

# Writing plugins

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

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

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

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

## Plugin discovery order at tool startup

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

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

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

- by recursively loading all plugins specified by the `pytest_plugins` variable in `conftest.py` files

## conftest.py: local per-directory plugins

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

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

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

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

Here is how you might run it:

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

Note:

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

See also: [pytest import mechanisms and sys.path/PYTHONPATH](#).

## Writing your own plugin

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

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

All of these plugins implement the documented [well specified hooks](#) to extend and add functionality.

Note:

Make sure to check out the excellent [cookiecutter-pytest-plugin](#) project, which is a [cookiecutter template](#) for authoring plugins.

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

Also consider [contributing your plugin to pytest-dev](#) once it has some happy users other than yourself.

## Making your plugin installable by others

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

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

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

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

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

 v: latest ▼

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

Note:

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

## Assertion Rewriting

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

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

**register\_assert\_rewrite(\*names)** [source]

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

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

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

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

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

With the following typical `setup.py` extract:

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

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

```
import pytest

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

## Requiring/Loading plugins in a test module or conftest file

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

 v: latest ▼

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

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

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

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

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

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

## Accessing another plugin by name

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

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

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

## Testing plugins

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

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

```
# content of conftest.py
```

```
pytest_plugins = ["pytester"]
```

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

This will allow you to use the [`testdir`](#) fixture for testing your plugin code.

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

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

```
import pytest
```

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

 v: latest ▾

```

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

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

    return _hello

```

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

```

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

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

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

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

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

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

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

```

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

## Writing hook functions

### hook function validation and execution

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

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

Let's look at a possible implementation:

 v: latest ▼

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

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

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

## firstresult: stop at first non-None result

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

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

## hookwrapper: executing around other hooks

*New in version 2.7.*

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

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

Here is an example definition of a hook wrapper:

```
import pytest

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

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

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

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

## Hook function ordering / call example

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

```
# Plugin 1
@pytest.hookimpl(tryfirst=True)
```

 v: latest ▼

```
def pytest_collection_modifyitems(items):
    # will execute as early as possible

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

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

Here is the order of execution:

1. Plugin3's `pytest_collection_modifyitems` called until the yield point because it is a hook wrapper.
2. Plugin1's `pytest_collection_modifyitems` is called because it is marked with `tryfirst=True`.
3. Plugin2's `pytest_collection_modifyitems` is called because it is marked with `trylast=True` (but even without this mark it would come after Plugin1).
4. Plugin3's `pytest_collection_modifyitems` then executing the code after the yield point. The yield receives a `CallOutcome` instance which encapsulates the result from calling the non-wrappers. Wrappers shall not modify the result.

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

## Declaring new hooks

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

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

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

For an example, see `newhooks.py` from `xdist`.

## Optionally using hooks from 3rd party plugins

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

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

```
# contents of myplugin.py

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

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

def pytest_configure(config):
```

 v: latest ▼

```
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.

## pytest hook reference

### Initialization, command line and configuration hooks

**pytest\_load\_initial\_conftests**(*early\_config, parser, args*) [source]

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

**pytest\_cmdline\_preparse**(*config, args*) [source]

(deprecated) modify command line arguments before option parsing.

**pytest\_cmdline\_parse**(*pluginmanager, args*) [source]

return initialized config object, parsing the specified args.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_adoption**(*parser*) [source]

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

#### Note:

This function should be implemented only in plugins or `conftest.py` files situated at the tests root directory due to how [pytest discovers plugins during startup](#).

Parameters: *parser* – To add command line options, call [parser.adoption\(...\)](#). To add ini-file values call [parser.addini\(...\)](#).

Options can later be accessed through the [config](#) object, respectively:

- [config.getoption\(name\)](#) to retrieve the value of a command line option.
- [config.getini\(name\)](#) to retrieve a value read from an ini-style file.

The config object is passed around on many internal objects via the `.config` attribute or can be retrieved as the `pytestconfig` fixture or accessed via (deprecated) `pytest.config`.

