2.7 Detailed summary report

The `-r` flag can be used to display a "short test summary info" at the end of the test session, making it easy in large test suites to get a clear picture of all failures, skips, xfails, etc.

It defaults to `fE` to list failures and errors.

Example:

```python
# content of test_example.py
import pytest


@pytest.fixture
def error_fixture():
    assert 0


def test_ok():
    print("ok")


def test_fail():
    assert 0
```

(continues on next page)

```python
def test_error(error_fixture):
    pass


def test_skip():
    pytest.skip("skipping this test")


def test_xfail():
    pytest.xfail("xfailing this test")


@pytest.mark.xfail(reason="always xfail")
def test_xpass():
    pass
```

```
$ pytest -ra
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 6 items

test_example.py .FEsxX                                                   [100%]

================================== ERRORS ==================================
_____ ERROR at setup of test_error _____

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
================================= FAILURES =================================
_____ test_fail _____

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
========================= short test summary info ==========================
SKIPPED [1] test_example.py:22: skipping this test
XFAIL test_example.py::test_xfail
  reason: xfailing this test
XPASS test_example.py::test_xpass always xfail
ERROR test_example.py::test_error - assert 0
FAILED test_example.py::test_fail - assert 0
== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.12s ===
```

The `-r` options accepts a number of characters after it, with `a` used above meaning "all except passes".

Here is the full list of available characters that can be used:

- `f` - failed

- `E` - error

- `s` - skipped

- `x` - xfailed

- `X` - xpassed

- `p` - passed

- `P` - passed with output

Special characters for (de)selection of groups:

- `a` - all except `pP`

- `A` - all

- `N` - none, this can be used to display nothing (since `fE` is the default)

More than one character can be used, so for example to only see failed and skipped tests, you can execute:

```
$ pytest -rfs
========================= test session starts =========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 6 items

test_example.py .FEsxX                                          [100%]

================================ ERRORS ================================
_____ ERROR at setup of test_error _____

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
=============================== FAILURES ===============================
_____ test_fail _____

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
======================= short test summary info =======================
FAILED test_example.py::test_fail - assert 0
SKIPPED [1] test_example.py:22: skipping this test
== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.12s ===
```

Using `p` lists the passing tests, whilst `P` adds an extra section "PASSES" with those tests that passed but had captured output:

```
$ pytest -rpP
========================= test session starts =========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
```

(continues on next page)

```
collected 6 items

test_example.py .FEsxX                                                [100%]

================================ ERRORS ====================================
_____ ERROR at setup of test_error _____

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
================================ FAILURES ==================================
_____ test_fail _____

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
================================ PASSES ====================================
_____ test_ok _____
-------------------------- Captured stdout call ---------------------------
ok
======================= short test summary info ===========================
PASSED test_example.py::test_ok
== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.12s ===
```

2.8 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 (or KeyboardInterrupt). 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"',)
```

2.9 Dropping to PDB (Python Debugger) at the start of a test

`pytest` allows one to drop into the PDB prompt immediately at the start of each test via a command line option:

```
pytest --trace
```

This will invoke the Python debugger at the start of every test.

2.10 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.

- Output capture gets resumed when ending the debugger session (via the `continue` command).

2.11 Using the builtin breakpoint function

Python 3.7 introduces a builtin `breakpoint()` function. Pytest supports the use of `breakpoint()` with the following behaviours:

- When `breakpoint()` is called and `PYTHONBREAKPOINT` is set to the default value, pytest will use the custom internal PDB trace UI instead of the system default `Pdb`.

- When tests are complete, the system will default back to the system `Pdb` trace UI.

- With `--pdb` passed to pytest, the custom internal Pdb trace UI is used with both `breakpoint()` and failed tests/unhandled exceptions.

- `--pdbcls` can be used to specify a custom debugger class.

2.12 Profiling test execution duration

Changed in version 6.0.

To get a list of the slowest 10 test durations over 1.0s long:

```
pytest --durations=10 --durations-min=1.0
```

By default, pytest will not show test durations that are too small (<0.005s) unless `-vv` is passed on the command-line.

2.13 Fault Handler

New in version 5.0.

The faulthandler standard module can be used to dump Python tracebacks on a segfault or after a timeout.

The module is automatically enabled for pytest runs, unless the `-p no:faulthandler` is given on the command-line.

Also the `faulthandler_timeout=X` configuration option can be used to dump the traceback of all threads if a test takes longer than `X` seconds to finish (not available on Windows).

Note: This functionality has been integrated from the external pytest-faulthandler plugin, with two small differences:

- To disable it, use `-p no:faulthandler` instead of `--no-faulthandler`: the former can be used with any plugin, so it saves one option.

- The `--faulthandler-timeout` command-line option has become the `faulthandler_timeout` configuration option. It can still be configured from the command-line using `-o faulthandler_timeout=X`.

2.14 Warning about unraisable exceptions and unhandled thread exceptions

New in version 6.2.

Note: These features only work on Python>=3.8.

Unhandled exceptions are exceptions that are raised in a situation in which they cannot propagate to a caller. The most common case is an exception raised in a `__del__` implementation.

Unhandled thread exceptions are exceptions raised in a `Thread` but not handled, causing the thread to terminate uncleanly.

Both types of exceptions are normally considered bugs, but may go unnoticed because they don't cause the program itself to crash. Pytest detects these conditions and issues a warning that is visible in the test run summary.

The plugins are automatically enabled for pytest runs, unless the `-p no:unraisableexception` (for unraisable exceptions) and `-p no:threadexception` (for thread exceptions) options are given on the command-line.

The warnings may be silenced selectivly using the *pytest.mark.filterwarnings* mark. The warning categories are `pytest.PytestUnraisableExceptionWarning` and `pytest.PytestUnhandledThreadExceptionWarning`.

2.15 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`.

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
```

New in version 4.0.

JUnit XML specification seems to indicate that `"time"` attribute should report total test execution times, including setup and teardown (1, 2). It is the default pytest behavior. To report just call durations instead, configure the `junit_duration_report` option like this:

```
[pytest]
junit_duration_report = call
```

2.15.1 record_property

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

```
def test_function(record_property):
    record_property("example_key", 1)
    assert True
```

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>
```

Alternatively, you can integrate this functionality with custom markers:

```
# content of conftest.py


def pytest_collection_modifyitems(session, config, items):
    for item in items:
        for marker in item.iter_markers(name="test_id"):
            test_id = marker.args[0]
            item.user_properties.append(("test_id", test_id))
```

And in your tests:

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

```python
@pytest.mark.test_id(1501)
def test_function():
    assert True
```

Will result in:

```xml
<testcase classname="test_function" file="test_function.py" line="0" name="test_
→function" time="0.0009">
  <properties>
    <property name="test_id" value="1501" />
  </properties>
</testcase>
```

> **Warning:** Please note that using this feature will break schema verifications for the latest JUnitXML schema. This might be a problem when used with some CI servers.

2.15.2 record_xml_attribute

To add an additional xml attribute to a testcase element, you can use `record_xml_attribute` fixture. This can also be used to override existing values:

```python
def test_function(record_xml_attribute):
    record_xml_attribute("assertions", "REQ-1234")
    record_xml_attribute("classname", "custom_classname")
    print("hello world")
    assert True
```

Unlike `record_property`, this will not add a new child element. Instead, this will add an attribute `assertions="REQ-1234"` inside the generated `testcase` tag and override the default `classname` with `"classname=custom_classname"`:

```xml
<testcase classname="custom_classname" file="test_function.py" line="0" name="test_
→function" time="0.003" assertions="REQ-1234">
    <system-out>
        hello world
    </system-out>
</testcase>
```

> **Warning:** `record_xml_attribute` 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.
>
> Using this over `record_xml_property` can help when using ci tools to parse the xml report. However, some parsers are quite strict about the elements and attributes that are allowed. Many tools use an xsd schema (like the example below) to validate incoming xml. Make sure you are using attribute names that are allowed by your parser.
>
> Below is the Scheme used by Jenkins to validate the XML report:
>
> ```xml
> <xs:element name="testcase">
>     <xs:complexType>
>         <xs:sequence>
>             <xs:element ref="skipped" minOccurs="0" maxOccurs="1"/>
>             <xs:element ref="error" minOccurs="0" maxOccurs="unbounded"/>
> ```

```
            <xs:element ref="failure" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="system-out" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="system-err" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="assertions" type="xs:string" use="optional"/>
        <xs:attribute name="time" type="xs:string" use="optional"/>
        <xs:attribute name="classname" type="xs:string" use="optional"/>
        <xs:attribute name="status" type="xs:string" use="optional"/>
    </xs:complexType>
</xs:element>
```

> **Warning:** Please note that using this feature will break schema verifications for the latest JUnitXML schema.
> This might be a problem when used with some CI servers.

2.15.3 record_testsuite_property

New in version 4.5.

If you want to add a properties node at the test-suite level, which may contains properties that are relevant to all tests, you can use the `record_testsuite_property` session-scoped fixture:

The `record_testsuite_property` session-scoped fixture can be used to add properties relevant to all tests.

```python
import pytest


@pytest.fixture(scope="session", autouse=True)
def log_global_env_facts(record_testsuite_property):
    record_testsuite_property("ARCH", "PPC")
    record_testsuite_property("STORAGE_TYPE", "CEPH")


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

The fixture is a callable which receives `name` and `value` of a `<property>` tag added at the test-suite level of the generated xml:

```xml
<testsuite errors="0" failures="0" name="pytest" skipped="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>
```

`name` must be a string, `value` will be converted to a string and properly xml-escaped.

The generated XML is compatible with the latest `xunit` standard, contrary to *record_property* and *record_xml_attribute*.

2.16 Creating resultlog format files

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.

> **Warning:** This option is rarely used and is scheduled for removal in pytest 6.0.
>
> If you use this option, consider using the new pytest-reportlog plugin instead.
>
> See the deprecation docs for more information.

2.17 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.

Changed in version 5.2.

If creating the URL fails for any reason, a warning is generated instead of failing the entire test suite.

2.18 Early loading plugins

You can early-load plugins (internal and external) explicitly in the command-line with the `-p` option:

```
pytest -p mypluginmodule
```

The option receives a `name` parameter, which can be:

- A full module dotted name, for example `myproject.plugins`. This dotted name must be importable.
- The entry-point name of a plugin. This is the name passed to `setuptools` when the plugin is registered. For example to early-load the pytest-cov plugin you can use:

```
pytest -p pytest_cov
```

2.19 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
```

2.20 Calling pytest from Python code

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`:

```python
# content of myinvoke.py
import pytest


class MyPlugin:
    def pytest_sessionfinish(self):
        print("*** test run reporting finishing")


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

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

```
$ python myinvoke.py
.FEsxX.                                                      [100%]*** test
→run reporting finishing


================================ ERRORS ================================
_____ ERROR at setup of test_error _____

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
================================ FAILURES ================================
_____ test_fail _____

    def test_fail():
>       assert 0
E       assert 0
```

```
test_example.py:14: AssertionError
========================= short test summary info =========================
FAILED test_example.py::test_fail - assert 0
ERROR test_example.py::test_error - assert 0
```

Note: Calling `pytest.main()` will result in importing your tests and any modules that they import. Due to the caching mechanism of python's import system, making subsequent calls to `pytest.main()` from the same process will not reflect changes to those files between the calls. For this reason, making multiple calls to `pytest.main()` from the same process (in order to re-run tests, for example) is not recommended.

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*.

3.1 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*.

THE WRITING AND REPORTING OF ASSERTIONS IN TESTS

4.1 Asserting with the `assert` statement

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

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_assert1.py F                                                     [100%]

================================= FAILURES =================================
_____ test_function _____

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

test_assert1.py:6: AssertionError
========================= short test summary info =========================
FAILED test_assert1.py::test_function - assert 3 == 4
========================== 1 failed in 0.12s ==========================
```

`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 *Assertion introspection details* for more information on assertion introspection.

4.2 Assertions about expected exceptions

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

```python
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:

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

        def f():
            f()

        f()
    assert "maximum recursion" in str(excinfo.value)
```

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

You can pass a match keyword parameter to the context-manager to test that a regular expression matches on the string representation of an exception (similar to the TestCase.assertRaisesRegexp method from unittest):

```python
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.

There's an alternate form of the `pytest.raises()` function where you pass a function that will be executed with the given *args and **kwargs and assert that the given exception is raised:

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

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.

4.3 Assertions about expected warnings

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

4.4 Making use of context-sensitive comparisons

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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_assert2.py F                                                    [100%]

================================= 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:6: AssertionError
========================= short test summary info ==========================
FAILED test_assert2.py::test_set_comparison - AssertionError: assert {'0'...
========================== 1 failed in 0.12s ===============================
```

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.

4.5 Defining your own explanation for failed assertions

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

pytest_assertrepr_compare (*config: Config*, *op: str*, *left: object*, *right: object*) → Optional[List[str]]
    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.

        Parameters config (`_pytest.config.Config`) – The pytest config object.

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

```python
# 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: {} != {}".format(left.val, right.val),
        ]
```

now, given this test module:

```python
# content of test_foocompare.py
class Foo:
    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                                                                        [100%]
================================ FAILURES ================================
```
(continues on next page)

```
_____ 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:12: AssertionError
========================= short test summary info ==========================
FAILED test_foocompare.py::test_compare - assert Comparing Foo instances:
1 failed in 0.12s
```

4.6 Assertion introspection details

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**.

You can manually enable assertion rewriting for an imported module by calling register_assert_rewrite before you import it (a good place to do that is in your root `conftest.py`).

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

4.6.1 Assertion rewriting caches files on disk

`pytest` will write back the rewritten modules to disk for caching. You can disable this behavior (for example to avoid leaving stale `.pyc` files around in projects that move files around a lot) by adding this to the top of your `conftest.py` file:

```python
import sys

sys.dont_write_bytecode = True
```

Note that you still get the benefits of assertion introspection, the only change is that the `.pyc` files won't be cached on disk.

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

4.6.2 Disabling assert rewriting

`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 working with the import machinery 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`.

FIVE

PYTEST FIXTURES: EXPLICIT, MODULAR, SCALABLE

Software test fixtures initialize test functions. They provide a fixed baseline so that tests execute reliably and produce consistent, repeatable, results. Initialization may setup services, state, or other operating environments. These are accessed by test functions through arguments; for each fixture used by a test function there is typically a parameter (named after the fixture) in the test function's definition.

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 function, class, module or whole test session scopes.

- teardown logic can be easily, and safely managed, no matter how many fixtures are used, without the need to carefully handle errors by hand or micromanage the order that cleanup steps are added.

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 are defined using the *@pytest.fixture* decorator, *described below*. Pytest has useful built-in fixtures, listed here for reference:

`capfd` Capture, as text, output to file descriptors `1` and `2`.

`capfdbinary` Capture, as bytes, output to file descriptors `1` and `2`.

`caplog` Control logging and access log entries.

`capsys` Capture, as text, output to `sys.stdout` and `sys.stderr`.

`capsysbinary` Capture, as bytes, output to `sys.stdout` and `sys.stderr`.

`cache` Store and retrieve values across pytest runs.

`doctest_namespace` Provide a dict injected into the docstests namespace.

`monkeypatch` Temporarily modify classes, functions, dictionaries, `os.environ`, and other objects.

`pytestconfig` Access to configuration values, pluginmanager and plugin hooks.

`record_property` Add extra properties to the test.

`record_testsuite_property` Add extra properties to the test suite.

`recwarn` Record warnings emitted by test functions.

`request` Provide information on the executing test function.

testdir Provide a temporary test directory to aid in running, and testing, pytest plugins.

tmp_path Provide a `pathlib.Path` object to a temporary directory which is unique to each test function.

tmp_path_factory Make session-scoped temporary directories and return `pathlib.Path` objects.

tmpdir Provide a `py.path.local` object to a temporary directory which is unique to each test function; replaced by *tmp_path*.

tmpdir_factory Make session-scoped temporary directories and return `py.path.local` objects; replaced by `tmp_path_factory`.

5.1 What fixtures are

Before we dive into what fixtures are, let's first look at what a test is.

In the simplest terms, a test is meant to look at the result of a particular behavior, and make sure that result aligns with what you would expect. Behavior is not something that can be empirically measured, which is why writing tests can be challenging.

"Behavior" is the way in which some system **acts in response** to a particular situation and/or stimuli. But exactly *how* or *why* something is done is not quite as important as *what* was done.

You can think of a test as being broken down into four steps:

1. **Arrange**
2. **Act**
3. **Assert**
4. **Cleanup**

Arrange is where we prepare everything for our test. This means pretty much everything except for the "**act**". It's lining up the dominoes so that the **act** can do its thing in one, state-changing step. This can mean preparing objects, starting/killing services, entering records into a database, or even things like defining a URL to query, generating some credentials for a user that doesn't exist yet, or just waiting for some process to finish.

Act is the singular, state-changing action that kicks off the **behavior** we want to test. This behavior is what carries out the changing of the state of the system under test (SUT), and it's the resulting changed state that we can look at to make a judgement about the behavior. This typically takes the form of a function/method call.

Assert is where we look at that resulting state and check if it looks how we'd expect after the dust has settled. It's where we gather evidence to say the behavior does or does not aligns with what we expect. The `assert` in our test is where we take that measurement/observation and apply our judgement to it. If something should be green, we'd say `assert thing == "green"`.

Cleanup is where the test picks up after itself, so other tests aren't being accidentally influenced by it.

At it's core, the test is ultimately the **act** and **assert** steps, with the **arrange** step only providing the context. **Behavior** exists between **act** and **assert**.

5.1.1 Back to fixtures

"Fixtures", in the literal sense, are each of the **arrange** steps and data. They're everything that test needs to do its thing.

At a basic level, test functions request fixtures by declaring them as arguments, as in the `test_ehlo(smtp_connection):` in the previous example.

In pytest, "fixtures" are functions you define that serve this purpose. But they don't have to be limited to just the **arrange** steps. They can provide the **act** step, as well, and this can be a powerful technique for designing more complex tests, especially given how pytest's fixture system works. But we'll get into that further down.

We can tell pytest that a particular function is a fixture by decorating it with `@pytest.fixture`. Here's a simple example of what a fixture in pytest might look like:

```python
import pytest


class Fruit:
    def __init__(self, name):
        self.name = name

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


@pytest.fixture
def my_fruit():
    return Fruit("apple")


@pytest.fixture
def fruit_basket(my_fruit):
    return [Fruit("banana"), my_fruit]


def test_my_fruit_in_basket(my_fruit, fruit_basket):
    assert my_fruit in fruit_basket
```

Tests don't have to be limited to a single fixture, either. They can depend on as many fixtures as you want, and fixtures can use other fixtures, as well. This is where pytest's fixture system really shines.

Don't be afraid to break things up if it makes things cleaner.

5.2 "Requesting" fixtures

So fixtures are how we *prepare* for a test, but how do we tell pytest what tests and fixtures need which fixtures?

At a basic level, test functions request fixtures by declaring them as arguments, as in the `test_my_fruit_in_basket(my_fruit, fruit_basket):` in the previous example.

At a basic level, pytest depends on a test to tell it what fixtures it needs, so we have to build that information into the test itself. We have to make the test "**request**" the fixtures it depends on, and to do this, we have to list those fixtures as parameters in the test function's "signature" (which is the `def test_something(blah, stuff, more):` line).

When pytest goes to run a test, it looks at the parameters in that test function's signature, and then searches for fixtures that have the same names as those parameters. Once pytest finds them, it runs those fixtures, captures what they

returned (if anything), and passes those objects into the test function as arguments.

5.2.1 Quick example

```python
import pytest


class Fruit:
    def __init__(self, name):
        self.name = name
        self.cubed = False

    def cube(self):
        self.cubed = True


class FruitSalad:
    def __init__(self, *fruit_bowl):
        self.fruit = fruit_bowl
        self._cube_fruit()

    def _cube_fruit(self):
        for fruit in self.fruit:
            fruit.cube()


# Arrange
@pytest.fixture
def fruit_bowl():
    return [Fruit("apple"), Fruit("banana")]


def test_fruit_salad(fruit_bowl):
    # Act
    fruit_salad = FruitSalad(*fruit_bowl)

    # Assert
    assert all(fruit.cubed for fruit in fruit_salad.fruit)
```

In this example, `test_fruit_salad` "**requests**" `fruit_bowl` (i.e. `def test_fruit_salad(fruit_bowl):`), and when pytest sees this, it will execute the `fruit_bowl` fixture function and pass the object it returns into `test_fruit_salad` as the `fruit_bowl` argument.

Here's roughly what's happening if we were to do it by hand:

```python
def fruit_bowl():
    return [Fruit("apple"), Fruit("banana")]


def test_fruit_salad(fruit_bowl):
    # Act
    fruit_salad = FruitSalad(*fruit_bowl)

    # Assert
    assert all(fruit.cubed for fruit in fruit_salad.fruit)
```

```python
# Arrange
bowl = fruit_bowl()
test_fruit_salad(fruit_bowl=bowl)
```

5.2.2 Fixtures can request other fixtures

One of pytest's greatest strengths is its extremely flexible fixture system. It allows us to boil down complex requirements for tests into more simple and organized functions, where we only need to have each one describe the things they are dependent on. We'll get more into this further down, but for now, here's a quick example to demonstrate how fixtures can use other fixtures:

```python
# contents of test_append.py
import pytest


# Arrange
@pytest.fixture
def first_entry():
    return "a"


# Arrange
@pytest.fixture
def order(first_entry):
    return [first_entry]


def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]
```

Notice that this is the same example from above, but very little changed. The fixtures in pytest **request** fixtures just like tests. All the same **requesting** rules apply to fixtures that do for tests. Here's how this example would work if we did it by hand:

```python
def first_entry():
    return "a"


def order(first_entry):
    return [first_entry]


def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]
```

```
entry = first_entry()
the_list = order(first_entry=entry)
test_string(order=the_list)
```

5.2.3 Fixtures are reusable

One of the things that makes pytest's fixture system so powerful, is that it gives us the abilty to define a generic setup step that can reused over and over, just like a normal function would be used. Two different tests can request the same fixture and have pytest give each test their own result from that fixture.

This is extremely useful for making sure tests aren't affected by each other. We can use this system to make sure each test gets its own fresh batch of data and is starting from a clean state so it can provide consistent, repeatable results.

Here's an example of how this can come in handy:

```python
# contents of test_append.py
import pytest


# Arrange
@pytest.fixture
def first_entry():
    return "a"


# Arrange
@pytest.fixture
def order(first_entry):
    return [first_entry]


def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]


def test_int(order):
    # Act
    order.append(2)

    # Assert
    assert order == ["a", 2]
```

Each test here is being given its own copy of that `list` object, which means the `order` fixture is getting executed twice (the same is true for the `first_entry` fixture). If we were to do this by hand as well, it would look something like this:

```python
def first_entry():
    return "a"


def order(first_entry):
```

```python
    return [first_entry]


def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]


def test_int(order):
    # Act
    order.append(2)

    # Assert
    assert order == ["a", 2]


entry = first_entry()
the_list = order(first_entry=entry)
test_string(order=the_list)

entry = first_entry()
the_list = order(first_entry=entry)
test_int(order=the_list)
```

5.2.4 A test/fixture can request more than one fixture at a time

Tests and fixtures aren't limited to **requesting** a single fixture at a time. They can request as many as they like. Here's another quick example to demonstrate:

```python
# contents of test_append.py
import pytest


# Arrange
@pytest.fixture
def first_entry():
    return "a"


# Arrange
@pytest.fixture
def second_entry():
    return 2


# Arrange
@pytest.fixture
def order(first_entry, second_entry):
    return [first_entry, second_entry]


# Arrange
```

```python
@pytest.fixture
def expected_list():
    return ["a", 2, 3.0]


def test_string(order, expected_list):
    # Act
    order.append(3.0)

    # Assert
    assert order == expected_list
```

5.2.5 Fixtures can be requested more than once per test (return values are cached)

Fixtures can also be **requested** more than once during the same test, and pytest won't execute them again for that test. This means we can **request** fixtures in multiple fixtures that are dependent on them (and even again in the test itself) without those fixtures being executed more than once.

```python
# contents of test_append.py
import pytest


# Arrange
@pytest.fixture
def first_entry():
    return "a"


# Arrange
@pytest.fixture
def order():
    return []


# Act
@pytest.fixture
def append_first(order, first_entry):
    return order.append(first_entry)


def test_string_only(append_first, order, first_entry):
    # Assert
    assert order == [first_entry]
```

If a **requested** fixture was executed once for every time it was **requested** during a test, then this test would fail because both `append_first` and `test_string_only` would see `order` as an empty list (i.e. `[]`), but since the return value of `order` was cached (along with any side effects executing it may have had) after the first time it was called, both the test and `append_first` were referencing the same object, and the test saw the effect `append_first` had on that object.

5.3 Autouse fixtures (fixtures you don't have to request)

Sometimes you may want to have a fixture (or even several) that you know all your tests will depend on. "Autouse" fixtures are a convenient way to make all tests automatically **request** them. This can cut out a lot of redundant **requests**, and can even provide more advanced fixture usage (more on that further down).

We can make a fixture an autouse fixture by passing in `autouse=True` to the fixture's decorator. Here's a simple example for how they can be used:

```python
# contents of test_append.py
import pytest


@pytest.fixture
def first_entry():
    return "a"


@pytest.fixture
def order(first_entry):
    return []


@pytest.fixture(autouse=True)
def append_first(order, first_entry):
    return order.append(first_entry)


def test_string_only(order, first_entry):
    assert order == [first_entry]


def test_string_and_int(order, first_entry):
    order.append(2)
    assert order == [first_entry, 2]
```

In this example, the `append_first` fixture is an autouse fixture. Because it happens automatically, both tests are affected by it, even though neither test **requested** it. That doesn't mean they *can't* be **requested** though; just that it isn't *necessary*.

5.4 Scope: sharing fixtures across classes, modules, packages or 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 a `smtp_connection` fixture function, responsible to create a connection to a preexisting SMTP server, to only be invoked once per test *module* (the default is to invoke once per test *function*). Multiple test functions in a test module will thus each receive the same `smtp_connection` fixture instance, thus saving time. Possible values for `scope` are: `function`, `class`, `module`, `package` or `session`.

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:

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


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

```python
# content of test_module.py


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


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

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

```
$ pytest test_module.py
=========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py FF                                                    [100%]

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

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

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

test_module.py:7: AssertionError
_____ test_noop _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

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

(continues on next page)

```
E       assert 0

test_module.py:13: AssertionError
========================= short test summary info ==========================
FAILED test_module.py::test_ehlo - assert 0
FAILED test_module.py::test_noop - assert 0
========================== 2 failed in 0.12s ===============================
```

You see the two `assert 0` failing and more importantly you can also see that the **exactly same** `smtp_connection` 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_connection` 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_connection` instance, you can simply declare it:

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

5.4.1 Fixture scopes

Fixtures are created when first requested by a test, and are destroyed based on their `scope`:

- `function`: the default scope, the fixture is destroyed at the end of the test.
- `class`: the fixture is destroyed during teardown of the last test in the class.
- `module`: the fixture is destroyed during teardown of the last test in the module.
- `package`: the fixture is destroyed during teardown of the last test in the package.
- `session`: the fixture is destroyed at the end of the test session.

Note: Pytest only caches one instance of a fixture at a time, which means that when using a parametrized fixture, pytest may invoke a fixture more than once in the given scope.

5.4.2 Dynamic scope

New in version 5.2.

In some cases, you might want to change the scope of the fixture without changing the code. To do that, pass a callable to `scope`. The callable must return a string with a valid scope and will be executed only once - during the fixture definition. It will be called with two keyword arguments - `fixture_name` as a string and `config` with a configuration object.

This can be especially useful when dealing with fixtures that need time for setup, like spawning a docker container. You can use the command-line argument to control the scope of the spawned containers for different environments. See the example below.

```python
def determine_scope(fixture_name, config):
    if config.getoption("--keep-containers", None):
        return "session"
```

```python
    return "function"


@pytest.fixture(scope=determine_scope)
def docker_container():
    yield spawn_container()
```

5.5 Fixture errors

pytest does its best to put all the fixtures for a given test in a linear order so that it can see which fixture happens first, second, third, and so on. If an earlier fixture has a problem, though, and raises an exception, pytest will stop executing fixtures for that test and mark the test as having an error.

When a test is marked as having an error, it doesn't mean the test failed, though. It just means the test couldn't even be attempted because one of the things it depends on had a problem.

This is one reason why it's a good idea to cut out as many unnecessary dependencies as possible for a given test. That way a problem in something unrelated isn't causing us to have an incomplete picture of what may or may not have issues.

Here's a quick example to help explain:

```python
import pytest


@pytest.fixture
def order():
    return []


@pytest.fixture
def append_first(order):
    order.append(1)


@pytest.fixture
def append_second(order, append_first):
    order.extend([2])


@pytest.fixture(autouse=True)
def append_third(order, append_second):
    order += [3]


def test_order(order):
    assert order == [1, 2, 3]
```

If, for whatever reason, `order.append(1)` had a bug and it raises an exception, we wouldn't be able to know if `order.extend([2])` or `order += [3]` would also have problems. After `append_first` throws an exception, pytest won't run any more fixtures for `test_order`, and it won't even try to run `test_order` itself. The only things that would've run would be `order` and `append_first`.

5.6 Teardown/Cleanup (AKA Fixture finalization)

When we run our tests, we'll want to make sure they clean up after themselves so they don't mess with any other tests (and also so that we don't leave behind a mountain of test data to bloat the system). Fixtures in pytest offer a very useful teardown system, which allows us to define the specific steps necessary for each fixture to clean up after itself.

This system can be leveraged in two ways.

5.6.1 1. `yield` fixtures (recommended)

"Yield" fixtures `yield` instead of `return`. With these fixtures, we can run some code and pass an object back to the requesting fixture/test, just like with the other fixtures. The only differences are:

1. `return` is swapped out for `yield`.

2. Any teardown code for that fixture is placed *after* the `yield`.

Once pytest figures out a linear order for the fixtures, it will run each one up until it returns or yields, and then move on to the next fixture in the list to do the same thing.

Once the test is finished, pytest will go back down the list of fixtures, but in the *reverse order*, taking each one that yielded, and running the code inside it that was *after* the `yield` statement.

As a simple example, let's say we want to test sending email from one user to another. We'll have to first make each user, then send the email from one user to the other, and finally assert that the other user received that message in their inbox. If we want to clean up after the test runs, we'll likely have to make sure the other user's mailbox is emptied before deleting that user, otherwise the system may complain.

Here's what that might look like:

```python
import pytest

from emaillib import Email, MailAdminClient


@pytest.fixture
def mail_admin():
    return MailAdminClient()


@pytest.fixture
def sending_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    admin_client.delete_user(user)


@pytest.fixture
def receiving_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    admin_client.delete_user(user)


def test_email_received(receiving_user, email):
    email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_email(_email, receiving_user)
    assert email in receiving_user.inbox
```

Because `receiving_user` is the last fixture to run during setup, it's the first to run during teardown.

There is a risk that even having the order right on the teardown side of things doesn't guarantee a safe cleanup. That's covered in a bit more detail in *Safe teardowns*.

Handling errors for yield fixture

If a yield fixture raises an exception before yielding, pytest won't try to run the teardown code after that yield fixture's `yield` statement. But, for every fixture that has already run successfully for that test, pytest will still attempt to tear them down as it normally would.

5.6.2 2. Adding finalizers directly

While yield fixtures are considered to be the cleaner and more straighforward option, there is another choice, and that is to add "finalizer" functions directly to the test's *request-context* object. It brings a similar result as yield fixtures, but requires a bit more verbosity.

In order to use this approach, we have to request the *request-context* object (just like we would request another fixture) in the fixture we need to add teardown code for, and then pass a callable, containing that teardown code, to its `addfinalizer` method.

We have to be careful though, because pytest will run that finalizer once it's been added, even if that fixture raises an exception after adding the finalizer. So to make sure we don't run the finalizer code when we wouldn't need to, we would only add the finalizer once the fixture would have done something that we'd need to teardown.

Here's how the previous example would look using the `addfinalizer` method:

```python
import pytest

from emaillib import Email, MailAdminClient


@pytest.fixture
def mail_admin():
    return MailAdminClient()


@pytest.fixture
def sending_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    admin_client.delete_user(user)


@pytest.fixture
def receiving_user(mail_admin, request):
    user = mail_admin.create_user()

    def delete_user():
        admin_client.delete_user(user)

    request.addfinalizer(delete_user)
    return user


@pytest.fixture
```

```python
def email(sending_user, receiving_user, request):
    _email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_email(_email, receiving_user)

    def empty_mailbox():
        receiving_user.delete_email(_email)

    request.addfinalizer(empty_mailbox)
    return _email


def test_email_received(receiving_user, email):
    assert email in receiving_user.inbox
```

It's a bit longer than yield fixtures and a bit more complex, but it does offer some nuances for when you're in a pinch.

5.7 Safe teardowns

The fixture system of pytest is *very* powerful, but it's still being run by a computer, so it isn't able to figure out how to safely teardown everything we throw at it. If we aren't careful, an error in the wrong spot might leave stuff from our tests behind, and that can cause further issues pretty quickly.

For example, consider the following tests (based off of the mail example from above):

```python
import pytest

from emaillib import Email, MailAdminClient


@pytest.fixture
def setup():
    mail_admin = MailAdminClient()
    sending_user = mail_admin.create_user()
    receiving_user = mail_admin.create_user()
    email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_emai(email, receiving_user)
    yield receiving_user, email
    receiving_user.delete_email(email)
    admin_client.delete_user(sending_user)
    admin_client.delete_user(receiving_user)


def test_email_received(setup):
    receiving_user, email = setup
    assert email in receiving_user.inbox
```

This version is a lot more compact, but it's also harder to read, doesn't have a very descriptive fixture name, and none of the fixtures can be reused easily.

There's also a more serious issue, which is that if any of those steps in the setup raise an exception, none of the teardown code will run.

One option might be to go with the `addfinalizer` method instead of yield fixtures, but that might get pretty complex and difficult to maintain (and it wouldn't be compact anymore).

5.7.1 Safe fixture structure

The safest and simplest fixture structure requires limiting fixtures to only making one state-changing action each, and then bundling them together with their teardown code, as *the email examples above* showed.

The chance that a state-changing operation can fail but still modify state is neglibible, as most of these operations tend to be transaction-based (at least at the level of testing where state could be left behind). So if we make sure that any successful state-changing action gets torn down by moving it to a separate fixture function and separating it from other, potentially failing state-changing actions, then our tests will stand the best chance at leaving the test environment the way they found it.

For an example, let's say we have a website with a login page, and we have access to an admin API where we can generate users. For our test, we want to:

1. Create a user through that admin API
2. Launch a browser using Selenium
3. Go to the login page of our site
4. Log in as the user we created
5. Assert that their name is in the header of the landing page

We wouldn't want to leave that user in the system, nor would we want to leave that browser session running, so we'll want to make sure the fixtures that create those things clean up after themselves.

Here's what that might look like:

Note: For this example, certain fixtures (i.e. `base_url` and `admin_credentials`) are implied to exist elsewhere. So for now, let's assume they exist, and we're just not looking at them.

```python
from uuid import uuid4
from urllib.parse import urljoin

from selenium.webdriver import Chrome
import pytest

from src.utils.pages import LoginPage, LandingPage
from src.utils import AdminApiClient
from src.utils.data_types import User


@pytest.fixture
def admin_client(base_url, admin_credentials):
    return AdminApiClient(base_url, **admin_credentials)


@pytest.fixture
def user(admin_client):
    _user = User(name="Susan", username=f"testuser-{uuid4()}", password="P4$$word")
    admin_client.create_user(_user)
    yield _user
    admin_client.delete_user(_user)


@pytest.fixture
def driver():
```

(continues on next page)

```python
    _driver = Chrome()
    yield _driver
    _driver.quit()


@pytest.fixture
def login(driver, base_url, user):
    driver.get(urljoin(base_url, "/login"))
    page = LoginPage(driver)
    page.login(user)


@pytest.fixture
def landing_page(driver, login):
    return LandingPage(driver)


def test_name_on_landing_page_after_login(landing_page, user):
    assert landing_page.header == f"Welcome, {user.name}!"
```

The way the dependencies are laid out means it's unclear if the `user` fixture would execute before the `driver` fixture. But that's ok, because those are atomic operations, and so it doesn't matter which one runs first because the sequence of events for the test is still linearizable. But what *does* matter is that, no matter which one runs first, if the one raises an exception while the other would not have, neither will have left anything behind. If `driver` executes before `user`, and `user` raises an exception, the driver will still quit, and the user was never made. And if `driver` was the one to raise the exception, then the driver would never have been started and the user would never have been made.

5.8 Fixture availabiility

Fixture availability is determined from the perspective of the test. A fixture is only available for tests to request if they are in the scope that fixture is defined in. If a fixture is defined inside a class, it can only be requested by tests inside that class. But if a fixture is defined inside the global scope of the module, than every test in that module, even if it's defined inside a class, can request it.

Similarly, a test can also only be affected by an autouse fixture if that test is in the same scope that autouse fixture is defined in (see *Autouse fixtures are executed first within their scope*).

A fixture can also request any other fixture, no matter where it's defined, so long as the test requesting them can see all fixtures involved.

For example, here's a test file with a fixture (`outer`) that requests a fixture (`inner`) from a scope it wasn't defined in:

```python
import pytest


@pytest.fixture
def order():
    return []


@pytest.fixture
def outer(order, inner):
```

```python
        order.append("outer")


class TestOne:
    @pytest.fixture
    def inner(self, order):
        order.append("one")

    def test_order(self, order, outer):
        assert order == ["one", "outer"]


class TestTwo:
    @pytest.fixture
    def inner(self, order):
        order.append("two")

    def test_order(self, order, outer):
        assert order == ["two", "outer"]
```

From the tests' perspectives, they have no problem seeing each of the fixtures they're dependent on:

So when they run, `outer` will have no problem finding `inner`, because pytest searched from the tests' perspectives.

Note: The scope a fixture is defined in has no bearing on the order it will be instantiated in: the order is mandated by the logic described *here*.

5.8.1 `conftest.py`: sharing fixtures across multiple files

The `conftest.py` file serves as a means of providing fixtures for an entire directory. Fixtures defined in a `conftest.py` can be used by any test in that package without needing to import them (pytest will automatically discover them).

You can have multiple nested directories/packages containing your tests, and each directory can have its own `conftest.py` with its own fixtures, adding on to the ones provided by the `conftest.py` files in parent directories.

For example, given a test file structure like this:

```
tests/
    __init__.py

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

        @pytest.fixture
        def order():
            return []

        @pytest.fixture
        def top(order, innermost):
            order.append("top")
```

```
test_top.py
    # content of tests/test_top.py
    import pytest


    @pytest.fixture
    def innermost(order):
        order.append("innermost top")


    def test_order(order, top):
        assert order == ["innermost top", "top"]

subpackage/
    __init__.py

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


        @pytest.fixture
        def mid(order):
            order.append("mid subpackage")

    test_subpackage.py
        # content of tests/subpackage/test_subpackage.py
        import pytest


        @pytest.fixture
        def innermost(order, mid):
            order.append("innermost subpackage")


        def test_order(order, top):
            assert order == ["mid subpackage", "innermost subpackage", "top"]
```

The boundaries of the scopes can be visualized like this:

The directories become their own sort of scope where fixtures that are defined in a `conftest.py` file in that directory become available for that whole scope.

Tests are allowed to search upward (stepping outside a circle) for fixtures, but can never go down (stepping inside a circle) to continue their search. So `tests/subpackage/test_subpackage.py::test_order` would be able to find the `innermost` fixture defined in `tests/subpackage/test_subpackage.py`, but the one defined in `tests/test_top.py` would be unavailable to it because it would have to step down a level (step inside a circle) to find it.

The first fixture the test finds is the one that will be used, so *fixtures can be overriden* if you need to change or extend what one does for a particular scope.

You can also use the `conftest.py` file to implement *local per-directory plugins*.

5.8.2 Fixtures from third-party plugins

Fixtures don't have to be defined in this structure to be available for tests, though. They can also be provided by third-party plugins that are installed, and this is how many pytest plugins operate. As long as those plugins are installed, the fixtures they provide can be requested from anywhere in your test suite.

Because they're provided from outside the structure of your test suite, third-party plugins don't really provide a scope like `conftest.py` files and the directories in your test suite do. As a result, pytest will search for fixtures stepping out through scopes as explained previously, only reaching fixtures defined in plugins *last*.

For example, given the following file structure:

```
tests/
    __init__.py

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

        @pytest.fixture
        def order():
            return []

    subpackage/
        __init__.py

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

            @pytest.fixture(autouse=True)
            def mid(order, b_fix):
                order.append("mid subpackage")

        test_subpackage.py
            # content of tests/subpackage/test_subpackage.py
            import pytest

            @pytest.fixture
            def inner(order, mid, a_fix):
                order.append("inner subpackage")

            def test_order(order, inner):
                assert order == ["b_fix", "mid subpackage", "a_fix", "inner subpackage
→"]
```

If `plugin_a` is installed and provides the fixture `a_fix`, and `plugin_b` is installed and provides the fixture `b_fix`, then this is what the test's search for fixtures would look like:

pytest will only search for `a_fix` and `b_fix` in the plugins after searching for them first in the scopes inside `tests/`.

5.9 Sharing test data

If you want to make test data from files available to your tests, a good way to do this is by loading these data in a fixture for use by your tests. This makes use of the automatic caching mechanisms of pytest.

Another good approach is by adding the data files in the `tests` folder. There are also community plugins available to help managing this aspect of testing, e.g. pytest-datadir and pytest-datafiles.

5.10 Fixture instantiation order

When pytest wants to execute a test, once it knows what fixtures will be executed, it has to figure out the order they'll be executed in. To do this, it considers 3 factors:

1. scope

2. dependencies

3. autouse

Names of fixtures or tests, where they're defined, the order they're defined in, and the order fixtures are requested in have no bearing on execution order beyond coincidence. While pytest will try to make sure coincidences like these stay consistent from run to run, it's not something that should be depended on. If you want to control the order, it's safest to rely on these 3 things and make sure dependencies are clearly established.

5.10.1 Higher-scoped fixtures are executed first

Within a function request for fixtures, those of higher-scopes (such as `session`) are executed before lower-scoped fixtures (such as `function` or `class`).

Here's an example:

```python
import pytest


@pytest.fixture(scope="session")
def order():
    return []


@pytest.fixture
def func(order):
    order.append("function")


@pytest.fixture(scope="class")
def cls(order):
    order.append("class")


@pytest.fixture(scope="module")
def mod(order):
    order.append("module")


@pytest.fixture(scope="package")
```

```python
def pack(order):
    order.append("package")


@pytest.fixture(scope="session")
def sess(order):
    order.append("session")


class TestClass:
    def test_order(self, func, cls, mod, pack, sess, order):
        assert order == ["session", "package", "module", "class", "function"]
```

The test will pass because the larger scoped fixtures are executing first.

The order breaks down to this:

5.10.2 Fixtures of the same order execute based on dependencies

When a fixture requests another fixture, the other fixture is executed first. So if fixture a requests fixture b, fixture b will execute first, because a depends on b and can't operate without it. Even if a doesn't need the result of b, it can still request b if it needs to make sure it is executed after b.

For example:

```python
import pytest


@pytest.fixture
def order():
    return []


@pytest.fixture
def a(order):
    order.append("a")


@pytest.fixture
def b(a, order):
    order.append("b")


@pytest.fixture
def c(a, b, order):
    order.append("c")


@pytest.fixture
def d(c, b, order):
    order.append("d")


@pytest.fixture
```

```python
def e(d, b, order):
    order.append("e")


@pytest.fixture
def f(e, order):
    order.append("f")


@pytest.fixture
def g(f, c, order):
    order.append("g")


def test_order(g, order):
    assert order == ["a", "b", "c", "d", "e", "f", "g"]
```

If we map out what depends on what, we get something that look like this:

The rules provided by each fixture (as to what fixture(s) each one has to come after) are comprehensive enough that it can be flattened to this:

Enough information has to be provided through these requests in order for pytest to be able to figure out a clear, linear chain of dependencies, and as a result, an order of operations for a given test. If there's any ambiguity, and the order of operations can be interpreted more than one way, you should assume pytest could go with any one of those interpretations at any point.

For example, if d didn't request c, i.e.the graph would look like this:

Because nothing requested c other than g, and g also requests f, it's now unclear if c should go before/after f, e, or d. The only rules that were set for c is that it must execute after b and before g.

pytest doesn't know where c should go in the case, so it should be assumed that it could go anywhere between g and b.

This isn't necessarily bad, but it's something to keep in mind. If the order they execute in could affect the behavior a test is targetting, or could otherwise influence the result of a test, then the order should be defined explicitly in a way that allows pytest to linearize/"flatten" that order.

5.10.3 Autouse fixtures are executed first within their scope

Autouse fixtures are assumed to apply to every test that could reference them, so they are executed before other fixtures in that scope. Fixtures that are requested by autouse fixtures effectively become autouse fixtures themselves for the tests that the real autouse fixture applies to.

So if fixture a is autouse and fixture b is not, but fixture a requests fixture b, then fixture b will effectively be an autouse fixture as well, but only for the tests that a applies to.

In the last example, the graph became unclear if d didn't request c. But if c was autouse, then b and a would effectively also be autouse because c depends on them. As a result, they would all be shifted above non-autouse fixtures within that scope.

So if the test file looked like this:

```python
import pytest


@pytest.fixture
def order():
    return []


@pytest.fixture
def a(order):
    order.append("a")


@pytest.fixture
def b(a, order):
    order.append("b")


@pytest.fixture(autouse=True)
def c(b, order):
    order.append("c")


@pytest.fixture
def d(b, order):
    order.append("d")


@pytest.fixture
def e(d, order):
    order.append("e")


@pytest.fixture
def f(e, order):
    order.append("f")


@pytest.fixture
def g(f, c, order):
    order.append("g")


def test_order_and_g(g, order):
    assert order == ["a", "b", "c", "d", "e", "f", "g"]
```

the graph would look like this:

Because `c` can now be put above `d` in the graph, pytest can once again linearize the graph to this:

In this example, `c` makes `b` and `a` effectively autouse fixtures as well.

Be careful with autouse, though, as an autouse fixture will automatically execute for every test that can reach it, even if they don't request it. For example, consider this file:

```python
import pytest
```

```python
@pytest.fixture(scope="class")
def order():
    return []


@pytest.fixture(scope="class", autouse=True)
def c1(order):
    order.append("c1")


@pytest.fixture(scope="class")
def c2(order):
    order.append("c2")


@pytest.fixture(scope="class")
def c3(order, c1):
    order.append("c3")


class TestClassWithC1Request:
    def test_order(self, order, c1, c3):
        assert order == ["c1", "c3"]


class TestClassWithoutC1Request:
    def test_order(self, order, c2):
        assert order == ["c1", "c2"]
```

Even though nothing in `TestClassWithC1Request` is requesting `c1`, it still is executed for the tests inside it anyway:

But just because one autouse fixture requested a non-autouse fixture, that doesn't mean the non-autouse fixture becomes an autouse fixture for all contexts that it can apply to. It only effectively becomes an auotuse fixture for the contexts the real autouse fixture (the one that requested the non-autouse fixture) can apply to.

For example, take a look at this test file:

```python
import pytest


@pytest.fixture
def order():
    return []


@pytest.fixture
def c1(order):
    order.append("c1")


@pytest.fixture
def c2(order):
    order.append("c2")
```

```python
class TestClassWithAutouse:
    @pytest.fixture(autouse=True)
    def c3(self, order, c2):
        order.append("c3")

    def test_req(self, order, c1):
        assert order == ["c2", "c3", "c1"]

    def test_no_req(self, order):
        assert order == ["c2", "c3"]


class TestClassWithoutAutouse:
    def test_req(self, order, c1):
        assert order == ["c1"]

    def test_no_req(self, order):
        assert order == []
```

It would break down to something like this:

For `test_req` and `test_no_req` inside `TestClassWithAutouse`, `c3` effectively makes `c2` an autouse fixture, which is why `c2` and `c3` are executed for both tests, despite not being requested, and why `c2` and `c3` are executed before `c1` for `test_req`.

If this made `c2` an *actual* autouse fixture, then `c2` would also execute for the tests inside `TestClassWithoutAutouse`, since they can reference `c2` if they wanted to. But it doesn't, because from the perspective of the `TestClassWithoutAutouse` tests, `c2` isn't an autouse fixture, since they can't see `c3`.

5.11 Running multiple `assert` statements safely

Sometimes you may want to run multiple asserts after doing all that setup, which makes sense as, in more complex systems, a single action can kick off multiple behaviors. pytest has a convenient way of handling this and it combines a bunch of what we've gone over so far.

All that's needed is stepping up to a larger scope, then having the **act** step defined as an autouse fixture, and finally, making sure all the fixtures are targetting that highler level scope.

Let's pull *an example from above*, and tweak it a bit. Let's say that in addition to checking for a welcome message in the header, we also want to check for a sign out button, and a link to the user's profile.

Let's take a look at how we can structure that so we can run multiple asserts without having to repeat all those steps again.

Note: For this example, certain fixtures (i.e. `base_url` and `admin_credentials`) are implied to exist elsewhere. So for now, let's assume they exist, and we're just not looking at them.

```python
# contents of tests/end_to_end/test_login.py
from uuid import uuid4
from urllib.parse import urljoin
```

```python
from selenium.webdriver import Chrome
import pytest

from src.utils.pages import LoginPage, LandingPage
from src.utils import AdminApiClient
from src.utils.data_types import User


@pytest.fixture(scope="class")
def admin_client(base_url, admin_credentials):
    return AdminApiClient(base_url, **admin_credentials)


@pytest.fixture(scope="class")
def user(admin_client):
    _user = User(name="Susan", username=f"testuser-{uuid4()}", password="P4$$word")
    admin_client.create_user(_user)
    yield _user
    admin_client.delete_user(_user)


@pytest.fixture(scope="class")
def driver():
    _driver = Chrome()
    yield _driver
    _driver.quit()


@pytest.fixture(scope="class")
def landing_page(driver, login):
    return LandingPage(driver)


class TestLandingPageSuccess:
    @pytest.fixture(scope="class", autouse=True)
    def login(self, driver, base_url, user):
        driver.get(urljoin(base_url, "/login"))
        page = LoginPage(driver)
        page.login(user)

    def test_name_in_header(self, landing_page, user):
        assert landing_page.header == f"Welcome, {user.name}!"

    def test_sign_out_button(self, landing_page):
        assert landing_page.sign_out_button.is_displayed()

    def test_profile_link(self, landing_page, user):
        profile_href = urljoin(base_url, f"/profile?id={user.profile_id}")
        assert landing_page.profile_link.get_attribute("href") == profile_href
```

Notice that the methods are only referencing `self` in the signature as a formality. No state is tied to the actual test class as it might be in the `unittest.TestCase` framework. Everything is managed by the pytest fixture system.

Each method only has to request the fixtures that it actually needs without worrying about order. This is because the **act** fixture is an autouse fixture, and it made sure all the other fixtures executed before it. There's no more changes of state that need to take place, so the tests are free to make as many non-state-changing queries as they want without

risking stepping on the toes of the other tests.

The `login` fixture is defined inside the class as well, because not every one of the other tests in the module will be expecting a successful login, and the **act** may need to be handled a little differently for another test class. For example, if we wanted to write another test scenario around submitting bad credentials, we could handle it by adding something like this to the test file:

```python
class TestLandingPageBadCredentials:
    @pytest.fixture(scope="class")
    def faux_user(self, user):
        _user = deepcopy(user)
        _user.password = "badpass"
        return _user

    def test_raises_bad_credentials_exception(self, login_page, faux_user):
        with pytest.raises(BadCredentialsException):
            login_page.login(faux_user)
```

5.12 Fixtures can introspect the requesting test context

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

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


@pytest.fixture(scope="module")
def smtp_connection(request):
    server = getattr(request.module, "smtpserver", "smtp.gmail.com")
    smtp_connection = smtplib.SMTP(server, 587, timeout=5)
    yield smtp_connection
    print("finalizing {} ({})".format(smtp_connection, server))
    smtp_connection.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)

========================= short test summary info ==========================
FAILED test_module.py::test_ehlo - assert 0
FAILED test_module.py::test_noop - assert 0
2 failed in 0.12s
```

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

```python
# content of test_anothersmtp.py

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

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

Running it:

```
$ pytest -qq --tb=short test_anothersmtp.py
F                                                                        [100%]
================================= FAILURES =================================
_____ test_showhelo _____
test_anothersmtp.py:6: in test_showhelo
    assert 0, smtp_connection.helo()
E   AssertionError: (250, b'mail.python.org')
E   assert 0
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef> (mail.python.org)
========================= short test summary info ==========================
FAILED test_anothersmtp.py::test_showhelo - AssertionError: (250, b'mail....
```

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

5.13 Using markers to pass data to fixtures

Using the `request` object, a fixture can also access markers which are applied to a test function. This can be useful to pass data into a fixture from a test:

```python
import pytest


@pytest.fixture
def fixt(request):
    marker = request.node.get_closest_marker("fixt_data")
    if marker is None:
        # Handle missing marker in some way...
        data = None
    else:
        data = marker.args[0]

    # Do something with the data
    return data


@pytest.mark.fixt_data(42)
def test_fixt(fixt):
    assert fixt == 42
```

5.14 Factories as fixtures

The "factory as fixture" pattern can help in situations where the result of a fixture is needed multiple times in a single test. Instead of returning data directly, the fixture instead returns a function which generates the data. This function can then be called multiple times in the test.

Factories can have parameters as needed:

```python
@pytest.fixture
def make_customer_record():
    def _make_customer_record(name):
        return {"name": name, "orders": []}

    return _make_customer_record


def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

If the data created by the factory requires managing, the fixture can take care of that:

```python
@pytest.fixture
def make_customer_record():

    created_records = []

    def _make_customer_record(name):
        record = models.Customer(name=name, orders=[])
        created_records.append(record)
        return record

    yield _make_customer_record

    for record in created_records:
        record.destroy()


def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

5.15 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 usually do 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_connection` 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:

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


@pytest.fixture(scope="module", params=["smtp.gmail.com", "mail.python.org"])
def smtp_connection(request):
    smtp_connection = smtplib.SMTP(request.param, 587, timeout=5)
    yield smtp_connection
    print("finalizing {}".format(smtp_connection))
    smtp_connection.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                                                                     [100%]
================================= FAILURES =================================
_____ test_ehlo[smtp.gmail.com] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

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

test_module.py:7: AssertionError
_____ test_noop[smtp.gmail.com] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

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

test_module.py:13: AssertionError
_____ test_ehlo[mail.python.org] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.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\nCHUNKING'

test_module.py:6: AssertionError
------------------------- Captured stdout setup ---------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
```

```
_____ test_noop[mail.python.org] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

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

test_module.py:13: AssertionError
----------------------- Captured stdout teardown ------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
========================= short test summary info ==========================
FAILED test_module.py::test_ehlo[smtp.gmail.com] - assert 0
FAILED test_module.py::test_noop[smtp.gmail.com] - assert 0
FAILED test_module.py::test_ehlo[mail.python.org] - AssertionError: asser...
FAILED test_module.py::test_noop[mail.python.org] - assert 0
4 failed in 0.12s
```

We see that our two test functions each ran twice, against the different `smtp_connection` 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:

```python
# 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 returns `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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
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]>

====================== 10 tests collected in 0.12s =======================
```

5.16 Using marks with parametrized fixtures

pytest.param() can be used to apply marks in values sets of parametrized fixtures in the same way that they can be used with *@pytest.mark.parametrize*.

Example:

```
# content of test_fixture_marks.py
import pytest


@pytest.fixture(params=[0, 1, pytest.param(2, marks=pytest.mark.skip)])
def data_set(request):
    return request.param


def test_data(data_set):
    pass
```

Running this test will *skip* the invocation of `data_set` with value `2`:

```
$ pytest test_fixture_marks.py -v
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 3 items

test_fixture_marks.py::test_data[0] PASSED                               [ 33%]
test_fixture_marks.py::test_data[1] PASSED                               [ 66%]
test_fixture_marks.py::test_data[2] SKIPPED (unconditional skip)     [100%]

====================== 2 passed, 1 skipped in 0.12s ======================
```

5.17 Modularity: using fixtures from a fixture function

In addition to using fixtures in test functions, 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_connection resource into it:

```python
# content of test_appsetup.py

import pytest


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


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


def test_smtp_connection_exists(app):
    assert app.smtp_connection
```

Here we declare an app fixture which receives the previously defined smtp_connection 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-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 2 items

test_appsetup.py::test_smtp_connection_exists[smtp.gmail.com] PASSED [ 50%]
test_appsetup.py::test_smtp_connection_exists[mail.python.org] PASSED [100%]

=========================== 2 passed in 0.12s ============================
```

Due to the parametrization of `smtp_connection`, 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_connection` parametrization because pytest will fully analyse the fixture dependency graph.

Note that the `app` fixture has a scope of `module` and uses a module-scoped `smtp_connection` fixture. The example would still work if `smtp_connection` 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.

5.18 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 fixtures, one of which is scoped on a per-module basis, and all the functions perform `print` calls to show the setup/teardown flow:

```python
# content of test_module.py
import pytest


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


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


def test_0(otherarg):
    print("  RUN test0 with otherarg", otherarg)


def test_1(modarg):
    print("  RUN test1 with modarg", modarg)


def test_2(otherarg, modarg):
    print("  RUN test2 with otherarg {} and modarg {}".format(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-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
```

(continues on next page)

```
rootdir: $REGENDOC_TMPDIR
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[mod1-1]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod1
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[mod1-2]   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[mod2-1]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod2
PASSED  TEARDOWN otherarg 1

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


=========================== 8 passed in 0.12s ===============================
```

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.

5.19 Use fixtures in classes and modules with `usefixtures`

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:

```python
# content of conftest.py

import os
import shutil
import tempfile

import pytest


@pytest.fixture
def cleandir():
    old_cwd = os.getcwd()
    newpath = tempfile.mkdtemp()
    os.chdir(newpath)
    yield
    os.chdir(old_cwd)
    shutil.rmtree(newpath)
```

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

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


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

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

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

```
$ pytest -q
..                                                                       [100%]
2 passed in 0.12s
```

You can specify multiple fixtures like this:

```python
@pytest.mark.usefixtures("cleandir", "anotherfixture")
def test():
    ...
```

and you may specify fixture usage at the test module level using *pytestmark*:

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

It is also possible to put fixtures required by all tests in your project into an ini-file:

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

> **Warning:** Note this mark has no effect in **fixture functions**. For example, this **will not work as expected**:
>
> ```python
> @pytest.mark.usefixtures("my_other_fixture")
> @pytest.fixture
> def my_fixture_that_sadly_wont_use_my_other_fixture():
>     ...
> ```
>
> Currently this will not generate any error or warning, but this is intended to be handled by #3664.

5.20 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.

5.20.1 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
```

(continues on next page)

```
            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.

5.20.2 Override a fixture on a test module 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
        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.

5.20.3 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).

5.20.4 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
```

```python
    @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.

5.21 Using fixtures from other projects

Usually projects that provide pytest support will use *entry points*, so just installing those projects into an environment will make those fixtures available for use.

In case you want to use fixtures from a project that does not use entry points, you can define `pytest_plugins` in your top `conftest.py` file to register that module as a plugin.

Suppose you have some fixtures in `mylibrary.fixtures` and you want to reuse them into your `app/tests` directory.

All you need to do is to define `pytest_plugins` in `app/tests/conftest.py` pointing to that module.

```python
pytest_plugins = "mylibrary.fixtures"
```

This effectively registers `mylibrary.fixtures` as a plugin, making all its fixtures and hooks available to tests in `app/tests`.

Note: Sometimes users will *import* fixtures from other projects for use, however this is not recommended: importing fixtures into a module will register them in pytest as *defined* in that module.

This has minor consequences, such as appearing multiple times in `pytest --help`, but it is not **recommended** because this behavior might change/stop working in future versions.

MARKING TEST FUNCTIONS WITH ATTRIBUTES

By using the `pytest.mark` helper you can easily set metadata on your test functions. You can find the full list of builtin markers in the *API Reference*. Or you can list all the markers, including builtin and custom, using the CLI - `pytest --markers`.

Here are some of the builtin markers:

- *usefixtures* - use fixtures on a test function or class
- *filterwarnings* - filter certain warnings of a test function
- *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* - 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. Those markers can be used by plugins, and also are commonly used to *select tests* on the command-line with the −m option.

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*.

6.1 Registering marks

You can register custom marks in your `pytest.ini` file like this:

```
[pytest]
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    serial
```

or in your `pyproject.toml` file like this:

```
[tool.pytest.ini_options]
markers = [
    "slow: marks tests as slow (deselect with '-m \"not slow\"')",
    "serial",
]
```

Note that everything past the `:` after the mark name is an optional description.

Alternatively, you can register new markers programmatically in a *pytest_configure* hook:

```python
def pytest_configure(config):
    config.addinivalue_line(
        "markers", "env(name): mark test to run only on named environment"
    )
```

Registered marks appear in pytest's help text and do not emit warnings (see the next section). It is recommended that third-party plugins always *register their markers*.

6.2 Raising errors on unknown marks

Unregistered marks applied with the `@pytest.mark.name_of_the_mark` decorator will always emit a warning in order to avoid silently doing something surprising due to mistyped names. As described in the previous section, you can disable the warning for custom marks by registering them in your `pytest.ini` file or using a custom `pytest_configure` hook.

When the `--strict-markers` command-line flag is passed, any unknown marks applied with the `@pytest.mark.name_of_the_mark` decorator will trigger an error. You can enforce this validation in your project by adding `--strict-markers` to `addopts`:

```ini
[pytest]
addopts = --strict-markers
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    serial
```

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.

The `monkeypatch` fixture provides these helper methods for safely patching and mocking functionality in tests:

```
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, 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 target of the set/deletion operation does not exist.

Consider the following scenarios:

1. Modifying the behavior of a function or the property of a class for a test e.g. there is an API call or database connection you will not make for a test but you know what the expected output should be. Use `monkeypatch.setattr` to patch the function or property with your desired testing behavior. This can include your own functions. Use `monkeypatch.delattr` to remove the function or property for the test.

2. Modifying the values of dictionaries e.g. you have a global configuration that you want to modify for certain test cases. Use `monkeypatch.setitem` to patch the dictionary for the test. `monkeypatch.delitem` can be used to remove items.

3. Modifying environment variables for a test e.g. to test program behavior if an environment variable is missing, or to set multiple values to a known variable. `monkeypatch.setenv` and `monkeypatch.delenv` can be used for these patches.

4. Use `monkeypatch.setenv("PATH", value, prepend=os.pathsep)` to modify `$PATH`, and `monkeypatch.chdir` to change the context of the current working directory during a test.

5. Use `monkeypatch.syspath_prepend` to modify `sys.path` which will also call `pkg_resources.fixup_namespace_packages` and `importlib.invalidate_caches()`.

See the monkeypatch blog post for some introduction material and a discussion of its motivation.

7.1 Simple example: monkeypatching functions

Consider a scenario where you are working with user directories. In the context of testing, you do not want your test to depend on the running user. `monkeypatch` can be used to patch functions dependent on the user to always return a specific value.

In this example, `monkeypatch.setattr` is used to patch `Path.home` so that the known testing path `Path("/abc")` is always used when the test is run. This removes any dependency on the running user for testing purposes. `monkeypatch.setattr` must be called before the function which will use the patched function is called. After the test function finishes the `Path.home` modification will be undone.

```python
# contents of test_module.py with source code and the test
from pathlib import Path


def getssh():
    """Simple function to return expanded homedir ssh path."""
    return Path.home() / ".ssh"


def test_getssh(monkeypatch):
    # mocked return function to replace Path.home
    # always return '/abc'
    def mockreturn():
        return Path("/abc")

    # Application of the monkeypatch to replace Path.home
    # with the behavior of mockreturn defined above.
    monkeypatch.setattr(Path, "home", mockreturn)

    # Calling getssh() will use mockreturn in place of Path.home
    # for this test with the monkeypatch.
    x = getssh()
    assert x == Path("/abc/.ssh")
```

7.2 Monkeypatching returned objects: building mock classes

`monkeypatch.setattr` can be used in conjunction with classes to mock returned objects from functions instead of values. Imagine a simple function to take an API url and return the json response.

```python
# contents of app.py, a simple API retrieval example
import requests


def get_json(url):
    """Takes a URL, and returns the JSON."""
    r = requests.get(url)
    return r.json()
```

We need to mock `r`, the returned response object for testing purposes. The mock of `r` needs a `.json()` method which returns a dictionary. This can be done in our test file by defining a class to represent `r`.

```python
# contents of test_app.py, a simple test for our API retrieval
# import requests for the purposes of monkeypatching
```

```python
import requests

# our app.py that includes the get_json() function
# this is the previous code block example
import app

# custom class to be the mock return value
# will override the requests.Response returned from requests.get
class MockResponse:

    # mock json() method always returns a specific testing dictionary
    @staticmethod
    def json():
        return {"mock_key": "mock_response"}


def test_get_json(monkeypatch):

    # Any arguments may be passed and mock_get() will always return our
    # mocked object, which only has the .json() method.
    def mock_get(*args, **kwargs):
        return MockResponse()

    # apply the monkeypatch for requests.get to mock_get
    monkeypatch.setattr(requests, "get", mock_get)

    # app.get_json, which contains requests.get, uses the monkeypatch
    result = app.get_json("https://fakeurl")
    assert result["mock_key"] == "mock_response"
```

monkeypatch applies the mock for requests.get with our mock_get function. The mock_get function returns an instance of the MockResponse class, which has a json() method defined to return a known testing dictionary and does not require any outside API connection.

You can build the MockResponse class with the appropriate degree of complexity for the scenario you are testing. For instance, it could include an ok property that always returns True, or return different values from the json() mocked method based on input strings.

This mock can be shared across tests using a fixture:

```python
# contents of test_app.py, a simple test for our API retrieval
import pytest
import requests

# app.py that includes the get_json() function
import app

# custom class to be the mock return value of requests.get()
class MockResponse:
    @staticmethod
    def json():
        return {"mock_key": "mock_response"}


# monkeypatched requests.get moved to a fixture
@pytest.fixture
def mock_response(monkeypatch):
```

7.2. Monkeypatching returned objects: building mock classes

```python
    """Requests.get() mocked to return {'mock_key':'mock_response'}."""

    def mock_get(*args, **kwargs):
        return MockResponse()

    monkeypatch.setattr(requests, "get", mock_get)


# notice our test uses the custom fixture instead of monkeypatch directly
def test_get_json(mock_response):
    result = app.get_json("https://fakeurl")
    assert result["mock_key"] == "mock_response"
```

Furthermore, if the mock was designed to be applied to all tests, the `fixture` could be moved to a `conftest.py` file and use the with `autouse=True` option.

7.3 Global patch 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:

```python
# contents of conftest.py
import pytest


@pytest.fixture(autouse=True)
def no_requests(monkeypatch):
    """Remove requests.sessions.Session.request for all tests."""
    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.

Note: Mind that patching `stdlib` functions and some third-party libraries used by pytest might break pytest itself, therefore in those cases it is recommended to use `MonkeyPatch.context()` to limit the patching to the block you want tested:

```python
import functools


def test_partial(monkeypatch):
    with monkeypatch.context() as m:
        m.setattr(functools, "partial", 3)
        assert functools.partial == 3
```

See issue #3290 for details.

7.4 Monkeypatching environment variables

If you are working with environment variables you often need to safely change the values or delete them from the system for testing purposes. `monkeypatch` provides a mechanism to do this using the `setenv` and `delenv` method. Our example code to test:

```python
# contents of our original code file e.g. code.py
import os


def get_os_user_lower():
    """Simple retrieval function.
    Returns lowercase USER or raises OSError."""
    username = os.getenv("USER")

    if username is None:
        raise OSError("USER environment is not set.")

    return username.lower()
```

There are two potential paths. First, the `USER` environment variable is set to a value. Second, the `USER` environment variable does not exist. Using `monkeypatch` both paths can be safely tested without impacting the running environment:

```python
# contents of our test file e.g. test_code.py
import pytest


def test_upper_to_lower(monkeypatch):
    """Set the USER env var to assert the behavior."""
    monkeypatch.setenv("USER", "TestingUser")
    assert get_os_user_lower() == "testinguser"


def test_raise_exception(monkeypatch):
    """Remove the USER env var and assert OSError is raised."""
    monkeypatch.delenv("USER", raising=False)

    with pytest.raises(OSError):
        _ = get_os_user_lower()
```

This behavior can be moved into `fixture` structures and shared across tests:

```python
# contents of our test file e.g. test_code.py
import pytest


@pytest.fixture
def mock_env_user(monkeypatch):
    monkeypatch.setenv("USER", "TestingUser")


@pytest.fixture
def mock_env_missing(monkeypatch):
    monkeypatch.delenv("USER", raising=False)
```

(continues on next page)

```python
# notice the tests reference the fixtures for mocks
def test_upper_to_lower(mock_env_user):
    assert get_os_user_lower() == "testinguser"


def test_raise_exception(mock_env_missing):
    with pytest.raises(OSError):
        _ = get_os_user_lower()
```

7.5 Monkeypatching dictionaries

`monkeypatch.setitem` can be used to safely set the values of dictionaries to specific values during tests. Take this simplified connection string example:

```python
# contents of app.py to generate a simple connection string
DEFAULT_CONFIG = {"user": "user1", "database": "db1"}


def create_connection_string(config=None):
    """Creates a connection string from input or defaults."""
    config = config or DEFAULT_CONFIG
    return f"User Id={config['user']}; Location={config['database']};"
```

For testing purposes we can patch the `DEFAULT_CONFIG` dictionary to specific values.

```python
# contents of test_app.py
# app.py with the connection string function (prior code block)
import app


def test_connection(monkeypatch):

    # Patch the values of DEFAULT_CONFIG to specific
    # testing values only for this test.
    monkeypatch.setitem(app.DEFAULT_CONFIG, "user", "test_user")
    monkeypatch.setitem(app.DEFAULT_CONFIG, "database", "test_db")

    # expected result based on the mocks
    expected = "User Id=test_user; Location=test_db;"

    # the test uses the monkeypatched dictionary settings
    result = app.create_connection_string()
    assert result == expected
```

You can use the `monkeypatch.delitem` to remove values.

```python
# contents of test_app.py
import pytest

# app.py with the connection string function
import app


def test_missing_user(monkeypatch):
```

```python
    # patch the DEFAULT_CONFIG t be missing the 'user' key
    monkeypatch.delitem(app.DEFAULT_CONFIG, "user", raising=False)

    # Key error expected because a config is not passed, and the
    # default is now missing the 'user' entry.
    with pytest.raises(KeyError):
        _ = app.create_connection_string()
```

The modularity of fixtures gives you the flexibility to define separate fixtures for each potential mock and reference them in the needed tests.

```python
# contents of test_app.py
import pytest

# app.py with the connection string function
import app

# all of the mocks are moved into separated fixtures
@pytest.fixture
def mock_test_user(monkeypatch):
    """Set the DEFAULT_CONFIG user to test_user."""
    monkeypatch.setitem(app.DEFAULT_CONFIG, "user", "test_user")


@pytest.fixture
def mock_test_database(monkeypatch):
    """Set the DEFAULT_CONFIG database to test_db."""
    monkeypatch.setitem(app.DEFAULT_CONFIG, "database", "test_db")


@pytest.fixture
def mock_missing_default_user(monkeypatch):
    """Remove the user key from DEFAULT_CONFIG"""
    monkeypatch.delitem(app.DEFAULT_CONFIG, "user", raising=False)


# tests reference only the fixture mocks that are needed
def test_connection(mock_test_user, mock_test_database):

    expected = "User Id=test_user; Location=test_db;"

    result = app.create_connection_string()
    assert result == expected


def test_missing_user(mock_missing_default_user):

    with pytest.raises(KeyError):
        _ = app.create_connection_string()
```

7.6 API Reference

Consult the docs for the `MonkeyPatch` class.

EIGHT

TEMPORARY DIRECTORIES AND FILES

8.1 The `tmp_path` fixture

You can use the `tmp_path` fixture which will provide a temporary directory unique to the test invocation, created in the *base temporary directory*.

`tmp_path` is a `pathlib.Path` object. Here is an example test usage:

```python
# content of test_tmp_path.py
CONTENT = "content"


def test_create_file(tmp_path):
    d = tmp_path / "sub"
    d.mkdir()
    p = d / "hello.txt"
    p.write_text(CONTENT)
    assert p.read_text() == CONTENT
    assert len(list(tmp_path.iterdir())) == 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_tmp_path.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_tmp_path.py F                                                   [100%]

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

tmp_path = PosixPath('PYTEST_TMPDIR/test_create_file0')

    def test_create_file(tmp_path):
        d = tmp_path / "sub"
        d.mkdir()
        p = d / "hello.txt"
        p.write_text(CONTENT)
        assert p.read_text() == CONTENT
        assert len(list(tmp_path.iterdir())) == 1
```

```
>       assert 0
E       assert 0

test_tmp_path.py:11: AssertionError
========================= short test summary info ==========================
FAILED test_tmp_path.py::test_create_file - assert 0
========================== 1 failed in 0.12s ===============================
```

8.2 The `tmp_path_factory` fixture

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

It is intended to replace `tmpdir_factory`, and returns `pathlib.Path` instances.

See *tmp_path_factory API* for details.

8.3 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:

```python
# content of test_tmpdir.py
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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_tmpdir.py F                                                  [100%]

================================ 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:6: AssertionError
========================= short test summary info ==========================
FAILED test_tmpdir.py::test_create_file - assert 0
========================== 1 failed in 0.12s ===============================
```

8.4 The 'tmpdir_factory' fixture

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:

```python
# 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
```

See *tmpdir_factory API* for details.

8.5 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
```

> **Warning:** The contents of `mydir` will be completely removed, so make sure to use a directory for that purpose only.

When distributing tests on the local machine using `pytest-xdist`, care is taken to automatically configure a basetemp directory for the sub processes such that all temporary data lands below a single per-test run basetemp directory.

CAPTURING OF THE STDOUT/STDERR OUTPUT

9.1 Default stdout/stderr/stdin capturing behaviour

During test execution any output sent to `stdout` and `stderr` is captured. If a test or a setup method fails its according captured output will usually be shown along with the failure traceback. (this behavior can be configured by the `--show-capture` command-line option).

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.

9.2 Setting capturing methods or disabling capturing

There are three ways in which `pytest` can perform capturing:

- `fd` (file descriptor) 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.

- `tee-sys` capturing: Python writes to `sys.stdout` and `sys.stderr` will be captured, however the writes will also be passed-through to the actual `sys.stdout` and `sys.stderr`. This allows output to be 'live printed' and captured for plugin use, such as junitxml (new in pytest 5.4).

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
pytest --capture=tee-sys     # combines 'sys' and '-s', capturing sys.stdout/stderr
                             # and passing it along to the actual sys.stdout/stderr
```

9.3 Using print statements for debugging

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

```python
# content of test_module.py


def setup_function(function):
    print("setting up", 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py .F                                                    [100%]

================================= FAILURES =================================
_____ test_func2 _____

    def test_func2():
>       assert False
E       assert False

test_module.py:12: AssertionError
-------------------------- Captured stdout setup ---------------------------
setting up <function test_func2 at 0xdeadbeef>
========================= short test summary info ==========================
FAILED test_module.py::test_func2 - assert False
======================= 1 failed, 1 passed in 0.12s ========================
```

9.4 Accessing captured output from a test function

The `capsys`, `capsysbinary`, `capfd`, and `capfdbinary` fixtures allow access to stdout/stderr output created during test execution. Here is an example test function that performs some output related checks:

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

(continues on next page)

```
    print("next")
    captured = capsys.readouterr()
    assert captured.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` fixture 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).

The return value from `readouterr` changed to a `namedtuple` with two attributes, `out` and `err`.

If the code under test writes non-textual data, you can capture this using the `capsysbinary` fixture which instead returns `bytes` from the `readouterr` method.

If the code under test writes non-textual data, you can capture this using the `capfdbinary` fixture which instead returns `bytes` from the `readouterr` method. The `capfdbinary` fixture operates on the filedescriptor level.

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:

```python
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")
```

TEN

WARNINGS CAPTURE

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

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_show_warnings.py .                                              [100%]

============================= warnings summary =============================
test_show_warnings.py::test_one
  $REGENDOC_TMPDIR/test_show_warnings.py:5: UserWarning: api v1, should use functions␣
→from v2
    warnings.warn(UserWarning("api v1, should use functions from v2"))

-- Docs: https://docs.pytest.org/en/stable/warnings.html
======================= 1 passed, 1 warning in 0.12s =======================
```

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                                                                    [100%]
================================= FAILURES =================================
_____ test_one _____

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

```
test_show_warnings.py:10:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    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:5: UserWarning
========================= short test summary info ==========================
FAILED test_show_warnings.py::test_one - UserWarning: api v1, should use ...
1 failed in 0.12s
```

The same option can be set in the `pytest.ini` or `pyproject.toml` file using the `filterwarnings` ini option.
For example, the configuration below will ignore all user warnings and specific deprecation warnings matching a regex,
but will transform all other warnings into errors.

```
# pytest.ini
[pytest]
filterwarnings =
    error
    ignore::UserWarning
    ignore:function ham\(\) is deprecated:DeprecationWarning
```

```
# pyproject.toml
[tool.pytest.ini_options]
filterwarnings = [
    "error",
    "ignore::UserWarning",
    # note the use of single quote below to denote "raw" strings in TOML
    'ignore:function ham\(\) is deprecated:DeprecationWarning',
]
```

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.

10.1 `@pytest.mark.filterwarnings`

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:

```python
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")
```

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

10.2 Disabling warnings summary

Although not recommended, you can use the `--disable-warnings` command-line option to suppress the warning summary entirely from the test run output.

10.3 Disabling warning capture entirely

This plugin 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. This might be useful if your test suites handles warnings using an external system.

10.4 DeprecationWarning and PendingDeprecationWarning

By default pytest will display `DeprecationWarning` and `PendingDeprecationWarning` warnings from user code and third-party libraries, as recommended by PEP-0565. This helps users keep their code modern and avoid breakages when deprecated warnings are effectively removed.

Sometimes it is useful to hide some specific deprecation warnings that happen in code that you have no control over (such as third-party libraries), in which case you might use the warning filters options (ini or marks) to ignore those warnings.

For example:

```
[pytest]
filterwarnings =
    ignore:.*U.*mode is deprecated:DeprecationWarning
```

This will ignore all warnings of type `DeprecationWarning` where the start of the message matches the regular expression `".*U.*mode is deprecated"`.

Note: If warnings are configured at the interpreter level, using the PYTHONWARNINGS environment variable or the `-W` command-line option, pytest will not configure any filters by default.

Also pytest doesn't follow `PEP-0506` suggestion of resetting all warning filters because it might break test suites that configure warning filters themselves by calling `warnings.simplefilter` (see issue #2430 for an example of that).

10.5 Ensuring code triggers a deprecation warning

You can also use *pytest.deprecated_call()* for checking that a certain function call triggers a DeprecationWarning or PendingDeprecationWarning:

```python
import pytest


def test_myfunction_deprecated():
    with pytest.deprecated_call():
        myfunction(17)
```

This test will fail if myfunction does not issue a deprecation warning when called with a 17 argument.

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

```python
import warnings
import pytest


def test_deprecation(recwarn):
    warnings.simplefilter("always")
    myfunction(17)
    assert len(recwarn) == 1
    assert recwarn.pop(DeprecationWarning)
```

The *recwarn* fixture automatically ensures to reset the warnings filter at the end of the test, so no global state is leaked.

10.6 Asserting warnings with the warns function

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

```python
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. The keyword argument match to assert that the exception matches a text or regex:

```python
>>> with warns(UserWarning, match='must be 0 or None'):
...     warnings.warn("value must be 0 or None", UserWarning)

>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("value must be 42", UserWarning)

>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("this is not here", UserWarning)
Traceback (most recent call last):
```

(continues on next page)

```
    ...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

You can also call func:`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 code triggers a deprecation warning*.

10.7 Recording warnings

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

To record with func:`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 func:`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.

Full API: `WarningsRecorder`.

10.8 Custom failure messages

Recording warnings provides an opportunity to produce custom test failure messages for when no warnings are issued or other conditions are met.

```python
def test():
    with pytest.warns(Warning) as record:
        f()
        if not record:
            pytest.fail("Expected a warning!")
```

If no warnings are issued when calling `f`, then `not record` will evaluate to `True`. You can then call `pytest.fail()` with a custom error message.

10.9 Internal pytest warnings

pytest may generate its own warnings in some situations, such as improper usage or deprecated features.

For example, pytest will emit a warning if it encounters a class that matches `python_classes` but also defines an `__init__` constructor, as this prevents the class from being instantiated:

```python
# content of test_pytest_warnings.py
class Test:
    def __init__(self):
        pass

    def test_foo(self):
        assert 1 == 1
```

```
$ pytest test_pytest_warnings.py -q

=========================== warnings summary ===========================
test_pytest_warnings.py:1
  $REGENDOC_TMPDIR/test_pytest_warnings.py:1: PytestCollectionWarning: cannot collect␣
→test class 'Test' because it has a __init__ constructor (from: test_pytest_warnings.
→py)
    class Test:

-- Docs: https://docs.pytest.org/en/stable/warnings.html
1 warning in 0.12s
```

These warnings might be filtered using the same builtin mechanisms used to filter other types of warnings.

Please read our *Backwards Compatibility Policy* to learn how we proceed about deprecating and eventually removing features.

The full list of warnings is listed in *the reference documentation*.

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. `--doctest-glob` can be given multiple times in the command-line.

If you then have a text file like this:

```
# content of test_example.txt

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

then you can just invoke `pytest` directly:

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_example.txt .                                                   [100%]

============================ 1 passed in 0.12s =============================
```

By default, pytest will collect `test*.txt` files looking for doctest directives, but you can pass additional globs using the `--doctest-glob` option (multi-allowed).

In addition to text files, you can also execute doctests directly from docstrings of your classes and functions, including from test modules:

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

```
$ pytest --doctest-modules
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

mymodule.py .                                                        [ 50%]
test_example.txt .                                                   [100%]


============================ 2 passed in 0.12s =============================
```

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
```

11.1 Encoding

The default encoding is **UTF-8**, but 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
```

11.2 Using 'doctest' options

Python's standard `doctest` module provides some options to configure the strictness of doctest tests. In pytest, you can enable those flags using the configuration file.

For example, to make pytest ignore trailing whitespaces and ignore lengthy exception stack traces you can just write:

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

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

```
>>> something_that_raises()  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
ValueError: ...
```

pytest also introduces new options:

- `ALLOW_UNICODE`: when enabled, the `u` prefix is stripped from unicode strings in expected doctest output. This allows doctests to run in Python 2 and Python 3 unchanged.

- `ALLOW_BYTES`: similarly, the `b` prefix is stripped from byte strings in expected doctest output.

- `NUMBER`: when enabled, floating-point numbers only need to match as far as the precision you have written in the expected doctest output. For example, the following output would only need to match to 2 decimal places:

```
>>> math.pi
3.14
```

If you wrote `3.1416` then the actual output would need to match to 4 decimal places; and so on.

This avoids false positives caused by limited floating-point precision, like this:

```
Expected:
    0.233
Got:
    0.23300000000000001
```

`NUMBER` also supports lists of floating-point numbers – in fact, it matches floating-point numbers appearing anywhere in the output, even inside a string! This means that it may not be appropriate to enable globally in `doctest_optionflags` in your configuration file.

New in version 5.1.

11.3 Continue on failure

By default, pytest would report only the first failure for a given doctest. If you want to continue the test even when you have failures, do:

```
pytest --doctest-modules --doctest-continue-on-failure
```

11.4 Output format

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
```

11.5 pytest-specific features

Some features are provided to make writing doctests easier or with better integration with your existing test suite. Keep in mind however that by using those features you will make your doctests incompatible with the standard `doctests` module.

