Understanding Hooks in Pytest

[Parag Kamble](https://paragkamble.medium.com)
·
·
Jan 6, 2023

https://paragkamble.medium.com/understanding-hooks-in-pytest-892e91edbdb7#:~:text=This%20hook%20is%20called%20when,the%20behavior%20of%20the%20script.

Hooks are a key part of pytest’s plugin system and are used by plugins and by pytest itself to extend the functionality of pytest. Hooks allow plugins to register custom code to be run at specific points during the execution of pytest, such as before and after tests are run, or when exceptions are raised. They provide a flexible way to customize the behavior of pytest and to extend its functionality.

Pytest is a popular testing framework for Python that allows us to write tests for our code and run them automatically. One of the powerful features of pytest is the ability to use hooks, which are functions that are called at specific points during the execution of a test. These hooks use to perform actions before or after a test, or to modify the behavior of a test.

Using hooks in pytest is easy. To define a hook, simply need to create a function with the appropriate name and signature and then register it as a hook by adding it to `pytest.ini` file or by decorating it with `@pytest.hookimpl`.

For example, here is an example of define a hook that is called before a test is set up to run:

```
@pytest.hookimpl(hookwrapper=True)
def pytest_runtest_setup(item):
    # Set up resources or perform other tasks before the test is run
    yield
    # Tear down resources or perform other tasks after the test is run
```

The `hookwrapper` argument tells pytest to wrap the hook function in a try-except block, so that any exceptions that are raised during the hook's execution will be caught and reported as test failures.

One of the benefits of using hooks in pytest is that they allow you to modularize and reuse code across multiple tests. This can make it easier to maintain our tests and to add new tests as our code evolves.

Hooks are mainly categories in following parts`Bootstrapping Hooks,` `initialization hooks` , `Colletion Hooks` , `Test running(runtest) hooks` , `Reporting Hooks` and `Debugging and Interaction Hooks` . Lets see all of these with examples.

BootStrapping hooks

In pytest, bootstrapping hooks are special functions that are called at the very beginning and end of the test run.

Here is a list of some of the bootstrapping hooks available in pytest:

- `pytest_load_initial_conftests`: called to load the initial conftest files that are used to configure the test run. Conftest files are Python modules that contain hook functions that are used to customize the behavior of the tests.
- `pytest_cmdline_parse` is a hook provided by the Pytest testing framework. This hook is called when the Pytest command-line tool is run, and it allows plugins to modify the command-line arguments that are passed to Pytest.
- `pytest_cmdline_main`: called when the pytest command line script is run, allowing you to customize the behavior of the script.

In following example, the `pytest_load_initial_conftests` hook is used to register an additional plugin module called "my_plugin". This plugin module will be loaded when the test runner starts up, and will be available to all tests that are run.

```
import pytest

def pytest_load_initial_conftests(early_config, parser, args):
    # Load an additional plugin module called "my_plugin"
    early_config.pluginmanager.register(name="my_plugin", 
module="my_plugin")
```

Bootstrapping hooks can be very useful for tasks such as setting up and tearing down resources that are needed by all of our tests, or for aggregating test results.

Initialization hooks

Initialization hooks are called at the beginning of the test run, after the bootstrapping hooks have been executed. These hooks are used to initialize resources or perform other tasks that are needed before the tests are run.

Here are the some example of Initialization Hooks in pytest:

1. `pytest_addoption`: this hook is called when the test runner is starting up, and allows us to add additional command line options to the test runner.
2. `pytest_configure`: this hook is called after the command line options have been parsed, and allows us to configure the test runner or perform any additional setup tasks.
3. `pytest_sessionstart`: this hook is called when the test session is starting, and allows us to perform any setup tasks that are needed before the tests are run.
4. `pytest_sessionfinish`: this hook is called after all tests have been run, and allows us to perform any teardown tasks or generate reports.

```
@pytest.hookimpl
def pytest_sessionstart(session):
    print("Test session is starting")
    # Perform setup tasks here
```

Initialization hooks hooks can be used to perform additional setup or teardown tasks, or to modify the behavior of the test runner in some way.

Collection Hooks.

collection hooks are special functions that are called during the test collection process, which is the process of discovering and collecting the test functions and test modules that make up the test suite. These hooks can be use to customize the way that tests are collected and to add additional tests to the collection.

There are several types of collection hooks that we can use in pytest:

1. `pytest_collectstart`: this hook is called when the test collection process is starting, and allows us to modify the configuration or perform any setup tasks before the tests are collected.
2. `pytest_collectreport`: this hook is called after a test collection has been completed, and allows us to access the results of the collection. We can use this hook to inspect the collected tests, or to report any errors or warnings that occurred during collection.

3. `pytest_ignore_collect`: this hook is called when a test or test module is being marked as ignored, and allows us to override the default ignore behavior.