**pytest\_cmdline\_main**(*config*) [source]

called for performing the main command line action. The default implementation will invoke the configure hooks and `runtest_mainloop`.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_configure**(*config*) [source]

Allows plugins and conftest files to perform initial configuration.

This hook is called for every plugin and initial conftest file after command line options have been parsed.

After that, the hook is called for other conftest files as they are imported.

Parameters: *config* ([\\_pytest.config.Config](#)) – pytest config object

**pytest\_unconfigure**(*config*) [source]

called before test process is exited.

 v: latest ▼



## Generic “runtest” hooks

All runtest related hooks receive a `pytest.Item` object.

**pytest\_runtest\_protocol**(*item*, *nextitem*) [source]

implements the runtest\_setup/call/teardown protocol for the given test item, including capturing exceptions and calling reporting hooks.

Parameters:

- *item* – test item for which the runtest protocol is performed.
- *nextitem* – the scheduled-to-be-next test item (or None if this is the end my friend). This argument is passed on to `pytest_runtest_teardown()`.

Return: True if no further hook implementations should be invoked.  
boolean:

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_runtest\_setup**(*item*) [source]

called before `pytest_runtest_call(item)`.

**pytest\_runtest\_call**(*item*) [source]

called to execute the test item.

**pytest\_runtest\_teardown**(*item*, *nextitem*) [source]

called after `pytest_runtest_call`.

Parameters:

- *nextitem* – the scheduled-to-be-next test item (None if no further test item is scheduled). This argument can be used to perform exact teardowns, i.e. calling just enough finalizers so that *nextitem* only needs to call setup-functions.

**pytest\_runtest\_makereport**(*item*, *call*) [source]

return a `pytest.runner.TestReport` object for the given `pytest.Item` and `pytest.runner.CallInfo`.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

For deeper understanding you may look at the default implementation of these hooks in `_pytest.runner` and maybe also in `_pytest.pdb` which interacts with `_pytest.capture` and its input/output capturing in order to immediately drop into interactive debugging when a test failure occurs.

The `_pytest.terminal` reported specifically uses the reporting hook to print information about a test run.

## Collection hooks

pytest calls the following hooks for collecting files and directories:

**pytest\_ignore\_collect**(*path*, *config*) [source]

return True to prevent considering this path for collection. This hook is consulted for all files and directories prior to calling more specific hooks.


Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_collect\_directory**(*path*, *parent*) [source]

called before traversing a directory for collection files.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_collect\_file**(*path*, *parent*) [source]

return collection Node or None for the given path. Any new node needs to have the specified parent as a parent  [v: latest](#) ▼

For influencing the collection of objects in Python modules you can use the following hook:

**pytest\_pycollect\_makeitem**(*collector, name, obj*) [source]

return custom item/collector for a python object in a module, or None.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_generate\_tests**(*metafunc*) [source]

generate (multiple) parametrized calls to a test function.

**pytest\_make\_parametrize\_id**(*config, val, argname*) [source]

Return a user-friendly string representation of the given *val* that will be used by `@pytest.mark.parametrize` calls. Return None if the hook doesn't know about *val*. The parameter name is available as *argname*, if required.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

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

**pytest\_collection\_modifyitems**(*session, config, items*) [source]

called after collection has been performed, may filter or re-order the items in-place.

## Reporting hooks

Session related reporting hooks:

**pytest\_collectstart**(*collector*) [source]

collector starts collecting.

**pytest\_itemcollected**(*item*) [source]

we just collected a test item.

**pytest\_collectreport**(*report*) [source]

collector finished collecting.

**pytest\_deselected**(*items*) [source]

called for test items deselected by keyword.

**pytest\_report\_header**(*config, startdir*) [source]

return a string or list of strings to be displayed as header info for terminal reporting.

Parameters:

- *config* – the pytest config object.
- *startdir* – `py.path` object with the starting dir

### Note:

This function should be implemented only in plugins or `conftest.py` files situated at the tests root directory due to how [pytest discovers plugins during startup](#).

**pytest\_report\_collectionfinish**(*config, startdir, items*) [source]

*New in version 3.2.*

return a string or list of strings to be displayed after collection has finished successfully.