11.5.1 Using fixtures

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

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

Note that the fixture needs to be defined in a place visible by pytest, for example, a `conftest.py` file or plugin; normal python files containing docstrings are not normally scanned for fixtures unless explicitly configured by *python_files*.

Also, the *usefixtures* mark and fixtures marked as *autouse* are supported when executing text doctest files.

11.5.2 'doctest_namespace' fixture

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:

```python
# 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:

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

Note that like the normal `conftest.py`, the fixtures are discovered in the directory tree conftest is in. Meaning that if you put your doctest with your source code, the relevant conftest.py needs to be in the same directory tree. Fixtures will not be discovered in a sibling directory tree!

11.5.3 Skipping tests

For the same reasons one might want to skip normal tests, it is also possible to skip tests inside doctests.

To skip a single check inside a doctest you can use the standard doctest.SKIP directive:

```python
def test_random(y):
    """
    >>> random.random()  # doctest: +SKIP
    0.156231223

    >>> 1 + 1
    2
    """
```

This will skip the first check, but not the second.

pytest also allows using the standard pytest functions `pytest.skip()` and `pytest.xfail()` inside doctests, which might be useful because you can then skip/xfail tests based on external conditions:

```python
>>> import sys, pytest
>>> if sys.platform.startswith('win'):
...     pytest.skip('this doctest does not work on Windows')
...
>>> import fcntl
>>> ...
```

However using those functions is discouraged because it reduces the readability of the docstring.

Note: `pytest.skip()` and `pytest.xfail()` behave differently depending if the doctests are in a Python file (in docstrings) or a text file containing doctests intermingled with text:

- Python modules (docstrings): the functions only act in that specific docstring, letting the other docstrings in the same module execute as normal.

- Text files: the functions will skip/xfail the checks for the rest of the entire file.

11.6 Alternatives

While the built-in pytest support provides a good set of functionalities for using doctests, if you use them extensively you might be interested in those external packages which add many more features, and include pytest integration:

- pytest-doctestplus: provides advanced doctest support and enables the testing of reStructuredText (".rst") files.

- Sybil: provides a way to test examples in your documentation by parsing them from the documentation source and evaluating the parsed examples as part of your normal test run.

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).

An **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. When a test passes despite being expected to fail (marked with `pytest.mark.xfail`), it's an **xpass** and will be reported in the test summary.

`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 -rxXs  # show extra info on xfailed, xpassed, and skipped tests
```

More details on the `-r` option can be found by running `pytest -h`.

(See *Builtin configuration file options*)

12.1 Skipping test functions

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

```python
@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:

```python
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.

It is also possible to skip the whole module using `pytest.skip(reason, allow_module_level=True)` at the module level:

```python
import sys
import pytest

if not sys.platform.startswith("win"):
    pytest.skip("skipping windows-only tests", allow_module_level=True)
```

Reference: *pytest.mark.skip*

12.1.1 `skipif`

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 an interpreter earlier than Python3.6:

```python
import sys


@pytest.mark.skipif(sys.version_info < (3, 7), reason="requires python3.7 or higher")
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:

```python
# 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:

```python
# 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.

Reference: *pytest.mark.skipif*

12.1.2 Skip all test functions of a class or module

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

```python
@pytest.mark.skipif(sys.platform == "win32", reason="does not run on windows")
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

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

If you want to skip all test functions of a module, you may use the *pytestmark* global:

```python
# 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.

12.1.3 Skipping files or directories

Sometimes you may need to skip an entire file or directory, for example if the tests rely on Python version-specific features or contain code that you do not wish pytest to run. In this case, you must exclude the files and directories from collection. Refer to *Customizing test collection* for more information.

12.1.4 Skipping on a missing import dependency

You can skip tests on a missing import by using *pytest.importorskip* at module level, within a test, or test setup function.

```python
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:

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

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

12.1.5 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:

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

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

   ```python
   pytestmark = pytest.mark.skipif(sys.platform == "win32", reason="tests for
   ↪linux only")
   ```

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

   ```python
   pexpect = pytest.importorskip("pexpect")
   ```

12.2 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 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 the test or its setup function imperatively:

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

```
def test_function2():
    import slow_module

    if slow_module.slow_function():
        pytest.xfail("slow_module taking too long")
```

These two examples illustrate situations where you don't want to check for a condition at the module level, which is when a condition would otherwise be evaluated for marks.

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

Reference: *pytest.mark.xfail*

12.2.1 `condition` parameter

If a test is only expected to fail under a certain condition, you can pass that condition as the first parameter:

```
@pytest.mark.xfail(sys.platform == "win32", reason="bug in a 3rd party library")
def test_function():
    ...
```

Note that you have to pass a reason as well (see the parameter description at *pytest.mark.xfail*).

12.2.2 `reason` parameter

You can specify the motive of an expected failure with the `reason` parameter:

```
@pytest.mark.xfail(reason="known parser issue")
def test_function():
    ...
```

12.2.3 `raises` parameter

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

```python
@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`.

12.2.4 `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`:

```python
@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.

12.2.5 `strict` parameter

Both `XFAIL` and `XPASS` don't fail the test suite by default. You can change this by setting the `strict` keyword-only parameter to `True`:

```python
@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:

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

12.2.6 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.

12.2.7 Examples

Here is a simple test file with the several usages:

```python
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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/example
collected 7 items

xfail_demo.py xxxxxxx                                                  [100%]

========================= short test summary info =========================
XFAIL xfail_demo.py::test_hello
XFAIL xfail_demo.py::test_hello2
  reason: [NOTRUN]
```

(continues on next page)

```
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 0.12s ============================
```

12.3 Skip/xfail with parametrize

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

```python
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
```

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.

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

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:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 3 items

test_expectation.py ..F                                                  [100%]

================================= 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):
```

(continues on next page)

```
>       assert eval(test_input) == expected
E       AssertionError: assert 54 == 42
E        +  where 54 = eval('6*9')

test_expectation.py:6: AssertionError
========================= short test summary info =========================
FAILED test_expectation.py::test_eval[6*9-42] - AssertionError: assert 54...
======================= 1 failed, 2 passed in 0.12s =======================
```

Note: Parameter values are passed as-is to tests (no copy whatsoever).

For example, if you pass a list or a dict as a parameter value, and the test case code mutates it, the mutations will be reflected in subsequent test case calls.

Note: pytest by default escapes any non-ascii characters used in unicode strings for the parametrization because it has several downsides. If however you would like to use unicode strings in parametrization and see them in the terminal as is (non-escaped), use this option in your `pytest.ini`:

```ini
[pytest]
disable_test_id_escaping_and_forfeit_all_rights_to_community_support = True
```

Keep in mind however that this might cause unwanted side effects and even bugs depending on the OS used and plugins currently installed, so use it at your own risk.

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`:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 3 items

test_expectation.py ..x                                              [100%]
```

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

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

In case the values provided to parametrize result in an empty list - for example, if they're dynamically generated by some function - the behaviour of pytest is defined by the *empty_parameter_set_mark* option.

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

```python
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=1/y=2, x=0/y=3, and x=1/y=3 exhausting parameters in the order of the decorators.

13.2 Basic `pytest_generate_tests` example

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

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

```python
# 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:

```python
# 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
..                                                                      [100%]
2 passed in 0.12s
```

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

```
$ pytest -q --stringinput="!" test_strings.py
F                                                                       [100%]
================================ 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:4: AssertionError
========================= short test summary info =========================
FAILED test_strings.py::test_valid_string[!] - AssertionError: assert False
1 failed in 0.12s
```

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                                                                       [100%]
========================= short test summary info =========================
SKIPPED [1] test_strings.py: got empty parameter set ['stringinput'], function test_
↪valid_string at $REGENDOC_TMPDIR/test_strings.py:2
1 skipped in 0.12s
```

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.

13.3 More examples

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

CACHE: WORKING WITH CROSS-TESTRUN STATE

14.1 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`).

14.2 Rerunning only failures or failures first

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

```python
# 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.......................                         [100%]
================================= 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:7: 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:7: Failed
========================= short test summary info =========================
FAILED test_50.py::test_num[17] - Failed: bad luck
FAILED test_50.py::test_num[25] - Failed: bad luck
2 failed, 48 passed in 0.12s
```

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

```
$ pytest --lf
========================= test session starts =========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items
run-last-failure: rerun previous 2 failures

test_50.py FF                                                   [100%]

=============================== 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:7: 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:7: Failed
========================= short test summary info =========================
```

```
FAILED test_50.py::test_num[17] - Failed: bad luck
FAILED test_50.py::test_num[25] - Failed: bad luck
=========================== 2 failed in 0.12s ===============================
```

You have run only the two failing tests from the last run, while the 48 passing 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 50 items
run-last-failure: rerun previous 2 failures first

test_50.py FF..............................................     [100%]

================================ 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:7: 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:7: Failed
======================= short test summary info ==========================
FAILED test_50.py::test_num[17] - Failed: bad luck
FAILED test_50.py::test_num[25] - Failed: bad luck
===================== 2 failed, 48 passed in 0.12s =======================
```

New `--nf`, `--new-first` options: run new tests first followed by the rest of the tests, in both cases tests are also sorted by the file modified time, with more recent files coming first.

14.2. Rerunning only failures or failures first 117

14.3 Behavior when no tests failed in the last run

When no tests failed in the last run, or when no cached `lastfailed` data was found, `pytest` can be configured either to run all of the tests or no tests, using the `--last-failed-no-failures` option, which takes one of the following values:

```
pytest --last-failed --last-failed-no-failures all    # run all tests (default
↪behavior)
pytest --last-failed --last-failed-no-failures none   # run no tests and exit
```

14.4 The new config.cache object

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

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


def expensive_computation():
    print("running expensive computation...")


@pytest.fixture
def mydata(request):
    val = request.config.cache.get("example/value", None)
    if val is None:
        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 for the first time, you can see the print statement:

```
$ pytest -q
F                                                                        [100%]
================================= FAILURES =================================
_____ test_function _____

mydata = 42

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

test_caching.py:20: AssertionError
-------------------------- Captured stdout setup --------------------------
running expensive computation...
========================= short test summary info =========================
```

```
FAILED test_caching.py::test_function - assert 42 == 23
1 failed in 0.12s
```

If you run it a second time, the value will be retrieved from the cache and nothing will be printed:

```
$ pytest -q
F                                                                    [100%]
================================= FAILURES =================================
_____ test_function _____

mydata = 42

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

test_caching.py:20: AssertionError
========================= short test summary info =========================
FAILED test_caching.py::test_function - assert 42 == 23
1 failed in 0.12s
```

See the `config.cache fixture` for more details.

14.5 Inspecting Cache content

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

```
$ pytest --cache-show
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
cachedir: $PYTHON_PREFIX/.pytest_cache
-------------------------- cache values for '*' --------------------------
cache/lastfailed contains:
  {'test_50.py::test_num[17]': True,
   'test_50.py::test_num[25]': True,
   'test_assert1.py::test_function': True,
   'test_assert2.py::test_set_comparison': True,
   'test_caching.py::test_function': True,
   'test_foocompare.py::test_compare': True}
cache/nodeids contains:
  ['test_50.py::test_num[0]',
   'test_50.py::test_num[10]',
   'test_50.py::test_num[11]',
   'test_50.py::test_num[12]',
   'test_50.py::test_num[13]',
   'test_50.py::test_num[14]',
   'test_50.py::test_num[15]',
   'test_50.py::test_num[16]',
   'test_50.py::test_num[17]',
   'test_50.py::test_num[18]',
   'test_50.py::test_num[19]',
   'test_50.py::test_num[1]',
```

```
      'test_50.py::test_num[20]',
      'test_50.py::test_num[21]',
      'test_50.py::test_num[22]',
      'test_50.py::test_num[23]',
      'test_50.py::test_num[24]',
      'test_50.py::test_num[25]',
      'test_50.py::test_num[26]',
      'test_50.py::test_num[27]',
      'test_50.py::test_num[28]',
      'test_50.py::test_num[29]',
      'test_50.py::test_num[2]',
      'test_50.py::test_num[30]',
      'test_50.py::test_num[31]',
      'test_50.py::test_num[32]',
      'test_50.py::test_num[33]',
      'test_50.py::test_num[34]',
      'test_50.py::test_num[35]',
      'test_50.py::test_num[36]',
      'test_50.py::test_num[37]',
      'test_50.py::test_num[38]',
      'test_50.py::test_num[39]',
      'test_50.py::test_num[3]',
      'test_50.py::test_num[40]',
      'test_50.py::test_num[41]',
      'test_50.py::test_num[42]',
      'test_50.py::test_num[43]',
      'test_50.py::test_num[44]',
      'test_50.py::test_num[45]',
      'test_50.py::test_num[46]',
      'test_50.py::test_num[47]',
      'test_50.py::test_num[48]',
      'test_50.py::test_num[49]',
      'test_50.py::test_num[4]',
      'test_50.py::test_num[5]',
      'test_50.py::test_num[6]',
      'test_50.py::test_num[7]',
      'test_50.py::test_num[8]',
      'test_50.py::test_num[9]',
      'test_assert1.py::test_function',
      'test_assert2.py::test_set_comparison',
      'test_caching.py::test_function',
      'test_foocompare.py::test_compare']
cache/stepwise contains:
  []
example/value contains:
  42


========================= no tests ran in 0.12s ==========================
```

`--cache-show` takes an optional argument to specify a glob pattern for filtering:

```
$ pytest --cache-show example/*
========================= test session starts =========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
cachedir: $PYTHON_PREFIX/.pytest_cache
```

```
---------------------- cache values for 'example/*' ----------------------
example/value contains:
  42


========================= no tests ran in 0.12s =========================
```

14.6 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.

14.7 Stepwise

As an alternative to `--lf -x`, especially for cases where you expect a large part of the test suite will fail, `--sw`, `--stepwise` allows you to fix them one at a time. The test suite will run until the first failure and then stop. At the next invocation, tests will continue from the last failing test and then run until the next failing test. You may use the `--stepwise-skip` option to ignore one failing test and stop the test execution on the second failing test instead. This is useful if you get stuck on a failing test and just want to ignore it until later.

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`;

- `setUpModule/tearDownModule`;

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

- load_tests protocol;

- subtests;

15.1 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);

15.2 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.

15.3 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:

```python
# content of conftest.py

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

import pytest


@pytest.fixture(scope="class")
def db_class(request):
    class DummyDB:
        pass

    # set a class attribute on the invoking test context
    request.cls.db = DummyDB()
```

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

```python
# content of test_unittest_db.py

import unittest
import pytest


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

(continues on next page)

```python
        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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_unittest_db.py FF                                                [100%]

================================= 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:10: 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:13: AssertionError
========================= short test summary info ==========================
FAILED test_unittest_db.py::MyTest::test_method1 - AssertionError: <conft...
FAILED test_unittest_db.py::MyTest::test_method2 - AssertionError: <conft...
========================== 2 failed in 0.12s ===========================
```

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.

15.4 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:

```python
# 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
.                                                                        [100%]
1 passed in 0.12s
```

. . . 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: Due to architectural differences between the two frameworks, setup and teardown for `unittest`-based tests is performed during the `call` phase of testing instead of in `pytest`'s standard `setup` and `teardown` stages. This can be important to understand in some situations, particularly when reasoning about errors. For example, if a `unittest`-based suite exhibits errors during setup, `pytest` will report no errors during its `setup` phase and will instead raise the error during `call`.

RUNNING TESTS WRITTEN FOR NOSE

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

16.1 Usage

After *Install pytest* 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.

16.2 Supported nose Idioms

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

16.3 Unsupported idioms / known issues

- unittest-style `setUp`, `tearDown`, `setUpClass`, `tearDownClass` are recognized only on `unittest.TestCase` classes but not on plain classes. `nose` supports these methods also on plain classes but pytest deliberately does not. As nose and pytest already both support `setup_class`, `teardown_class`, `setup_method`, `teardown_method` it doesn't seem useful to duplicate the unittest-API like nose does. If you however rather think pytest should support the unittest-spelling on plain classes please post to this issue.
- nose imports test modules with the same import path (e.g. `tests.test_mode`) 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 are unsupported as of pytest 4.1.0. They are fundamentally incompatible with pytest because they don't support fixtures properly since collection and test execution are separated.

16.4 Migrating from nose to pytest

nose2pytest is a Python script and pytest plugin to help convert Nose-based tests into pytest-based tests. Specifically, the script transforms nose.tools.assert_* function calls into raw assert statements, while preserving format of original arguments as much as possible.

128   **Chapter 16. Running tests written for nose**

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.

17.1 Module level setup/teardown

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

```python
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.

17.2 Class level setup/teardown

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

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


@classmethod
```

(continues on next page)

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

17.3 Method and function level setup/teardown

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

```python
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:

```python
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.

- Prior to pytest-4.2, xunit-style functions did not obey the scope rules of fixtures, so it was possible, for example, for a `setup_method` to be called before a session-scoped autouse fixture.

  Now the xunit-style functions are integrated with the fixture mechanism and obey the proper scope rules of fixtures involved in the call.

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-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: 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.org search.

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

You can require plugins in a test module or a conftest file using *pytest_plugins*:

```
pytest_plugins = ("myapp.testsupport.myplugin",)
```

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

Note: Requiring plugins using a `pytest_plugins` variable in non-root `conftest.py` files is deprecated. See *full explanation* in the Writing plugins section.

Note: The name `pytest_plugins` is reserved and should not be used as a name for a custom plugin module.

18.2 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.

18.3 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.

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:

- builtin plugins: loaded from pytest's internal `_pytest` directory.

- *external plugins*: modules discovered through *setuptools entry points*

- *conftest.py plugins*: modules auto-discovered in test directories

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.

19.1 Plugin discovery order at tool startup

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

1. by scanning the command line for the `-p no:name` option and *blocking* that plugin from being loaded (even builtin plugins can be blocked this way). This happens before normal command-line parsing.

2. by loading all builtin plugins.

3. by scanning the command line for the `-p name` option and loading the specified plugin. This happens before normal command-line parsing.

4. by loading all plugins registered through *setuptools entry points*.

5. by loading all plugins specified through the *PYTEST_PLUGINS* environment variable.

6. by loading all `conftest.py` files as inferred by the command line invocation:

   - if no test paths are specified, use the 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. After the `conftest.py` file is loaded, load all plugins specified in its *pytest_plugins* variable if present.

   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.

7. by recursively loading all plugins specified by the *pytest_plugins* variable in `conftest.py` files.

19.2 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 --capture=no  # will not show "setting up"
pytest a/test_sub.py --capture=no  # 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*.

Note: Some hooks should be implemented only in plugins or conftest.py files situated at the tests root directory due to how pytest discovers plugins during startup, see the documentation of each hook for details.

19.3 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*
- builtin plugins which provide pytest's own functionality
- many external plugins providing additional features

All of these plugins implement *hooks* and/or *fixtures* 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.

19.4 Making your plugin installable by others

If you want to make your plugin externally available, you may define a so-called entry point for your distribution so that `pytest` finds your plugin module. Entry points are a feature that is provided by setuptools. 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:

```python
# 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 *hooks*.

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

19.5 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 rewriting when modules get imported. However, since we do not want to test different bytecode from what you will run in production, this hook only rewrites test modules themselves (as defined by the `python_files` configuration option), and any modules which are part of plugins. Any other imported module will not be rewritten 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 rewrite this module before it gets imported.

register_assert_rewrite(*names: str*) → None
> 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 rewritten. If the helper module also contains assert statements which need to be rewritten it needs to be marked as such, before it gets imported. This is easiest by marking it for rewriting 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:

```python
import pytest

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

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

You can require plugins in a test module or a `conftest.py` file using *pytest_plugins*:

```
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 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.

Note: Requiring plugins using *pytest_plugins* variable in non-root `conftest.py` files is deprecated.

This is important because `conftest.py` files implement per-directory hook implementations, but once a plugin is imported, it will affect the entire directory tree. In order to avoid confusion, defining *pytest_plugins* in any `conftest.py` file which is not located in the tests root directory is deprecated, and will raise a warning.

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 rewritten. 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.

19.7 Accessing another plugin by name

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

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

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

19.8 Registering custom markers

If your plugin uses any markers, you should register them so that they appear in pytest's help text and do not *cause spurious warnings*. For example, the following plugin would register `cool_marker` and `mark_with` for all users:

```python
def pytest_configure(config):
    config.addinivalue_line("markers", "cool_marker: this one is for cool tests.")
    config.addinivalue_line(
        "markers", "mark_with(arg, arg2): this marker takes arguments."
    )
```

19.9 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:

```python
# 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.

```python
import pytest


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

(continues on next page)

```python
@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.

```python
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)
```

Additionally it is possible to copy examples for an example folder before running pytest on it.

```ini
# content of pytest.ini
[pytest]
pytester_example_dir = .
```

```python
# content of test_example.py
```

```python
def test_plugin(testdir):
    testdir.copy_example("test_example.py")
    testdir.runpytest("-k", "test_example")


def test_example():
    pass
```

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, configfile: pytest.ini
collected 2 items

test_example.py ..                                                   [100%]

============================ 2 passed in 0.12s =============================
```

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

WRITING HOOK FUNCTIONS

20.1 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.

20.2 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.

20.3 hookwrapper: executing around other hooks

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 `Result` 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:

```python
import pytest


@pytest.hookimpl(hookwrapper=True)
def pytest_pyfunc_call(pyfuncitem):
    do_something_before_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

    post_process_result(res)

    outcome.force_result(new_res)  # to override the return value to the plugin system
```

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.

For more information, consult the pluggy documentation about hookwrappers.

20.4 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:

```python
# 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
```

```python
@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 `Result` 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.

20.5 Declaring new hooks

Note: This is a quick overview on how to add new hooks and how they work in general, but a more complete overview can be found in the pluggy documentation.

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: PytestPluginManager*) → None
    Called at plugin registration time to allow adding new hooks via a call to `pluginmanager.add_hookspecs(module_or_class, prefix)`.

> **Parameters pluginmanager** (`_pytest.config.PytestPluginManager`) – pytest plugin manager.

> **Note:** This hook is incompatible with `hookwrapper=True`.

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. The names of the functions must start with `pytest_` otherwise pytest won't recognize them.

Here's an example. Let's assume this code is in the `sample_hook.py` module.

```python
def pytest_my_hook(config):
    """
    Receives the pytest config and does things with it
    """
```

To register the hooks with pytest they need to be structured in their own module or class. This class or module can then be passed to the `pluginmanager` using the `pytest_addhooks` function (which itself is a hook exposed by pytest).

```
def pytest_addhooks(pluginmanager):
    """ This example assumes the hooks are grouped in the 'sample_hook' module. """
    from my_app.tests import sample_hook

    pluginmanager.add_hookspecs(sample_hook)
```

For a real world example, see newhooks.py from xdist.

Hooks may be called both from fixtures or from other hooks. In both cases, hooks are called through the `hook` object, available in the `config` object. Most hooks receive a `config` object directly, while fixtures may use the `pytestconfig` fixture which provides the same object.

```
@pytest.fixture()
def my_fixture(pytestconfig):
    # call the hook called "pytest_my_hook"
    # 'result' will be a list of return values from all registered functions.
    result = pytestconfig.hook.pytest_my_hook(config=pytestconfig)
```

Note: Hooks receive parameters using only keyword arguments.

Now your hook is ready to be used. To register a function at the hook, other plugins or users must now simply define the function `pytest_my_hook` with the correct signature in their `conftest.py`.

Example:

```
def pytest_my_hook(config):
    """
    Print all active hooks to the screen.
    """
    print(config.hook)
```

20.6 Using hooks in pytest_addoption

Occasionally, it is necessary to change the way in which command line options are defined by one plugin based on hooks in another plugin. For example, a plugin may expose a command line option for which another plugin needs to define the default value. The pluginmanager can be used to install and use hooks to accomplish this. The plugin would define and add the hooks and use pytest_addoption as follows:

```
# contents of hooks.py

# Use firstresult=True because we only want one plugin to define this
# default value
@hookspec(firstresult=True)
def pytest_config_file_default_value():
    """ Return the default value for the config file command line option. """


# contents of myplugin.py


def pytest_addhooks(pluginmanager):
    """ This example assumes the hooks are grouped in the 'hooks' module. """
    from . import hook
```

```
    pluginmanager.add_hookspecs(hook)


def pytest_addoption(parser, pluginmanager):
    default_value = pluginmanager.hook.pytest_config_file_default_value()
    parser.addoption(
        "--config-file",
        help="Config file to use, defaults to %(default)s",
        default=default_value,
    )
```

The conftest.py that is using myplugin would simply define the hook as follows:

```
def pytest_config_file_default_value():
    return "config.yaml"
```

20.7 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:
    """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.

LOGGING

pytest captures log messages of level `WARNING` or above automatically and displays them in their own section for each failed test in the same manner as captured stdout and stderr.

Running without options:

```
pytest
```

Shows failed tests like so:

```
--------------------- Captured stdlog call ---------------------
test_reporting.py    26 WARNING  text going to logger
--------------------- Captured stdout call ---------------------
text going to stdout
--------------------- Captured stderr call ---------------------
text going to stderr
=================== 2 failed in 0.02 seconds ===================
```

By default each captured log message shows the module, line number, log level and message.

If desired the log and date format can be specified to anything that the logging module supports by passing specific formatting options:

```
pytest --log-format="%(asctime)s %(levelname)s %(message)s" \
       --log-date-format="%Y-%m-%d %H:%M:%S"
```

Shows failed tests like so:

```
--------------------- Captured stdlog call ---------------------
2010-04-10 14:48:44 WARNING text going to logger
--------------------- Captured stdout call ---------------------
text going to stdout
--------------------- Captured stderr call ---------------------
text going to stderr
=================== 2 failed in 0.02 seconds ===================
```

These options can also be customized through `pytest.ini` file:

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
log_date_format = %Y-%m-%d %H:%M:%S
```

Further it is possible to disable reporting of captured content (stdout, stderr and logs) on failed tests completely with:

```
pytest --show-capture=no
```

21.1 caplog fixture

Inside tests it is possible to change the log level for the captured log messages. This is supported by the `caplog` fixture:

```python
def test_foo(caplog):
    caplog.set_level(logging.INFO)
    pass
```

By default the level is set on the root logger, however as a convenience it is also possible to set the log level of any logger:

```python
def test_foo(caplog):
    caplog.set_level(logging.CRITICAL, logger="root.baz")
    pass
```

The log levels set are restored automatically at the end of the test.

It is also possible to use a context manager to temporarily change the log level inside a `with` block:

```python
def test_bar(caplog):
    with caplog.at_level(logging.INFO):
        pass
```

Again, by default the level of the root logger is affected but the level of any logger can be changed instead with:

```python
def test_bar(caplog):
    with caplog.at_level(logging.CRITICAL, logger="root.baz"):
        pass
```

Lastly all the logs sent to the logger during the test run are made available on the fixture in the form of both the `logging.LogRecord` instances and the final log text. This is useful for when you want to assert on the contents of a message:

```python
def test_baz(caplog):
    func_under_test()
    for record in caplog.records:
        assert record.levelname != "CRITICAL"
    assert "wally" not in caplog.text
```

For all the available attributes of the log records see the `logging.LogRecord` class.

You can also resort to `record_tuples` if all you want to do is to ensure, that certain messages have been logged under a given logger name with a given severity and message:

```python
def test_foo(caplog):
    logging.getLogger().info("boo %s", "arg")

    assert caplog.record_tuples == [("root", logging.INFO, "boo arg")]
```

You can call `caplog.clear()` to reset the captured log records in a test:

```python
def test_something_with_clearing_records(caplog):
    some_method_that_creates_log_records()
    caplog.clear()
    your_test_method()
    assert ["Foo"] == [rec.message for rec in caplog.records]
```

The `caplog.records` attribute contains records from the current stage only, so inside the `setup` phase it contains only setup logs, same with the `call` and `teardown` phases.

To access logs from other stages, use the `caplog.get_records(when)` method. As an example, if you want to make sure that tests which use a certain fixture never log any warnings, you can inspect the records for the `setup` and `call` stages during teardown like so:

```python
@pytest.fixture
def window(caplog):
    window = create_window()
    yield window
    for when in ("setup", "call"):
        messages = [
            x.message for x in caplog.get_records(when) if x.levelno == logging.
→WARNING
        ]
        if messages:
            pytest.fail(
                "warning messages encountered during testing: {}".format(messages)
            )
```

The full API is available at `_pytest.logging.LogCaptureFixture`.

21.2 Live Logs

By setting the `log_cli` configuration option to `true`, pytest will output logging records as they are emitted directly into the console.

You can specify the logging level for which log records with equal or higher level are printed to the console by passing `--log-cli-level`. This setting accepts the logging level names as seen in python's documentation or an integer as the logging level num.

Additionally, you can also specify `--log-cli-format` and `--log-cli-date-format` which mirror and default to `--log-format` and `--log-date-format` if not provided, but are applied only to the console logging handler.

All of the CLI log options can also be set in the configuration INI file. The option names are:

- `log_cli_level`
- `log_cli_format`
- `log_cli_date_format`

If you need to record the whole test suite logging calls to a file, you can pass `--log-file=/path/to/log/file`. This log file is opened in write mode which means that it will be overwritten at each run tests session.

You can also specify the logging level for the log file by passing `--log-file-level`. This setting accepts the logging level names as seen in python's documentation(ie, uppercased level names) or an integer as the logging level num.

Additionally, you can also specify `--log-file-format` and `--log-file-date-format` which are equal to `--log-format` and `--log-date-format` but are applied to the log file logging handler.

All of the log file options can also be set in the configuration INI file. The option names are:

- `log_file`
- `log_file_level`
- `log_file_format`

• log_file_date_format

You can call set_log_path() to customize the log_file path dynamically. This functionality is considered **experimental**.

21.3 Release notes

This feature was introduced as a drop-in replacement for the pytest-catchlog plugin and they conflict with each other. The backward compatibility API with pytest-capturelog has been dropped when this feature was introduced, so if for that reason you still need pytest-catchlog you can disable the internal feature by adding to your pytest.ini:

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

21.4 Incompatible changes in pytest 3.4

This feature was introduced in 3.3 and some **incompatible changes** have been made in 3.4 after community feedback:

- Log levels are no longer changed unless explicitly requested by the *log_level* configuration or --log-level command-line options. This allows users to configure logger objects themselves. Setting *log_level* will set the level that is captured globally so if a specific test requires a lower level than this, use the caplog.set_level() functionality otherwise that test will be prone to failure.

- *Live Logs* is now disabled by default and can be enabled setting the *log_cli* configuration option to true. When enabled, the verbosity is increased so logging for each test is visible.

- *Live Logs* are now sent to sys.stdout and no longer require the -s command-line option to work.

If you want to partially restore the logging behavior of version 3.3, you can add this options to your ini file:

```
[pytest]
log_cli=true
log_level=NOTSET
```

More details about the discussion that lead to this changes can be read in issue #3013.

TWENTYTWO

API REFERENCE

This page contains the full reference to pytest's API.

22.1 Functions

22.1.1 pytest.approx

approx(*expected*, *rel=None*, *abs=None*, *nan_ok:* *bool* *= False*) → _pytest.python_api.ApproxBase

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
```

`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
```

And for a `numpy` array against a scalar:

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

You can also use `approx` to compare nonnumeric types, or dicts and sequences containing nonnumeric types, in which case it falls back to strict equality. This can be useful for comparing dicts and sequences that can contain optional values:

```
>>> {"required": 1.0000005, "optional": None} == approx({"required": 1, "optional
↪": None})
True
>>> [None, 1.0000005] == approx([None,1])
True
>>> ["foo", 1.0000005] == approx([None,1])
False
```

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. 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...

Changed in version 3.7.1: `approx` raises `TypeError` when it encounters a dict value or sequence element of nonnumeric type.

Changed in version 6.1.0: `approx` falls back to strict equality for nonnumeric types instead of raising `TypeError`.

22.1.2 pytest.fail

Tutorial: *Skip and xfail: dealing with tests that cannot succeed*

fail (*msg: str = '', pytrace: bool = True*) → NoReturn
Explicitly fail an executing test with the given message.

>   **Parameters**
>
>   - **msg** (*str*) – The message to show the user as reason for the failure.
>
>   - **pytrace** (*bool*) – If False, msg represents the full failure information and no python traceback will be reported.

22.1.3 pytest.skip

skip (*msg*[, *allow_module_level=False*])
Skip an executing test with the given message.

This function should be called only during testing (setup, call or teardown) or during collection by using the `allow_module_level` flag. This function can be called in doctests as well.

>   **Parameters allow_module_level** (*bool*) – Allows this function to be called at module level, skipping the rest of the module. Defaults to False.

Note: It is better to use the *pytest.mark.skipif* marker when possible to declare a test to be skipped under certain conditions like mismatching platforms or dependencies. Similarly, use the `# doctest:  +SKIP` directive (see doctest.SKIP) to skip a doctest statically.

22.1.4 pytest.importorskip

importorskip (*modname: str*, *minversion: Optional[str] = None*, *reason: Optional[str] = None*) → Any
Import and return the requested module `modname`, or skip the current test if the module cannot be imported.

>   **Parameters**
>
>   - **modname** (*str*) – The name of the module to import.
>
>   - **minversion** (*str*) – If given, the imported module's __version__ attribute must be at least this minimal version, otherwise the test is still skipped.
>
>   - **reason** (*str*) – If given, this reason is shown as the message when the module cannot be imported.
>
>   **Returns** The imported module. This should be assigned to its canonical name.

Example:

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

22.1.5 pytest.xfail

xfail(*reason: str = ''*) → NoReturn
> Imperatively xfail an executing test or setup function with the given reason.
>
> This function should be called only during testing (setup, call or teardown).
>
> ---
>
> **Note:** It is better to use the *pytest.mark.xfail* marker when possible to declare a test to be xfailed under certain conditions like known bugs or missing features.
>
> ---

22.1.6 pytest.exit

exit(*msg: str*, *returncode: Optional[int] = None*) → NoReturn
> Exit testing process.
>
> **Parameters**
> - **msg** (`str`) – Message to display upon exit.
> - **returncode** (`int`) – Return code to be used when exiting pytest.

22.1.7 pytest.main

main(*args: Optional[Union[List[str], os.PathLike[str]]] = None*, *plugins: Optional[Sequence[Union[str, object]]] = None*) → Union[int, _pytest.config.ExitCode]
> Perform an in-process test run.
>
> **Parameters**
> - **args** – List of command line arguments.
> - **plugins** – List of plugin objects to be auto-registered during initialization.
>
> **Returns** An exit code.

22.1.8 pytest.param

param(**values*[, *id*][, *marks*])
> Specify a parameter in *pytest.mark.parametrize* calls or *parametrized fixtures*.

```python
@pytest.mark.parametrize(
    "test_input,expected",
    [
        ("3+5", 8),
        pytest.param("6*9", 42, marks=pytest.mark.xfail),
    ],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

> **Parameters**
> - **values** – Variable args of the values of the parameter set, in order.
> - **marks** – A single mark or a list of marks to be applied to this parameter set.

- **id** (*str*) – The id to attribute to this parameter set.

22.1.9 pytest.raises

Tutorial: *Assertions about expected exceptions*.

with raises(*expected_exception: Union[Type[_E], Tuple[Type[_E], ...]], *, match: Optional[Union[str, Pattern[str]]] = '...'*) → RaisesContext[_E] **as excinfo**

with raises(*expected_exception: Union[Type[_E], Tuple[Type[_E], ...]], func: Callable[[...], Any], *args: Any, **kwargs: Any*) → _pytest._code.code.ExceptionInfo[_E] **as excinfo**

Assert that a code block/function call raises `expected_exception` or raise a failure exception otherwise.

> **Parameters match** – If specified, a string containing a regular expression, or a regular expression object, that is tested against the string representation of the exception using `re.search`. To match a literal string that may contain special characters, the pattern can first be escaped with `re.escape`.
>
> (This is only used when `pytest.raises` is used as a context manager, and passed through to the function otherwise. When using `pytest.raises` as a function, you can use: `pytest.raises(Exc, func, match="passed on").match("my pattern")`.)

Use `pytest.raises` as a context manager, which will capture the exception of the given type:

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

If the code block does not raise the expected exception (`ZeroDivisionError` in the example above), or no exception at all, the check will fail instead.

You can also use the keyword argument `match` to assert that the exception matches a text or regex:

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

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

The context manager produces an *ExceptionInfo* object which can be used to inspect the details of the captured exception:

```
>>> with pytest.raises(ValueError) as exc_info:
...     raise ValueError("value must be 42")
>>> assert exc_info.type is ValueError
>>> assert exc_info.value.args[0] == "value must be 42"
```

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

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

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

Using with `pytest.mark.parametrize`

When using *pytest.mark.parametrize* it is possible to parametrize tests such that some runs raise an exception and others do not.

See *Parametrizing conditional raising* for an example.

Legacy form

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 ...>
```

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

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. More detailed information can be found in the official Python documentation for the try statement.

22.1.10 pytest.deprecated_call

Tutorial: *Ensuring code triggers a deprecation warning*.

with deprecated_call(*, *match: Optional[Union[str, Pattern[str]]] = '...'*) → WarningsRecorder
with deprecated_call(*func: Callable[[. . .], T]*, *args: Any*, **kwargs: Any*) → T
   Assert that code produces a `DeprecationWarning` or `PendingDeprecationWarning`.

   This function can be used as a context manager:

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

(continues on next page)

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

It 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. The return value is the return value of the function.

In the context manager form you may use the keyword argument `match` to assert that the warning matches a text or regex.

The context manager produces a list of `warnings.WarningMessage` objects, one for each warning raised.

22.1.11 pytest.register_assert_rewrite

Tutorial: *Assertion Rewriting*.

register_assert_rewrite(**names: str*) → None

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.

22.1.12 pytest.warns

Tutorial: *Asserting warnings with the warns function*

with warns(*expected_warning: Optional[Union[Type[Warning], Tuple[Type[Warning], . . .]]], *, match: Optional[Union[str, Pattern[str]]] = '...'*) → WarningsChecker

with warns(*expected_warning: Optional[Union[Type[Warning], Tuple[Type[Warning], . . .]]], func: Callable[[. . .], T], *args: Any, **kwargs: Any*) → T

Assert that code raises a particular class of warning.

Specifically, the parameter `expected_warning` can be a warning class or sequence of warning classes, and the inside the `with` block must issue a warning of that class or classes.

This helper produces a list of `warnings.WarningMessage` objects, one for each warning raised.

This function can be used as a context manager, or any of the other ways *pytest.raises()* can be used:

```
>>> import pytest
>>> with pytest.warns(RuntimeWarning):
...     warnings.warn("my warning", RuntimeWarning)
```

In the context manager form you may use the keyword argument `match` to assert that the warning matches a text or regex:

```
>>> with pytest.warns(UserWarning, match='must be 0 or None'):
...     warnings.warn("value must be 0 or None", UserWarning)

>>> with pytest.warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("value must be 42", UserWarning)
```

```
>>> with pytest.warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("this is not here", UserWarning)
Traceback (most recent call last):
  ...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

22.1.13 pytest.freeze_includes

Tutorial: *Freezing pytest*.

freeze_includes() → List[str]

Return a list of module names used by pytest that should be included by cx_freeze.

22.2 Marks

Marks can be used apply meta data to *test functions* (but not fixtures), which can then be accessed by fixtures or plugins.

22.2.1 pytest.mark.filterwarnings

Tutorial: *@pytest.mark.filterwarnings*.

Add warning filters to marked test items.

pytest.mark.**filterwarnings**(*filter*)

> **Parameters filter** (*str*) – A *warning specification string*, which is composed of contents of the tuple (action, message, category, module, lineno) as specified in The Warnings filter section of the Python documentation, separated by ":". Optional fields can be omitted. Module names passed for filtering are not regex-escaped.
>
> For example:

```
@pytest.mark.filterwarnings("ignore:.*usage will be deprecated.
↪*:DeprecationWarning")
def test_foo():
    ...
```

22.2.2 pytest.mark.parametrize

Tutorial: *Parametrizing fixtures and test functions*.

This mark has the same signature as *pytest.Metafunc.parametrize()*; see there.

22.2.3 pytest.mark.skip

Tutorial: *Skipping test functions*.

Unconditionally skip a test function.

pytest.mark.**skip**(*, *reason=None*)

> **Parameters** **reason** (*str*) – Reason why the test function is being skipped.

22.2.4 pytest.mark.skipif

Tutorial: *Skipping test functions*.

Skip a test function if a condition is `True`.

pytest.mark.**skipif**(*condition*, *, *reason=None*)

> **Parameters**
>
> - **condition** (*bool or str*) – True/False if the condition should be skipped or a *condition string*.
> - **reason** (*str*) – Reason why the test function is being skipped.

