# Usage and Invocations

## Calling pytest through `python -m pytest`

*New in version 2.0.*

You can invoke testing through the Python interpreter from the command line:

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

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

## Possible exit codes

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

Exit code 0: All tests were collected and passed successfully
Exit code 1: Tests were collected and run but some of the tests failed
Exit code 2: Test execution was interrupted by the user
Exit code 3: Internal error happened while executing tests
Exit code 4: pytest command line usage error
Exit code 5: No tests were collected

## Getting help on version, option names, environment variables

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

## Stopping after the first (or N) failures

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

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

## Specifying tests / selecting tests

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

Run tests in a module

```
pytest test_mod.py
```

Run tests in a directory

```
pytest testing/
```

Run tests by keyword expressions

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

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

`TestMyClass.test_method_simple`.

Run tests by node ids

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

To run a specific test within a module:

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

Another example specifying a test method in the command line:

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

Run tests by marker expressions

```
pytest -m slow
```

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

For more information see marks.

Run tests from packages

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

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

## Modifying Python traceback printing

Examples for modifying traceback printing:

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

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

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

## Dropping to PDB (Python Debugger) on failures

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

```
pytest --pdb
```

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

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

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

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

## Setting breakpoints

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

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

## Profiling test execution duration

To get a list of the slowest 10 test durations:

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

## Creating JUnitXML format files

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

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

to create an XML file at `path`.

*New in version 3.1.*

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

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

### record_xml_property

*New in version 2.8.*

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

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

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

```
<testcase classname="test_function" file="test_function.py" line="0" name="test_function" time="0.0009">
  <properties>
    <property name="example_key" value="1" />
  </properties>
</testcase>
```

Warning:

`record_xml_property` is an experimental feature, and its interface might be replaced by something more powerful and general in future versions. The functionality per-se will be kept, however.

Currently it does not work when used with the `pytest-xdist` plugin.

Also please note that using this feature will break any schema verification. This might be a problem when used with some CI servers.

### LogXML: add_global_property

*New in version 3.0.*

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

```
import pytest

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

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

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

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

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

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

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

Warning:

This is an experimental feature, and its interface might be replaced by something more powerful and general in future versions. The functionality per-se will be kept.

## Creating resultlog format files

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

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

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

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

```
pytest --resultlog=path
```

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

## Sending test report to online pastebin service

Creating a URL for each test failure:

```
pytest --pastebin=failed
```

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

Creating a URL for a whole test session log:

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

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

## Disabling plugins

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

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

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

## Calling pytest from Python code

*New in version 2.0.*

You can invoke `pytest` from Python code directly:

```
pytest.main()
```

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

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

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

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

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

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

```
$ python myinvoke.py
*** test run reporting finishing
```