This strings will be displayed after the standard “collected X items” message.

Parameters:

- *config* – the pytest config object.
- *startdir* – `py.path` object with the starting dir

 [v: latest](#) ▼

- `items` – list of pytest items that are going to be executed; this list should not be modified.

**pytest\_report\_teststatus**(*report*) [source]

return result-category, shortletter and verbose word for reporting.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_terminal\_summary**(*terminalreporter, exitstatus*) [source]

add additional section in terminal summary reporting.

**pytest\_fixture\_setup**(*fixturedef, request*) [source]

performs fixture setup execution.

Stops at first non-None result, see [firstresult: stop at first non-None result](#)

**pytest\_fixture\_post\_finalizer**(*fixturedef*) [source]

called after fixture teardown, but before the cache is cleared so the fixture result cache `fixturedef.cached_result` can still be accessed.

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

**pytest\_runtest\_logreport**(*report*) [source]

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

You can also use this hook to customize assertion representation for some types:

**pytest\_assertrepr\_compare**(*config, op, left, right*) [source]

return explanation for comparisons in failing assert expressions.

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

## Debugging/Interaction hooks

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

**pytest\_internalerror**(*excrepr, excinfo*) [source]

called for internal errors.

**pytest\_keyboard\_interrupt**(*excinfo*) [source]

called for keyboard interrupt.

**pytest\_exception\_interact**(*node, call, report*) [source]

called when an exception was raised which can potentially be interactively handled.

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

**pytest\_enter\_pdb**(*config*) [source]

called upon `pdb.set_trace()`, can be used by plugins to take special action just before the python debugger enters in interactive mode.

Parameters: `config` ([\\_pytest.config.Config](#)) – pytest config object

## Reference of objects involved in hooks

class **Config**

 v: latest ▾

access to configuration values, pluginmanager and plugin hooks.

**option** = None

access to command line option as attributes. (deprecated), use `getoption()` instead

**pluginmanager** = None

a pluginmanager instance

**add\_cleanup**(*func*)

[source]

Add a function to be called when the config object gets out of use (usually coinciding with `pytest_unconfigure`).

**warn**(*code, message, fslocation=None, nodeid=None*)

[source]

generate a warning for this test session.

*classmethod* **fromdictargs**(*option\_dict, args*)

[source]

constructor useable for subprocesses.

**addini***value\_line*(*name, line*)

[source]

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*)

[source]

return configuration value from an ini file. If the specified name hasn't been registered through a prior `parser.addini` call (usually from a plugin), a `ValueError` is raised.

**getoption**(*name, default=<NOTSET>, skip=False*)

[source]

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 exist or has a None value.

**getvalue**(*name, path=None*)

[source]

(deprecated, use `getoption()`)

**getvalueorskip**(*name, path=None*)

[source]

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

*class* **Parser**

[source]

Parser for command line arguments and ini-file values.

Variables: `extra_info` – dict of generic param -> value to display in case there's an error processing the command line arguments.

**getgroup**(*name, description="", after=None*)

[source]

get (or create) a named option Group.

Name: name of the option group.  
 Description: long description for `-help` output.  
 After: name of other group, used for ordering `-help` output.

The returned group object has an `addoption` method with the same signature as `parser.addoption` but will be shown in the respective group in the output of `pytest -h`.

**addoption**(*\*opts, \*\*attrs*)

[source]

register a command line option.

Opts: option names, can be short or long options.  
 Attrs: same attributes which the `add_option()` function of the `argparse` library accepts.

 v: latest ▾

After command line parsing options are available on the pytest config object via `config.option.NAME` where `NAME` is usually set by passing a `dest` attribute, for example `addoption("--long", dest="NAME", ...)`.

**parse\_known\_args**(*args, namespace=None*) [source]

parses and returns a namespace object with known arguments at this point.

**parse\_known\_and\_unknown\_args**(*args, namespace=None*) [source]

parses and returns a namespace object with known arguments, and the remaining arguments unknown at this point.

**addini**(*name, help, type=None, default=None*) [source]

register an ini-file option.