22.2.5 pytest.mark.usefixtures

Tutorial: *Use fixtures in classes and modules with usefixtures*.

Mark a test function as using the given fixture names.

pytest.mark.**usefixtures**(**names*)

> **Parameters** **args** – The names of the fixture to use, as strings.

Note: When using `usefixtures` in hooks, it can only load fixtures when applied to a test function before test setup (for example in the `pytest_collection_modifyitems` hook).

Also note that this mark has no effect when applied to **fixtures**.

22.2.6 pytest.mark.xfail

Tutorial: *XFail: mark test functions as expected to fail*.

Marks a test function as *expected to fail*.

pytest.mark.**xfail**(*condition=None*, *, *reason=None*, *raises=None*, *run=True*, *strict=False*)

> **Parameters**
>
> - **condition** (*bool or str*) – Condition for marking the test function as xfail (True/ False or a *condition string*). If a bool, you also have to specify reason (see *condition string*).
> - **reason** (*str*) – Reason why the test function is marked as xfail.
> - **raises** (*Type[Exception]*) – Exception subclass expected to be raised by the test function; other exceptions will fail the test.

- **run** (*bool*) – If the test function should actually be executed. If False, the function will always xfail and will not be executed (useful if a function is segfaulting).

- **strict** (*bool*) –

  - If False (the default) the function will be shown in the terminal output as xfailed if it fails and as xpass if it passes. In both cases this will not cause the test suite to fail as a whole. This is particularly useful to mark *flaky* tests (tests that fail at random) to be tackled later.

  - If True, the function will be shown in the terminal output as xfailed if it fails, but if it unexpectedly passes then it will **fail** the test suite. This is particularly useful to mark functions that are always failing and there should be a clear indication if they unexpectedly start to pass (for example a new release of a library fixes a known bug).

22.2.7 Custom marks

Marks are created dynamically using the factory object pytest.mark and applied as a decorator.

For example:

```
@pytest.mark.timeout(10, "slow", method="thread")
def test_function():
    ...
```

Will create and attach a *Mark* object to the collected *Item*, which can then be accessed by fixtures or hooks with *Node.iter_markers*. The mark object will have the following attributes:

```
mark.args == (10, "slow")
mark.kwargs == {"method": "thread"}
```

Example for using multiple custom markers:

```
@pytest.mark.timeout(10, "slow", method="thread")
@pytest.mark.slow
def test_function():
    ...
```

When *Node.iter_markers* or *Node.iter_markers* is used with multiple markers, the marker closest to the function will be iterated over first. The above example will result in @pytest.mark.slow followed by @pytest.mark.timeout(...).

22.3 Fixtures

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

Fixtures are requested by test functions or other fixtures by declaring them as argument names.

Example of a test requiring a fixture:

```
def test_output(capsys):
    print("hello")
    out, err = capsys.readouterr()
    assert out == "hello\n"
```

Example of a fixture requiring another fixture:

```
@pytest.fixture
def db_session(tmpdir):
    fn = tmpdir / "db.file"
    return connect(str(fn))
```

For more details, consult the full *fixtures docs*.

22.3.1 @pytest.fixture

@**fixture**(*fixture_function: _FixtureFunction, *, scope: Union[_Scope, Callable[[str, Config], _Scope]]
= '...', params: Optional[Iterable[object]] = '...', autouse: bool = '...', ids: Op-
tional[Union[Iterable[Union[None, str, float, int, bool]], Callable[[Any], Optional[object]]]] =
'...', name: Optional[str] = '...'*) → _FixtureFunction

@**fixture**(*fixture_function: None = '...', *, scope: Union[_Scope, Callable[[str, Config], _Scope]]
= '...', params: Optional[Iterable[object]] = '...', autouse: bool = '...', ids: Op-
tional[Union[Iterable[Union[None, str, float, int, bool]], Callable[[Any], Optional[object]]]] =
'...', name: Optional[str] = 'None'*) → _pytest.fixtures.FixtureFunctionMarker

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.

Fixtures can provide their values to test functions using `return` or `yield` statements. When using `yield`
the code block after the `yield` statement is executed as teardown code regardless of the test outcome, and must
yield exactly once.

> **Parameters**
>
> - **scope** – The scope for which this fixture is shared; one of `"function"` (default),
>   `"class"`, `"module"`, `"package"` or `"session"`.
>
>   This parameter may also be a callable which receives `(fixture_name, config)` as
>   parameters, and must return a `str` with one of the values mentioned above.
>
>   See *Dynamic scope* in the docs for more information.
>
> - **params** – An optional list of parameters which will cause multiple invocations of the fixture
>   function and all of the tests using it. The current parameter is available in `request.`
>   `param`.
>
> - **autouse** – If True, the fixture func is activated for all tests that can see it. If False (the
>   default), 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>')`.

pytest Documentation, Release 6.3

22.3.2 config.cache

Tutorial: *Cache: working with cross-testrun state*.

The `config.cache` object allows other plugins and fixtures to store and retrieve values across test runs. To access it from fixtures request `pytestconfig` into your fixture and get it with `pytestconfig.cache`.

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

`config.cache` is an instance of `pytest.Cache`:

final class Cache

makedir(*name: str*) → py._path.local.LocalPath
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 to 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.

get(*key: str*, *default*)
Return the 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** – The value to return in case of a cache-miss or invalid cache value.

set(*key: str*, *value: object*) → None
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 lists of dictionaries.

22.3.3 capsys

Tutorial: *Capturing of the stdout/stderr output*.

capsys()
Enable text capturing of writes to `sys.stdout` and `sys.stderr`.

The captured output is made available via `capsys.readouterr()` method calls, which return a `(out, err)` namedtuple. `out` and `err` will be `text` objects.

Returns an instance of `CaptureFixture[str]`.

Example:

```
def test_output(capsys):
    print("hello")
    captured = capsys.readouterr()
    assert captured.out == "hello\n"
```

class CaptureFixture

Object returned by the *capsys*, *capsysbinary*, *capfd* and *capfdbinary* fixtures.

readouterr() → _pytest.capture.CaptureResult

Read and return the captured output so far, resetting the internal buffer.

> **Returns** The captured content as a namedtuple with out and err string attributes.

with disabled() → Generator[None, None, None]

Temporarily disable capturing while inside the with block.

22.3.4 capsysbinary

Tutorial: *Capturing of the stdout/stderr output*.

capsysbinary()

Enable bytes capturing of writes to sys.stdout and sys.stderr.

The captured output is made available via capsysbinary.readouterr() method calls, which return a (out, err) namedtuple. out and err will be bytes objects.

Returns an instance of *CaptureFixture[bytes]*.

Example:

```
def test_output(capsysbinary):
    print("hello")
    captured = capsysbinary.readouterr()
    assert captured.out == b"hello\n"
```

22.3.5 capfd

Tutorial: *Capturing of the stdout/stderr output*.

capfd()

Enable text capturing of writes to file descriptors 1 and 2.

The captured output is made available via capfd.readouterr() method calls, which return a (out, err) namedtuple. out and err will be text objects.

Returns an instance of *CaptureFixture[str]*.

Example:

```
def test_system_echo(capfd):
    os.system('echo "hello"')
    captured = capfd.readouterr()
    assert captured.out == "hello\n"
```

22.3.6 capfdbinary

Tutorial: *Capturing of the stdout/stderr output*.

capfdbinary()

Enable bytes capturing of writes to file descriptors 1 and 2.

The captured output is made available via `capfd.readouterr()` method calls, which return a `(out, err)` namedtuple. `out` and `err` will be `byte` objects.

Returns an instance of *CaptureFixture[bytes]*.

Example:

```python
def test_system_echo(capfdbinary):
    os.system('echo "hello"')
    captured = capfdbinary.readouterr()
    assert captured.out == b"hello\n"
```

22.3.7 doctest_namespace

Tutorial: *Doctest integration for modules and test files*.

doctest_namespace()

Fixture that returns a `dict` that will be injected into the namespace of doctests.

Usually this fixture is used in conjunction with another `autouse` fixture:

```python
@pytest.fixture(autouse=True)
def add_np(doctest_namespace):
    doctest_namespace["np"] = numpy
```

For more details: *'doctest_namespace' fixture*.

22.3.8 request

Tutorial: *Pass different values to a test function, depending on command line options*.

The `request` fixture is a special fixture providing information of the requesting test function.

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: Optional[str]

Fixture for which this request is being performed.

scope: _Scope

Scope string, one of "function", "class", "module", "session".

fixturenames

Names of all active fixtures in this request.

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: Callable[], object]*) → None
    Add finalizer/teardown function to be called after the last test within the requesting test context finished execution.

applymarker(*marker: Union[str, _pytest.mark.structures.MarkDecorator]*) → None
    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.MarkDecorator` object created by a call to `pytest.mark.NAME(...)`.

raiseerror(*msg: Optional[str]*) → NoReturn
    Raise a FixtureLookupError with the given message.

getfixturevalue(*argname: str*) → Any
    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.

        Raises `pytest.FixtureLookupError` – If the given fixture could not be found.

22.3.9 pytestconfig

pytestconfig()
    Session-scoped fixture that returns the `_pytest.config.Config` object.

    Example:

```
def test_foo(pytestconfig):
    if pytestconfig.getoption("verbose") > 0:
        ...
```

22.3.10 record_property

Tutorial: *record_property*.

record_property()
    Add extra properties to the calling test.

    User properties become part of the test report and are available to the configured reporters, like JUnit XML.

    The fixture is callable with `name, value`. The value is automatically XML-encoded.

    Example:

```python
def test_function(record_property):
    record_property("example_key", 1)
```

22.3.11 record_testsuite_property

Tutorial: *record_testsuite_property*.

record_testsuite_property()
    Record a new `<property>` tag as child of the root `<testsuite>`.

    This is suitable to writing global information regarding the entire test suite, and is compatible with `xunit2` JUnit family.

    This is a `session`-scoped fixture which is called with `(name, value)`. Example:

```python
def test_foo(record_testsuite_property):
    record_testsuite_property("ARCH", "PPC")
    record_testsuite_property("STORAGE_TYPE", "CEPH")
```

    `name` must be a string, `value` will be converted to a string and properly xml-escaped.

> **Warning:** Currently this fixture **does not work** with the pytest-xdist plugin. See issue #7767 for details.

22.3.12 caplog

Tutorial: *Logging*.

caplog()
    Access and control log capturing.

    Captured logs are available through the following properties/methods:

```
* caplog.messages        -> list of format-interpolated log messages
* caplog.text            -> string containing formatted log output
* caplog.records         -> list of logging.LogRecord instances
* caplog.record_tuples   -> list of (logger_name, level, message) tuples
* caplog.clear()         -> clear captured records and formatted log output string
```

    Returns a *pytest.LogCaptureFixture* instance.

final class LogCaptureFixture
    Provides access and control of log capturing.

handler

> Get the logging handler used by the fixture.
>
> > **Return type** LogCaptureHandler

get_records(*when: str*) → List[logging.LogRecord]

> Get the logging records for one of the possible test phases.
>
> > **Parameters when** (`str`) – Which test phase to obtain the records from. Valid values are: "setup", "call" and "teardown".
> >
> > **Returns** The list of captured records at the given stage.
> >
> > **Return type** List[logging.LogRecord]
>
> New in version 3.4.

text

> The formatted log text.

records

> The list of log records.

record_tuples

> A list of a stripped down version of log records intended for use in assertion comparison.
>
> The format of the tuple is:
>
> > (logger_name, log_level, message)

messages

> A list of format-interpolated log messages.
>
> Unlike 'records', which contains the format string and parameters for interpolation, log messages in this list are all interpolated.
>
> Unlike 'text', which contains the output from the handler, log messages in this list are unadorned with levels, timestamps, etc, making exact comparisons more reliable.
>
> Note that traceback or stack info (from `logging.exception()` or the `exc_info` or `stack_info` arguments to the logging functions) is not included, as this is added by the formatter in the handler.
>
> New in version 3.7.

clear() → None

> Reset the list of log records and the captured log text.

set_level(*level: Union[int, str]*, *logger: Optional[str] = None*) → None

> Set the level of a logger for the duration of a test.
>
> Changed in version 3.4: The levels of the loggers changed by this function will be restored to their initial values at the end of the test.
>
> > **Parameters**
> >
> > - **level** (`int`) – The level.
> >
> > - **logger** (`str`) – The logger to update. If not given, the root logger.

with at_level(*level: int*, *logger: Optional[str] = None*) → Generator[None, None, None]

> Context manager that sets the level for capturing of logs. After the end of the 'with' statement the level is restored to its original value.
>
> > **Parameters**
> >
> > - **level** (`int`) – The level.

- **logger** (`str`) – The logger to update. If not given, the root logger.

22.3.13 monkeypatch

Tutorial: *Monkeypatching/mocking modules and environments*.

monkeypatch()
A convenient fixture for monkey-patching.

The fixture provides these 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, 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.

Returns a *MonkeyPatch* instance.

final class MonkeyPatch
Helper to conveniently monkeypatch attributes/items/environment variables/syspath.

Returned by the *monkeypatch* fixture.

> **Versionchanged:** 6.2 Can now also be used directly as `pytest.MonkeyPatch()`, for when the fixture is not available. In this case, use `with MonkeyPatch.context() as mp:` or remember to call `undo()` explicitly.

classmethod with context() → Generator[_pytest.monkeypatch.MonkeyPatch, None, None]
Context manager that returns a new *MonkeyPatch* object which undoes any patching done inside the `with` block upon exit.

Example:

```python
import functools


def test_partial(monkeypatch):
    with monkeypatch.context() as m:
        m.setattr(functools, "partial", 3)
```

Useful in situations where it is desired to undo some patches before the test ends, such as mocking `stdlib` functions that might break pytest itself if mocked (for examples of this see #3290.

setattr(*target: str*, *name: object*, *value: _pytest.monkeypatch.Notset = '...'*, *raising: bool = '...'*) →
        None
setattr(*target: object*, *name: str*, *value: object*, *raising: bool = '...'*) → None
Set attribute value on target, memorizing the old value.

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. For example, `monkeypatch.setattr("os.getcwd", lambda: "/")` would set the `getcwd` function of the `os` module.

Raises AttributeError if the attribute does not exist, unless `raising` is set to False.

delattr(*target: Union[object, str], name: Union[str, _pytest.monkeypatch.Notset] = <notset>, raising: bool = True*) → None
    Delete attribute `name` from `target`.

    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.

    Raises AttributeError it the attribute does not exist, unless `raising` is set to False.

setitem(*dic: MutableMapping[K, V], name: K, value: V*) → None
    Set dictionary entry `name` to value.

delitem(*dic: MutableMapping[K, V], name: K, raising: bool = True*) → None
    Delete `name` from dict.

    Raises `KeyError` if it doesn't exist, unless `raising` is set to False.

setenv(*name: str, value: str, prepend: Optional[str] = None*) → 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: str, raising: bool = True*) → None
    Delete `name` from the environment.

    Raises `KeyError` if it does not exist, unless `raising` is set to False.

syspath_prepend(*path*) → None
    Prepend `path` to `sys.path` list of import locations.

chdir(*path: Union[str, os.PathLike[str]]*) → None
    Change the current working directory to the specified path.

    Path can be a string or a path object.

undo() → None
    Undo previous changes.

    This call consumes the undo stack. Calling it a second time has no effect unless you do more monkey-patching 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.

22.3.14 pytester

New in version 6.2.

Provides a *Pytester* instance that can be used to run and test pytest itself.

It provides an empty directory where pytest can be executed in isolation, and contains facilities to write tests, configuration files, and match against expected output.

To use it, include in your topmost `conftest.py` file:

```
pytest_plugins = "pytester"
```

final class Pytester

> Facilities to write tests/configuration files, execute pytest in isolation, and match against expected output, perfect for black-box testing of pytest plugins.
>
> It attempts to isolate the test run from external factors as much as possible, modifying the current working directory to `path` and environment variables during initialization.
>
> Attributes:
>
> > **Variables**
> >
> > - **_path_** (`Path`) – temporary directory path used to create files/run tests from, etc.
> >
> > - **_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 depends on the method using them so refer to them for details.
>
> **CLOSE_STDIN**
>
> > alias of `object`
>
> **exception TimeoutExpired**
>
> **path**
>
> > Temporary directory where files are created and pytest is executed.
>
> **make_hook_recorder**(_pluginmanager:_ __pytest.config.PytestPluginManager_) → __pytest.pytester.HookRecorder_
>
> > Create a new `HookRecorder` for a PluginManager.
>
> **chdir**() → None
>
> > Cd into the temporary directory.
> >
> > This is done automatically upon instantiation.
>
> **makefile**(_ext: str_, _*args: str_, _**kwargs: str_) → pathlib.Path
>
> > Create new file(s) in the test directory.
> >
> > > **Parameters**
> > >
> > > - **ext** (`str`) – The extension the file(s) should use, including the dot, e.g. `.py`.
> > >
> > > - **args** – All args are treated as strings and joined using newlines. The result is written as contents to the file. The name of the file is based on the test function requesting this fixture.
> > >
> > > - **kwargs** – Each keyword is the name of a file, while the value of it will be written as contents of the file.
> >
> > Examples:
> >
> > ```
> > pytester.makefile(".txt", "line1", "line2")
> >
> > pytester.makefile(".ini", pytest="[pytest]\naddopts=-rs\n")
> > ```
>
> **makeconftest**(_source: str_) → pathlib.Path
>
> > Write a contest.py file with 'source' as contents.
>
> **makeini**(_source: str_) → pathlib.Path
>
> > Write a tox.ini file with 'source' as contents.
>
> **getinicfg**(_source: str_) → iniconfig.SectionWrapper
>
> > Return the pytest section from the tox.ini config file.

makepyprojecttoml(*source: str*) → pathlib.Path
    Write a pyproject.toml file with 'source' as contents.

    New in version 6.0.

makepyfile(**args*, ***kwargs*) → pathlib.Path
    Shortcut for .makefile() with a .py extension.

    Defaults to the test name with a '.py' extension, e.g test_foobar.py, overwriting existing files.

    Examples:

```python
def test_something(pytester):
    # Initial file is created test_something.py.
    pytester.makepyfile("foobar")
    # To create multiple files, pass kwargs accordingly.
    pytester.makepyfile(custom="foobar")
    # At this point, both 'test_something.py' & 'custom.py' exist in the test
    →directory.
```

maketxtfile(**args*, ***kwargs*) → pathlib.Path
    Shortcut for .makefile() with a .txt extension.

    Defaults to the test name with a '.txt' extension, e.g test_foobar.txt, overwriting existing files.

    Examples:

```python
def test_something(pytester):
    # Initial file is created test_something.txt.
    pytester.maketxtfile("foobar")
    # To create multiple files, pass kwargs accordingly.
    pytester.maketxtfile(custom="foobar")
    # At this point, both 'test_something.txt' & 'custom.txt' exist in the
    →test directory.
```

syspathinsert(*path: Optional[Union[str, os.PathLike[str]]] = None*) → None
    Prepend a directory to sys.path, defaults to tmpdir.

    This is undone automatically when this object dies at the end of each test.

mkdir(*name: str*) → pathlib.Path
    Create a new (sub)directory.

mkpydir(*name: str*) → pathlib.Path
    Create a new python package.

    This creates a (sub)directory with an empty __init__.py file so it gets recognised as a Python package.

copy_example(*name: Optional[str] = None*) → pathlib.Path
    Copy file from project's directory into the testdir.

        Parameters name (`str`) – The name of the file to copy.

        Returns path to the copied directory (inside `self.path`).

class Session(**k*, ***kw*)

    exception Failed
        Signals a stop as failed test run.

    exception Interrupted
        Signals that the test run was interrupted.

for ... in collect() → Iterator[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
Return a list of children (items and collectors) for this collection node.

classmethod from_config(*config: _pytest.config.Config*) → _pytest.main.Session

for ... in genitems(*node: Union[_pytest.nodes.Item, _pytest.nodes.Collector]*) → Iterator[_pytest.nodes.Item]

gethookproxy(*fspath: os.PathLike[str]*)

isinitpath(*path: Union[str, os.PathLike[str]]*) → bool

perform_collect(*args: Optional[Sequence[str]] = None, genitems: bool = True*) → Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
Perform the collection phase for this session.

This is called by the default *pytest_collection* hook implementation; see the documentation of this hook for more details. For testing purposes, it may also be called directly on a fresh Session.

This function normally recursively expands any collectors collected from the session to their items, and only items are returned. For testing purposes, this may be suppressed by passing genitems=False, in which case the return value contains these collectors unexpanded, and session.items is empty.

pytest_collectreport(*report: Union[_pytest.reports.TestReport, _pytest.reports.CollectReport]*) → None

pytest_collectstart() → None

pytest_runtest_logreport(*report: Union[_pytest.reports.TestReport, _pytest.reports.CollectReport]*) → None

getnode(*config: _pytest.config.Config, arg: Union[str, os.PathLike[str]]*) → Optional[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
Return the collection node of a file.

Parameters

- **config**(_pytest.config.Config) – A pytest config. See *parseconfig()* and *parseconfigure()* for creating it.

- **arg**(py.path.local) – Path to the file.

getpathnode(*path: Union[str, os.PathLike[str]]*)
Return the collection node of a file.

This is like *getnode()* but uses *parseconfigure()* to create the (configured) pytest Config instance.

Parameters path(py.path.local) – Path to the file.

genitems(*colitems: Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]*) → List[_pytest.nodes.Item]
Generate all test items from a collection node.

This recurses into the collection node and returns a list of all the test items contained within.

runitem(*source: str*) → Any
Run the "test_func" Item.

The calling test instance (class containing the test method) must provide a .getrunner() method which should return a runner which can run the test protocol for a single item, e.g. _pytest.runner.runtestprotocol().

inline_runsource(*source: str*, **cmdlineargs*) → *_pytest.pytester.HookRecorder*
    Run a test module in process using `pytest.main()`.

    This run writes "source" into a temporary file and runs `pytest.main()` on it, returning a `HookRecorder` instance for the result.

        Parameters

            • **source** – The source code of the test module.

            • **cmdlineargs** – Any extra command line arguments to use.

        Returns `HookRecorder` instance of the result.

inline_genitems(**args*) → Tuple[List[_pytest.nodes.Item], *_pytest.pytester.HookRecorder*]
    Run `pytest.main(['--collectonly'])` in-process.

    Runs the `pytest.main()` function to run all of pytest inside the test process itself like `inline_run()`, but returns a tuple of the collected items and a `HookRecorder` instance.

inline_run(**args: Union[str, os.PathLike[str]]*, *plugins=()*, *no_reraise_ctrlc: bool = False*) → *_pytest.pytester.HookRecorder*
    Run `pytest.main()` in-process, returning a HookRecorder.

    Runs the `pytest.main()` function to run all of pytest inside the test process itself. This means it can return a `HookRecorder` instance which gives more detailed results from that run than can be done by matching stdout/stderr from `runpytest()`.

        Parameters

            • **args** – Command line arguments to pass to `pytest.main()`.

            • **plugins** – Extra plugin instances the `pytest.main()` instance should use.

            • **no_reraise_ctrlc** – Typically we reraise keyboard interrupts from the child run. If True, the KeyboardInterrupt exception is captured.

        Returns A `HookRecorder` instance.

runpytest_inprocess(**args: Union[str, os.PathLike[str]]*, ***kwargs: Any*) → *_pytest.pytester.RunResult*
    Return result of running pytest in-process, providing a similar interface to what self.runpytest() provides.

runpytest(**args: Union[str, os.PathLike[str]]*, ***kwargs: Any*) → *_pytest.pytester.RunResult*
    Run pytest inline or in a subprocess, depending on the command line option "–runpytest" and return a `RunResult`.

parseconfig(**args: Union[str, os.PathLike[str]]*) → *_pytest.config.Config*
    Return a new pytest Config instance from given commandline args.

    This invokes the pytest bootstrapping code in _pytest.config to create a new `_pytest.core.PluginManager` and call the pytest_cmdline_parse hook to create a new `_pytest.config.Config` instance.

    If `plugins` has been populated they should be plugin modules to be registered with the PluginManager.

parseconfigure(**args: Union[str, os.PathLike[str]]*) → *_pytest.config.Config*
    Return a new pytest configured Config instance.

    Returns a new `_pytest.config.Config` instance like `parseconfig()`, but also calls the pytest_configure hook.

getitem(*source: Union[str, os.PathLike[str]]*, *funcname: str = 'test_func'*) → _pytest.nodes.Item
    Return the test item for a test function.

Writes the source to a python file and runs pytest's collection on the resulting module, returning the test item for the requested function name.

> **Parameters**
>
> - **source** – The module source.
> - **funcname** – The name of the test function for which to return a test item.

getitems(*source: Union[str, os.PathLike[str]]*) → List[_pytest.nodes.Item]
    Return all test items collected from the module.

Writes the source to a Python file and runs pytest's collection on the resulting module, returning all test items contained within.

getmodulecol(*source: Union[str, os.PathLike[str]], configargs=(), *, withinit: bool = False*)
    Return the module collection node for `source`.

Writes `source` to a file using *makepyfile()* and then runs the pytest collection on it, returning the collection node for the test module.

> **Parameters**
>
> - **source** – The source code of the module to collect.
> - **configargs** – Any extra arguments to pass to *parseconfigure()*.
> - **withinit** – Whether to also write an `__init__.py` file to the same directory to ensure it is a package.

collect_by_name(*modcol: _pytest.nodes.Collector, name: str*) → Optional[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
    Return the collection node for name from the module collection.

Searchs a module collection node for a collection node matching the given name.

> **Parameters**
>
> - **modcol** – A module collection node; see *getmodulecol()*.
> - **name** – The name of the node to return.

popen(*cmdargs, stdout: Union[int, TextIO] = -1, stderr: Union[int, TextIO] = -1, stdin=<class 'object'>, **kw*)
    Invoke subprocess.Popen.

Calls subprocess.Popen making sure the current working directory is in the PYTHONPATH.

You probably want to use *run()* instead.

run(**cmdargs: Union[str, os.PathLike[str]], timeout: Optional[float] = None, stdin=<class 'object'>*) → *_pytest.pytester.RunResult*
    Run a command with arguments.

Run a process using subprocess.Popen saving the stdout and stderr.

> **Parameters**
>
> - **cmdargs** – The sequence of arguments to pass to `subprocess.Popen()`, with path-like objects being converted to `str` automatically.
> - **timeout** – The period in seconds after which to timeout and raise *Pytester.TimeoutExpired*.
> - **stdin** – Optional standard input. Bytes are being send, closing the pipe, otherwise it is passed through to popen. Defaults to CLOSE_STDIN, which translates to using a pipe (`subprocess.PIPE`) that gets closed.

> **Return type** *RunResult*

runpython(*script*) → *_pytest.pytester.RunResult*
> Run a python script using sys.executable as interpreter.
>
> > **Return type** *RunResult*

runpython_c(*command*)
> Run python -c "command".
>
> > **Return type** *RunResult*

runpytest_subprocess(**args*, *timeout: Optional[float] = None*) → *_pytest.pytester.RunResult*
> Run pytest as a subprocess with given arguments.
>
> Any plugins added to the `plugins` list will be added using the `-p` command line option. Additionally `--basetemp` is used to put any temporary files and directories in a numbered directory prefixed with "runpytest-" to not conflict with the normal numbered pytest location for temporary files and directories.
>
> > **Parameters**
> >
> > - **args** – The sequence of arguments to pass to the pytest subprocess.
> > - **timeout** – The period in seconds after which to timeout and raise `Pytester.TimeoutExpired`.
> >
> > **Return type** *RunResult*

spawn_pytest(*string: str*, *expect_timeout: float = 10.0*) → pexpect.spawn
> Run pytest using pexpect.
>
> This makes sure to use the right pytest and sets up the temporary directory locations.
>
> The pexpect child is returned.

spawn(*cmd: str*, *expect_timeout: float = 10.0*) → pexpect.spawn
> Run a command using pexpect.
>
> The pexpect child is returned.

class RunResult
> The result of running a command.

ret: Union[int, _pytest.config.ExitCode]
> The return value.

outlines
> List of lines captured from stdout.

errlines
> List of lines captured from stderr.

stdout
> *LineMatcher* of stdout.
>
> Use e.g. `str(stdout)` to reconstruct stdout, or the commonly used `stdout.fnmatch_lines()` method.

stderr
> *LineMatcher* of stderr.

duration
> Duration in seconds.

parseoutcomes() → Dict[str, int]
 Return a dictionary of outcome noun -> count from parsing the terminal output that the test process pro-
 duced.

 The returned nouns will always be in plural form:

 ```
 ======= 1 failed, 1 passed, 1 warning, 1 error in 0.13s ====
 ```

 Will return {"failed": 1, "passed": 1, "warnings": 1, "errors": 1}.

classmethod parse_summary_nouns(*lines*) → Dict[str, int]
 Extract the nouns from a pytest terminal summary line.

 It always returns the plural noun for consistency:

 ```
 ======= 1 failed, 1 passed, 1 warning, 1 error in 0.13s ====
 ```

 Will return {"failed": 1, "passed": 1, "warnings": 1, "errors": 1}.

assert_outcomes(*passed: int = 0, skipped: int = 0, failed: int = 0, errors: int = 0, xpassed: int = 0,*
 xfailed: int = 0) → None
 Assert that the specified outcomes appear with the respective numbers (0 means it didn't occur) in the text
 output from a test run.

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().

 __str__() → *str*
  Return the entire original text.

  New in version 6.2: You can use *str()* in older versions.

 fnmatch_lines_random(*lines2: Sequence[str]*) → None
  Check lines exist in the output in any order (using fnmatch.fnmatch()).

 re_match_lines_random(*lines2: Sequence[str]*) → None
  Check lines exist in the output in any order (using re.match()).

 get_lines_after(*fnline: str*) → Sequence[*str*]
  Return all lines following the given line in the text.

  The given line can contain glob wildcards.

 fnmatch_lines(*lines2: Sequence[str], *, consecutive: bool = False*) → None
  Check lines exist in the output (using fnmatch.fnmatch()).

  The argument is a list of lines which have to match and can use glob wildcards. If they do not match a
  pytest.fail() is called. The matches and non-matches are also shown as part of the error message.

  Parameters

   - **lines2** – String patterns to match.

   - **consecutive** – Match lines consecutively?

 re_match_lines(*lines2: Sequence[str], *, consecutive: bool = False*) → None
  Check lines exist in the output (using re.match()).

  The argument is a list of lines which have to match using re.match. If they do not match a pytest.fail()
  is called.

The matches and non-matches are also shown as part of the error message.

> Parameters
>
> > - **lines2** – string patterns to match.
> >
> > - **consecutive** – match lines consecutively?

no_fnmatch_line(*pat:* str) → None
> Ensure captured lines do not match the given pattern, using `fnmatch.fnmatch`.
>
> > Parameters **pat** (`str`) – The pattern to match lines.

no_re_match_line(*pat:* str) → None
> Ensure captured lines do not match the given pattern, using `re.match`.
>
> > Parameters **pat** (`str`) – The regular expression to match lines.

str() → *str*
> Return the entire original text.

class HookRecorder
> Record all hooks called in a plugin manager.
>
> This wraps all the hook calls in the plugin manager, recording each call before propagating the normal calls.
>
> **matchreport**(*inamepart:* *str* = *''*, *names:* *Union[str,* *Iterable[str]]* = *('pytest_runtest_logreport',* *'pytest_collectreport'),* *when:* *Optional[str]* = *None*) → Union[*_pytest.reports.CollectReport*, *_pytest.reports.TestReport*]
> Return a testreport whose dotted import path matches.

22.3.15 testdir

Identical to [*pytester*](#), but provides an instance whose methods return legacy `py.path.local` objects instead when applicable.

New code should avoid using [*testdir*](#) in favor of [*pytester*](#).

final class Testdir
> Similar to [*Pytester*](#), but this class works with legacy py.path.local objects instead.
>
> All methods just forward to an internal [*Pytester*](#) instance, converting results to `py.path.local` objects as necessary.
>
> **CLOSE_STDIN**
> > alias of [`object`](#)
>
> **exception TimeoutExpired**
>
> **class Session**(**k*, ***kw*)
>
> > **exception Failed**
> > > Signals a stop as failed test run.
> >
> > **exception Interrupted**
> > > Signals that the test run was interrupted.
> >
> > **for ... in collect**() → Iterator[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
> > > Return a list of children (items and collectors) for this collection node.
> >
> > **classmethod from_config**(*config:* _pytest.config.Config) → _pytest.main.Session

for ... in genitems(*node: Union[_pytest.nodes.Item, _pytest.nodes.Collector]*) → Iterator[_pytest.nodes.Item]

gethookproxy(*fspath: os.PathLike[str]*)

isinitpath(*path: Union[str, os.PathLike[str]]*) → bool

perform_collect(*args: Optional[Sequence[str]] = None*, *genitems:* bool *= True*) → Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
　　Perform the collection phase for this session.

　　This is called by the default `pytest_collection` hook implementation; see the documentation of this hook for more details. For testing purposes, it may also be called directly on a fresh `Session`.

　　This function normally recursively expands any collectors collected from the session to their items, and only items are returned. For testing purposes, this may be suppressed by passing `genitems=False`, in which case the return value contains these collectors unexpanded, and `session.items` is empty.

pytest_collectreport(*report:* Union[_pytest.reports.TestReport, _pytest.reports.CollectReport]) → None

pytest_collectstart() → None

pytest_runtest_logreport(*report:* Union[_pytest.reports.TestReport, _pytest.reports.CollectReport]) → None

tmpdir
　　Temporary directory where tests are executed.

make_hook_recorder(*pluginmanager*) → *_pytest.pytester.HookRecorder*
　　See *Pytester.make_hook_recorder()*.

chdir() → None
　　See *Pytester.chdir()*.

finalize() → None
　　See Pytester._finalize().

makefile(*ext*, **args*, ***kwargs*) → py._path.local.LocalPath
　　See *Pytester.makefile()*.

makeconftest(*source*) → py._path.local.LocalPath
　　See *Pytester.makeconftest()*.

makeini(*source*) → py._path.local.LocalPath
　　See *Pytester.makeini()*.

getinicfg(*source: str*) → iniconfig.SectionWrapper
　　See *Pytester.getinicfg()*.

makepyprojecttoml(*source*) → py._path.local.LocalPath
　　See *Pytester.makepyprojecttoml()*.

makepyfile(**args*, ***kwargs*) → py._path.local.LocalPath
　　See *Pytester.makepyfile()*.

maketxtfile(**args*, ***kwargs*) → py._path.local.LocalPath
　　See *Pytester.maketxtfile()*.

syspathinsert(*path=None*) → None
　　See *Pytester.syspathinsert()*.

mkdir(*name*) → py._path.local.LocalPath
　　See *Pytester.mkdir()*.

mkpydir(*name*) → py._path.local.LocalPath
> See *Pytester.mkpydir()*.

copy_example(*name=None*) → py._path.local.LocalPath
> See *Pytester.copy_example()*.

getnode(*config: _pytest.config.Config*,   *arg*)   →   Optional[Union[_pytest.nodes.Item,
> _pytest.nodes.Collector]]
> See *Pytester.getnode()*.

getpathnode(*path*)
> See *Pytester.getpathnode()*.

genitems(*colitems: List[Union[_pytest.nodes.Item,   _pytest.nodes.Collector]]*)   →
> List[_pytest.nodes.Item]
> See *Pytester.genitems()*.

runitem(*source*)
> See *Pytester.runitem()*.

inline_runsource(*source*, **cmdlineargs*)
> See *Pytester.inline_runsource()*.

inline_genitems(**args*)
> See *Pytester.inline_genitems()*.

inline_run(**args*, *plugins=()*, *no_reraise_ctrlc: bool = False*)
> See *Pytester.inline_run()*.

runpytest_inprocess(**args*, ***kwargs*) → *_pytest.pytester.RunResult*
> See *Pytester.runpytest_inprocess()*.

runpytest(**args*, ***kwargs*) → *_pytest.pytester.RunResult*
> See *Pytester.runpytest()*.

parseconfig(**args*) → *_pytest.config.Config*
> See *Pytester.parseconfig()*.

parseconfigure(**args*) → *_pytest.config.Config*
> See *Pytester.parseconfigure()*.

getitem(*source*, *funcname='test_func'*)
> See *Pytester.getitem()*.

getitems(*source*)
> See *Pytester.getitems()*.

getmodulecol(*source*, *configargs=()*, *withinit=False*)
> See *Pytester.getmodulecol()*.

collect_by_name(*modcol: _pytest.nodes.Collector*,   *name: str*)   →   Op-
> tional[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
> See *Pytester.collect_by_name()*.

popen(*cmdargs*, *stdout: Union[int, TextIO] = -1*, *stderr: Union[int, TextIO] = -1*, *stdin=<class 'ob-
> ject'>*, ***kw*)
> See *Pytester.popen()*.

run(**cmdargs*, *timeout=None*, *stdin=<class 'object'>*) → *_pytest.pytester.RunResult*
> See *Pytester.run()*.

runpython(*script*) → *_pytest.pytester.RunResult*
> See *Pytester.runpython()*.

runpython_c(*command*)
 See *Pytester.runpython_c()*.

runpytest_subprocess(**args*, *timeout=None*) → *_pytest.pytester.RunResult*
 See *Pytester.runpytest_subprocess()*.

spawn_pytest(*string: str*, *expect_timeout: float = 10.0*) → pexpect.spawn
 See *Pytester.spawn_pytest()*.

spawn(*cmd: str*, *expect_timeout: float = 10.0*) → pexpect.spawn
 See *Pytester.spawn()*.

22.3.16 recwarn

Tutorial: *Asserting warnings with the warns function*

recwarn()
 Return a `WarningsRecorder` instance that records all warnings emitted by test functions.

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

class WarningsRecorder
 A context manager to record raised warnings.

 Adapted from `warnings.catch_warnings`.

 list
  The list of recorded warnings.

 pop(*cls: Type[Warning] = <class 'Warning'>*) → warnings.WarningMessage
  Pop the first recorded warning, raise exception if not exists.

 clear() → None
  Clear the list of recorded warnings.

Each recorded warning is an instance of `warnings.WarningMessage`.

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

22.3.17 tmp_path

Tutorial: *Temporary directories and files*

tmp_path()
 Return a temporary directory path object which is unique to each test function invocation, created as a sub directory of the base temporary directory.

 By default, a new base temporary directory is created each test session, and old bases are removed after 3 sessions, to aid in debugging. If `--basetemp` is used then it is cleared each session. See *The default base temporary directory*.

 The returned object is a `pathlib.Path` object.

22.3.18 tmp_path_factory

Tutorial: *The tmp_path_factory fixture*

tmp_path_factory is an instance of `TempPathFactory`:

final class TempPathFactory
> Factory for temporary directories under the common base temp directory.
>
> The base directory can be configured using the `--basetemp` option.

22.3.19 tmpdir

Tutorial: *Temporary directories and files*

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.
>
> By default, a new base temporary directory is created each test session, and old bases are removed after 3 sessions, to aid in debugging. If `--basetemp` is used then it is cleared each session. See *The default base temporary directory*.
>
> The returned object is a py.path.local path object.

22.3.20 tmpdir_factory

Tutorial: *The 'tmpdir_factory' fixture*

tmp_path_factory is an instance of `TempdirFactory`:

final class TempdirFactory
> Backward comptibility wrapper that implements :class:py.path.local for :class:TempPathFactory.

22.4 Hooks

Tutorial: *Writing plugins*.

Reference to all hooks which can be implemented by *conftest.py files* and *plugins*.

22.4.1 Bootstrapping hooks

Bootstrapping hooks called for plugins registered early enough (internal and setuptools plugins).

pytest_load_initial_conftests(*early_config: Config*, *parser: Parser*, *args: List[str]*) → None
> Called to implement the loading of initial conftest files ahead of command line option parsing.

> ---
> **Note:** This hook will not be called for `conftest.py` files, only for setuptools plugins.
> ---

> **Parameters**
> - **early_config** (`_pytest.config.Config`) – The pytest config object.
> - **args** (`List[str]`) – Arguments passed on the command line.

- **parser** (`_pytest.config.argparsing.Parser`) – To add command line options.

pytest_cmdline_preparse(*config: Config*, *args: List[str]*) → [None](#)
 (**Deprecated**) modify command line arguments before option parsing.

 This hook is considered deprecated and will be removed in a future pytest version. Consider using `pytest_load_initial_conftests()` instead.

 Note: This hook will not be called for `conftest.py` files, only for setuptools plugins.

 Parameters

- **config** (`_pytest.config.Config`) – The pytest config object.
- **args** (`List[str]`) – Arguments passed on the command line.

pytest_cmdline_parse(*pluginmanager: PytestPluginManager*, *args: List[str]*) → Optional[Config]
 Return an initialized config object, parsing the specified args.

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

 Note: This hook will only be called for plugin classes passed to the `plugins` arg when using *pytest.main* to perform an in-process test run.

 Parameters

- **pluginmanager** (`_pytest.config.PytestPluginManager`) – Pytest plugin manager.
- **args** (`List[str]`) – List of arguments passed on the command line.

pytest_cmdline_main(*config: Config*) → Optional[Union[ExitCode, [int](#)]]
 Called for performing the main command line action. The default implementation will invoke the configure hooks and runtest_mainloop.

 Note: This hook will not be called for `conftest.py` files, only for setuptools plugins.

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

 Parameters config (`_pytest.config.Config`) – The pytest config object.

22.4.2 Initialization hooks

Initialization hooks called for plugins and `conftest.py` files.

pytest_addoption(*parser: Parser*, *pluginmanager: PytestPluginManager*) → [None](#)
 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** (`_pytest.config.argparsing.Parser`) – To add command line options, call `parser.addoption(...)`. To add ini-file values call `parser.addini(...)`.