4. `pytest_collect_file`: this hook is called when a test file is being collected, and allows us to customize the way that tests are discovered and collected from the file.

In Following example, the `pytest_collect_file` hook is used to collect tests from files with the `.myext` extension. When a file with this extension is found, the hook creates a custom `MyFile` object to represent the file, and returns it to the test runner. The `MyFile` class then implements a `collect` method to discover the tests in the file, and creates custom `MyItem` objects to represent each test. Finally, the `MyItem` class implements a `runtest` method to run the test.

```
import pytest

def pytest_collect_file(path, parent):
    if path.ext == ".myext":
        # Collect tests from files with the ".myext" extension
        return MyFile(path, parent)

class MyFile(pytest.File):
    def collect(self):
        # Implement custom logic to discover tests in this file
        tests = []
        for name in self.get_test_names():
            tests.append(MyItem(name, self))
        return tests

class MyItem(pytest.Item):
    def runtest(self):
        # Implement custom logic to run this test
        pass
```

Collection hooks can be very useful for tasks such as ignoring specific test items or adding additional tests to the test collection.

Test running (runtest) hooks.

Test running (runtest) hooks can be used to customize the way that tests are run and to perform actions before or after a test is run.

Here is some examples of runtest hooks that we can use in pytest:

1. `pytest_runtest_setup`: this hook is called before a test is set up to run, and allows to perform any additional setup tasks that are needed.
2. `pytest_runtest_call`: this hook is called when a test is called, and allow to modify the arguments that are passed to the test or to customize the way that the test is called.
3. `pytest_runtest_teardown`: this hook is called after a test has completed, and allow to perform any teardown tasks that are needed.
4. `pytest_runtest_logreport`: this hook is called after a test has completed, and allows to access the test report and modify it before it is logged.

In Following example, the `pytest_runtest_setup` hook is used to perform additional setup tasks before the `test_something` test is run. The hook function is called with a `item` argument, which represents the test that is about to be run. In this case, the hook simply prints a message to the console, but you can use it to perform any necessary setup tasks before the test is called.

```
import pytest

@pytest.hookimpl
def pytest_runtest_setup(item):
    print("Setting up test:", item.name)
    # Perform setup tasks here

def test_something():
    # Test code goes here
    pass
```

This hook function would be called for every test that is about to be run, and it would allow us to perform any necessary setup tasks before the test is called.

Reporting Hooks

In pytest, reporting hooks are called at various points during the test run to report the progress and results of the tests. These hooks allow us to customize the way that test results are reported and to perform actions based on the test results.

There are several types of reporting hooks that we can use in pytest:

1. `pytest_report_header`: this hook is called at the beginning of the test run, and allows us to add additional information to the test report header.

2. `pytest_report_teststatus`: this hook is called after a test has completed, and allows us to modify the test's status and outcome.

3. `pytest_terminal_summary`: this hook is called after all tests have completed, and allows us to customize the way that the test results are summarized in the terminal.

4. `pytest_logreport`: this hook is called for every test report that is generated, and allows us to access the report and modify it before it is logged.

In following example, the `pytest_report_teststatus` hook is used to customize the way that test results are reported. The hook function is called with a `report` argument, which contains information about the test that has just completed. If the test has failed, the hook function returns the tuple `("FAIL", "F", "FAILED")`, which tells pytest to report the test as a failure with the status "FAIL" and the short-form status "F". If the test has passed, the hook function returns the tuple `("PASS", "P", "PASSED")`, which tells pytest to report the test as a success with the status "PASS" and the short-form status "P".

```
import pytest

@pytest.hookimpl
def pytest_report_teststatus(report):
    if report.when == "call":
        if report.outcome == "failed":
            return "FAIL", "F", "FAILED"
        elif report.outcome == "passed":
            return "PASS", "P", "PASSED"

def test_something():
    assert 1 + 1 == 2
```

This hook function would be called for every test that completes, and it would modify the test's status and outcome based on whether it passed or failed.

Debugging/Interaction Hooks

In pytest, debugging/interaction hooks are allow us to interact with the test process or to debug issues that may arise.

Here is a list of some of the debugging/interaction hooks that are available in pytest, along with a brief description of each:

- `pytest_exception_interact`: called when an exception is raised during the test run and allow to interact with the exception.
- `pytest_enter_pdb`: called when entering the Python debugger during the test run and allow to customize the debugger's behavior.
- `pytest_leave_pdb`: called when leaving the Python debugger during the test run and allow to customize the debugger's behavior.
- `pytest_keyboard_interrupt`: called when a keyboard interrupt is received during the test run and allow to handle the interrupt.
- `pytest_internalerror`: called when an internal error occurs during the test run and allow to handle the error.