Name: name of the ini-variable

Type: type of the variable, can be `pathlist`, `args`, `linelist` or `bool`.

Default: default value if no ini-file option exists but is queried.

The value of ini-variables can be retrieved via a call to `config.getini(name)`.

**class Node** [source]

base class for Collector and Item the test collection tree. Collector subclasses have children, Items are terminal nodes.

**name** = *None*

a unique name within the scope of the parent node

**parent** = *None*

the parent collector node.

**config** = *None*

the pytest config object

**session** = *None*

the session this node is part of

**fspath** = *None*

filesystem path where this node was collected from (can be *None*)

**keywords** = *None*

keywords/markers collected from all scopes

**extra\_keyword\_matches** = *None*

allow adding of extra keywords to use for matching

**ihook**

fspath sensitive hook proxy used to call pytest hooks

**warn**(*code, message*) [source]

generate a warning with the given code and message for this item.

**nodeid**

a ::-separated string denoting its collection tree address.

**listchain**() [source]

return list of all parent collectors up to self, starting from root of collection tree.

**add\_marker**(*marker*) [source]

dynamically add a marker object to the node.

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

 [v: latest](#) ▼

**get\_marker(*name*)** [source]

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

**listextrakeywords()** [source]

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

**addfinalizer(*fin*)** [source]

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*)** [source]

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

**class Collector** [source]

Bases: `__pytest.main.Node`

Collector instances create children through collect() and thus iteratively build a tree.

**exception CollectError** [source]

Bases: `exceptions.Exception`

an error during collection, contains a custom message.

Collector.**collect()** [source]

returns a list of children (items and collectors) for this collection node.

Collector.**repr\_failure(*excinfo*)** [source]

represent a collection failure.

**class Item** [source]

Bases: `__pytest.main.Node`

a basic test invocation item. Note that for a single function there might be multiple test invocation items.

**add\_report\_section(*when, key, content*)** [source]

Adds a new report section, similar to what's done internally to add stdout and stderr captured output:

```
item.add_report_section("call", "stdout", "report section contents")
```

Parameters:

- *when* (`str`) – One of the possible capture states, "setup", "call", "teardown".
- *key* (`str`) – Name of the section, can be customized at will. Pytest uses "stdout" and "stderr" internally.
- *content* (`str`) – The full contents as a string.

**class Module** [source]

Bases: `__pytest.main.File`, `__pytest.python.PyCollector`

Collector for test classes and functions.

**class Class** [source]

Bases: `__pytest.python.PyCollector`

Collector for test methods.

**class Function** [source]

Bases: `__pytest.python.FunctionMixin`, `__pytest.main.Item`, `__pytest.compat.FuncargnamesCompatAttr`

a Function Item is responsible for setting up and executing a Python test function.

 v: latest ▾

**originalname** = *None*

original function name, without any decorations (for example parametrization adds a "[...]" suffix to function names).

*New in version 3.0.*

**function**

underlying python 'function' object

**runtest()**[\[source\]](#)

execute the underlying test function.

*class* **FixtureDef**[\[source\]](#)

A container for a factory definition.

*class* **CallInfo**[\[source\]](#)

Result/Exception info a function invocation.

**when** = *None*

context of invocation: one of "setup", "call", "teardown", "memocollect"

**excinfo** = *None*

None or ExceptionInfo object.

*class* **TestReport**[\[source\]](#)

Basic test report object (also used for setup and teardown calls if they fail).

**nodeid** = *None*

normalized collection node id

**location** = *None*

a (filesystempath, lineno, domaininfo) tuple indicating the actual location of a test item - it might be different from the collected one e.g. if a method is inherited from a different module.

**keywords** = *None*

a name -> value dictionary containing all keywords and markers associated with a test invocation.

**outcome** = *None*

test outcome, always one of "passed", "failed", "skipped".

**longrepr** = *None*

None or a failure representation.

**when** = *None*

one of 'setup', 'call', 'teardown' to indicate runtest phase.

**sections** = *None*

list of pairs (str, str) of extra information which needs to be marshallable. Used by pytest to add captured text from stdout and stderr, but may be used by other plugins to add arbitrary information to reports.