- **pluginmanager** (`_pytest.config.PytestPluginManager`) – pytest plugin manager, which can be used to install `hookspec()`'s or `hookimpl()`'s and allow one plugin to call another plugin's hooks to change how command line options are added.

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.

Note: This hook is incompatible with `hookwrapper=True`.

---

pytest_addhooks(*pluginmanager: PytestPluginManager*) → None
  Called at plugin registration time to allow adding new hooks via a call to `pluginmanager.add_hookspecs(module_or_class, prefix)`.

  Parameters pluginmanager (`_pytest.config.PytestPluginManager`) – pytest plugin manager.

Note: This hook is incompatible with `hookwrapper=True`.

---

pytest_configure(*config: Config*) → None
  Allow 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.

Note: This hook is incompatible with `hookwrapper=True`.

  Parameters config (`_pytest.config.Config`) – The pytest config object.

pytest_unconfigure(*config: Config*) → None
  Called before test process is exited.

  Parameters config (`_pytest.config.Config`) – The pytest config object.

pytest_sessionstart(*session: Session*) → None
  Called after the `Session` object has been created and before performing collection and entering the run test loop.

  Parameters session (`pytest.Session`) – The pytest session object.

pytest_sessionfinish(*session: Session*, *exitstatus: Union[int, ExitCode]*) → None
  Called after whole test run finished, right before returning the exit status to the system.

  Parameters

  - **session** (`pytest.Session`) – The pytest session object.

  - **exitstatus** (`int`) – The status which pytest will return to the system.

---

pytest_plugin_registered(*plugin: _PluggyPlugin*, *manager: PytestPluginManager*) → None
    A new pytest plugin got registered.

> **Parameters**
>
> > - **plugin** – The plugin module or instance.
> >
> > - **manager** (`_pytest.config.PytestPluginManager`) – pytest plugin manager.

> **Note:** This hook is incompatible with `hookwrapper=True`.

22.4.3 Collection hooks

`pytest` calls the following hooks for collecting files and directories:

pytest_collection(*session: Session*) → Optional[object]
    Perform the collection phase for the given session.

Stops at first non-None result, see *firstresult: stop at first non-None result*. The return value is not used, but only stops further processing.

The default collection phase is this (see individual hooks for full details):

1. Starting from `session` as the initial collector:

   1. `pytest_collectstart(collector)`

   2. `report = pytest_make_collect_report(collector)`

   3. `pytest_exception_interact(collector, call, report)` if an interactive exception occurred

   4. For each collected node:

      1. If an item, `pytest_itemcollected(item)`

      2. If a collector, recurse into it.

   5. `pytest_collectreport(report)`

2. `pytest_collection_modifyitems(session, config, items)`

   1. `pytest_deselected(items)` for any deselected items (may be called multiple times)

3. `pytest_collection_finish(session)`

4. Set `session.items` to the list of collected items

5. Set `session.testscollected` to the number of collected items

You can implement this hook to only perform some action before collection, for example the terminal plugin uses it to start displaying the collection counter (and returns `None`).

> **Parameters session** (`pytest.Session`) – The pytest session object.

pytest_ignore_collect(*fspath: pathlib.Path*, *path: py._path.local.LocalPath*, *config: Config*) → Optional[bool]
    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*.

> **Parameters**
>
> - **fspath** (`pathlib.Path`) – The path to analyze.
>
> - **path** (`py.path.local`) – The path to analyze.
>
> - **config** (`_pytest.config.Config`) – The pytest config object.

Changed in version 6.3.0: The `fspath` parameter was added as a `pathlib.Path` equivalent of the `path` parameter.

pytest_collect_file(*fspath: pathlib.Path*, *path: py._path.local.LocalPath*, *parent: Collector*) → Optional[Collector]
  Create a Collector for the given path, or None if not relevant.

The new node needs to have the specified `parent` as a parent.

> **Parameters**
>
> - **fspath** (`pathlib.Path`) – The path to analyze.
>
> - **path** (`py.path.local`) – The path to collect.

Changed in version 6.3.0: The `fspath` parameter was added as a `pathlib.Path` equivalent of the `path` parameter.

pytest_pycollect_makemodule(*fspath: pathlib.Path*, *path: py._path.local.LocalPath*, *parent*) → Optional[Module]
  Return a Module collector or None for the given path.

This hook will be called for each matching test module path. The pytest_collect_file hook needs to be used if you want to create test modules for files that do not match as a test module.

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

> **Parameters**
>
> - **fspath** (`pathlib.Path`) – The path of the module to collect.
>
> - **path** (`py.path.local`) – The path of the module to collect.

Changed in version 6.3.0: The `fspath` parameter was added as a `pathlib.Path` equivalent of the `path` parameter.

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

pytest_pycollect_makeitem(*collector: PyCollector*, *name: str*, *obj: object*) → Union[None, Item, Collector, List[Union[Item, Collector]]]
  Return a 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: Metafunc*) → None
  Generate (multiple) parametrized calls to a test function.

pytest_make_parametrize_id(*config: Config*, *val: object*, *argname: str*) → Optional[str]
  Return a user-friendly string representation of the given `val` that will be used by @pytest.mark.parametrize calls, or 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*.

> **Parameters**
>
> - **config** (`_pytest.config.Config`) – The pytest config object.

- **val** – The parametrized value.

- **argname** (*str*) – The automatic parameter name produced by pytest.

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

pytest_collection_modifyitems (*session: Session*, *config: Config*, *items: List[Item]*) → None
    Called after collection has been performed. May filter or re-order the items in-place.

> **Parameters**
>
> - **session** (`pytest.Session`) – The pytest session object.
>
> - **config** (`_pytest.config.Config`) – The pytest config object.
>
> - **items** (`List[pytest.Item]`) – List of item objects.

Note: If this hook is implemented in `conftest.py` files, it always receives all collected items, not only those under the `conftest.py` where it is implemented.

---

pytest_collection_finish (*session: Session*) → None
    Called after collection has been performed and modified.

> **Parameters** **session** (`pytest.Session`) – The pytest session object.

22.4.4 Test running (runtest) hooks

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

pytest_runtestloop (*session: Session*) → Optional[object]
    Perform the main runtest loop (after collection finished).

The default hook implementation performs the runtest protocol for all items collected in the session (`session.items`), unless the collection failed or the `collectonly` pytest option is set.

If at any point *`pytest.exit()`* is called, the loop is terminated immediately.

If at any point `session.shouldfail` or `session.shouldstop` are set, the loop is terminated after the runtest protocol for the current item is finished.

> **Parameters** **session** (`pytest.Session`) – The pytest session object.

Stops at first non-None result, see *firstresult: stop at first non-None result*. The return value is not used, but only stops further processing.

pytest_runtest_protocol (*item: Item*, *nextitem: Optional[Item]*) → Optional[object]
    Perform the runtest protocol for a single test item.

The default runtest protocol is this (see individual hooks for full details):

- `pytest_runtest_logstart(nodeid, location)`

- **Setup phase:**

    - `call = pytest_runtest_setup(item)` (wrapped in `CallInfo(when="setup")`)

    - `report = pytest_runtest_makereport(item, call)`

    - `pytest_runtest_logreport(report)`

    - `pytest_exception_interact(call, report)` if an interactive exception occurred

- **Call phase, if the the setup passed and the `setuponly` pytest option is not set:**

- **call = pytest_runtest_call(item)** (wrapped in `CallInfo(when="call")`)

- **report = pytest_runtest_makereport(item, call)**

- **pytest_runtest_logreport(report)**

- **pytest_exception_interact(call, report)** if an interactive exception occurred

- **Teardown phase:**

  - **call = pytest_runtest_teardown(item, nextitem)** (wrapped in `CallInfo(when="teardown")`)

  - **report = pytest_runtest_makereport(item, call)**

  - **pytest_runtest_logreport(report)**

  - **pytest_exception_interact(call, report)** if an interactive exception occurred

- **pytest_runtest_logfinish(nodeid, location)**

  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).

Stops at first non-None result, see *firstresult: stop at first non-None result*. The return value is not used, but only stops further processing.

pytest_runtest_logstart(*nodeid: str*, *location: Tuple[str, Optional[int], str]*) → None
  Called at the start of running the runtest protocol for a single item.

  See *pytest_runtest_protocol()* for a description of the runtest protocol.

  Parameters

  - **nodeid** (*str*) – Full node ID of the item.

  - **location** – A tuple of (filename, lineno, testname).

pytest_runtest_logfinish(*nodeid: str*, *location: Tuple[str, Optional[int], str]*) → None
  Called at the end of running the runtest protocol for a single item.

  See *pytest_runtest_protocol()* for a description of the runtest protocol.

  Parameters

  - **nodeid** (*str*) – Full node ID of the item.

  - **location** – A tuple of (filename, lineno, testname).

pytest_runtest_setup(*item: Item*) → None
  Called to perform the setup phase for a test item.

  The default implementation runs `setup()` on `item` and all of its parents (which haven't been setup yet). This includes obtaining the values of fixtures required by the item (which haven't been obtained yet).

pytest_runtest_call(*item: Item*) → None
  Called to run the test for test item (the call phase).

  The default implementation calls `item.runtest()`.

pytest_runtest_teardown(*item: Item*, *nextitem: Optional[Item]*) → None
  Called to perform the teardown phase for a test item.

The default implementation runs the finalizers and calls `teardown()` on `item` and all of its parents (which need to be torn down). This includes running the teardown phase of fixtures required by the item (if they go out of scope).

> **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: Item*, *call: CallInfo[None]*) → Optional[TestReport]

Called to create a *_pytest.reports.TestReport* for each of the setup, call and teardown runtest phases of a test item.

See *pytest_runtest_protocol()* for a description of the runtest protocol.

> **Parameters call** (`CallInfo[None]`) – The CallInfo for the phase.

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.

pytest_pyfunc_call(*pyfuncitem: Function*) → Optional[object]

Call underlying test function.

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

22.4.5 Reporting hooks

Session related reporting hooks:

pytest_collectstart(*collector: Collector*) → None

Collector starts collecting.

pytest_make_collect_report(*collector: Collector*) → Optional[CollectReport]

Perform `collector.collect()` and return a CollectReport.

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

pytest_itemcollected(*item: Item*) → None

We just collected a test item.

pytest_collectreport(*report: CollectReport*) → None

Collector finished collecting.

pytest_deselected(*items: Sequence[Item]*) → None

Called for deselected test items, e.g. by keyword.

May be called multiple times.

pytest_report_header(*config: Config*, *startpath: pathlib.Path*, *startdir: py._path.local.LocalPath*) → Union[str, List[str]]

Return a string or list of strings to be displayed as header info for terminal reporting.

> **Parameters**
>
> - **config** (`_pytest.config.Config`) – The pytest config object.
>
> - **startpath** (`Path`) – The starting dir.
>
> - **startdir** (`py.path.local`) – The starting dir.

Note: Lines returned by a plugin are displayed before those of plugins which ran before it. If you want to have your line(s) displayed first, use *trylast=True*.

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*.

Changed in version 6.3.0: The `startpath` parameter was added as a `pathlib.Path` equivalent of the `startdir` parameter.

pytest_report_collectionfinish(*config: Config, startpath: pathlib.Path, startdir: py._path.local.LocalPath, items: Sequence[Item]*) → Union[str, List[str]]
Return a string or list of strings to be displayed after collection has finished successfully.

These strings will be displayed after the standard "collected X items" message.

New in version 3.2.

> **Parameters**
> * **config** (`_pytest.config.Config`) – The pytest config object.
> * **startpath** (`Path`) – The starting path.
> * **startdir** (`py.path.local`) – The starting dir.
> * **items** – List of pytest items that are going to be executed; this list should not be modified.

Note: Lines returned by a plugin are displayed before those of plugins which ran before it. If you want to have your line(s) displayed first, use *trylast=True*.

Changed in version 6.3.0: The `startpath` parameter was added as a `pathlib.Path` equivalent of the `startdir` parameter.

pytest_report_teststatus(*report: Union[CollectReport, TestReport], config: Config*) → Tuple[str, str, Union[str, Mapping[str, bool]]]
Return result-category, shortletter and verbose word for status reporting.

The result-category is a category in which to count the result, for example "passed", "skipped", "error" or the empty string.

The shortletter is shown as testing progresses, for example ".", "s", "E" or the empty string.

The verbose word is shown as testing progresses in verbose mode, for example "PASSED", "SKIPPED", "ERROR" or the empty string.

pytest may style these implicitly according to the report outcome. To provide explicit styling, return a tuple for the verbose word, for example `"rerun"`, `"R"`, `("RERUN", {"yellow": True})`.

> **Parameters**
> * **report** – The report object whose status is to be returned.
> * **config** (`_pytest.config.Config`) – The pytest config object.

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

pytest_terminal_summary(*terminalreporter: TerminalReporter, exitstatus: ExitCode, config: Config*) → None
Add a section to terminal summary reporting.

Parameters

- **terminalreporter** (`_pytest.terminal.TerminalReporter`) – The internal terminal reporter object.

- **exitstatus** (`int`) – The exit status that will be reported back to the OS.

- **config** (`_pytest.config.Config`) – The pytest config object.

New in version 4.2: The `config` parameter.

pytest_fixture_setup (*fixturedef: FixtureDef[Any], request: SubRequest*) → Optional[object]
Perform fixture setup execution.

>   **Returns** The return value of the call to the fixture function.

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

Note: If the fixture function returns None, other implementations of this hook function will continue to be called, according to the behavior of the *firstresult: stop at first non-None result* option.

---

pytest_fixture_post_finalizer (*fixturedef: FixtureDef[Any], request: SubRequest*) → None
Called after fixture teardown, but before the cache is cleared, so the fixture result `fixturedef.cached_result` is still available (not None).

pytest_warning_captured (*warning_message: warnings.WarningMessage, when: Literal['config', 'collect', 'runtest'], item: Optional[Item], location: Optional[Tuple[str, int, str]]*) → None
(**Deprecated**) Process a warning captured by the internal pytest warnings plugin.

Deprecated since version 6.0.

This hook is considered deprecated and will be removed in a future pytest version. Use `pytest_warning_recorded()` instead.

>   Parameters
>
>   - **warning_message** (`warnings.WarningMessage`) – The captured warning. This is the same object produced by `warnings.catch_warnings()`, and contains the same attributes as the parameters of `warnings.showwarning()`.
>
>   - **when** (`str`) – Indicates when the warning was captured. Possible values:
>
>   – `"config"`: during pytest configuration/initialization stage.
>
>   – `"collect"`: during test collection.
>
>   – `"runtest"`: during test execution.
>
>   - **item** (`pytest.Item|None`) – The item being executed if `when` is `"runtest"`, otherwise `None`.
>
>   - **location** (`tuple`) – When available, holds information about the execution context of the captured warning (filename, linenumber, function). `function` evaluates to <module> when the execution context is at the module level.

pytest_warning_recorded (*warning_message: warnings.WarningMessage, when: Literal['config', 'collect', 'runtest'], nodeid: str, location: Optional[Tuple[str, int, str]]*) → None
Process a warning captured by the internal pytest warnings plugin.

>   Parameters

- **warning_message**(*warnings.WarningMessage*) – The captured warning. This is the same object produced by warnings.catch_warnings(), and contains the same attributes as the parameters of warnings.showwarning().

- **when**(*str*) – Indicates when the warning was captured. Possible values:

  - "config": during pytest configuration/initialization stage.

  - "collect": during test collection.

  - "runtest": during test execution.

- **nodeid**(*str*) – Full id of the item.

- **location**(*tuple|None*) – When available, holds information about the execution context of the captured warning (filename, linenumber, function). function evaluates to <module> when the execution context is at the module level.

New in version 6.0.

Central hook for reporting about test execution:

pytest_runtest_logreport(*report: TestReport*) → None
  Process the *_pytest.reports.TestReport* produced for each of the setup, call and teardown runtest phases of an item.

  See *pytest_runtest_protocol()* for a description of the runtest protocol.

Assertion related hooks:

pytest_assertrepr_compare(*config: Config*, *op: str*, *left: object*, *right: object*) → Optional[List[str]]
  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.

  Parameters config (*_pytest.config.Config*) – The pytest config object.

pytest_assertion_pass(*item: Item*, *lineno: int*, *orig: str*, *expl: str*) → None
  (**Experimental**) Called whenever an assertion passes.

  New in version 5.0.

  Use this hook to do some processing after a passing assertion. The original assertion information is available in the orig string and the pytest introspected assertion information is available in the expl string.

  This hook must be explicitly enabled by the enable_assertion_pass_hook ini-file option:

```
[pytest]
enable_assertion_pass_hook=true
```

  You need to **clean the .pyc** files in your project directory and interpreter libraries when enabling this option, as assertions will require to be re-written.

  Parameters

  - **item** (*pytest.Item*) – pytest item object of current test.

  - **lineno** (*int*) – Line number of the assert statement.

  - **orig** (*str*) – String with the original assertion.

  - **expl** (*str*) – String with the assert explanation.

Note: This hook is **experimental**, so its parameters or even the hook itself might be changed/removed without warning in any future pytest release.

If you find this hook useful, please share your feedback in an issue.

22.4.6 Debugging/Interaction hooks

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

pytest_internalerror(*excrepr: ExceptionRepr*, *excinfo: ExceptionInfo[BaseException]*) → Optional[bool]
Called for internal errors.

Return True to suppress the fallback handling of printing an INTERNALERROR message directly to sys.stderr.

pytest_keyboard_interrupt(*excinfo: ExceptionInfo[Union[KeyboardInterrupt, Exit]]*) → None
Called for keyboard interrupt.

pytest_exception_interact(*node: Union[Item, Collector]*, *call: CallInfo[Any]*, *report: Union[CollectReport, TestReport]*) → None
Called when an exception was raised which can potentially be interactively handled.

May be called during collection (see `pytest_make_collect_report()`), in which case `report` is a `_pytest.reports.CollectReport`.

May be called during runtest of an item (see `pytest_runtest_protocol()`), in which case `report` is a `_pytest.reports.TestReport`.

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

pytest_enter_pdb(*config: Config*, *pdb: pdb.Pdb*) → None
Called upon pdb.set_trace().

Can be used by plugins to take special action just before the python debugger enters interactive mode.

> **Parameters**
>> • **config** (`_pytest.config.Config`) – The pytest config object.
>> • **pdb** (`pdb.Pdb`) – The Pdb instance.

22.5 Objects

Full reference to objects accessible from *fixtures* or *hooks*.

22.5.1 CallInfo

final class CallInfo
Result/Exception info of a function invocation.

excinfo: Optional[_pytest._code.code.ExceptionInfo[BaseException]]
The captured exception of the call, if it raised.

start: float
The system time when the call started, in seconds since the epoch.

stop: `float`
>   The system time when the call ended, in seconds since the epoch.

duration: `float`
>   The call duration, in seconds.

when: `Literal['collect', 'setup', 'call', 'teardown']`
>   The context of invocation: "collect", "setup", "call" or "teardown".

result
>   The return value of the call, if it didn't raise.
>
>   Can only be accessed if excinfo is None.

classmethod from_call(*func: Callable[], TResult], when: Literal['collect', 'setup', 'call', 'teardown'], reraise: Optional[Union[Type[BaseException], Tuple[Type[BaseException], ...]]] = None*) → *CallInfo*[TResult]
>   Call func, wrapping the result in a CallInfo.
>
>   > **Parameters**
>   >
>   > - **func** – The function to call. Called without arguments.
>   >
>   > - **when** – The phase in which the function is called.
>   >
>   > - **reraise** – Exception or exceptions that shall propagate if raised by the function, instead of being wrapped in the CallInfo.

22.5.2 Class

class Class
>   Bases: `_pytest.python.PyCollector`
>
>   Collector for test methods.
>
>   **classmethod from_parent**(*parent*, ***, *name*, *obj=None*)
>   >   The public constructor.
>
>   **collect**() → Iterable[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
>   >   Return a list of children (items and collectors) for this collection node.

22.5.3 Collector

class Collector
>   Bases: *`_pytest.nodes.Node`*
>
>   Collector instances create children through collect() and thus iteratively build a tree.
>
>   **exception CollectError**
>   >   Bases: `Exception`
>   >
>   >   An error during collection, contains a custom message.
>
>   **collect**() → Iterable[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
>   >   Return a list of children (items and collectors) for this collection node.
>
>   **repr_failure**(*excinfo: _pytest._code.code.ExceptionInfo[BaseException]*) → Union[str, _pytest._code.code.TerminalRepr]
>   >   Return a representation of a collection failure.
>   >
>   >   > **Parameters** **excinfo** – Exception information for the failure.

22.5.4 CollectReport

final class CollectReport
    Bases: `_pytest.reports.BaseReport`

    Collection report object.

    nodeid: **str**
        Normalized collection nodeid.

    outcome
        Test outcome, always one of "passed", "failed", "skipped".

    longrepr: **Union[None, _pytest._code.code.ExceptionInfo[BaseException], Tuple[str, int**
        None or a failure representation.

    result
        The collected items and collection nodes.

    caplog
        Return captured log lines, if log capturing is enabled.

        New in version 3.5.

    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.

    count_towards_summary
        Experimental Whether this report should be counted towards the totals shown at the end of the test session: "1 passed, 1 failure, etc".

        Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

    head_line
        Experimental The head line shown with longrepr output for this report, more commonly during traceback representation during failures:

```
_____ Test.foo _____
```

        In the example above, the head_line is "Test.foo".

        Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

    longreprtext
        Read-only property that returns the full string representation of `longrepr`.

        New in version 3.0.

22.5.5 Config

final class Config
Access to configuration values, pluginmanager and plugin hooks.

> **Parameters**
>
> - **pluginmanager** (`PytestPluginManager`) –
>
> - **invocation_params** (`InvocationParams`) – Object containing parameters regarding the `pytest.main()` invocation.

final class InvocationParams(*args*, *plugins: Optional[Sequence[Union[str, object]]]*, *dir: pathlib.Path*)
Holds parameters passed during `pytest.main()`.

The object attributes are read-only.

New in version 5.1.

Note: Note that the environment variable PYTEST_ADDOPTS and the `addopts` ini option are handled by pytest, not being included in the `args` attribute.

Plugins accessing `InvocationParams` must be aware of that.

---

args
The command-line arguments as passed to `pytest.main()`.
> **Type** Tuple[str, ..]

plugins
Extra plugins, might be `None`.
> **Type** Optional[Sequence[Union[str, plugin]]]

dir
The directory from which `pytest.main()` was invoked.
> **Type** pathlib.Path

option
Access to command line option as attributes.

> **Type** argparse.Namespace

invocation_params
The parameters with which pytest was invoked.

> **Type** *InvocationParams*

pluginmanager
The plugin manager handles plugin registration and hook invocation.

> **Type** *PytestPluginManager*

invocation_dir
The directory from which pytest was invoked.

Prefer to use `invocation_params.dir`, which is a `pathlib.Path`.

> **Type** py.path.local

rootpath
The path to the *rootdir*.

> **Type** pathlib.Path

New in version 6.1.

rootdir
> The path to the *rootdir*.
>
> Prefer to use `rootpath`, which is a `pathlib.Path`.
>
> > **Type** py.path.local

inipath
> The path to the *configfile*.
>
> > **Type** Optional[pathlib.Path]
>
> New in version 6.1.

inifile
> The path to the *configfile*.
>
> Prefer to use `inipath`, which is a `pathlib.Path`.
>
> > **Type** Optional[py.path.local]

add_cleanup(*func: Callable[], None]*) → None
> Add a function to be called when the config object gets out of use (usually coninciding with pytest_unconfigure).

classmethod fromdictargs(*option_dict*, *args*) → *_pytest.config.Config*
> Constructor usable for subprocesses.

for ... in pytest_collection() → Generator[None, None, None]
> Validate invalid ini keys after collection is done so we take in account options added by late-loading conftest files.

issue_config_time_warning(*warning: Warning*, *stacklevel: int*) → None
> Issue and handle a warning during the "configure" stage.
>
> During `pytest_configure` we can't capture warnings using the `catch_warnings_for_item` function because it is not possible to have hookwrappers around `pytest_configure`.
>
> This function is mainly intended for plugins that need to issue warnings during `pytest_configure` (or similar stages).
>
> > **Parameters**
> >
> > - **warning** – The warning instance.
> > - **stacklevel** – stacklevel forwarded to warnings.warn.

addinivalue_line(*name: str*, *line: str*) → None
> 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 first line in its value.

getini(*name: str*)
> 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: str*, *default=<NOTSET>*, *skip: bool = 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: str*, *path=None*)
    Deprecated, use getoption() instead.

getvalueorskip(*name: str*, *path=None*)
    Deprecated, use getoption(skip=True) instead.

22.5.6 ExceptionInfo

final class ExceptionInfo(*excinfo: Optional[Tuple[Type[_E], _E, traceback]]*, *striptext: str = ''*, *traceback: Optional[_pytest._code.code.Traceback] = None*)
    Wraps sys.exc_info() objects and offers help for navigating the traceback.

classmethod from_exc_info(*exc_info: Tuple[Type[_E], _E, traceback]*, *exprinfo: Optional[str] = None*) → _pytest._code.code.ExceptionInfo[_E]
    Return an ExceptionInfo for an existing exc_info tuple.

> **Warning:** Experimental API

> **Parameters exprinfo** – A text string helping to determine if we should strip AssertionError from the output. Defaults to the exception message/`__str__()`.

classmethod from_current(*exprinfo: Optional[str] = None*) → _pytest._code.code.ExceptionInfo[BaseException]
    Return an ExceptionInfo matching the current traceback.

> **Warning:** Experimental API

> **Parameters exprinfo** – A text string helping to determine if we should strip AssertionError from the output. Defaults to the exception message/`__str__()`.

classmethod for_later() → _pytest._code.code.ExceptionInfo[_E]
    Return an unfilled ExceptionInfo.

fill_unfilled(*exc_info: Tuple[Type[_E], _E, traceback]*) → None
    Fill an unfilled ExceptionInfo created with `for_later()`.

type
    The exception class.

value
    The exception value.

tb
    The exception raw traceback.

typename
    The type name of the exception.

traceback
    The traceback.

exconly (*tryshort:* *bool* *= False*) → str
  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: Union[Type[BaseException], Tuple[Type[BaseException], ...]]*) → bool
  Return True if the exception is an instance of exc.

  Consider using `isinstance(excinfo.value, exc)` instead.

getrepr (*showlocals:* *bool* *= False*, *style:* *_TracebackStyle* *= 'long'*, *abspath:* *bool* *= False*, *tbfilter:* *bool* *= True*, *funcargs:* *bool* *= False*, *truncate_locals:* *bool* *= True*, *chain:* *bool* *= True*) → Union[ReprExceptionInfo, ExceptionChainRepr]
  Return str()able representation of this exception info.

  Parameters

  - **showlocals** (`bool`) – Show locals per traceback entry. Ignored if `style=="native"`.

  - **style** (`str`) – long|short|no|native|value traceback style.

  - **abspath** (`bool`) – If paths should be changed to absolute or left unchanged.

  - **tbfilter** (`bool`) – Hide entries that contain a local variable `__tracebackhide__==True`. Ignored if `style=="native"`.

  - **funcargs** (`bool`) – Show fixtures ("funcargs" for legacy purposes) per traceback entry.

  - **truncate_locals** (`bool`) – With `showlocals==True`, make sure locals can be safely represented as strings.

  - **chain** (`bool`) – If chained exceptions in Python 3 should be shown.

  Changed in version 3.9: Added the `chain` parameter.

match (*regexp: Union[str, Pattern[str]]*) → Literal[True]
  Check whether the regular expression `regexp` matches the string representation of the exception using `re.search()`.

  If it matches `True` is returned, otherwise an `AssertionError` is raised.

22.5.7 ExitCode

final class ExitCode (*value*)
  Encodes the valid exit codes by pytest.

  Currently users and plugins may supply other exit codes as well.

  New in version 5.0.

  OK = 0
    Tests passed.

  TESTS_FAILED = 1
    Tests failed.

  INTERRUPTED = 2
    pytest was interrupted.

  INTERNAL_ERROR = 3
    An internal error got in the way.

USAGE_ERROR = 4
> pytest was misused.

NO_TESTS_COLLECTED = 5
> pytest couldn't find tests.

22.5.8 File

class File
> Bases: *_pytest.nodes.FSCollector*

> Base class for collecting tests from a file.

> *Working with non-python tests*.

22.5.9 FixtureDef

final class FixtureDef
> Bases: Generic[_pytest.fixtures._FixtureValue]

> A container for a factory definition.

22.5.10 FSCollector

class FSCollector
> Bases: _pytest.nodes.Collector

> **classmethod from_parent** (*parent*, ***, *fspath*, ***kw*)
>> The public constructor.

22.5.11 Function

class Function
> Bases: _pytest.python.PyobjMixin, _pytest.nodes.Item

> An Item responsible for setting up and executing a Python test function.

> **param name:** The full function name, including any decorations like those added by parametrization (my_func[my_param]).

> **param parent:** The parent Node.

> **param config:** The pytest Config object.

> **param callspec:** If given, this is function has been parametrized and the callspec contains meta information about the parametrization.

> **param callobj:** If given, the object which will be called when the Function is invoked, otherwise the callobj will be obtained from parent using originalname.

> **param keywords:** Keywords bound to the function object for "-k" matching.

> **param session:** The pytest Session object.

> **param fixtureinfo:** Fixture information already resolved at this fixture node..

param originalname: The attribute name to use for accessing the underlying function object. Defaults to `name`. Set this if name is different from the original name, for example when it contains decorations like those added by parametrization (`my_func[my_param]`).

originalname
Original function name, without any decorations (for example parametrization adds a `"[...]"` suffix to function names), used to access the underlying function object from `parent` (in case `callobj` is not given explicitly).

New in version 3.0.

classmethod from_parent(*parent*, ***kw*)
The public constructor.

function
Underlying python 'function' object.

runtest() → None
Execute the underlying test function.

repr_failure(*excinfo:* _pytest._code.code.ExceptionInfo[*BaseException*]) → Union[str, _pytest._code.code.TerminalRepr]
Return a representation of a collection or test failure.

> **Parameters excinfo** – Exception information for the failure.

22.5.12 FunctionDefinition

class FunctionDefinition
Bases: `_pytest.python.Function`

This class is a step gap solution until we evolve to have actual function definition nodes and manage to get rid of `metafunc`.

runtest() → None
Execute the underlying test function.

setup() → None
Execute the underlying test function.

22.5.13 Item

class Item
Bases: `_pytest.nodes.Node`

A basic test invocation item.

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

user_properties: List[Tuple[str, object]]
A list of tuples (name, value) that holds user defined properties for this test.

add_report_section(*when: str*, *key: str*, *content: str*) → None
Add 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.

22.5.14 MarkDecorator

class MarkDecorator

A decorator for applying a mark on test functions and classes.

`MarkDecorators` are created with `pytest.mark`:

```
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` 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 the mark 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 the mark to the function, containing all the arguments already stored internally in the `MarkDecorator`.

3. When called in any other case, 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 a `MarkDecorator` from storing only a single function or class reference as its positional argument with no additional keyword or positional arguments. You can work around this by using `with_args()`.

name

Alias for mark.name.

args

Alias for mark.args.

kwargs

Alias for mark.kwargs.

with_args(**args: object*, ***kwargs: object*) → _pytest.mark.structures.MarkDecorator

Return a MarkDecorator with extra arguments added.

Unlike calling the MarkDecorator, with_args() can be used even if the sole argument is a callable/class.

22.5.15 MarkGenerator

final class MarkGenerator

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

Example:

```python
import pytest

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

applies a 'slowtest' *Mark* on `test_function`.

22.5.16 Mark

final class Mark

name: str

Name of the mark.

args: Tuple[Any, ...]

Positional arguments of the mark decorator.

kwargs: Mapping[str, Any]

Keyword arguments of the mark decorator.

combined_with(*other: _pytest.mark.structures.Mark*) → _pytest.mark.structures.Mark

Return a new Mark which is a combination of this Mark and another Mark.

Combines by appending args and merging kwargs.

> **Parameters other** (*Mark*) – The mark to combine with.

> **Return type** *Mark*

22.5.17 Metafunc

final class Metafunc

Objects 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.

definition

Access to the underlying *_pytest.python.FunctionDefinition*.

config

Access to the *_pytest.config.Config* object for the test session.

module

The module object where the test function is defined in.

function

Underlying Python test function.

fixturenames

Set of fixture names required by the test function.

cls
> Class object where the test function is defined in or `None`.

parametrize(*argnames: Union[str,   List[str],   Tuple[str,   …]],   argvalues: Iterable[Union[_pytest.mark.structures.ParameterSet, Sequence[object], object]], indirect: Union[bool, Sequence[str]] = False, ids: Optional[Union[Iterable[Union[None, str, float, int, bool]]], Callable[[Any], Optional[object]]]] = None, scope: Optional[_Scope] = None, *, _param_mark: Optional[_pytest.mark.structures.Mark] = None*) → 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** – A list of arguments' names (subset of argnames) or a boolean. 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** – Sequence of (or generator for) ids for `argvalues`, or a callable to return part of the id for each argvalue.
>
>   With sequences (and generators like `itertools.count()`) the returned ids should be of type `string`, `int`, `float`, `bool`, or `None`. They are mapped to the corresponding index in `argvalues`. `None` means to use the auto-generated id.
>
>   If it is a callable it will be called for each entry in `argvalues`, and the return value is used as part of the auto-generated id for the whole set (where parts are joined with dashes ("-")). This is useful to provide more specific ids for certain items, e.g. dates. Returning `None` will use an auto-generated id.
>
>   If no ids are provided they will be generated automatically from the argvalues.
>
> - **scope** – If specified it denotes the scope of the parameters. The scope is used for grouping tests by parameter instances. It will also override any fixture-function defined scope, allowing to set a dynamic scope using test context or configuration.

22.5.18 Module

class Module
> Bases: `_pytest.nodes.File, _pytest.python.PyCollector`
>
> Collector for test classes and functions.
>
> **collect**() → Iterable[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]
> > Return a list of children (items and collectors) for this collection node.

22.5.19 Node

class Node

Base class for Collector and Item, the components of the test collection tree.

Collector subclasses have children; Items are leaf nodes.

name

A unique name within the scope of the parent node.

parent

The parent collector node.

fspath

Filesystem path where this node was collected from (can be None).

keywords: MutableMapping[str, Any]

Keywords/markers collected from all scopes.

own_markers: List[_pytest.mark.structures.Mark]

The marker objects belonging to this node.

extra_keyword_matches: Set[str]

Allow adding of extra keywords to use for matching.

classmethod from_parent(*parent: _pytest.nodes.Node*, ***kw*)

Public constructor for Nodes.

This indirection got introduced in order to enable removing the fragile logic from the node constructors.

Subclasses can use super().from_parent(...) when overriding the construction.

> **Parameters parent** – The parent node of this Node.

ihook

fspath-sensitive hook proxy used to call pytest hooks.

warn(*warning: Warning*) → None

Issue a warning for this Node.

Warnings will be displayed after the test session, unless explicitly suppressed.

> **Parameters warning** (*Warning*) – The warning instance to issue.
>
> **Raises ValueError** – If warning instance is not a subclass of Warning.

Example usage:

```
node.warn(PytestWarning("some message"))
node.warn(UserWarning("some message"))
```

Changed in version 6.2: Any subclass of Warning is now accepted, rather than only *PytestWarning* subclasses.

nodeid

A ::-separated string denoting its collection tree address.

listchain() → List[*_pytest.nodes.Node*]

Return list of all parent collectors up to self, starting from the root of collection tree.

add_marker(*marker: Union[str, _pytest.mark.structures.MarkDecorator]*, *append: bool = True*) → None

Dynamically add a marker object to the node.

> **Parameters append** – Whether to append the marker, or prepend it.

iter_markers(*name: Optional[str] = None*) → Iterator[_pytest.mark.structures.Mark]
Iterate over all markers of the node.

>**Parameters name** – If given, filter the results by the name attribute.

for ... in iter_markers_with_node(*name:*   *Optional[str]*   =   *None*)   →
Iterator[Tuple[*_pytest.nodes.Node*,
_pytest.mark.structures.Mark]]
Iterate over all markers of the node.

>**Parameters name** – If given, filter the results by the name attribute.

>**Returns** An iterator of (node, mark) tuples.

get_closest_marker(*name: str*) → Optional[_pytest.mark.structures.Mark]
get_closest_marker(*name:*   *str*,   *default:*   *_pytest.mark.structures.Mark*)   →
_pytest.mark.structures.Mark
Return the first marker matching the name, from closest (for example function) to farther level (for example module level).

>**Parameters**

>>• **default** – Fallback return value if no marker was found.

>>• **name** – Name to filter by.

listextrakeywords() → Set[str]
Return a set of all extra keywords in self and any parents.

addfinalizer(*fin: Callable[], object]*) → None
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: Type[_NodeType]*) → Optional[_NodeType]
Get the next parent node (including self) which is an instance of the given class.

repr_failure(*excinfo:*   *_pytest._code.code.ExceptionInfo[BaseException]*,   *style:*   *Optional[_TracebackStyle] = None*) → Union[str, _pytest._code.code.TerminalRepr]
Return a representation of a collection or test failure.

>**Parameters excinfo** – Exception information for the failure.

22.5.20 Parser

final 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:*   *str*,   *description:*   *str*   =   *''*,   *after:*   *Optional[str] = None*)   →
_pytest.config.argparsing.OptionGroup
Get (or create) a named option Group.

>**Name** Name of the option group.

>**Description** Long description for –help output.

>**After** Name of another 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: str*, ***attrs: Any*) → None
 Register a command line option.

> **Opts** Option names, can be short or long options.
>
> **Attrs** Same attributes which the add_argument() 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: Sequence[Union[str, os.PathLike[str]]]*, *namespace: Optional[argparse.Namespace] = None*) → argparse.Namespace
 Parse and return a namespace object with known arguments at this point.

parse_known_and_unknown_args(*args: Sequence[Union[str, os.PathLike[str]]]*, *namespace: Optional[argparse.Namespace] = None*) → Tuple[argparse.Namespace, List[str]]
 Parse and return a namespace object with known arguments, and the remaining arguments unknown at this point.

addini(*name: str*, *help: str*, *type: Optional[Literal['string', 'pathlist', 'args', 'linelist', 'bool']] = None*, *default=None*) → None
 Register an ini-file option.

> **Name** Name of the ini-variable.
>
> **Type** Type of the variable, can be string, pathlist, args, linelist or bool. Defaults to string if None or not passed.
>
> **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)*.

22.5.21 PytestPluginManager

final class PytestPluginManager
 Bases: pluggy.manager.PluginManager

 A pluggy.PluginManager with additional pytest-specific functionality:

- Loading plugins from the command line, PYTEST_PLUGINS env variable and pytest_plugins global variables found in plugins being loaded.

- conftest.py loading during start-up.

parse_hookimpl_opts(*plugin: object*, *name: str*)

parse_hookspec_opts(*module_or_class*, *name: str*)

register(*plugin: object*, *name: Optional[str] = None*) → Optional[str]
 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.

getplugin(*name: str*)

hasplugin(*name: str*) → bool
 Return whether a plugin with the given name is registered.

consider_preparse(*args: Sequence[str]*, ***, *exclude_only: bool = False*) → None

consider_pluginarg(*arg: str*) → None

consider_conftest(*conftestmodule: module*) → None

consider_env() → None

consider_module(*mod: module*) → None

import_plugin(*modname: str*, *consider_entry_points: bool = False*) → None
 Import a plugin with `modname`.

 If `consider_entry_points` is True, entry point names are also considered to find a 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 *pluggy.callers._Result* object which represents the result of the overall hook call.

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.

check_pending()
 Verify that all hooks which have not been verified against a hook specification are optional, otherwise raise `PluginValidationError`.

enable_tracing()
 enable tracing of hook calls and return an undo function.

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_hookcallers(*plugin*)
 get all hook callers for the specified plugin.

get_name(*plugin*)
 Return name for registered plugin or `None` if not registered.

get_plugin(*name*)
 Return a plugin or `None` for the given name.

get_plugins()
 return the set of registered plugins.

has_plugin(*name*)
 Return `True` if a plugin with the given name is registered.

is_blocked(*name*)
 return `True` if the given plugin name is blocked.

is_registered(*plugin*)
 Return `True` if the plugin is already registered.

list_name_plugin()
 return list of name/plugin pairs.

list_plugin_distinfo()
 return list of distinfo/plugin tuples for all setuptools registered plugins.

load_setuptools_entrypoints(*group*, *name=None*)

Load modules from querying the specified setuptools `group`.

> **Parameters**
>
> - **group** (`str`) – entry point group to load plugins
>
> - **name** (`str`) – if given, loads only plugins with the given `name`.
>
> **Return type** int
>
> **Returns** return the number of loaded plugins by this call.

set_blocked(*name*)

block registrations of the given name, unregister if already registered.

subset_hook_caller(*name*, *remove_plugins*)

Return a new `hooks._HookCaller` instance for the named method which manages calls to all registered plugins except the ones from remove_plugins.

unregister(*plugin=None*, *name=None*)

unregister a plugin object and all its contained hook implementations from internal data structures.

22.5.22 Session

final class Session

Bases: `_pytest.nodes.FSCollector`

exception Interrupted

Bases: `KeyboardInterrupt`

Signals that the test run was interrupted.

exception Failed

Bases: `Exception`

Signals a stop as failed test run.

perform_collect(*args: Optional[Sequence[str]] = '...'*, *genitems: Literal[True] = '...'*) → Sequence[_pytest.nodes.Item]

perform_collect(*args: Optional[Sequence[str]] = '...'*, *genitems: bool = '...'*) → Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]

Perform the collection phase for this session.