In Following example, the `pytest_exception_interact` hook is used to customize the way that `MyCustomException` is handled during the test run. The hook function is called with a `node` argument, which represents the test that is being run, a `call` argument, which contains information about the exception that was raised, and a `report` argument, which contains the test report for the test. If the exception that was raised is an instance of `MyCustomException`, the hook function modifies the test report to indicate that the test was skipped, and sets the `longrepr` field of the report to a custom message.

```
import pytest

@pytest.hookimpl
def pytest_exception_interact(node, call, report):
    if isinstance(call.excinfo.value, MyCustomException):
        # Customize the way that MyCustomException is handled
        report.outcome = "skipped"
        report.longrepr = "MyCustomException was raised"

def test_something():
    raise MyCustomException
```

This is just a simple example of how we can use the `pytest_exception_interact` hook to customize the way that exceptions are handled during a test run. We can use this hook to do much more complex tasks, such as logging additional information about the exception or modifying the test outcome based on custom criteria. Overall, `pytest_exception_interact` hook is a very useful hook which provides the ability to customize the way that exceptions are handled during a test run.

How and where to define hook in pytest?

hooks are defined using the `@pytest.hookimpl` decorator, which is applied to a function that is called at a specific point during the test run. To define a hook in pytest we need to specify the hook name as an argument to the `@pytest.hookimpl` decorator.

To define a test case hook we can use the `@pytest.hookimpl` decorator and specify the hook name as an argument, as shown in the following code:

```
@pytest.hookimpl(hookwrapper=True)
def pytest_runtest_setup(item):
    # Your code here
```

Hooks can be defined in any module or package in the test suite, and they will be automatically discovered by pytest when the tests are run. It is generally a good idea to define hooks in a separate module or package, so that they can be easily shared and reused across multiple tests.

In addition to defining hooks in our own code, we can also use hooks provided by pytest plugins or third-party libraries to extend the functionality of your tests. To use these hooks, you will need to install the appropriate plugin or library and import the hook functions into our test code.

Pytest Hooks Tree

hooks are organized into a tree structure that reflects the order in which they are called during the test run. At the root of the tree are the bootstrapping hooks, which are called at the very beginning and end of the test run. These hooks are used to set up and tear down resources that are needed by all of the tests.

Below the bootstrapping hooks in the tree are the initialization hooks, which are called after the bootstrapping hooks and are used to initialize resources or perform other tasks that are needed before the tests are run.

Below the initialization hooks in the tree are the collection hooks, which are called during the test collection process and are used to customize the way that tests are collected and to add additional tests to the collection.

Below the collection hooks in the tree are the test running (runtest) hooks, which are called at various points during the execution of a test and are used to customize the way that tests are run and to perform actions before or after a test is run.

Below the test running hooks in the tree are the reporting hooks, which are called at various points during the test run to report the progress and results of the tests and to customize the way that test results are reported.

Finally, at the bottom of the tree are the debugging/interaction hooks, which are called at various points during the test run to allow you to interact with the test process or to debug issues that may arise.

This hierarchy of hooks allows you to customize the behavior of our tests at different levels, depending on our needs. We can use the hooks at the root of the tree to set up and tear down resources that are needed by all of our tests, and it can use the hooks lower down in the tree to customize the way that specific tests are run and reported.

```
root
└── pytest_cmdline_main
 ├── pytest_plugin_registered
 ├── pytest_configure
 │ └── pytest_plugin_registered
 ├── pytest_sessionstart
 │ ├── pytest_plugin_registered
 │ └── pytest_report_header
 ├── pytest_collection
 │ ├── pytest_collectstart
 │ ├── pytest_make_collect_report
 │ │ ├── pytest_collect_file
 │ │ │ └── pytest_pycollect_makemodule
 │ │ └── pytest_pycollect_makeitem
 │ │ └── pytest_generate_tests
 │ │ └── pytest_make_parametrize_id
 │ ├── pytest_collectreport
 │ ├── pytest_itemcollected
 │ ├── pytest_collection_modifyitems
 │ └── pytest_collection_finish
 │ └── pytest_report_collectionfinish
 ├── pytest_runtestloop
 │ └── pytest_runtest_protocol
 │ ├── pytest_runtest_logstart
 │ ├── pytest_runtest_setup
 │ │ └── pytest_fixture_setup
 │ ├── pytest_runtest_makereport
 │ ├── pytest_runtest_logreport
 │ │ └── pytest_report_teststatus
 │ ├── pytest_runtest_call
 │ │ └── pytest_pyfunc_call
 │ ├── pytest_runtest_teardown
 │ │ └── pytest_fixture_post_finalizer
 │ └── pytest_runtest_logfinish
 ├── pytest_sessionfinish
 │ └── pytest_terminal_summary
 └── pytest_unconfigure
```

By using these hooks, We can make our tests more powerful and flexible, and better able to handle the needs of the project. So, hooks are very important and useful feature of pytest which provides the ability to modify and customize the test runner as per the requirements.

1

Written by Parag Kamble

18 Followers