**duration** = *None*

time it took to run just the test

**capstderr**

Return captured text from stderr, if capturing is enabled

*New in version 3.0.*

**capstdout** [v: latest](#) ▼

Return captured text from stdout, if capturing is enabled

*New in version 3.0.*

## longreprtext

Read-only property that returns the full string representation of longrepr.

*New in version 3.0.*

**class \_CallOutcome** [source]

Outcome of a function call, either an exception or a proper result. Calling the `get_result` method will return the result or reraise the exception raised when the function was called.

**get\_plugin\_manager()** [source]

Obtain a new instance of the `pytest.config.PytestPluginManager`, with default plugins already loaded.

This function can be used by integration with other tools, like hooking into pytest to run tests into an IDE.

**class PytestPluginManager** [source]

Bases: `pytest.vendored_packages.pluggy.PluginManager`

Overwrites `pluggy.PluginManager` to add pytest-specific functionality:

- loading plugins from the command line, `PYTEST_PLUGIN` env variable and `pytest_plugins` global variables found in plugins being loaded;
- `conftest.py` loading during start-up;

**addhooks(*module\_or\_class*)** [source]

*Deprecated since version 2.8.*

Use `pluggy.PluginManager.add_hookspecs` instead.

**parse\_hookimpl\_opts(*plugin*, *name*)** [source]

**parse\_hookspec\_opts(*module\_or\_class*, *name*)** [source]

**register(*plugin*, *name*=None)** [source]

**getplugin(*name*)** [source]

**hasplugin(*name*)** [source]

Return True if the plugin with the given name is registered.

**pytest\_configure(*config*)** [source]

**consider\_preparse(*args*)** [source]

**consider\_pluginarg(*arg*)** [source]

**consider\_conftest(*conftestmodule*)** [source]

**consider\_env()** [source]

**consider\_module(*mod*)** [source]

**import\_plugin(*modname*)** [source]

**class PluginManager** [source]

Core Pluginmanager class which manages registration of plugin objects and 1:N hook calling.

You can register new hooks by calling `add_hookspec(module_or_class)`. You can register plugin objects (which contain hooks) by calling `register(plugin)`. The Pluginmanager is initialized with a prefix that is searched for in the names of the dict of registered plugin objects. An optional `excludefunc` allows to blacklist names which are not considered as hooks despite a matching prefix.

 v: latest ▼



For debugging purposes you can call `enable_tracing()` which will subsequently send debug information to the trace helper.

**register(plugin, name=None)** [source]  
 Register a plugin and return its canonical name or None if the name is blocked from registering. Raise a `ValueError` if the plugin is already registered.

**unregister(plugin=None, name=None)** [source]  
 unregister a plugin object and all its contained hook implementations from internal data structures.

**set\_blocked(name)** [source]  
 block registrations of the given name, unregister if already registered.

**is\_blocked(name)** [source]  
 return True if the name blocks registering plugins of that name.

**add\_hookspecs(module\_or\_class)** [source]  
 add new hook specifications defined in the given module\_or\_class. Functions are recognized if they have been decorated accordingly.

**get\_plugins()** [source]  
 return the set of registered plugins.

**is\_registered(plugin)** [source]  
 Return True if the plugin is already registered.

**get\_canonical\_name(plugin)** [source]  
 Return canonical name for a plugin object. Note that a plugin may be registered under a different name which was specified by the caller of `register(plugin, name)`. To obtain the name of an registered plugin use `get_name(plugin)` instead.

**get\_plugin(name)** [source]  
 Return a plugin or None for the given name.

**has\_plugin(name)** [source]  
 Return True if a plugin with the given name is registered.

**get\_name(plugin)** [source]  
 Return name for registered plugin or None if not registered.



**check\_pending()** [source]  
 Verify that all hooks which have not been verified against a hook specification are optional, otherwise raise `PluginValidationError`

**load\_setuptools\_entrypoints(entrypoint\_name)** [source]  
 Load modules from querying the specified setuptools entrypoint name. Return the number of loaded plugins.