This is called by the default *pytest_collection* hook implementation; see the documentation of this hook for more details. For testing purposes, it may also be called directly on a fresh `Session`.

This function normally recursively expands any collectors collected from the session to their items, and only items are returned. For testing purposes, this may be suppressed by passing `genitems=False`, in which case the return value contains these collectors unexpanded, and `session.items` is empty.

for ... in collect() → Iterator[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]

Return a list of children (items and collectors) for this collection node.

22.5.23 TestReport

final class TestReport

Bases: `_pytest.reports.BaseReport`

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

nodeid: `str`

Normalized collection nodeid.

location: `Optional[Tuple[str, Optional[int], str]]`

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

A name -> value dictionary containing all keywords and markers associated with a test invocation.

outcome

Test outcome, always one of "passed", "failed", "skipped".

longrepr: `Union[None, _pytest._code.code.ExceptionInfo[BaseException], Tuple[str, int`

None or a failure representation.

when: `Optional[str]`

One of 'setup', 'call', 'teardown' to indicate runtest phase.

user_properties

User properties is a list of tuples (name, value) that holds user defined properties of the test.

sections: `List[Tuple[str, str]]`

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

Time it took to run just the test.

classmethod from_item_and_call(*item: _pytest.nodes.Item*, *call: CallInfo[None]*) → *TestReport*

Create and fill a TestReport with standard item and call info.

caplog

Return captured log lines, if log capturing is enabled.

New in version 3.5.

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.

count_towards_summary

Experimental Whether this report should be counted towards the totals shown at the end of the test session: "1 passed, 1 failure, etc".

Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

---

head_line

Experimental The head line shown with longrepr output for this report, more commonly during traceback representation during failures:

```
_____ Test.foo _____
```

In the example above, the head_line is "Test.foo".

Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

---

longreprtext

Read-only property that returns the full string representation of `longrepr`.

New in version 3.0.

22.5.24 _Result

Result used within *hook wrappers*.

class _Result(*result*, *excinfo*)

_Result.**get_result**()

Get the result(s) for this hook call.

If the hook was marked as a `firstresult` only a single value will be returned otherwise a list of results.

_Result.**force_result**(*result*)

Force the result(s) to `result`.

If the hook was marked as a `firstresult` a single value should be set otherwise set a (modified) list of results. Any exceptions found during invocation will be deleted.

22.6 Global Variables

pytest treats some global variables in a special manner when defined in a test module or `conftest.py` files.

collect_ignore

Tutorial: *Customizing test collection*

Can be declared in *conftest.py files* to exclude test directories or modules. Needs to be `list[str]`.

```
collect_ignore = ["setup.py"]
```

collect_ignore_glob

Tutorial: *Customizing test collection*

Can be declared in *conftest.py files* to exclude test directories or modules with Unix shell-style wildcards. Needs to be `list[str]` where `str` can contain glob patterns.

```
collect_ignore_glob = ["*_ignore.py"]
```

pytest_plugins

Tutorial: *Requiring/Loading plugins in a test module or conftest file*

Can be declared at the **global** level in *test modules* and *conftest.py files* to register additional plugins. Can be either a
`str` or `Sequence[str]`.

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

```
pytest_plugins = ("myapp.testsupport.tools", "myapp.testsupport.regression")
```

pytestmark

Tutorial: *Marking whole classes or modules*

Can be declared at the **global** level in *test modules* to apply one or more *marks* to all test functions and methods. Can
be either a single mark or a list of marks (applied in left-to-right order).

```python
import pytest

pytestmark = pytest.mark.webtest
```

```python
import pytest

pytestmark = [pytest.mark.integration, pytest.mark.slow]
```

22.7 Environment Variables

Environment variables that can be used to change pytest's behavior.

PYTEST_ADDOPTS

This contains a command-line (parsed by the py:mod:`shlex` module) that will be **prepended** to the command line
given by the user, see *Builtin configuration file options* for more information.

PYTEST_CURRENT_TEST

This is not meant to be set by users, but is set by pytest internally with the name of the current test so other processes
can inspect it, see *PYTEST_CURRENT_TEST environment variable* for more information.

PYTEST_DEBUG

When set, pytest will print tracing and debug information.

PYTEST_DISABLE_PLUGIN_AUTOLOAD

When set, disables plugin auto-loading through setuptools entrypoints. Only explicitly specified plugins will be loaded.

PYTEST_PLUGINS

Contains comma-separated list of modules that should be loaded as plugins:

```
export PYTEST_PLUGINS=mymodule.plugin,xdist
```

PY_COLORS

When set to `1`, pytest will use color in terminal output. When set to `0`, pytest will not use color. `PY_COLORS` takes
precedence over `NO_COLOR` and `FORCE_COLOR`.

NO_COLOR

When set (regardless of value), pytest will not use color in terminal output. `PY_COLORS` takes precedence over `NO_COLOR`, which takes precedence over `FORCE_COLOR`. See no-color.org for other libraries supporting this community standard.

FORCE_COLOR

When set (regardless of value), pytest will use color in terminal output. `PY_COLORS` and `NO_COLOR` take precedence over `FORCE_COLOR`.

22.8 Exceptions

final class UsageError

> Bases: `Exception`

> Error in pytest usage or invocation.

22.9 Warnings

Custom warnings generated in some situations such as improper usage or deprecated features.

class PytestWarning

> Bases: `UserWarning`

> Base class for all warnings emitted by pytest.

class PytestAssertRewriteWarning

> Bases: `pytest.PytestWarning`

> Warning emitted by the pytest assert rewrite module.

class PytestCacheWarning

> Bases: `pytest.PytestWarning`

> Warning emitted by the cache plugin in various situations.

class PytestCollectionWarning

> Bases: `pytest.PytestWarning`

> Warning emitted when pytest is not able to collect a file or symbol in a module.

class PytestConfigWarning

> Bases: `pytest.PytestWarning`

> Warning emitted for configuration issues.

class PytestDeprecationWarning

> Bases: `pytest.PytestWarning`, `DeprecationWarning`

> Warning class for features that will be removed in a future version.

class PytestExperimentalApiWarning

> Bases: `pytest.PytestWarning`, `FutureWarning`

> Warning category used to denote experiments in pytest.

> Use sparingly as the API might change or even be removed completely in a future version.

class PytestUnhandledCoroutineWarning

> Bases: `pytest.PytestWarning`

> Warning emitted for an unhandled coroutine.

A coroutine was encountered when collecting test functions, but was not handled by any async-aware plugin. Coroutine test functions are not natively supported.

class PytestUnknownMarkWarning

Bases: *pytest.PytestWarning*

Warning emitted on use of unknown markers.

See *Marking test functions with attributes* for details.

class PytestUnraisableExceptionWarning

Bases: *pytest.PytestWarning*

An unraisable exception was reported.

Unraisable exceptions are exceptions raised in `__del__` implementations and similar situations when the exception cannot be raised as normal.

class PytestUnhandledThreadExceptionWarning

Bases: *pytest.PytestWarning*

An unhandled exception occurred in a `Thread`.

Such exceptions don't propagate normally.

Consult the *Internal pytest warnings* section in the documentation for more information.

22.10 Configuration Options

Here is a list of builtin configuration options that may be written in a `pytest.ini`, `pyproject.toml`, `tox.ini` or `setup.cfg` file, usually located at the root of your repository. To see each file format in details, see *Configuration file formats*.

> **Warning:** Usage of `setup.cfg` is not recommended except for very simple use cases. `.cfg` files use a different parser than `pytest.ini` and `tox.ini` which might cause hard to track down problems. When possible, it is recommended to use the latter files, or `pyproject.toml`, to hold your pytest configuration.

Configuration options may be overwritten in the command-line by using `-o/--override-ini`, which can also be passed multiple times. The expected format is `name=value`. For example:

```
pytest -o console_output_style=classic -o cache_dir=/tmp/mycache
```

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:

```
# content of pytest.ini
[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.

cache_dir

Sets a directory where stores content of cache plugin. Default directory is `.pytest_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*.

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.

console_output_style

Sets the console output style while running tests:

- `classic`: classic pytest output.

- `progress`: like classic pytest output, but with a progress indicator.

- `count`: like progress, but shows progress as the number of tests completed instead of a percent.

The default is `progress`, but you can fallback to `classic` if you prefer or the new mode is causing unexpected problems:

```
# content of pytest.ini
[pytest]
console_output_style = classic
```

doctest_encoding

Default encoding to use to decode text files with docstrings. *See how pytest handles doctests*.

doctest_optionflags

One or more doctest flag names from the standard `doctest` module. *See how pytest handles doctests*.

empty_parameter_set_mark

Allows to pick the action for empty parametersets in parameterization

- `skip` skips tests with an empty parameterset (default)

- `xfail` marks tests with an empty parameterset as xfail(run=False)

- `fail_at_collect` raises an exception if parametrize collects an empty parameter set

```
# content of pytest.ini
[pytest]
empty_parameter_set_mark = xfail
```

Note: The default value of this option is planned to change to `xfail` in future releases as this is considered less error prone, see #3155 for more details.

---

faulthandler_timeout

Dumps the tracebacks of all threads if a test takes longer than X seconds to run (including fixture setup and teardown). Implemented using the faulthandler.dump_traceback_later function, so all caveats there apply.

```
# content of pytest.ini
[pytest]
faulthandler_timeout=5
```

For more information please refer to *Fault Handler*.

---

filterwarnings

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.

```ini
# 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*.

junit_duration_report

New in version 4.1.

Configures how durations are recorded into the JUnit XML report:

- `total` (the default): duration times reported include setup, call, and teardown times.

- `call`: duration times reported include only call times, excluding setup and teardown.

```ini
[pytest]
junit_duration_report = call
```

junit_family

New in version 4.2.

Changed in version 6.1: Default changed to `xunit2`.

Configures the format of the generated JUnit XML file. The possible options are:

- `xunit1` (or `legacy`): produces old style output, compatible with the xunit 1.0 format.

- `xunit2`: produces xunit 2.0 style output, which should be more compatible with latest Jenkins versions. **This is the default**.

```ini
[pytest]
junit_family = xunit2
```

junit_logging

New in version 3.5.

Changed in version 5.4: `log`, `all`, `out-err` options added.

Configures if captured output should be written to the JUnit XML file. Valid values are:

- `log`: write only `logging` captured output.

- `system-out`: write captured `stdout` contents.

- `system-err`: write captured `stderr` contents.

- `out-err`: write both captured `stdout` and `stderr` contents.

- `all`: write captured `logging`, `stdout` and `stderr` contents.

- `no` (the default): no captured output is written.

```ini
[pytest]
junit_logging = system-out
```

junit_log_passing_tests
New in version 4.6.

If `junit_logging != "no"`, configures if the captured output should be written to the JUnit XML file for **passing** tests. Default is `True`.

```
[pytest]
junit_log_passing_tests = False
```

junit_suite_name
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
```

log_auto_indent
Allow selective auto-indentation of multiline log messages.

Supports command line option `--log-auto-indent [value]` and config option `log_auto_indent = [value]` to set the auto-indentation behavior for all logging.

[value] can be:

- True or "On" - Dynamically auto-indent multiline log messages
- False or "Off" or 0 - Do not auto-indent multiline log messages (the default behavior)
- [positive integer] - auto-indent multiline log messages by [value] spaces

```
[pytest]
log_auto_indent = False
```

Supports passing kwarg `extra={"auto_indent": [value]}` to calls to `logging.log()` to specify auto-indentation behavior for a specific entry in the log. `extra` kwarg overrides the value specified on the command line or in the config.

log_cli
Enable log display during test run (also known as *"live logging"*). The default is `False`.

```
[pytest]
log_cli = True
```

log_cli_date_format
Sets a `time.strftime()`-compatible string that will be used when formatting dates for live logging.

```
[pytest]
log_cli_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *Live Logs*.

log_cli_format
Sets a `logging`-compatible string used to format live logging messages.

```
[pytest]
log_cli_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *Live Logs*.

log_cli_level

Sets the minimum log message level that should be captured for live logging. The integer value or the names of the levels can be used.

```
[pytest]
log_cli_level = INFO
```

For more information, see *Live Logs*.

log_date_format

Sets a `time.strftime()`-compatible string that will be used when formatting dates for logging capture.

```
[pytest]
log_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *Logging*.

log_file

Sets a file name relative to the `pytest.ini` file where log messages should be written to, in addition to the other logging facilities that are active.

```
[pytest]
log_file = logs/pytest-logs.txt
```

For more information, see *Logging*.

log_file_date_format

Sets a `time.strftime()`-compatible string that will be used when formatting dates for the logging file.

```
[pytest]
log_file_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *Logging*.

log_file_format

Sets a `logging`-compatible string used to format logging messages redirected to the logging file.

```
[pytest]
log_file_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *Logging*.

log_file_level

Sets the minimum log message level that should be captured for the logging file. The integer value or the names of the levels can be used.

```
[pytest]
log_file_level = INFO
```

For more information, see *Logging*.

log_format

Sets a `logging`-compatible string used to format captured logging messages.

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *Logging*.

log_level

Sets the minimum log message level that should be captured for logging capture. The integer value or the names of the levels can be used.

```
[pytest]
log_level = INFO
```

For more information, see *Logging*.

markers

When the `--strict-markers` or `--strict` command-line arguments are used, only known markers - defined in code by core pytest or some plugin - are allowed.

You can list additional markers in this setting to add them to the whitelist, in which case you probably want to add `--strict-markers` to `addopts` to avoid future regressions:

```
[pytest]
addopts = --strict-markers
markers =
    slow
    serial
```

Note: The use of `--strict-markers` is highly preferred. `--strict` was kept for backward compatibility only and may be confusing for others as it only applies to markers and not to other options.

---

minversion

Specifies a minimal pytest version required for running tests.

```
# content of pytest.ini
[pytest]
minversion = 3.0  # will fail if we run with pytest-2.8
```

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 `'*.egg'`, `'.*'`, `'_darcs'`, `'build'`, `'CVS'`, `'dist'`, `'node_modules'`, `'venv'`, `'{arch}'`. Setting a `norecursedirs` replaces the default. Here is an example of how to avoid certain directories:

```
[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 `--collect-in-virtualenv` is given. Note also that `norecursedirs` takes precedence over `--collect-in-virtualenv`; 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 `--collect-in-virtualenv` flag.

python_classes

One or more name prefixes or glob-style patterns determining which classes are considered for test collection. Search for multiple glob patterns by adding a space between patterns. 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`:

```
[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_files

One or more Glob-style file patterns determining which python files are considered as test modules. Search for multiple glob patterns by adding a space between patterns:

```
[pytest]
python_files = test_*.py check_*.py example_*.py
```

Or one per line:

```
[pytest]
python_files =
    test_*.py
    check_*.py
    example_*.py
```

By default, files matching `test_*.py` and `*_test.py` will be considered test modules.

python_functions

One or more name prefixes or glob-patterns determining which test functions and methods are considered tests. Search for multiple glob patterns by adding a space between patterns. 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`:

```
[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.

required_plugins

A space separated list of plugins that must be present for pytest to run. Plugins can be listed with or without version specifiers directly following their name. Whitespace between different version specifiers is not allowed. If any one of the plugins is not found, emit an error.

```
[pytest]
required_plugins = pytest-django>=3.0.0,<4.0.0 pytest-html pytest-xdist>=1.0.0
```

testpaths

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.

```
[pytest]
testpaths = testing doc
```

This tells pytest to only look for tests in `testing` and `doc` directories when executing from the root directory.

usefixtures

List of fixtures that will be applied to all test functions; this is semantically the same to apply the `@pytest.mark.usefixtures` marker to all test functions.

```
[pytest]
usefixtures =
    clean_db
```

xfail_strict

If set to `True`, tests marked with `@pytest.mark.xfail` that actually succeed will by default fail the test suite. For more information, see *strict parameter*.

```
[pytest]
xfail_strict = True
```

22.11 Command-line Flags

All the command-line flags can be obtained by running `pytest --help`:

```
$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]

positional arguments:
  file_or_dir

general:
  -k EXPRESSION         only run tests which match the given substring
                        expression. An expression is a python evaluatable
                        expression where all names are substring-matched
                        against test names and their parent classes.
                        Example: -k 'test_method or test_other' matches all
                        test functions and classes whose name contains
                        'test_method' or 'test_other', while -k 'not
                        test_method' matches those that don't contain
                        'test_method' in their names. -k 'not test_method
                        and not test_other' will eliminate the matches.
                        Additionally keywords are matched to classes and
                        functions containing extra names in their
                        'extra_keyword_matches' set, as well as functions
                        which have names assigned directly to them. The
                        matching is case-insensitive.
  -m MARKEXPR           only run tests matching given mark expression.
                        For example: -m 'mark1 and not mark2'.
  --markers            show markers (builtin, plugin and per-project ones).
  -x, --exitfirst      exit instantly on first error or failed test.
  --fixtures, --funcargs
                        show available fixtures, sorted by plugin appearance
                        (fixtures with leading '_' are only shown with '-v')
  --fixtures-per-test   show fixtures per test
  --pdb                start the interactive Python debugger on errors or
                        KeyboardInterrupt.
  --pdbcls=modulename:classname
                        start a custom interactive Python debugger on
```

(continues on next page)

```
                        errors. For example:
                        --pdbcls=IPython.terminal.debugger:TerminalPdb
  --trace               Immediately break when running each test.
  --capture=method      per-test capturing method: one of fd|sys|no|tee-sys.
  -s                    shortcut for --capture=no.
  --runxfail            report the results of xfail tests as if they were
                        not marked
  --lf, --last-failed   rerun only the tests that failed at the last run (or
                        all if none failed)
  --ff, --failed-first  run all tests, but run the last failures first.
                        This may re-order tests and thus lead to repeated
                        fixture setup/teardown.
  --nf, --new-first     run tests from new files first, then the rest of the
                        tests sorted by file mtime
  --cache-show=[CACHESHOW]
                        show cache contents, don't perform collection or
                        tests. Optional argument: glob (default: '*').
  --cache-clear         remove all cache contents at start of test run.
  --lfnf={all,none}, --last-failed-no-failures={all,none}
                        which tests to run with no previously (known)
                        failures.
  --sw, --stepwise      exit on test failure and continue from last failing
                        test next time
  --sw-skip, --stepwise-skip
                        ignore the first failing test but stop on the next
                        failing test

reporting:
  --durations=N         show N slowest setup/test durations (N=0 for all).
  --durations-min=N     Minimal duration in seconds for inclusion in slowest
                        list. Default 0.005
  -v, --verbose         increase verbosity.
  --no-header           disable header
  --no-summary          disable summary
  -q, --quiet           decrease verbosity.
  --verbosity=VERBOSE   set verbosity. Default is 0.
  -r chars              show extra test summary info as specified by chars:
                        (f)ailed, (E)rror, (s)kipped, (x)failed, (X)passed,
                        (p)assed, (P)assed with output, (a)ll except passed
                        (p/P), or (A)ll. (w)arnings are enabled by default
                        (see --disable-warnings), 'N' can be used to reset
                        the list. (default: 'fE').
  --disable-warnings, --disable-pytest-warnings
                        disable warnings summary
  -l, --showlocals      show locals in tracebacks (disabled by default).
  --tb=style            traceback print mode
                        (auto/long/short/line/native/no).
  --show-capture={no,stdout,stderr,log,all}
                        Controls how captured stdout/stderr/log is shown on
                        failed tests. Default is 'all'.
  --full-trace          don't cut any tracebacks (default is to cut).
  --color=color         color terminal output (yes/no/auto).
  --code-highlight={yes,no}
                        Whether code should be highlighted (only if --color
                        is also enabled)
  --pastebin=mode       send failed|all info to bpaste.net pastebin service.
  --junit-xml=path      create junit-xml style report file at given path.
```

```
  --junit-prefix=str    prepend prefix to classnames in junit-xml output

pytest-warnings:
  -W PYTHONWARNINGS, --pythonwarnings=PYTHONWARNINGS
                        set which warnings to report, see -W option of
                        python itself.
  --maxfail=num         exit after first num failures or errors.
  --strict-config       any warnings encountered while parsing the `pytest`
                        section of the configuration file raise errors.
  --strict-markers      markers not registered in the `markers` section of
                        the configuration file raise errors.
  --strict              (deprecated) alias to --strict-markers.
  -c file               load configuration from `file` instead of trying to
                        locate one of the implicit configuration files.
  --continue-on-collection-errors
                        Force test execution even if collection errors
                        occur.
  --rootdir=ROOTDIR     Define root directory for tests. Can be relative
                        path: 'root_dir', './root_dir',
                        'root_dir/another_dir/'; absolute path:
                        '/home/user/root_dir'; path with variables:
                        '$HOME/root_dir'.

collection:
  --collect-only, --co  only collect tests, don't execute them.
  --pyargs              try to interpret all arguments as python packages.
  --ignore=path         ignore path during collection (multi-allowed).
  --ignore-glob=path    ignore path pattern during collection (multi-
                        allowed).
  --deselect=nodeid_prefix
                        deselect item (via node id prefix) during collection
                        (multi-allowed).
  --confcutdir=dir      only load conftest.py's relative to specified dir.
  --noconftest          Don't load any conftest.py files.
  --keep-duplicates     Keep duplicate tests.
  --collect-in-virtualenv
                        Don't ignore tests in a local virtualenv directory
  --import-mode={prepend,append,importlib}
                        prepend/append to sys.path when importing test
                        modules and conftest files, default is to prepend.
  --doctest-modules     run doctests in all .py modules
  --doctest-report={none,cdiff,ndiff,udiff,only_first_failure}
                        choose another output format for diffs on doctest
                        failure
  --doctest-glob=pat    doctests file matching pattern, default: test*.txt
  --doctest-ignore-import-errors
                        ignore doctest ImportErrors
  --doctest-continue-on-failure
                        for a given doctest, continue to run after the first
                        failure

test session debugging and configuration:
  --basetemp=dir        base temporary directory for this test run.(warning:
                        this directory is removed if it exists)
  -V, --version         display pytest version and information about
                        plugins.When given twice, also display information
                        about plugins.
```

```
  -h, --help            show help message and configuration info
  -p name               early-load given plugin module name or entry point
                        (multi-allowed).
                        To avoid loading of plugins, use the `no:` prefix,
                        e.g. `no:doctest`.
  --trace-config        trace considerations of conftest.py files.
  --debug               store internal tracing debug information in
                        'pytestdebug.log'.
  -o OVERRIDE_INI, --override-ini=OVERRIDE_INI
                        override ini option with "option=value" style, e.g.
                        `-o xfail_strict=True -o cache_dir=cache`.
  --assert=MODE         Control assertion debugging tools.
                        'plain' performs no assertion debugging.
                        'rewrite' (the default) rewrites assert statements
                        in test modules on import to provide assert
                        expression information.
  --setup-only          only setup fixtures, do not execute tests.
  --setup-show          show setup of fixtures while executing tests.
  --setup-plan          show what fixtures and tests would be executed but
                        don't execute anything.

logging:
  --log-level=LEVEL     level of messages to catch/display.
                        Not set by default, so it depends on the root/parent
                        log handler's effective level, where it is "WARNING"
                        by default.
  --log-format=LOG_FORMAT
                        log format as used by the logging module.
  --log-date-format=LOG_DATE_FORMAT
                        log date format as used by the logging module.
  --log-cli-level=LOG_CLI_LEVEL
                        cli logging level.
  --log-cli-format=LOG_CLI_FORMAT
                        log format as used by the logging module.
  --log-cli-date-format=LOG_CLI_DATE_FORMAT
                        log date format as used by the logging module.
  --log-file=LOG_FILE   path to a file when logging will be written to.
  --log-file-level=LOG_FILE_LEVEL
                        log file logging level.
  --log-file-format=LOG_FILE_FORMAT
                        log format as used by the logging module.
  --log-file-date-format=LOG_FILE_DATE_FORMAT
                        log date format as used by the logging module.
  --log-auto-indent=LOG_AUTO_INDENT
                        Auto-indent multiline messages passed to the logging
                        module. Accepts true|on, false|off or an integer.

[pytest] ini-options in the first pytest.ini|tox.ini|setup.cfg file found:

  markers (linelist):   markers for test functions
  empty_parameter_set_mark (string):
                        default marker for empty parametersets
  norecursedirs (args): directory patterns to avoid for recursion
  testpaths (args):     directories to search for tests when no files or
                        directories are given in the command line.
  filterwarnings (linelist):
                        Each line specifies a pattern for
```

```
                      warnings.filterwarnings. Processed after
                      -W/--pythonwarnings.
usefixtures (args):   list of default fixtures to be used with this
                      project
python_files (args):  glob-style file patterns for Python test module
                      discovery
python_classes (args):
                      prefixes or glob names for Python test class
                      discovery
python_functions (args):
                      prefixes or glob names for Python test function and
                      method discovery
disable_test_id_escaping_and_forfeit_all_rights_to_community_support (bool):
                      disable string escape non-ascii characters, might
                      cause unwanted side effects(use at your own risk)
console_output_style (string):
                      console output: "classic", or with additional
                      progress information ("progress" (percentage) |
                      "count").
xfail_strict (bool):  default for the strict parameter of xfail markers
                      when not given explicitly (default: False)
enable_assertion_pass_hook (bool):
                      Enables the pytest_assertion_pass hook.Make sure to
                      delete any previously generated pyc cache files.
junit_suite_name (string):
                      Test suite name for JUnit report
junit_logging (string):
                      Write captured log messages to JUnit report: one of
                      no|log|system-out|system-err|out-err|all
junit_log_passing_tests (bool):
                      Capture log information for passing tests to JUnit
                      report:
junit_duration_report (string):
                      Duration time to report: one of total|call
junit_family (string):
                      Emit XML for schema: one of legacy|xunit1|xunit2
doctest_optionflags (args):
                      option flags for doctests
doctest_encoding (string):
                      encoding used for doctest files
cache_dir (string):   cache directory path.
log_level (string):   default value for --log-level
log_format (string):  default value for --log-format
log_date_format (string):
                      default value for --log-date-format
log_cli (bool):       enable log display during test run (also known as
                      "live logging").
log_cli_level (string):
                      default value for --log-cli-level
log_cli_format (string):
                      default value for --log-cli-format
log_cli_date_format (string):
                      default value for --log-cli-date-format
log_file (string):    default value for --log-file
log_file_level (string):
                      default value for --log-file-level
log_file_format (string):
```

```
                        default value for --log-file-format
  log_file_date_format (string):
                        default value for --log-file-date-format
  log_auto_indent (string):
                        default value for --log-auto-indent
  faulthandler_timeout (string):
                        Dump the traceback of all threads if a test takes
                        more than TIMEOUT seconds to finish.
  addopts (args):       extra command line options
  minversion (string):  minimally required pytest version
  required_plugins (args):
                        plugins that must be present for pytest to run


environment variables:
  PYTEST_ADDOPTS           extra command line options
  PYTEST_PLUGINS           comma-separated plugins to load during startup
  PYTEST_DISABLE_PLUGIN_AUTOLOAD set to disable plugin auto-loading
  PYTEST_DEBUG             set to enable debug tracing of pytest's internals



to see available markers type: pytest --markers
to see available fixtures type: pytest --fixtures
(shown according to specified file_or_dir or current dir if not specified; fixtures␣
↪with leading '_' are only shown with the '-v' option
```

TWENTYTHREE

GOOD INTEGRATION PRACTICES

23.1 Install package with pip

For development, we recommend you use venv for virtual 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 your system Python installation.

Next, place a `setup.py` file in the root of your package with the following minimum content:

```python
from setuptools import setup, find_packages

setup(name="PACKAGENAME", packages=find_packages())
```

Where `PACKAGENAME` is the name of your package. You can then install your package in "editable" mode by running from the same directory:

```
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.

23.2 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.

23.3 Choosing a test layout / import rules

`pytest` supports two common test layouts:

23.3.1 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 has the following benefits:

- Your tests can run against an installed version after executing `pip install ..`
- Your tests can run against the local copy with an editable install after executing `pip install --editable ..`
- If you don't have a `setup.py` file and are relying on the fact that Python by default puts the current directory in `sys.path` to import your package, you can execute `python -m pytest` to execute the tests against the local copy directly, without using `pip`.

Note: See *Invoking pytest versus python -m pytest* for more information about the difference between calling `pytest` and `python -m pytest`.

Note that this scheme has a drawback if you are using `prepend` *import mode* (which is the default): 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 Măries.

Note: The new `--import-mode=importlib` (see *Import modes*) doesn't have any of the drawbacks above because `sys.path` and `sys.modules` are not changed when importing test modules, so users that run into this issue are strongly encouraged to try it and report if the new option works well for them.

The `src` directory layout is still strongly recommended however.

23.3.2 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: In prepend and append import-modes, 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.

With --import-mode=importlib things are less convoluted because pytest doesn't need to change sys.path or sys.modules, making things much less surprising.

23.4 tox

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.

FLAKY TESTS

A "flaky" test is one that exhibits intermittent or sporadic failure, that seems to have non-deterministic behaviour. Sometimes it passes, sometimes it fails, and it's not clear why. This page discusses pytest features that can help and other general strategies for identifying, fixing or mitigating them.

24.1 Why flaky tests are a problem

Flaky tests are particularly troublesome when a continuous integration (CI) server is being used, so that all tests must pass before a new code change can be merged. If the test result is not a reliable signal – that a test failure means the code change broke the test – developers can become mistrustful of the test results, which can lead to overlooking genuine failures. It is also a source of wasted time as developers must re-run test suites and investigate spurious failures.

24.2 Potential root causes

24.2.1 System state

Broadly speaking, a flaky test indicates that the test relies on some system state that is not being appropriately controlled - the test environment is not sufficiently isolated. Higher level tests are more likely to be flaky as they rely on more state.

Flaky tests sometimes appear when a test suite is run in parallel (such as use of pytest-xdist). This can indicate a test is reliant on test ordering.

- Perhaps a different test is failing to clean up after itself and leaving behind data which causes the flaky test to fail.

- The flaky test is reliant on data from a previous test that doesn't clean up after itself, and in parallel runs that previous test is not always present

- Tests that modify global state typically cannot be run in parallel.

24.2.2 Overly strict assertion

Overly strict assertions can cause problems with floating point comparison as well as timing issues. pytest.approx is useful here.

24.3 Pytest features

24.3.1 Xfail strict

pytest.mark.xfail with `strict=False` can be used to mark a test so that its failure does not cause the whole build to break. This could be considered like a manual quarantine, and is rather dangerous to use permanently.

24.3.2 PYTEST_CURRENT_TEST

PYTEST_CURRENT_TEST may be useful for figuring out "which test got stuck". See *PYTEST_CURRENT_TEST environment variable* for more details.

24.3.3 Plugins

Rerunning any failed tests can mitigate the negative effects of flaky tests by giving them additional chances to pass, so that the overall build does not fail. Several pytest plugins support this:

- flaky
- pytest-flakefinder - blog post
- pytest-rerunfailures
- pytest-replay: This plugin helps to reproduce locally crashes or flaky tests observed during CI runs.

Plugins to deliberately randomize tests can help expose tests with state problems:

- pytest-random-order
- pytest-randomly

24.4 Other general strategies

24.4.1 Split up test suites

It can be common to split a single test suite into two, such as unit vs integration, and only use the unit test suite as a CI gate. This also helps keep build times manageable as high level tests tend to be slower. However, it means it does become possible for code that breaks the build to be merged, so extra vigilance is needed for monitoring the integration test results.

24.4.2 Video/screenshot on failure

For UI tests these are important for understanding what the state of the UI was when the test failed. pytest-splinter can be used with plugins like pytest-bdd and can save a screenshot on test failure, which can help to isolate the cause.

24.4.3 Delete or rewrite the test

If the functionality is covered by other tests, perhaps the test can be removed. If not, perhaps it can be rewritten at a lower level which will remove the flakiness or make its source more apparent.

24.4.4 Quarantine

Mark Lapierre discusses the Pros and Cons of Quarantined Tests in a post from 2018.

24.4.5 CI tools that rerun on failure

Azure Pipelines (the Azure cloud CI/CD tool, formerly Visual Studio Team Services or VSTS) has a feature to identify flaky tests and rerun failed tests.

24.5 Research

This is a limited list, please submit an issue or pull request to expand it!

- Gao, Zebao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. "Making system user interactive tests repeatable: When and what should we control?." In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 55-65. IEEE, 2015. PDF

- Palomba, Fabio, and Andy Zaidman. "Does refactoring of test smells induce fixing flaky tests?." In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pp. 1-12. IEEE, 2017. PDF in Google Drive

- Bell, Jonathan, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. "De-Flaker: Automatically detecting flaky tests." In *Proceedings of the 2018 International Conference on Software Engineering*. 2018. PDF

24.6 Resources

- Eradicating Non-Determinism in Tests by Martin Fowler, 2011

- No more flaky tests on the Go team by Pavan Sudarshan, 2012

- The Build That Cried Broken: Building Trust in your Continuous Integration Tests talk (video) by Angie Jones at SeleniumConf Austin 2017

- Test and Code Podcast: Flaky Tests and How to Deal with Them by Brian Okken and Anthony Shaw, 2018

- Microsoft:

    - How we approach testing VSTS to enable continuous delivery by Brian Harry MS, 2017

    - Eliminating Flaky Tests blog and talk (video) by Munil Shah, 2017

- Google:

– Flaky Tests at Google and How We Mitigate Them by John Micco, 2016
– Where do Google's flaky tests come from? by Jeff Listfield, 2017

PYTEST IMPORT MECHANISMS AND `SYS.PATH/PYTHONPATH`

25.1 Import modes

pytest as a testing framework needs to import test modules and `conftest.py` files for execution.

Importing files in Python (at least until recently) is a non-trivial processes, often requiring changing sys.path. Some aspects of the import process can be controlled through the `--import-mode` command-line flag, which can assume these values:

- `prepend` (default): the directory path containing each module will be inserted into the *beginning* of `sys.path` if not already there, and then imported with the __import__ builtin.

  This requires test module names to be unique when the test directory tree is not arranged in packages, because the modules will put in `sys.modules` after importing.

  This is the classic mechanism, dating back from the time Python 2 was still supported.

- `append`: the directory containing each module is appended to the end of `sys.path` if not already there, and imported with __import__.

  This better allows to run test modules against installed versions of a package even if the package under test has the same import root. For example:

  ```
  testing/__init__.py
  testing/test_pkg_under_test.py
  pkg_under_test/
  ```

  the tests will run against the installed version of `pkg_under_test` when `--import-mode=append` is used whereas with `prepend` they would pick up the local version. This kind of confusion is why we advocate for using *src* layouts.

  Same as `prepend`, requires test module names to be unique when the test directory tree is not arranged in packages, because the modules will put in `sys.modules` after importing.

- `importlib`: new in pytest-6.0, this mode uses importlib to import test modules. This gives full control over the import process, and doesn't require changing sys.path or sys.modules at all.

  For this reason this doesn't require test module names to be unique at all, but also makes test modules non-importable by each other. This was made possible in previous modes, for tests not residing in Python packages, because of the side-effects of changing sys.path and sys.modules mentioned above. Users which require this should turn their tests into proper packages instead.

  We intend to make `importlib` the default in future releases.

25.2 `prepend` and `append` import modes scenarios

Here's a list of scenarios when using `prepend` or `append` import modes where pytest needs to change `sys.path` in order to import test modules or `conftest.py` files, and the issues users might encounter because of that.

25.2.1 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*.

25.2.2 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*.

25.3 Invoking `pytest` versus `python -m pytest`

Running pytest with `pytest [...]` instead of `python -m pytest [...]` yields nearly equivalent behaviour, except that the latter will add the current directory to `sys.path`, which is standard `python` behavior.

See also *Calling pytest through python -m pytest*.

CONFIGURATION

26.1 Command line options and configuration file settings

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

```
pytest -h   # prints options _and_ config file settings
```

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

26.2 Configuration file formats

Many *pytest settings* can be set in a *configuration file*, which by convention resides on the root of your repository or in your tests folder.

A quick example of the configuration files supported by pytest:

26.2.1 pytest.ini

`pytest.ini` files take precedence over other files, even when empty.

```
# pytest.ini
[pytest]
minversion = 6.0
addopts = -ra -q
testpaths =
    tests
    integration
```

26.2.2 pyproject.toml

New in version 6.0.

`pyproject.toml` are considered for configuration when they contain a `tool.pytest.ini_options` table.

```
# pyproject.toml
[tool.pytest.ini_options]
minversion = "6.0"
addopts = "-ra -q"
testpaths = [
```

```
    "tests",
    "integration",
]
```

Note: One might wonder why `[tool.pytest.ini_options]` instead of `[tool.pytest]` as is the case with other tools.

The reason is that the pytest team intends to fully utilize the rich TOML data format for configuration in the future, reserving the `[tool.pytest]` table for that. The `ini_options` table is being used, for now, as a bridge between the existing `.ini` configuration system and the future configuration format.

26.2.3 tox.ini

`tox.ini` files are the configuration files of the tox project, and can also be used to hold pytest configuration if they have a `[pytest]` section.

```
# tox.ini
[pytest]
minversion = 6.0
addopts = -ra -q
testpaths =
    tests
    integration
```

26.2.4 setup.cfg

`setup.cfg` files are general purpose configuration files, used originally by distutils, and can also be used to hold pytest configuration if they have a `[tool:pytest]` section.

```
# setup.cfg
[tool:pytest]
minversion = 6.0
addopts = -ra -q
testpaths =
    tests
    integration
```

Warning: Usage of `setup.cfg` is not recommended unless for very simple use cases. `.cfg` files use a different parser than `pytest.ini` and `tox.ini` which might cause hard to track down problems. When possible, it is recommended to use the latter files, or `pyproject.toml`, to hold your pytest configuration.

26.3 Initialization: determining rootdir and configfile

pytest determines a `rootdir` for each test run which depends on the command line arguments (specified test files, paths) and on the existence of configuration files. The determined `rootdir` and `configfile` 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 into account the 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 `.pytest_cache` subdirectory in `rootdir` to store its cross-test run state.

`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.

The `--rootdir=path` command-line option can be used to force a specific directory. Note that contrary to other command-line options, `--rootdir` cannot be used with *addopts* inside `pytest.ini` because the `rootdir` is used to *find* `pytest.ini` already.

26.3.1 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`, `pyproject.toml`, `tox.ini`, and `setup.cfg` files in the ancestor directory and upwards. If one is matched, it becomes the `configfile` and its directory becomes the `rootdir`.

- If no configuration 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`, `pyproject.toml`, `tox.ini`, and `setup.cfg` in each of the specified `args` and upwards. If one is matched, it becomes the `configfile` and its directory becomes the `rootdir`.

- If no `configfile` 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 configuration file.

If no `args` are given, pytest collects test below the current working directory and also starts determining the `rootdir` from there.

Files will only be matched for configuration if:

- `pytest.ini`: will always match and take precedence, even if empty.

- `pyproject.toml`: contains a `[tool.pytest.ini_options]` table.

- `tox.ini`: contains a `[pytest]` section.

- `setup.cfg`: contains a `[tool:pytest]` section.

The files are considered in the order above. Options from multiple `configfiles` candidates are never merged - the first match wins.

The internal *Config* object (accessible via hooks or through the *pytestconfig* fixture) will subsequently carry these attributes:

- *config.rootpath*: the determined root directory, guaranteed to exist.

- *config.inipath*: the determined `configfile`, may be `None` (it is named `inipath` for historical reasons).

New in version 6.1: The `config.rootpath` and `config.inipath` properties. They are `pathlib.Path` versions of the older `config.rootdir` and `config.inifile`, which have type `py.path.local`, and still exist for backward compatibility.

The `rootdir` is used as 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 configuration files as follows:

```
# first look for pytest.ini files
path/pytest.ini
path/pyproject.toml  # must contain a [tool.pytest.ini_options] table to match
path/tox.ini         # must contain [pytest] section to match
path/setup.cfg       # must contain [tool:pytest] section to match
pytest.ini
... # all the way up to the root

# now look for setup.py
path/setup.py
setup.py
... # all the way up to the root
```

> **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.

26.4 Builtin configuration file options

For the full list of options consult the *reference documentation*.

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
- *Fixtures* 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.

27.1 Demo of Python failure reports with pytest

Here is a nice run of several failures and how `pytest` presents things:

```
assertion $ pytest failure_demo.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/assertion
collected 44 items

failure_demo.py FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF          [100%]

================================= FAILURES =================================
_____ test_generative[3-6] _____

param1 = 3, param2 = 6

    @pytest.mark.parametrize("param1, param2", [(3, 6)])
    def test_generative(param1, param2):
>       assert param1 * 2 < param2
E       assert (3 * 2) < 6

failure_demo.py:19: 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:30: 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:33:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 42, b = 54

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

failure_demo.py:14: 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         - eggs
E         + spam
```

```
failure_demo.py:44: 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 2 bar
E         ?     ^
E         + foo 1 bar
E         ?     ^

failure_demo.py:47: 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         - eggs
E         + spam
E           bar

failure_demo.py:50: 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' == '111111111111...
→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         - 1111111111b222222222
E         ?           ^
E         + 1111111111a222222222
E         ?           ^

failure_demo.py:55: 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\n...n2\
→n2\n2\n2\n'
E         Skipping 190 identical leading characters in diff, use -v to show
E         Skipping 191 identical trailing characters in diff, use -v to show
```

```
E               1
E               1
E               1
E               1
E                 1...
E
E           ...Full output truncated (7 lines hidden), use '-vv' to show

failure_demo.py:60: 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:63: 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:68: 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 1 more item:
E         {'c': 0}
E         Right contains 1 more item:
E         {'d': 0}...
E
E           ...Full output truncated (2 lines hidden), use '-vv' to show

failure_demo.py:71: AssertionError
_____ TestSpecialisedExplanations.test_eq_set _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_set(self):
>       assert {0, 10, 11, 12} == {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:74: 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 one more item: 3
E         Use -v to get the full diff

failure_demo.py:77: 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:80: 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 multil...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:84: 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:88: 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 h...l 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:92: 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 'ffffffffffff...ffffffffffff' not in 'head head h...l␣
→tail tail '
E         'fffffffffffffffffff...ffffffffffffffffffff' is contained here:
E           head head␣
→ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail tail␣
→tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail␣
→tail tail
E         ?           ␣
→+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

failure_demo.py:96: AssertionError
_____ TestSpecialisedExplanations.test_eq_dataclass _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_dataclass(self):
        from dataclasses import dataclass

        @dataclass
        class Foo:
            a: int
            b: str

        left = Foo(1, "b")
        right = Foo(1, "c")
>       assert left == right
E       AssertionError: assert TestSpecialis...oo(a=1, b='b') == TestSpecialis...
→oo(a=1, b='c')
E
```

```
E           Omitting 1 identical items, use -vv to show
E           Differing attributes:
E           ['b']
E
E           Drill down into differing attribute b:
E             b: 'b' != 'c'...
E
E           ...Full output truncated (3 lines hidden), use '-vv' to show

failure_demo.py:108: AssertionError
_____ TestSpecialisedExplanations.test_eq_attrs _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_attrs(self):
        import attr

        @attr.s
        class Foo:
            a = attr.ib()
            b = attr.ib()

        left = Foo(1, "b")
        right = Foo(1, "c")
>       assert left == right
E       AssertionError: assert Foo(a=1, b='b') == Foo(a=1, b='c')
E
E           Omitting 1 identical items, use -vv to show
E           Differing attributes:
E           ['b']
E
E           Drill down into differing attribute b:
E             b: 'b' != 'c'...
E
E           ...Full output truncated (3 lines hidden), use '-vv' to show

failure_demo.py:120: AssertionError
_____ test_attribute _____

    def test_attribute():
        class Foo:
            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:128: AssertionError
_____ test_attribute_instance _____

    def test_attribute_instance():
        class Foo:
            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:135: AssertionError
_____ test_attribute_failure _____

    def test_attribute_failure():
        class Foo:
            def _get_b(self):
                raise Exception("Failed to get attrib")

            b = property(_get_b)

        i = Foo()
>       assert i.b == 2

failure_demo.py:146:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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:141: Exception
_____ test_attribute_multiple _____

    def test_attribute_multiple():
        class Foo:
            b = 1

        class Bar:
            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:156: AssertionError
_____ TestRaises.test_raises _____

self = <failure_demo.TestRaises object at 0xdeadbeef>

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

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

failure_demo.py:166: ValueError
_____ TestRaises.test_raises_doesnt _____

self = <failure_demo.TestRaises object at 0xdeadbeef>

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

failure_demo.py:169: 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:172: ValueError
_____ TestRaises.test_tupleerror _____

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_tupleerror(self):
>       a, b = [1]  # NOQA
E       ValueError: not enough values to unpack (expected 2, got 1)

failure_demo.py:175: 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):
        items = [1, 2, 3]
        print(f"items is {items!r}")
>       a, b = items.pop()
E       TypeError: cannot unpack non-iterable int object

failure_demo.py:180: TypeError
-------------------------- Captured stdout call ---------------------------
items is [1, 2, 3]
_____ TestRaises.test_some_error _____

self = <failure_demo.TestRaises object at 0xdeadbeef>

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

failure_demo.py:183: NameError
_____ test_dynamic_compile_shows_nicely _____

    def test_dynamic_compile_shows_nicely():
        import importlib.util
        import sys
```

```
        src = "def foo():\n assert 1 == 0\n"
        name = "abc-123"
        spec = importlib.util.spec_from_loader(name, loader=None)
        module = importlib.util.module_from_spec(spec)
        code = compile(src, name, "exec")
        exec(code, module.__dict__)
        sys.modules[name] = module
>       module.foo()

failure_demo.py:202:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

>   ???
E   AssertionError

abc-123:2: 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:213:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
failure_demo.py:10: 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):
        items = []
>       a, b = items
E       ValueError: not enough values to unpack (expected 2, got 0)

failure_demo.py:217: ValueError
_____ TestMoreErrors.test_z2_type_error _____

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

```
    def test_z2_type_error(self):
        items = 3
>       a, b = items
E       TypeError: cannot unpack non-iterable int object

failure_demo.py:221: 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:226: 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:235: 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:238: 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:242: 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:245: 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:250: AssertionError
_____ TestCustomAssertMsg.test_single_line _____

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    def test_single_line(self):
        class A:
            a = 1

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

failure_demo.py:261: AssertionError
_____ TestCustomAssertMsg.test_multiline _____

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    def test_multiline(self):
        class A:
            a = 1

        b = 2
>       assert (
            A.a == b
```

```
        ), "A.a appears not to be b\nor 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:268: AssertionError
_____ TestCustomAssertMsg.test_custom_repr _____

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    def test_custom_repr(self):
        class JSON:
            a = 1

            def __repr__(self):
                return "This is JSON\n{\n  'foo': 'bar'\n}"

        a = JSON()
        b = 2
>       assert a.a == b, a
E       AssertionError: This is JSON
E         {
E           'foo': 'bar'
E         }
E       assert 1 == 2
E        +  where 1 = This is JSON\n{\n  'foo': 'bar'\n}.a

failure_demo.py:281: AssertionError
========================= short test summary info =========================
FAILED failure_demo.py::test_generative[3-6] - assert (3 * 2) < 6
FAILED failure_demo.py::TestFailing::test_simple - assert 42 == 43
FAILED failure_demo.py::TestFailing::test_simple_multiline - assert 42 == 54
FAILED failure_demo.py::TestFailing::test_not - assert not 42
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_text - Asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_similar_text
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_multiline_text
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_long_text - ...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_long_text_multiline
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_list - asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_list_long - ...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_dict - Asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_set - Assert...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_longer_list
FAILED failure_demo.py::TestSpecialisedExplanations::test_in_list - asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_multiline
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_single
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_single_long
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_single_long_term
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_dataclass - ...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_attrs - Asse...
FAILED failure_demo.py::test_attribute - assert 1 == 2
FAILED failure_demo.py::test_attribute_instance - AssertionError: assert ...
FAILED failure_demo.py::test_attribute_failure - Exception: Failed to get...
FAILED failure_demo.py::test_attribute_multiple - AssertionError: assert ...
```

```
FAILED failure_demo.py::TestRaises::test_raises - ValueError: invalid lit...
FAILED failure_demo.py::TestRaises::test_raises_doesnt - Failed: DID NOT ...
FAILED failure_demo.py::TestRaises::test_raise - ValueError: demo error
FAILED failure_demo.py::TestRaises::test_tupleerror - ValueError: not eno...
FAILED failure_demo.py::TestRaises::test_reinterpret_fails_with_print_for_the_fun_of_
↪it
FAILED failure_demo.py::TestRaises::test_some_error - NameError: name 'na...
FAILED failure_demo.py::test_dynamic_compile_shows_nicely - AssertionError
FAILED failure_demo.py::TestMoreErrors::test_complex_error - assert 44 == 43
FAILED failure_demo.py::TestMoreErrors::test_z1_unpack_error - ValueError...
FAILED failure_demo.py::TestMoreErrors::test_z2_type_error - TypeError: c...
FAILED failure_demo.py::TestMoreErrors::test_startswith - AssertionError:...
FAILED failure_demo.py::TestMoreErrors::test_startswith_nested - Assertio...
FAILED failure_demo.py::TestMoreErrors::test_global_func - assert False
FAILED failure_demo.py::TestMoreErrors::test_instance - assert 42 != 42
FAILED failure_demo.py::TestMoreErrors::test_compare - assert 11 < 5
FAILED failure_demo.py::TestMoreErrors::test_try_finally - assert 1 == 0
FAILED failure_demo.py::TestCustomAssertMsg::test_single_line - Assertion...
FAILED failure_demo.py::TestCustomAssertMsg::test_multiline - AssertionEr...
FAILED failure_demo.py::TestCustomAssertMsg::test_custom_repr - Assertion...
=========================== 44 failed in 0.12s ===========================
```

27.2 Basic patterns and examples

27.2.1 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:

```ini
# content of pytest.ini
[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_ADDOPTS <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`.

27.2.2 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:

```python
# 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*:

```python
# 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                                                                        [100%]
================================= 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
========================= short test summary info =========================
FAILED test_sample.py::test_answer - assert 0
1 failed in 0.12s
```

And now with supplying a command line option:

```
$ pytest -q --cmdopt=type2
F                                                                        [100%]
================================= FAILURES =================================
_____ test_answer _____
```

(continues on next page)

```
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
========================= short test summary info =========================
FAILED test_sample.py::test_answer - assert 0
1 failed in 0.12s
```

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.

27.2.3 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:

```python
# setuptools plugin
import sys


def pytest_load_initial_conftests(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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 0 items

========================= no tests ran in 0.12s ===========================
```

27.2.4 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:

```python
# content of conftest.py

import pytest


def pytest_addoption(parser):
    parser.addoption(
        "--runslow", action="store_true", default=False, help="run slow tests"
    )


def pytest_configure(config):
    config.addinivalue_line("markers", "slow: mark test as slow to run")


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:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py .s                                                    [100%]

========================= short test summary info ==========================
SKIPPED [1] test_module.py:8: need --runslow option to run
======================= 1 passed, 1 skipped in 0.12s =======================
```

Or run it including the `slow` marked test:

```
$ pytest --runslow
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py ..                                                    [100%]

============================ 2 passed in 0.12s =============================
```

27.2.5 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:

```python
# content of test_checkconfig.py
import pytest


def checkconfig(x):
    __tracebackhide__ = True
    if not hasattr(x, "config"):
        pytest.fail("not configured: {}".format(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                                                                    [100%]
================================= FAILURES =================================
_____ test_something _____

    def test_something():
>       checkconfig(42)
E       Failed: not configured: 42

test_checkconfig.py:11: Failed
========================= short test summary info ==========================
FAILED test_checkconfig.py::test_something - Failed: not configured: 42
1 failed in 0.12s
```

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:

```python
import operator
import pytest
```

(continues on next page)

```python
class ConfigException(Exception):
    pass


def checkconfig(x):
    __tracebackhide__ = operator.methodcaller("errisinstance", ConfigException)
    if not hasattr(x, "config"):
        raise ConfigException("not configured: {}".format(x))


def test_something():
    checkconfig(42)
```

This will avoid hiding the exception traceback on unrelated exceptions (i.e. bugs in assertion helpers).

27.2.6 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:

```python
# content of your_module.py


_called_from_test = False
```

```python
# content of conftest.py


def pytest_configure(config):
    your_module._called_from_test = True
```

and then check for the `your_module._called_from_test` flag:

```python
if your_module._called_from_test:
    # called from within a test run
    ...
else:
    # called "normally"
    ...
```

accordingly in your application.

27.2.7 Adding info to test report header

It's easy to present extra information in a `pytest` run:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
project deps: mylib-1.1
rootdir: $REGENDOC_TMPDIR
collected 0 items


========================= no tests ran in 0.12s =========================
```

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:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
info1: did you know that ...
did you?
rootdir: $REGENDOC_TMPDIR
collecting ... collected 0 items


========================= no tests ran in 0.12s =========================
```

and nothing when run plainly:

```
$ pytest
========================= test session starts =========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 0 items


========================= no tests ran in 0.12s =========================
```

27.2.8 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:

```python
# content of test_some_are_slow.py
import time


def test_funcfast():
```

(continues on next page)

```
    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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 3 items

test_some_are_slow.py ...                                            [100%]

=========================== slowest 3 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 0.12s =============================
```

27.2.9 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:

```python
# content of conftest.py

from typing import Dict, Tuple
import pytest

# store history of failures per test class name and per index in parametrize (if
↪parametrize used)
_test_failed_incremental: Dict[str, Dict[Tuple[int, ...], str]] = {}


def pytest_runtest_makereport(item, call):
    if "incremental" in item.keywords:
        # incremental marker is used
        if call.excinfo is not None:
            # the test has failed
            # retrieve the class name of the test
            cls_name = str(item.cls)
            # retrieve the index of the test (if parametrize is used in combination
↪with incremental)
            parametrize_index = (
                tuple(item.callspec.indices.values())
                if hasattr(item, "callspec")
```

```python
                else ()
            )
            # retrieve the name of the test function
            test_name = item.originalname or item.name
            # store in _test_failed_incremental the original name of the failed test
            _test_failed_incremental.setdefault(cls_name, {}).setdefault(
                parametrize_index, test_name
            )


def pytest_runtest_setup(item):
    if "incremental" in item.keywords:
        # retrieve the class name of the test
        cls_name = str(item.cls)
        # check if a previous test has failed for this class
        if cls_name in _test_failed_incremental:
            # retrieve the index of the test (if parametrize is used in combination
→with incremental)
            parametrize_index = (
                tuple(item.callspec.indices.values())
                if hasattr(item, "callspec")
                else ()
            )
            # retrieve the name of the first test function to fail for this class
→name and index
            test_name = _test_failed_incremental[cls_name].get(parametrize_index,
→None)
            # if name found, test has failed for the combination of class name & test
→name
            if test_name is not None:
                pytest.xfail("previous test failed ({})".format(test_name))
```

These two hook implementations work together to abort incremental-marked tests in a class. Here is a test module example:

```python
# content of test_step.py

import pytest


@pytest.mark.incremental
class TestUserHandling:
    def test_login(self):
        pass

    def test_modification(self):
        assert 0

    def test_deletion(self):
        pass


def test_normal():
    pass
```

If we run this:

```
$ pytest -rx
========================= test session starts =========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items

test_step.py .Fx.                                                  [100%]

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

self = <test_step.TestUserHandling object at 0xdeadbeef>

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

test_step.py:11: AssertionError
========================= short test summary info =====================
XFAIL test_step.py::TestUserHandling::test_deletion
  reason: previous test failed (test_modification)
================== 1 failed, 2 passed, 1 xfailed in 0.12s ==============
```

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

27.2.10 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:

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


class DB:
    pass


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

and then a test module in that directory:

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

another test module:

```python
# 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:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 7 items

test_step.py .Fx.                                                    [ 57%]
a/test_db.py F                                                       [ 71%]
a/test_db2.py F                                                      [ 85%]
b/test_error.py E                                                    [100%]


================================== 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, capfdbinary, caplog, capsys, capsysbinary,␣
→doctest_namespace, monkeypatch, pytestconfig, record_property, record_testsuite_
→property, record_xml_attribute, recwarn, tmp_path, tmp_path_factory, 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:11: 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 _____
```

(continues on next page)

```
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
======================= short test summary info =========================
FAILED test_step.py::TestUserHandling::test_modification - assert 0
FAILED a/test_db.py::test_a1 - AssertionError: <conftest.DB object at 0x7...
FAILED a/test_db2.py::test_a2 - AssertionError: <conftest.DB object at 0x...
ERROR b/test_error.py::test_root
============= 3 failed, 2 passed, 1 xfailed, 1 error in 0.12s ==============
```

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).

27.2.11 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:

```python
# 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 = " ({})".format(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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py FF                                                    [100%]

================================= 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:6: AssertionError
========================= short test summary info ==========================
FAILED test_module.py::test_fail1 - assert 0
FAILED test_module.py::test_fail2 - assert 0
========================== 2 failed in 0.12s ===============================
```

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
```

27.2.12 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:

```python
# 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:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
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:7: AssertionError
=============================== FAILURES ===============================
_____ test_call_fails _____

something = None

    def test_call_fails(something):
>       assert 0
E       assert 0

test_module.py:15: AssertionError
_____ test_fail2 _____

    def test_fail2():
>       assert 0
E       assert 0

test_module.py:19: AssertionError
========================= short test summary info =========================
FAILED test_module.py::test_call_fails - assert 0
FAILED test_module.py::test_fail2 - assert 0
ERROR test_module.py::test_setup_fails - assert 0
======================= 2 failed, 1 error in 0.12s =======================
```

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

27.2.13 `PYTEST_CURRENT_TEST` environment variable

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 happens only sporadically, the famous "flaky" kind of tests.

pytest sets the *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:

```python
import psutil
```

```python
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, or 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.

27.2.14 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. Please note that the mechanism for plugin discovery used by pytest (setuptools entry points) doesn't work with frozen executables so pytest can't find any third party plugins automatically. To include third party plugins like pytest-timeout they must be imported explicitly and passed on to pytest.main.

```python
# contents of app_main.py
import sys
import pytest_timeout  # Third party plugin

if len(sys.argv) > 1 and sys.argv[1] == "--pytest":
    import pytest

    sys.exit(pytest.main(sys.argv[2:], plugins=[pytest_timeout]))
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/
```

27.3 Parametrizing tests

`pytest` allows to easily parametrize test functions. For basic docs, see *Parametrizing fixtures and test functions*.

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

27.3.1 Generating parameters combinations, depending on command line

Let's say we want to execute a test with different computation parameters and the parameter range shall be determined by a command line argument. Let's first write a simple (do-nothing) computation test:

```python
# content of test_compute.py


def test_compute(param1):
    assert param1 < 4
```

Now we add a test configuration like this:

```python
# 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
..                                                                       [100%]
2 passed in 0.12s
```

We run only two computations, so we see two dots. let's run the full monty:

```
$ pytest -q --all
....F                                                                    [100%]
================================= FAILURES =================================
_____ test_compute[4] _____

param1 = 4

    def test_compute(param1):
>       assert param1 < 4
E       assert 4 < 4
```

(continues on next page)

```
test_compute.py:4: AssertionError
========================= short test summary info =========================
FAILED test_compute.py::test_compute[4] - assert 4 < 4
1 failed, 4 passed in 0.12s
```

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

27.3.2 Different options for test IDs

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

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

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
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]>

======================== 8 tests collected in 0.12s ========================
```

In `test_timedistance_v3`, we used `pytest.param` to specify the test IDs together with the actual data, instead of listing them separately.

27.3.3 A quick port of "testscenarios"

Here is a quick port to run tests configured with test scenarios, an add-on from Robert Collins for the standard unittest framework. We only have to work a bit to construct the correct arguments for pytest's `Metafunc.parametrize()`:

```
# content of test_scenarios.py


def pytest_generate_tests(metafunc):
```

```python
    idlist = []
    argvalues = []
    for scenario in metafunc.cls.scenarios:
        idlist.append(scenario[0])
        items = scenario[1].items()
        argnames = [x[0] for x in items]
        argvalues.append([x[1] for x in items])
    metafunc.parametrize(argnames, argvalues, ids=idlist, scope="class")


scenario1 = ("basic", {"attribute": "value"})
scenario2 = ("advanced", {"attribute": "value2"})


class TestSampleWithScenarios:
    scenarios = [scenario1, scenario2]

    def test_demo1(self, attribute):
        assert isinstance(attribute, str)

    def test_demo2(self, attribute):
        assert isinstance(attribute, str)
```

this is a fully self-contained example which you can run with:

```
$ pytest test_scenarios.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items

test_scenarios.py ....                                               [100%]

============================ 4 passed in 0.12s =============================
```

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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items

<Module test_scenarios.py>
  <Class TestSampleWithScenarios>
      <Function test_demo1[basic]>
      <Function test_demo2[basic]>
      <Function test_demo1[advanced]>
      <Function test_demo2[advanced]>

======================= 4 tests collected in 0.12s =======================
```

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.

27.3.4 Deferring the setup of parametrized resources

The parametrization of test functions happens at collection time. It is a good idea to setup expensive resources like DB connections or subprocess only when the actual test is run. Here is a simple example how you can achieve that. This test requires a `db` object fixture:

```python
# 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:

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


def pytest_generate_tests(metafunc):
    if "db" in metafunc.fixturenames:
        metafunc.parametrize("db", ["d1", "d2"], indirect=True)


class DB1:
    "one database object"


class DB2:
    "alternative database object"


@pytest.fixture
def db(request):
    if request.param == "d1":
        return DB1()
    elif request.param == "d2":
        return DB2()
    else:
        raise ValueError("invalid internal test config")
```

Let's first see how it looks like at collection time:

```
$ pytest test_backends.py --collect-only
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

<Module test_backends.py>
  <Function test_db_initialized[d1]>
  <Function test_db_initialized[d2]>
```

(continues on next page)

```
======================= 2 tests collected in 0.12s =======================
```

And then when we run the test:

```
$ pytest -q test_backends.py
.F                                                                     [100%]
================================= 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:8: Failed
========================= short test summary info ==========================
FAILED test_backends.py::test_db_initialized[d2] - Failed: deliberately f...
1 failed, 1 passed in 0.12s
```

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.

27.3.5 Indirect parametrization

Using the `indirect=True` parameter when parametrizing a test allows to parametrize a test with a fixture receiving the values before passing them to a test:

```python
import pytest


@pytest.fixture
def fixt(request):
    return request.param * 3


@pytest.mark.parametrize("fixt", ["a", "b"], indirect=True)
def test_indirect(fixt):
    assert len(fixt) == 3
```

This can be used, for example, to do more expensive setup at test run time in the fixture, rather than having to run those setup steps at collection time.

27.3.6 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:

```python
# 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 -v test_indirect_list.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 1 item

test_indirect_list.py::test_indirect[a-b] PASSED                         [100%]

============================= 1 passed in 0.12s ============================
```

27.3.7 Parametrizing test methods through per-class configuration

Here is an example `pytest_generate_tests` function implementing a parametrization scheme similar to Michael Foord's unittest parametrizer but in a lot less code:

```python
# 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(
```

(continues on next page)

```
        argnames, [[funcargs[name] for name in argnames] for funcargs in funcarglist]
    )


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

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

    def test_zerodivision(self, a, b):
        with 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..                                                                      [100%]
================================= 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:21: AssertionError
========================= short test summary info =========================
FAILED test_parametrize.py::TestClass::test_equals[1-2] - assert 1 == 2
1 failed, 2 passed in 0.12s
```

27.3.8 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 shutil
import subprocess
import textwrap
```

```python
import pytest

pythonlist = ["python3.5", "python3.6", "python3.7"]


@pytest.fixture(params=pythonlist)
def python1(request, tmpdir):
    picklefile = tmpdir.join("data.pickle")
    return Python(request.param, picklefile)


@pytest.fixture(params=pythonlist)
def python2(request, python1):
    return Python(request.param, python1.picklefile)


class Python:
    def __init__(self, version, picklefile):
        self.pythonpath = shutil.which(version)
        if not self.pythonpath:
            pytest.skip(f"{version!r} not found")
        self.picklefile = picklefile

    def dumps(self, obj):
        dumpfile = self.picklefile.dirpath("dump.py")
        dumpfile.write(
            textwrap.dedent(
                r"""
                import pickle
                f = open({!r}, 'wb')
                s = pickle.dump({!r}, f, protocol=2)
                f.close()
                """.format(
                    str(self.picklefile), obj
                )
            )
        )
        subprocess.check_call((self.pythonpath, str(dumpfile)))

    def load_and_is_true(self, expression):
        loadfile = self.picklefile.dirpath("load.py")
        loadfile.write(
            textwrap.dedent(
                r"""
                import pickle
                f = open({!r}, 'rb')
                obj = pickle.load(f)
                f.close()
                res = eval({!r})
                if not res:
                    raise SystemExit(1)
                """.format(
                    str(self.picklefile), expression
                )
            )
        )
```

```
        print(loadfile)
        subprocess.check_call((self.pythonpath, str(loadfile)))


@pytest.mark.parametrize("obj", [42, {}, {1: 3}])
def test_basic_objects(python1, python2, obj):
    python1.dumps(obj)
    python2.load_and_is_true(f"obj == {obj}")
```

Running it results in some skips if we don't have all the python interpreters installed and otherwise runs all combinations (3 interpreters times 3 interpreters times 3 objects to serialize/deserialize):

```
. $ pytest -rs -q multipython.py
ssssssssssss...ssssssssssss                                     [100%]
========================= short test summary info =========================
SKIPPED [12] multipython.py:29: 'python3.5' not found
SKIPPED [12] multipython.py:29: 'python3.7' not found
3 passed, 24 skipped in 0.12s
```

27.3.9 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py .s                                                    [100%]

========================= short test summary info ==========================
SKIPPED [1] conftest.py:12: could not import 'opt2': No module named 'opt2'
======================= 1 passed, 1 skipped in 0.12s =======================
```

You'll see that we don't have an `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.

27.3.10 Set marks or test ID for individual parametrized test

Use `pytest.param` to apply marks or set test ID to individual parametrized test. For example:

```
# content of test_pytest_param_example.py
import pytest


@pytest.mark.parametrize(
    "test_input,expected",
    [
        ("3+5", 8),
        pytest.param("1+7", 8, marks=pytest.mark.basic),
        pytest.param("2+4", 6, marks=pytest.mark.basic, id="basic_2+4"),
        pytest.param(
            "6*9", 42, marks=[pytest.mark.basic, pytest.mark.xfail], id="basic_6*9"
        ),
    ],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

In this example, we have 4 parametrized tests. Except for the first test, we mark the rest three parametrized tests with the custom marker `basic`, and for the fourth test we also use the built-in mark `xfail` to indicate this test is expected to fail. For explicitness, we set test ids for some tests.

Then run `pytest` with verbose mode and with only the `basic` marker:

```
$ pytest -v -m basic
========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 14 items / 11 deselected / 3 selected

test_pytest_param_example.py::test_eval[1+7-8] PASSED                  [ 33%]
test_pytest_param_example.py::test_eval[basic_2+4] PASSED             [ 66%]
test_pytest_param_example.py::test_eval[basic_6*9] XFAIL              [100%]


=============== 2 passed, 11 deselected, 1 xfailed in 0.12s ================
```

As the result:

- Four tests were collected

- One test was deselected because it doesn't have the `basic` mark.

- Three tests with the `basic` mark was selected.

- The test `test_eval[1+7-8]` passed, but the name is autogenerated and confusing.

- The test `test_eval[basic_2+4]` passed.

- The test `test_eval[basic_6*9]` was expected to fail and did fail.

27.3.11 Parametrizing conditional raising

Use *pytest.raises()* with the *pytest.mark.parametrize* decorator to write parametrized tests in which some tests raise exceptions and others do not.

It is helpful to define a no-op context manager `does_not_raise` to serve as a complement to `raises`. For example:

```python
from contextlib import contextmanager
import pytest


@contextmanager
def does_not_raise():
    yield


@pytest.mark.parametrize(
    "example_input,expectation",
    [
        (3, does_not_raise()),
        (2, does_not_raise()),
        (1, does_not_raise()),
        (0, pytest.raises(ZeroDivisionError)),
    ],
)
def test_division(example_input, expectation):
    """Test how much I know division."""
```

```
    with expectation:
        assert (6 / example_input) is not None
```

In the example above, the first three test cases should run unexceptionally, while the fourth should raise `ZeroDivisionError`.

If you're only supporting Python 3.7+, you can simply use `nullcontext` to define `does_not_raise`:

```python
from contextlib import nullcontext as does_not_raise
```

Or, if you're supporting Python 3.3+ you can use:

```python
from contextlib import ExitStack as does_not_raise
```

Or, if desired, you can `pip install contextlib2` and use:

```python
from contextlib2 import nullcontext as does_not_raise
```

27.4 Working with custom markers

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

27.4.1 Marking test functions and selecting them for a run

You can "mark" a test function with custom metadata like this:

```python
# content of test_server.py

import pytest


@pytest.mark.webtest
def test_send_http():
    pass  # perform some webtest test for your app


def test_something_quick():
    pass


def test_another():
    pass


class TestClass:
    def test_method(self):
        pass
```

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-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
↪PREFIX/bin/python
```

```
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 3 deselected / 1 selected

test_server.py::test_send_http PASSED                                    [100%]

==================== 1 passed, 3 deselected in 0.12s =====================
```

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-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 1 deselected / 3 selected

test_server.py::test_something_quick PASSED                              [ 33%]
test_server.py::test_another PASSED                                      [ 66%]
test_server.py::TestClass::test_method PASSED                            [100%]

==================== 3 passed, 1 deselected in 0.12s =====================
```

27.4.2 Selecting tests based on their node ID

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

```
$ pytest -v test_server.py::TestClass::test_method
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED                            [100%]

============================ 1 passed in 0.12s =============================
```

You can also select on the class:

```
$ pytest -v test_server.py::TestClass
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED                            [100%]

============================ 1 passed in 0.12s =============================
```

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-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 2 items

test_server.py::TestClass::test_method PASSED                            [ 50%]
test_server.py::test_send_http PASSED                                    [100%]


========================== 2 passed in 0.12s =============================
```

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`.

27.4.3 Using `-k` `expr` to select tests based on their name

New in version 2.0/2.3.4.

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:

Changed in version 5.4.

The expression matching is now case-insensitive.

```
$ pytest -v -k http  # running with the above defined example module
========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 3 deselected / 1 selected

test_server.py::test_send_http PASSED                                    [100%]


===================== 1 passed, 3 deselected in 0.12s =====================
```

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-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 1 deselected / 3 selected

test_server.py::test_something_quick PASSED                              [ 33%]
```

```
test_server.py::test_another PASSED                              [ 66%]
test_server.py::TestClass::test_method PASSED                    [100%]


===================== 3 passed, 1 deselected in 0.12s =====================
```

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

```
$ pytest -k "http or quick" -v
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 2 deselected / 2 selected

test_server.py::test_send_http PASSED                            [ 50%]
test_server.py::test_something_quick PASSED                      [100%]


===================== 2 passed, 2 deselected in 0.12s =====================
```

You can use and, or, not and parentheses.

In addition to the test's name, -k also matches the names of the test's parents (usually, the name of the file and class it's in), attributes set on the test function, markers applied to it or its parents and any *extra keywords* explicitly added to it or its parents.

27.4.4 Registering markers

Registering markers for your test suite is simple:

```
# content of pytest.ini
[pytest]
markers =
    webtest: mark a test as a webtest.
    slow: mark test as slow.
```

Multiple custom markers can be registered, by defining each one in its own line, as shown in above example.

You can ask which markers exist for your test suite - the list includes our just defined webtest and slow markers:

```
$ pytest --markers
@pytest.mark.webtest: mark a test as a webtest.

@pytest.mark.slow: mark test as slow.

@pytest.mark.filterwarnings(warning): add a warning filter to the given test. see
→https://docs.pytest.org/en/stable/warnings.html#pytest-mark-filterwarnings

@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, ..., *, reason=...): skip the given test function if
→any of the conditions evaluate to True. Example: skipif(sys.platform == 'win32')
→skips the test if we are on the win32 platform. See https://docs.pytest.org/en/
→stable/reference.html#pytest-mark-skipif
```

```
@pytest.mark.xfail(condition, ..., *, reason=..., run=True, raises=None, strict=xfail_
↪strict): mark the test function as an expected failure if any of the conditions
↪evaluate to True. 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 https://docs.pytest.org/en/stable/reference.html
↪#pytest-mark-xfail

@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 https://docs.pytest.org/en/stable/parametrize.html for more info and
↪examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all
↪of the specified fixtures. see https://docs.pytest.org/en/stable/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-markers` option.

27.4.5 Marking whole classes or modules

You may use `pytest.mark` decorators with classes to apply markers to all of its test methods:

```python
# content of test_mark_classlevel.py
import pytest


@pytest.mark.webtest
class TestClass:
    def test_startup(self):
        pass

    def test_startup_and_more(self):
        pass
```

This is equivalent to directly applying the decorator to the two test functions.

To apply marks at the module level, use the `pytestmark` global variable:

```python
import pytest

pytestmark = pytest.mark.webtest
```

or multiple markers:

```python
pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]
```

Due to legacy reasons, before class decorators were introduced, it is possible to set the *pytestmark* attribute on a test class like this:

```python
import pytest


class TestClass:
    pytestmark = pytest.mark.webtest
```

27.4.6 Marking individual tests when using parametrize

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

```python
import pytest


@pytest.mark.foo
@pytest.mark.parametrize(
    ("n", "expected"), [(1, 2), pytest.param(1, 3, marks=pytest.mark.bar), (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*.

27.4.7 Custom marker and command line option to control test runs

Plugins can provide custom markers and implement specific behaviour based on it. This is a self-contained example which adds a command line option and a parametrized test function marker to run tests specifies via named environments:

```python
# 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.",
    )
```

(continues on next page)

```python
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):
    envnames = [mark.args[0] for mark in item.iter_markers(name="env")]
    if envnames:
        if item.config.getoption("-E") not in envnames:
            pytest.skip("test requires env in {!r}".format(envnames))
```

A test file using this local plugin:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_someenv.py s                                                    [100%]

============================ 1 skipped in 0.12s ============================
```

and here is one that specifies exactly the environment needed:

```
$ pytest -E stage1
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_someenv.py .                                                    [100%]

============================ 1 passed in 0.12s =============================
```

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.filterwarnings(warning): add a warning filter to the given test. see␣
→https://docs.pytest.org/en/stable/warnings.html#pytest-mark-filterwarnings
```

```
@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, ..., *, reason=...): skip the given test function if␣
→any of the conditions evaluate to True. Example: skipif(sys.platform == 'win32')␣
→skips the test if we are on the win32 platform. See https://docs.pytest.org/en/
→stable/reference.html#pytest-mark-skipif

@pytest.mark.xfail(condition, ..., *, reason=..., run=True, raises=None, strict=xfail_
→strict): mark the test function as an expected failure if any of the conditions␣
→evaluate to True. 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 https://docs.pytest.org/en/stable/reference.html
→#pytest-mark-xfail

@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 https://docs.pytest.org/en/stable/parametrize.html for more info and␣
→examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all␣
→of the specified fixtures. see https://docs.pytest.org/en/stable/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.
```

27.4.8 Passing a callable to custom markers

Below is the config file that will be used in the next examples:

```python
# content of conftest.py
import sys


def pytest_runtest_setup(item):
    for marker in item.iter_markers(name="my_marker"):
        print(marker)
        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
Mark(name='my_marker', args=(<function hello_world at 0xdeadbeef>,), kwargs={})
.
1 passed in 0.12s
```

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`.

27.4.9 Reading markers which were set from multiple places

If you are heavily using markers in your test suite you may encounter the case where a marker is applied several times to a test function. From plugin code you can read over all such settings. Example:

```
# content of test_mark_three_times.py
import pytest

pytestmark = pytest.mark.glob("module", x=1)


@pytest.mark.glob("class", x=2)
class TestClass:
    @pytest.mark.glob("function", x=3)
    def test_something(self):
        pass
```

Here we have the marker "glob" applied three times to the same test function. From a conftest file we can read it like this:

```
# content of conftest.py
import sys


def pytest_runtest_setup(item):
    for mark in item.iter_markers(name="glob"):
        print("glob args={} kwargs={}".format(mark.args, mark.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 0.12s
```

27.4.10 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:

```python
# content of conftest.py
#
import sys
import pytest


ALL = set("darwin linux win32".split())


def pytest_runtest_setup(item):
    supported_platforms = ALL.intersection(mark.name for mark in item.iter_markers())
    plat = sys.platform
    if supported_platforms and plat not in supported_platforms:
        pytest.skip("cannot run on platform {}".format(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:

```python
# 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items

test_plat.py s.s.                                                    [100%]

========================= short test summary info ==========================
SKIPPED [2] conftest.py:12: cannot run on platform linux
====================== 2 passed, 2 skipped in 0.12s ======================
```

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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items / 3 deselected / 1 selected

test_plat.py .                                                       [100%]

===================== 1 passed, 3 deselected in 0.12s ======================
```

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

27.4.11 Automatically adding markers based on test names

If you have 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:

```python
# 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:

```python
# content of conftest.py

import pytest
```

```python
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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items / 2 deselected / 2 selected

test_module.py FF                                                 [100%]

================================ FAILURES ================================
_____ test_interface_simple _____
test_module.py:4: in test_interface_simple
    assert 0
E   assert 0
_____ test_interface_complex _____
test_module.py:8: in test_interface_complex
    assert 0
E   assert 0
======================= short test summary info =========================
FAILED test_module.py::test_interface_simple - assert 0
FAILED test_module.py::test_interface_complex - assert 0
===================== 2 failed, 2 deselected in 0.12s ===================
```

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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items / 1 deselected / 3 selected

test_module.py FFF                                                [100%]

================================ FAILURES ================================
_____ test_interface_simple _____
test_module.py:4: in test_interface_simple
    assert 0
E   assert 0
_____ test_interface_complex _____
test_module.py:8: in test_interface_complex
    assert 0
E   assert 0
_____ test_event_simple _____
test_module.py:12: in test_event_simple
    assert 0
E   assert 0
```

```
========================= short test summary info ==========================
FAILED test_module.py::test_interface_simple - assert 0
FAILED test_module.py::test_interface_complex - assert 0
FAILED test_module.py::test_event_simple - assert 0
==================== 3 failed, 1 deselected in 0.12s =====================
```

27.5 A session-fixture which can look at all collected tests

A session-scoped fixture effectively has access to all collected test items. Here is an example of a fixture function which walks all collected tests and looks if their test class defines a `callme` method and calls it:

```python
# 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 = {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:

```python
# content of test_module.py


class TestHello:
    @classmethod
    def callme(cls):
        print("callme called!")

    def test_method1(self):
        print("test_method1 called")

    def test_method2(self):
        print("test_method1 called")


class TestOther:
    @classmethod
    def callme(cls):
        print("callme other called")

    def test_other(self):
        print("test other")
```

```python
# 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 0.12s
```

27.6 Changing standard (Python) test discovery

27.6.1 Ignore paths during test collection

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

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

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

```
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
```

```
rootdir: $REGENDOC_TMPDIR, inifile:
collected 5 items

tests/example/test_example_01.py .                                    [ 20%]
tests/example/test_example_02.py .                                    [ 40%]
tests/example/test_example_03.py .                                    [ 60%]
tests/foobar/test_foobar_01.py .                                      [ 80%]
tests/foobar/test_foobar_02.py .                                      [100%]


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

The `--ignore-glob` option allows to ignore test file paths based on Unix shell-style wildcards. If you want to exclude test-modules that end with `_01.py`, execute `pytest` with `--ignore-glob='*_01.py'`.

27.6.2 Deselect tests during test collection

Tests can individually be deselected during collection by passing the `--deselect=item` option. For example, say `tests/foobar/test_foobar_01.py` contains `test_a` and `test_b`. You can run all of the tests within `tests/` *except* for `tests/foobar/test_foobar_01.py::test_a` by invoking `pytest` with `--deselect tests/foobar/test_foobar_01.py::test_a`. pytest allows multiple `--deselect` options.

27.6.3 Keeping duplicate paths specified from command line

Default behavior of `pytest` is to ignore duplicate paths specified from the command line. Example:

```
pytest path_a path_a

...
collected 1 item
...
```

Just collect tests once.

To collect duplicate tests, use the `--keep-duplicates` option on the cli. Example:

```
pytest --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:

```
pytest test_a.py test_a.py

...
collected 2 items
...
```

27.6.4 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.

27.6.5 Changing naming conventions

You can configure different naming conventions by setting the *python_files*, *python_classes* and *python_functions* in your *configuration file*. Here is an example:

```
# content of pytest.ini
# Example 1: have pytest look for "check" instead of "test"
[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:

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

    def complex_check(self):
        pass
```

The test collection would look like this:

```
$ pytest --collect-only
========================= test session starts =========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, configfile: pytest.ini
collected 2 items

<Module check_myapp.py>
  <Class CheckMyApp>
      <Function simple_check>
      <Function complex_check>

======================== 2 tests collected in 0.12s ========================
```

You can check for multiple glob patterns by adding a space between the patterns:

```
# Example 2: have pytest look for files with "test" and "example"
# content of pytest.ini
[pytest]
python_files = test_*.py example_*.py
```

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

27.6.6 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.

27.6.7 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, configfile: pytest.ini
collected 3 items

<Module CWD/pythoncollection.py>
  <Function test_function>
  <Class TestClass>
      <Function test_method>
      <Function test_anothermethod>

======================= 3 tests collected in 0.12s ========================
```

27.6.8 Customizing test collection

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
```

If you run with a Python 2 interpreter then you 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'>

====== 1 tests found in 0.04 seconds ======
```

If you run with a Python 3 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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, configfile: pytest.ini
collected 0 items

======================= no tests collected in 0.12s =======================
```

It's also possible to ignore files based on Unix shell-style wildcards by adding patterns to *collect_ignore_glob*.

The following example `conftest.py` ignores the file `setup.py` and in addition all files that end with `*_py2.py` when executed with a Python 3 interpreter:

```
# content of conftest.py
import sys

collect_ignore = ["setup.py"]
if sys.version_info[0] > 2:
    collect_ignore_glob = ["*_py2.py"]
```

Since Pytest 2.6, users can prevent pytest from discovering classes that start with `Test` by setting a boolean `__test__` attribute to `False`.

```python
# Will not be discovered as a test
class TestClass:
    __test__ = False
```

27.7 Working with non-python tests

27.7.1 A basic example for specifying tests in Yaml files

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

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


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


class YamlFile(pytest.File):
    def collect(self):
        # We need a yaml parser, e.g. PyYAML.
        import yaml

        raw = yaml.safe_load(self.fspath.open())
        for name, spec in sorted(raw.items()):
            yield YamlItem.from_parent(self, name=name, spec=spec)


class YamlItem(pytest.Item):
    def __init__(self, name, parent, spec):
        super().__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: {1!r}: {2!r}".format(*excinfo.value.args),
                    "   no further details known at this point.",
                ]
            )

    def reportinfo(self):
        return self.fspath, 0, f"usecase: {self.name}"
```

```python
class YamlException(Exception):
    """Custom exception for error reporting."""
```

You can create a simple example file:

```yaml
# test_simple.yaml
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.yaml
=========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/nonpython
collected 2 items

test_simple.yaml F.                                                  [100%]

================================= FAILURES =================================
_____ usecase: hello _____
usecase execution failed
   spec failed: 'some': 'other'
   no further details known at this point.
========================= short test summary info =========================
FAILED test_simple.yaml::hello
====================== 1 failed, 1 passed in 0.12s ========================
```

You get one dot for the passing `sub1:   sub1` check and one failure. Obviously in the above `conftest.py` you'll want to implement a more interesting interpretation of the yaml-values. You can easily write your own domain specific testing language this way.