**list\_plugin\_distinfo()** [source]  
 return list of distinfo/plugin tuples for all setuptools registered plugins.

**list\_name\_plugin()** [source]  
 return list of name/plugin pairs.

**get\_hookcallers(plugin)** [source]  
 get all hook callers for the specified plugin.

**add\_hookcall\_monitoring(before, after)** [source]  
 add before/after tracing functions for all hooks and return an undo function which, when called, will remove  v: latest 

added tracers.

`before(hook_name, hook_impls, kwargs)` will be called ahead of all hook calls and receive a hookcaller instance, a list of `HookImpl` instances and the keyword arguments for the hook call.

`after(outcome, hook_name, hook_impls, kwargs)` receives the same arguments as before but also a `CallOutcome` object which represents the result of the overall hook call.

**enable\_tracing()** [source]

enable tracing of hook calls and return an undo function.

**subset\_hook\_caller(name, remove\_plugins)** [source]

Return a new `HookCaller` instance for the named method which manages calls to all registered plugins except the ones from `remove_plugins`.

**class Testdir** [source]

Temporary test directory with tools to test/run pytest itself.

This is based on the `tmpdir` fixture but provides a number of methods which aid with testing pytest itself. Unless `chdir()` is used all methods will use `tmpdir` as current working directory.

Attributes:

`tmpdir`: The `py.path.local` instance of the temporary directory.

`plugins`: A list of plugins to use with `parseconfig()` and `runpytest()`. Initially this is an empty list but plugins can be added to the list. The type of items to add to the list depend on the method which uses them so refer to them for details.

**makeconftest(source)** [source]

Write a `conftest.py` file with 'source' as contents.

**makepyfile(\*args, \*\*kwargs)** [source]

Shortcut for `.makefile()` with a `.py` extension.

**runpytest(\*args, \*\*kwargs)** [source]

Run pytest inline or in a subprocess, depending on the command line option `"--runpytest"` and return a `RunResult`.

**runpytest\_inprocess(\*args, \*\*kwargs)** [source]

Return result of running pytest in-process, providing a similar interface to what `self.runpytest()` provides.

**runpytest\_subprocess(\*args, \*\*kwargs)** [source]

Run pytest as a subprocess with given arguments.

Any plugins added to the `plugins` list will added using the `-p` command line option. Additionally `--basetemp` is used put any temporary files and directories in a numbered directory prefixed with `"runpytest-"` so they do not conflict with the normal numbered pytest location for temporary files and directories.

Returns a `RunResult`.

**class RunResult** [source]

The result of running a command.

Attributes:

`Ret`: The return value.

`Outlines`: List of lines captured from stdout.

`Errlines`: List of lines captures from stderr.

 v: latest ▾

Stdout: **LineMatcher** of stdout, use `stdout.str()` to reconstruct stdout or the commonly used `stdout.fnmatch_lines()` method.

Stderr: **LineMatcher** of stderr.

Duration: Duration in seconds.

**assert\_outcomes**(*passed=0, skipped=0, failed=0, error=0*) [source]

assert that the specified outcomes appear with the respective numbers (0 means it didn't occur) in the text output from a test run.

**parseoutcomes**() [source]

Return a dictionary of `outcomestring->num` from parsing the terminal output that the test process produced.

*class* **LineMatcher** [source]

Flexible matching of text.

This is a convenience class to test large texts like the output of commands.

The constructor takes a list of lines without their trailing newlines, i.e. `text.splitlines()`.

**fnmatch\_lines**(*lines2*) [source]

Search the text for matching lines.

The argument is a list of lines which have to match and can use glob wildcards. If they do not match an `pytest.fail()` is called. The matches and non-matches are also printed on stdout.

**fnmatch\_lines\_random**(*lines2*) [source]

Check lines exist in the output.

The argument is a list of lines which have to occur in the output, in any order. Each line can contain glob wildcards.

**get\_lines\_after**(*fnline*) [source]

Return all lines following the given line in the text.

The given line can contain glob wildcards.

**str**() [source]

Return the entire original text.