Note: `repr_failure(excinfo)` is called for representing test failures. If you create custom collection nodes you can return an error representation string of your choice. It will be reported as a (red) string.

`reportinfo()` is used for representing the test location and is also consulted when reporting in `verbose` mode:

```
nonpython $ pytest -v
=========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_
→PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/nonpython
collecting ... collected 2 items

test_simple.yaml::hello FAILED                                       [ 50%]
test_simple.yaml::ok PASSED                                          [100%]
```

```
================================ FAILURES ================================
_____ usecase: hello _____
usecase execution failed
   spec failed: 'some': 'other'
   no further details known at this point.
========================= short test summary info =========================
FAILED test_simple.yaml::hello
======================= 1 failed, 1 passed in 0.12s =======================
```

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-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/nonpython
collected 2 items

<Package nonpython>
  <YamlFile test_simple.yaml>
    <YamlItem hello>
    <YamlItem ok>

======================= 2 tests collected in 0.12s =======================
```

TWENTYEIGHT

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)"
```

BACKWARDS COMPATIBILITY POLICY

pytest is actively evolving and is a project that has been decades in the making, we keep learning about new and better structures to express different details about testing.

While we implement those modifications we try to ensure an easy transition and don't want to impose unnecessary churn on our users and community/plugin authors.

As of now, pytest considers multiple types of backward compatibility transitions:

a) trivial: APIs which trivially translate to the new mechanism, and do not cause problematic changes.

   We try to support those indefinitely while encouraging users to switch to newer/better mechanisms through documentation.

b) transitional: the old and new API don't conflict and we can help users transition by using warnings, while supporting both for a prolonged time.

   We will only start the removal of deprecated functionality in major releases (e.g. if we deprecate something in 3.0 we will start to 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 start to remove it in 5.0, not in 4.0).

   When the deprecation expires (e.g. 4.0 is released), we won't remove the deprecated functionality immediately, but will use the standard warning filters to turn them into **errors** by default. This approach makes it explicit that removal is imminent, and still gives you time to turn the deprecated feature into a warning instead of an error so it can be dealt with in your own time. In the next minor release (e.g. 4.1), the feature will be effectively removed.

c) true breakage: should only be considered when normal transition is unreasonably unsustainable and would offset important development/features by years. In addition, they should be limited to APIs where the number of actual users is very small (for example only impacting some plugins), and can be coordinated with the community in advance.

   Examples for such upcoming changes:

   - removal of `pytest_runtest_protocol/nextitem` - #895

   - rearranging of the node tree to include `FunctionDefinition`

   - rearranging of `SetupState` #895

   True breakages must be announced first in an issue containing:

   - Detailed description of the change

   - Rationale

   - Expected impact on users and plugin authors (example in #895)

   After there's no hard *-1* on the issue it should be followed up by an initial proof-of-concept Pull Request.

This POC serves as both a coordination point to assess impact and potential inspiration to come up with a transitional solution after all.

After a reasonable amount of time the PR can be merged to base a new major release.

For the PR to mature from POC to acceptance, it must contain: * Setup of deprecation errors/warnings that help users fix and port their code. If it is possible to introduce a deprecation period under the current series, before the true breakage, it should be introduced in a separate PR and be part of the current release stream. * Detailed description of the rationale and examples on how to port code in `doc/en/deprecations.rst`.

HISTORY

30.1 Focus primary on smooth transition - stance (pre 6.0)

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 issue deprecation warnings using a custom warning hierarchy (see *Internal pytest warnings*). These warnings may be suppressed using the standard means: `-W` command-line flag or `filterwarnings` ini options (see *Warnings Capture*), but we suggest to use these sparingly and temporarily, and heed the warnings when possible.

We will only start the removal of deprecated functionality in major releases (e.g. if we deprecate something in 3.0 we will start to 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 start to remove it in 5.0, not in 4.0).

When the deprecation expires (e.g. 4.0 is released), we won't remove the deprecated functionality immediately, but will use the standard warning filters to turn them into **errors** by default. This approach makes it explicit that removal is imminent, and still gives you time to turn the deprecated feature into a warning instead of an error so it can be dealt with in your own time. In the next minor release (e.g. 4.1), the feature will be effectively removed.

30.1.1 Deprecation Roadmap

Features currently deprecated and removed in previous releases can be found in *Deprecations and Removals*.

We track future deprecation and removal of features using milestones and the deprecation and removal labels on GitHub.

DEPRECATIONS AND REMOVALS

This page lists all pytest features that are currently deprecated or have been removed in past major releases. The objective is to give users a clear rationale why a certain feature has been removed, and what alternatives should be used instead.

31.1 Deprecated Features

Below is a complete list of all pytest features which are considered deprecated. Using those features will issue `PytestWarning` or subclasses, which can be filtered using *standard warning filters*.

31.1.1 The `--strict` command-line option

Deprecated since version 6.2.

The `--strict` command-line option has been deprecated in favor of `--strict-markers`, which better conveys what the option does.

We have plans to maybe in the future to reintroduce `--strict` and make it an encompassing flag for all strictness related options (`--strict-markers` and `--strict-config` at the moment, more might be introduced in the future).

31.1.2 The `yield_fixture` function/decorator

Deprecated since version 6.2.

`pytest.yield_fixture` is a deprecated alias for *`pytest.fixture()`*.

It has been so for a very long time, so can be search/replaced safely.

31.1.3 The `pytest_warning_captured` hook

Deprecated since version 6.0.

This hook has an `item` parameter which cannot be serialized by `pytest-xdist`.

Use the `pytest_warning_recored` hook instead, which replaces the `item` parameter by a `nodeid` parameter.

31.1.4 The `pytest.collect` module

Deprecated since version 6.0.

The `pytest.collect` module is no longer part of the public API, all its names should now be imported from `pytest` directly instead.

31.1.5 The `pytest._fillfuncargs` function

Deprecated since version 6.0.

This function was kept for backward compatibility with an older plugin.

It's functionality is not meant to be used directly, but if you must replace it, use `function._request._fillfixtures()` instead, though note this is not a public API and may break in the future.

31.2 Removed Features

As stated in our *Backwards Compatibility Policy* policy, deprecated features are removed only in major releases after an appropriate period of deprecation has passed.

31.2.1 `--no-print-logs` command-line option

Deprecated since version 5.4.

Removed in version 6.0.

The `--no-print-logs` option and `log_print` ini setting are removed. If you used them, please use `--show-capture` instead.

A `--show-capture` command-line option was added in `pytest 3.5.0` which allows to specify how to display captured output when tests fail: `no`, `stdout`, `stderr`, `log` or `all` (the default).

31.2.2 Result log (`--result-log`)

Deprecated since version 4.0.

Removed in version 6.0.

The `--result-log` option produces a stream of test reports which can be analysed at runtime, but it uses a custom format which requires users to implement their own parser.

The pytest-reportlog plugin provides a `--report-log` option, a more standard and extensible alternative, producing one JSON object per-line, and should cover the same use cases. Please try it out and provide feedback.

The `pytest-reportlog` plugin might even be merged into the core at some point, depending on the plans for the plugins and number of users using it.

31.2.3 `pytest_collect_directory` hook

Removed in version 6.0.

The `pytest_collect_directory` has not worked properly for years (it was called but the results were ignored). Users may consider using *pytest_collection_modifyitems* instead.

31.2.4 TerminalReporter.writer

Removed in version 6.0.

The `TerminalReporter.writer` attribute has been deprecated and should no longer be used. This was inadvertently exposed as part of the public API of that plugin and ties it too much with `py.io.TerminalWriter`.

Plugins that used `TerminalReporter.writer` directly should instead use `TerminalReporter` methods that provide the same functionality.

31.2.5 `junit_family` default value change to "xunit2"

Changed in version 6.0.

The default value of `junit_family` option will change to `xunit2` in pytest 6.0, which is an update of the old `xunit1` format and is supported by default in modern tools that manipulate this type of file (for example, Jenkins, Azure Pipelines, etc.).

Users are recommended to try the new `xunit2` format and see if their tooling that consumes the JUnit XML file supports it.

To use the new format, update your `pytest.ini`:

```
[pytest]
junit_family=xunit2
```

If you discover that your tooling does not support the new format, and want to keep using the legacy version, set the option to `legacy` instead:

```
[pytest]
junit_family=legacy
```

By using `legacy` you will keep using the legacy/xunit1 format when upgrading to pytest 6.0, where the default format will be `xunit2`.

In order to let users know about the transition, pytest will issue a warning in case the `--junitxml` option is given in the command line but `junit_family` is not explicitly configured in `pytest.ini`.

Services known to support the `xunit2` format:

- Jenkins with the JUnit plugin.

- Azure Pipelines.

31.2.6 Node Construction changed to `Node.from_parent`

Changed in version 6.0.

The construction of nodes now should use the named constructor `from_parent`. This limitation in api surface intends to enable better/simpler refactoring of the collection tree.

This means that instead of `MyItem(name="foo", parent=collector, obj=42)` one now has to invoke `MyItem.from_parent(collector, name="foo")`.

Plugins that wish to support older versions of pytest and suppress the warning can use `hasattr` to check if `from_parent` exists in that version:

```python
def pytest_pycollect_makeitem(collector, name, obj):
    if hasattr(MyItem, "from_parent"):
        item = MyItem.from_parent(collector, name="foo")
        item.obj = 42
        return item
    else:
        return MyItem(name="foo", parent=collector, obj=42)
```

Note that `from_parent` should only be called with keyword arguments for the parameters.

31.2.7 `pytest.fixture` arguments are keyword only

Removed in version 6.0.

Passing arguments to pytest.fixture() as positional arguments has been removed - pass them by keyword instead.

31.2.8 `funcargnames` alias for `fixturenames`

Removed in version 6.0.

The `FixtureRequest`, `Metafunc`, and `Function` classes track the names of their associated fixtures, with the aptly-named `fixturenames` attribute.

Prior to pytest 2.3, this attribute was named `funcargnames`, and we have kept that as an alias since. It is finally due for removal, as it is often confusing in places where we or plugin authors must distinguish between fixture names and names supplied by non-fixture things such as `pytest.mark.parametrize`.

31.2.9 `pytest.config` global

Removed in version 5.0.

The `pytest.config` global object is deprecated. Instead use `request.config` (via the `request` fixture) or if you are a plugin author use the `pytest_configure(config)` hook. Note that many hooks can also access the `config` object indirectly, through `session.config` or `item.config` for example.

31.2.10 `"message"` parameter of `pytest.raises`

Removed in version 5.0.

It is a common mistake to think this parameter will match the exception message, while in fact it only serves to provide a custom message in case the `pytest.raises` check fails. To prevent users from making this mistake, and because it is believed to be little used, pytest is deprecating it without providing an alternative for the moment.

If you have a valid use case for this parameter, consider that to obtain the same results you can just call `pytest.fail` manually at the end of the `with` statement.

For example:

```
with pytest.raises(TimeoutError, message="Client got unexpected message"):
    wait_for(websocket.recv(), 0.5)
```

Becomes:

```
with pytest.raises(TimeoutError):
    wait_for(websocket.recv(), 0.5)
    pytest.fail("Client got unexpected message")
```

If you still have concerns about this deprecation and future removal, please comment on issue #3974.

31.2.11 `raises` / `warns` with a string as the second argument

Removed in version 5.0.

Use the context manager form of these instead. When necessary, invoke `exec` directly.

Example:

```
pytest.raises(ZeroDivisionError, "1 / 0")
pytest.raises(SyntaxError, "a $ b")

pytest.warns(DeprecationWarning, "my_function()")
pytest.warns(SyntaxWarning, "assert(1, 2)")
```

Becomes:

```
with pytest.raises(ZeroDivisionError):
    1 / 0
with pytest.raises(SyntaxError):
    exec("a $ b")  # exec is required for invalid syntax

with pytest.warns(DeprecationWarning):
    my_function()
with pytest.warns(SyntaxWarning):
    exec("assert(1, 2)")  # exec is used to avoid a top-level warning
```

31.2.12 Using `Class` in custom Collectors

Removed in version 4.0.

Using objects named `"Class"` as a way to customize the type of nodes that are collected in `Collector` subclasses has been deprecated. Users instead should use `pytest_pycollect_makeitem` to customize node types during collection.

This issue should affect only advanced plugins who create new collection types, so if you see this warning message please contact the authors so they can change the code.

31.2.13 marks in `pytest.mark.parametrize`

Removed in version 4.0.

Applying marks to values of a `pytest.mark.parametrize` call is now deprecated. For example:

```python
@pytest.mark.parametrize(
    "a, b",
    [
        (3, 9),
        pytest.mark.xfail(reason="flaky")(6, 36),
        (10, 100),
        (20, 200),
        (40, 400),
        (50, 500),
    ],
)
def test_foo(a, b):
    ...
```

This code applies the `pytest.mark.xfail(reason="flaky")` mark to the `(6, 36)` value of the above parametrization call.

This was considered hard to read and understand, and also its implementation presented problems to the code preventing further internal improvements in the marks architecture.

To update the code, use `pytest.param`:

```python
@pytest.mark.parametrize(
    "a, b",
    [
        (3, 9),
        pytest.param(6, 36, marks=pytest.mark.xfail(reason="flaky")),
        (10, 100),
        (20, 200),
        (40, 400),
        (50, 500),
    ],
)
def test_foo(a, b):
    ...
```

31.2.14 `pytest_funcarg__` prefix

Removed in version 4.0.

In very early pytest versions fixtures could be defined using the `pytest_funcarg__` prefix:

```python
def pytest_funcarg__data():
    return SomeData()
```

Switch over to the `@pytest.fixture` decorator:

```python
@pytest.fixture
def data():
    return SomeData()
```

31.2.15 [pytest] section in setup.cfg files

Removed in version 4.0.

`[pytest]` sections in `setup.cfg` files should now be named `[tool:pytest]` to avoid conflicts with other distutils commands.

31.2.16 Metafunc.addcall

Removed in version 4.0.

`Metafunc.addcall` was a precursor to the current parametrized mechanism. Users should use *pytest.Metafunc.parametrize()* instead.

Example:

```python
def pytest_generate_tests(metafunc):
    metafunc.addcall({"i": 1}, id="1")
    metafunc.addcall({"i": 2}, id="2")
```

Becomes:

```python
def pytest_generate_tests(metafunc):
    metafunc.parametrize("i", [1, 2], ids=["1", "2"])
```

31.2.17 `cached_setup`

Removed in version 4.0.

`request.cached_setup` was the precursor of the setup/teardown mechanism available to fixtures.

Example:

```python
@pytest.fixture
def db_session():
    return request.cached_setup(
        setup=Session.create, teardown=lambda session: session.close(), scope="module"
    )
```

This should be updated to make use of standard fixture mechanisms:

```
@pytest.fixture(scope="module")
def db_session():
    session = Session.create()
    yield session
    session.close()
```

You can consult funcarg comparison section in the docs for more information.

31.2.18 pytest_plugins in non-top-level conftest files

Removed in version 4.0.

Defining *pytest_plugins* is now deprecated in non-top-level conftest.py files because they will activate referenced plugins *globally*, which is surprising because for all other pytest features `conftest.py` files are only *active* for tests at or below it.

31.2.19 `Config.warn` and `Node.warn`

Removed in version 4.0.

Those methods were part of the internal pytest warnings system, but since `3.8` pytest is using the builtin warning system for its own warnings, so those two functions are now deprecated.

`Config.warn` should be replaced by calls to the standard `warnings.warn`, example:

```
config.warn("C1", "some warning")
```

Becomes:

```
warnings.warn(pytest.PytestWarning("some warning"))
```

`Node.warn` now supports two signatures:

- `node.warn(PytestWarning("some message"))`: is now the **recommended** way to call this function. The warning instance must be a PytestWarning or subclass.

- `node.warn("CI", "some message")`: this code/message form has been **removed** and should be converted to the warning instance form above.

31.2.20 record_xml_property

Removed in version 4.0.

The `record_xml_property` fixture is now deprecated in favor of the more generic `record_property`, which can be used by other consumers (for example `pytest-html`) to obtain custom information about the test run.

This is just a matter of renaming the fixture as the API is the same:

```
def test_foo(record_xml_property):
    ...
```

Change to:

```
def test_foo(record_property):
    ...
```

31.2.21 Passing command-line string to `pytest.main()`

Removed in version 4.0.

Passing a command-line string to `pytest.main()` is deprecated:

```
pytest.main("-v -s")
```

Pass a list instead:

```
pytest.main(["-v", "-s"])
```

By passing a string, users expect that pytest will interpret that command-line using the shell rules they are working on (for example `bash` or `Powershell`), but this is very hard/impossible to do in a portable way.

31.2.22 Calling fixtures directly

Removed in version 4.0.

Calling a fixture function directly, as opposed to request them in a test function, is deprecated.

For example:

```python
@pytest.fixture
def cell():
    return ...


@pytest.fixture
def full_cell():
    cell = cell()
    cell.make_full()
    return cell
```

This is a great source of confusion to new users, which will often call the fixture functions and request them from test functions interchangeably, which breaks the fixture resolution model.

In those cases just request the function directly in the dependent fixture:

```python
@pytest.fixture
def cell():
    return ...


@pytest.fixture
def full_cell(cell):
    cell.make_full()
    return cell
```

Alternatively if the fixture function is called multiple times inside a test (making it hard to apply the above pattern) or if you would like to make minimal changes to the code, you can create a fixture which calls the original function together with the `name` parameter:

```python
def cell():
    return ...
```

<div align="right">(continues on next page)</div>

```python
@pytest.fixture(name="cell")
def cell_fixture():
    return cell()
```

31.2.23 `yield` tests

Removed in version 4.0.

pytest supported `yield`-style tests, where a test function actually `yield` functions and values that are then turned into proper test methods. Example:

```python
def check(x, y):
    assert x ** x == y


def test_squared():
    yield check, 2, 4
    yield check, 3, 9
```

This would result into two actual test functions being generated.

This form of test function doesn't support fixtures properly, and users should switch to `pytest.mark.parametrize`:

```python
@pytest.mark.parametrize("x, y", [(2, 4), (3, 9)])
def test_squared(x, y):
    assert x ** x == y
```

31.2.24 Internal classes accessed through `Node`

Removed in version 4.0.

Access of `Module`, `Function`, `Class`, `Instance`, `File` and `Item` through `Node` instances now issue this warning:

```
usage of Function.Module is deprecated, please use pytest.Module instead
```

Users should just `import pytest` and access those objects using the `pytest` module.

This has been documented as deprecated for years, but only now we are actually emitting deprecation warnings.

31.2.25 `Node.get_marker`

Removed in version 4.0.

As part of a large *Marker revamp and iteration*, `_pytest.nodes.Node.get_marker` is removed. See *the documentation* on tips on how to update your code.

31.2.26 `somefunction.markname`

Removed in version 4.0.

As part of a large *Marker revamp and iteration* we already deprecated using `MarkInfo` the only correct way to get markers of an element is via `node.iter_markers(name)`.

31.2.27 `pytest_namespace`

Removed in version 4.0.

This hook is deprecated because it greatly complicates the pytest internals regarding configuration and initialization, making some bug fixes and refactorings impossible.

Example of usage:

```python
class MySymbol:
    ...


def pytest_namespace():
    return {"my_symbol": MySymbol()}
```

Plugin authors relying on this hook should instead require that users now import the plugin modules directly (with an appropriate public API).

As a stopgap measure, plugin authors may still inject their names into pytest's namespace, usually during `pytest_configure`:

```python
import pytest


def pytest_configure():
    pytest.my_symbol = MySymbol()
```

PYTHON 2.7 AND 3.4 SUPPORT

It is demanding on the maintainers of an open source project to support many Python versions, as there's extra cost of keeping code compatible between all versions, while holding back on features only made possible on newer Python versions.

In case of Python 2 and 3, the difference between the languages makes it even more prominent, because many new Python 3 features cannot be used in a Python 2/3 compatible code base.

Python 2.7 EOL has been reached in 2020, with the last release made in April, 2020.

Python 3.4 EOL has been reached in 2019, with the last release made in March, 2019.

For those reasons, in Jun 2019 it was decided that **pytest 4.6** series will be the last to support Python 2.7 and 3.4.

32.1 What this means for general users

Thanks to the python_requires setuptools option, Python 2.7 and Python 3.4 users using a modern pip version will install the last pytest 4.6.X version automatically even if 5.0 or later versions are available on PyPI.

Users should ensure they are using the latest pip and setuptools versions for this to work.

32.2 Maintenance of 4.6.X versions

Until January 2020, the pytest core team ported many bug-fixes from the main release into the `4.6.x` branch, with several 4.6.X releases being made along the year.

From now on, the core team will **no longer actively backport patches**, but the `4.6.x` branch will continue to exist so the community itself can contribute patches.

The core team will be happy to accept those patches, and make new 4.6.X releases **until mid-2020** (but consider that date as a ballpark, after that date the team might still decide to make new releases for critical bugs).

32.2.1 Technical aspects

(This section is a transcript from #5275).

In this section we describe the technical aspects of the Python 2.7 and 3.4 support plan.

What goes into 4.6.X releases

New 4.6.X releases will contain bug fixes only.

When will 4.6.X releases happen

New 4.6.X releases will happen after we have a few bugs in place to release, or if a few weeks have passed (say a single bug has been fixed a month after the latest 4.6.X release).

No hard rules here, just ballpark.

Who will handle applying bug fixes

We core maintainers expect that people still using Python 2.7/3.4 and being affected by bugs to step up and provide patches and/or port bug fixes from the active branches.

We will be happy to guide users interested in doing so, so please don't hesitate to ask.

Backporting changes into 4.6

Please follow these instructions:

1. `git fetch --all --prune`

2. `git checkout origin/4.6.x -b backport-XXXX` # use the PR number here

3. Locate the merge commit on the PR, in the *merged* message, for example:

   nicoddemus merged commit 0f8b462 into pytest-dev:features

4. `git cherry-pick -m1 REVISION` # use the revision you found above (`0f8b462`).

5. Open a PR targeting `4.6.x`:

   - Prefix the message with `[4.6]` so it is an obvious backport
   - Delete the PR body, it usually contains a duplicate commit message.

Providing new PRs to 4.6

Fresh pull requests to `4.6.x` will be accepted provided that the equivalent code in the active branches does not contain that bug (for example, a bug is specific to Python 2 only).

Bug fixes that also happen in the mainstream version should be first fixed there, and then backported as per instructions above.

CONTRIBUTION GETTING STARTED

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

Contents

33.1 Feature requests and feedback

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

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

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

33.2 Report bugs

Report bugs for pytest in the issue tracker.

If you are reporting a bug, please include:

- Your operating system name and version.

- Any details about your local setup that might be helpful in troubleshooting, specifically 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.

33.3 Fix bugs

Look through the GitHub issues for bugs.

Talk to developers to find out how you can fix specific bugs. To indicate that you are going to work on a particular issue, add a comment to that effect on the specific issue.

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

33.4 Implement features

Look through the GitHub issues for enhancements.

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

33.5 Write documentation

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

- More complementary documentation. Have you perhaps found something unclear?

- Documentation translations. We currently have only English.

- Docstrings. There can never be too many of them.

- Blog posts, articles and such – they're all very appreciated.

You can also edit documentation files directly in the GitHub web interface, without 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 `doc/en/_build/html`, where 'en' refers to the documentation language.

Pytest has an API reference which in large part is generated automatically from the docstrings of the documented items. Pytest uses the Sphinx docstring format. For example:

```python
def my_function(arg: ArgType) -> Foo:
    """Do important stuff.

    More detailed info here, in separate paragraphs from the subject line.
    Use proper sentences -- start sentences with capital letters and end
    with periods.

    Can include annotated documentation:

    :param short_arg: An argument which determines stuff.
    :param long_arg:
        A long explanation which spans multiple lines, overflows
        like this.
    :returns: The result.
    :raises ValueError:
        Detailed information when this can happen.

    .. versionadded:: 6.0

    Including types into the annotations above is not necessary when
    type-hinting is being used (as in this example).
    """
```

33.6 Submitting Plugins to pytest-dev

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

- pytest-dev on GitHub

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 packaging metadata that contains a `pytest-` prefixed name, version number, authors, short and long description.
- a tox configuration for running tests using tox.
- a `README` describing how to use the plugin and on which platforms it runs.
- a `LICENSE` file containing the licensing information, with matching info in its packaging metadata.
- 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".

33.7 Preparing Pull Requests

33.7.1 Short version

1. Fork the repository.

2. Enable and install pre-commit to ensure style-guides and code checks are followed.

3. Follow **PEP-8** for naming and black for formatting.

4. Tests are run using `tox`:

```
tox -e linting,py37
```

The test environments above are usually enough to cover most cases locally.

5. Write a `changelog` entry: `changelog/2574.bugfix.rst`, use issue id number and one of `feature`, `improvement`, `bugfix`, `doc`, `deprecation`, `breaking`, `vendor` or `trivial` for the issue type.

6. Unless your change is a trivial or a documentation fix (e.g., a typo or reword of a small section) please add yourself to the `AUTHORS` file, in alphabetical order.

33.7.2 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, create your own branch off "master":

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

Given we have "major.minor.micro" version numbers, bug fixes 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 pre-commit and its hook on the pytest repo:

```
$ pip install --user pre-commit
$ pre-commit install
```

Afterwards `pre-commit` will run whenever you commit.

https://pre-commit.com/ is a framework for managing and maintaining multi-language pre-commit hooks to ensure code-style and code formatting is consistent.

4. 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
```

5. Run all the tests

You need to have Python 3.7 available in your system. Now running tests is as simple as issuing this command:

```
$ tox -e linting,py37
```

This command will run tests via the "tox" tool against Python 3.7 and also perform "lint" coding-style checks.

6. You can now edit your local working copy and run the tests again as necessary. Please follow PEP-8 for naming.

You can pass different options to `tox`. For example, to run tests on Python 3.7 and pass options to pytest (e.g. enter pdb on failure) to pytest you can do:

```
$ tox -e py37 -- --pdb
```

Or to only run tests in a particular test module on Python 3.7:

```
$ tox -e py37 -- testing/test_config.py
```

When committing, `pre-commit` will re-format the files if necessary.

7. If instead of using `tox` you prefer to run the tests directly, then we suggest to create a virtual environment and use an editable install with the `testing` extra:

```
$ python3 -m venv .venv
$ source .venv/bin/activate  # Linux
$ .venv/Scripts/activate.bat  # Windows
$ pip install -e ".[testing]"
```

Afterwards, you can edit the files and run pytest normally:

```
$ pytest testing/test_config.py
```

8. Create a new changelog entry in `changelog`. The file should be named `<issueid>.<type>.rst`, where *issueid* is the number of the issue related to the change and *type* is one of `feature`, `improvement`, `bugfix`, `doc`, `deprecation`, `breaking`, `vendor` or `trivial`. You may skip creating the changelog entry if the change doesn't affect the documented behaviour of pytest.

9. Add yourself to `AUTHORS` file if not there yet, in alphabetical order.

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

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

11. 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
```

33.7.3 Writing Tests

Writing tests for plugins or for pytest itself is often done using the testdir fixture, as a "black-box" test.

For example, to ensure a simple test passes you can write:

```python
def test_true_assertion(testdir):
    testdir.makepyfile(
        """
        def test_foo():
            assert True
        """
    )
    result = testdir.runpytest()
    result.assert_outcomes(failed=0, passed=1)
```

Alternatively, it is possible to make checks based on the actual output of the termal using *glob-like* expressions:

```python
def test_true_assertion(testdir):
    testdir.makepyfile(
        """
        def test_foo():
            assert False
        """
    )
    result = testdir.runpytest()
    result.stdout.fnmatch_lines(["*assert False*", "*1 failed*"])
```

When choosing a file where to write a new test, take a look at the existing files and see if there's one file which looks like a good fit. For example, a regression test about a bug in the `--lf` option should go into `test_cacheprovider.py`, given that this option is implemented in `cacheprovider.py`. If in doubt, go ahead and open a PR with your best guess and we can discuss this over the code.

33.8 Joining the Development Team

Anyone who has successfully seen through a pull request which did not require any extra work from the development team to merge will themselves gain commit access if they so wish (if we forget to ask please send a friendly reminder). This does not mean there is any change in your contribution workflow: everyone goes through the same pull-request-and-review process and no-one merges their own pull requests unless already approved. It does however mean you can participate in the development process more fully since you can merge pull requests from other contributors yourself after having reviewed them.

33.9 Backporting bug fixes for the next patch release

Pytest makes feature release every few weeks or months. In between, patch releases are made to the previous feature release, containing bug fixes only. The bug fixes usually fix regressions, but may be any change that should reach users before the next feature release.

Suppose for example that the latest release was 1.2.3, and you want to include a bug fix in 1.2.4 (check https://github.com/pytest-dev/pytest/releases for the actual latest release). The procedure for this is:

1. First, make sure the bug is fixed the `master` branch, with a regular pull request, as described above. An exception to this is if the bug fix is not applicable to `master` anymore.

2. `git checkout origin/1.2.x -b backport-XXXX` # use the master PR number here

3. Locate the merge commit on the PR, in the *merged* message, for example:

   nicoddemus merged commit 0f8b462 into pytest-dev:master

4. `git cherry-pick -x -m1 REVISION` # use the revision you found above (`0f8b462`).

5. Open a PR targeting `1.2.x`:

   - Prefix the message with `[1.2.x]`.
   - Delete the PR body, it usually contains a duplicate commit message.

33.9.1 Who does the backporting

As mentioned above, bugs should first be fixed on `master` (except in rare occasions that a bug only happens in a previous release). So who should do the backport procedure described above?

1. If the bug was fixed by a core developer, it is the main responsibility of that core developer to do the backport.

2. However, often the merge is done by another maintainer, in which case it is nice of them to do the backport procedure if they have the time.

3. For bugs submitted by non-maintainers, it is expected that a core developer will to do the backport, normally the one that merged the PR on `master`.

4. If a non-maintainers notices a bug which is fixed on `master` but has not been backported (due to maintainers forgetting to apply the *needs backport* label, or just plain missing it), they are also welcome to open a PR with the backport. The procedure is simple and really helps with the maintenance of the project.

All the above are not rules, but merely some guidelines/suggestions on what we should expect about backports.

33.10 Handling stale issues/PRs

Stale issues/PRs are those where pytest contributors have asked for questions/changes and the authors didn't get around to answer/implement them yet after a somewhat long time, or the discussion simply died because people seemed to lose interest.

There are many reasons why people don't answer questions or implement requested changes: they might get busy, lose interest, or just forget about it, but the fact is that this is very common in open source software.

The pytest team really appreciates every issue and pull request, but being a high-volume project with many issues and pull requests being submitted daily, we try to reduce the number of stale issues and PRs by regularly closing them. When an issue/pull request is closed in this manner, it is by no means a dismissal of the topic being tackled by the issue/pull request, but it is just a way for us to clear up the queue and make the maintainers' work more manageable. Submitters can always reopen the issue/pull request in their own time later if it makes sense.

33.10.1 When to close

Here are a few general rules the maintainers use to decide when to close issues/PRs because of lack of inactivity:

- Issues labeled `question` or `needs information`: closed after 14 days inactive.
- Issues labeled `proposal`: closed after six months inactive.
- Pull requests: after one month, consider pinging the author, update linked issue, or consider closing. For pull requests which are nearly finished, the team should consider finishing it up and merging it.

The above are **not hard rules**, but merely **guidelines**, and can be (and often are!) reviewed on a case-by-case basis.

33.10.2 Closing pull requests

When closing a Pull Request, it needs to be acknowledge the time, effort, and interest demonstrated by the person which submitted it. As mentioned previously, it is not the intent of the team to dismiss stalled pull request entirely but to merely to clear up our queue, so a message like the one below is warranted when closing a pull request that went stale:

> Hi <contributor>,
>
> First of all we would like to thank you for your time and effort on working on this, the pytest team deeply appreciates it.
>
> We noticed it has been awhile since you have updated this PR, however. pytest is a high activity project, with many issues/PRs being opened daily, so it is hard for us maintainers to track which PRs are ready for merging, for review, or need more attention.
>
> So for those reasons we think it is best to close the PR for now, but with the only intention to cleanup our queue, it is by no means a rejection of your changes. We still encourage you to re-open this PR (it is just a click of a button away) when you are ready to get back to it.
>
> Again we appreciate your time for working on this, and hope you might get back to this at a later time!
>
> <bye>

33.11 Closing Issues

When a pull request is submitted to fix an issue, add text like `closes #XYZW` to the PR description and/or commits (where `XYZW` is the issue number). See the GitHub docs for more information.

When an issue is due to user error (e.g. misunderstanding of a functionality), please politely explain to the user why the issue raised is really a non-issue and ask them to close the issue if they have no further questions. If the original requestor is unresponsive, the issue will be handled as described in the section *Handling stale issues/PRs* above.

THIRTYFOUR

DEVELOPMENT GUIDE

The contributing guidelines are to be found *here*. The release procedure for pytest is documented on GitHub.

CHAPTER

THIRTYFIVE

SPONSOR

pytest is maintained by a team of volunteers from all around the world in their free time. While we work on pytest because we love the project and use it daily at our daily jobs, monetary compensation when possible is welcome to justify time away from friends, family and personal time.

Money is also used to fund local sprints, merchandising (stickers to distribute in conferences for example) and every few years a large sprint involving all members.

35.1 OpenCollective

Open Collective is an online funding platform for open and transparent communities. It provide tools to raise money and share your finances in full transparency.

It is the platform of choice for individuals and companies that want to make one-time or monthly donations directly to the project.

See more details in the pytest collective.

PYTEST FOR ENTERPRISE

Tidelift is working with the maintainers of pytest and thousands of other open source projects to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use.

Get more details

The Tidelift Subscription is a managed open source subscription for application dependencies covering millions of open source projects across JavaScript, Python, Java, PHP, Ruby, .NET, and more.

Your subscription includes:

- **Security updates**

    - Tidelift's security response team coordinates patches for new breaking security vulnerabilities and alerts immediately through a private channel, so your software supply chain is always secure.

- **Licensing verification and indemnification**

    - Tidelift verifies license information to enable easy policy enforcement and adds intellectual property indemnification to cover creators and users in case something goes wrong. You always have a 100% up-to-date bill of materials for your dependencies to share with your legal team, customers, or partners.

- **Maintenance and code improvement**

    - Tidelift ensures the software you rely on keeps working as long as you need it to work. Your managed dependencies are actively maintained and we recruit additional maintainers where required.

- **Package selection and version guidance**

    - Tidelift helps you choose the best open source packages from the start—and then guide you through updates to stay on the best releases as new issues arise.

- **Roadmap input**

    - Take a seat at the table with the creators behind the software you use. Tidelift's participating maintainers earn more income as their software is used by more subscribers, so they're interested in knowing what you need.

- **Tooling and cloud integration**

    - Tidelift works with GitHub, GitLab, BitBucket, and every cloud platform (and other deployment targets, too).

The end result? All of the capabilities you expect from commercial-grade software, for the full breadth of open source you use. That means less time grappling with esoteric open source trivia, and more time building your own applications—and your business.

Request a demo

LICENSE

Distributed under the terms of the MIT license, pytest is free and open source software.

```
The MIT License (MIT)

Copyright (c) 2004-2020 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.
```

THIRTYEIGHT

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.

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.

39.1 Marker revamp and iteration

Changed in version 3.6.

pytest's marker implementation traditionally worked by simply updating the `__dict__` attribute of functions to cumulatively add markers. As a result, markers would unintentionally be passed along class hierarchies in surprising ways. Further, the API for retrieving them was inconsistent, as markers from parameterization would be stored differently than markers applied using the `@pytest.mark` decorator and markers added via `node.add_marker`.

This state of things made it technically next to impossible to use data from markers correctly without having a deep understanding of the internals, leading to subtle and hard to understand bugs in more advanced usages.

Depending on how a marker got declared/changed one would get either a `MarkerInfo` which might contain markers from sibling classes, `MarkDecorators` when marks came from parameterization or from a `node.add_marker` call, discarding prior marks. Also `MarkerInfo` acts like a single mark, when it in fact represents a merged view on multiple marks with the same name.

On top of that markers were not accessible in the same way for modules, classes, and functions/methods. In fact, markers were only accessible in functions, even if they were declared on classes/modules.

A new API to access markers has been introduced in pytest 3.6 in order to solve the problems with the initial design, providing the *_pytest.nodes.Node.iter_markers()* method to iterate over markers in a consistent manner and reworking the internals, which solved a great deal of problems with the initial design.

39.1.1 Updating code

The old `Node.get_marker(name)` function is considered deprecated because it returns an internal `MarkerInfo` object which contains the merged name, `*args` and `**kwargs` of all the markers which apply to that node.

In general there are two scenarios on how markers should be handled:

1. Marks overwrite each other. Order matters but you only want to think of your mark as a single item. E.g. `log_level('info')` at a module level can be overwritten by `log_level('debug')` for a specific test.

In this case, use `Node.get_closest_marker(name)`:

```
# replace this:
marker = item.get_marker("log_level")
if marker:
```

(continues on next page)

```
    level = marker.args[0]


# by this:
marker = item.get_closest_marker("log_level")
if marker:
    level = marker.args[0]
```

2. Marks compose in an additive manner. E.g. `skipif(condition)` marks mean you just want to evaluate all of them, order doesn't even matter. You probably want to think of your marks as a set here.

In this case iterate over each mark and handle their `*args` and `**kwargs` individually.

```
# replace this
skipif = item.get_marker("skipif")
if skipif:
    for condition in skipif.args:
        # eval condition
        ...

# by this:
for skipif in item.iter_markers("skipif"):
    condition = skipif.args[0]
    # eval condition
```

If you are unsure or have any questions, please consider opening an issue.

39.1.2 Related issues

Here is a non-exhaustive list of issues fixed by the new implementation:

- Marks don't pick up nested classes (#199).

- Markers stain on all related classes (#568).

- Combining marks - args and kwargs calculation (#2897).

- `request.node.get_marker('name')` returns `None` for markers applied in classes (#902).

- Marks applied in parametrize are stored as markdecorator (#2400).

- Fix marker interaction in a backward incompatible way (#1670).

- Refactor marks to get rid of the current "marks transfer" mechanism (#2363).

- Introduce FunctionDefinition node, use it in generate_tests (#2522).

- Remove named marker attributes and collect markers in items (#891).

- skipif mark from parametrize hides module level skipif mark (#1540).

- skipif + parametrize not skipping tests (#1296).

- Marker transfer incompatible with inheritance (#535).

More details can be found in the original PR.

Note: in a future major release of pytest we will introduce class based markers, at which point markers will no longer be limited to instances of `Mark`.

39.2 cache plugin integrated into the core

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.

39.3 funcargs and `pytest_funcarg__`

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.

39.4 `@pytest.yield_fixture` decorator

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.

39.5 `[pytest]` header in `setup.cfg`

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]`.

39.6 Applying marks to `@pytest.mark.parametrize` parameters

Prior to version 3.1 the supported mechanism for marking values used the syntax:

```python
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.

39.7 `@pytest.mark.parametrize` argument names as a tuple

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.

39.8 setup: is now an "autouse fixture"

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 (fixtures you don't have to request)*

39.9 Conditions as strings instead of booleans

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

```python
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:

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

The equivalent with "boolean conditions" is:

```python
@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.

39.10 `pytest.set_trace()`

Previous to version 2.4 to set a break point in code one needed to use `pytest.set_trace()`:

```python
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*.

39.11 "compat" properties

Access of `Module`, `Function`, `Class`, `Instance`, `File` and `Item` through `Node` instances have long been documented as deprecated, but started to emit warnings from pytest `3.9` and onward.

Users should just `import pytest` and access those objects using the `pytest` module.

TALKS AND TUTORIALS

40.1 Books

- pytest Quick Start Guide, by Bruno Oliveira (2018).
- Python Testing with pytest, by Brian Okken (2017).

40.2 Talks and blog postings

- Webinar: pytest: Test Driven Development für Python (German), Florian Bruhin, via mylearning.ch, 2020
- Webinar: Simplify Your Tests with Fixtures, Oliver Bestwalter, via JetBrains, 2020
- Training: Introduction to pytest - simple, rapid and fun testing with Python, Florian Bruhin, PyConDE 2019
- Abridged metaprogramming classics - this episode: pytest, Oliver Bestwalter, PyConDE 2019 (repository, recording)
- Testing PySide/PyQt code easily using the pytest framework, Florian Bruhin, Qt World Summit 2019 (slides, recording)
- pytest: recommendations, basic packages for testing in Python and Django, Andreu Vallbona, PyBCN June 2019.
- pytest: recommendations, basic packages for testing in Python and Django, Andreu Vallbona, PyconES 2017 (slides in english, video in spanish)
- pytest advanced, Andrew Svetlov (Russian, PyCon Russia, 2016).
- 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.
- 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.

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
- 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
- bbfreeze create standalone executables from Python scripts
- pdb++ a fancier version of PDB
- pudb full-screen console debugger for python
- 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

41.1 Some organisations using pytest

- Square Kilometre Array, Cape Town

- Some Mozilla QA people use pytest to distribute their Selenium tests

- 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*)