PyTest Fixture

Fixture is a pytest plugin for loading and referencing test data. It provides several utilities for achieving a *fixed state* when testing Python programs.

In pytest you write your tests as functions (or methods.) When writing a lot of tests, you frequently have the same boilerplate over and over as you setup data. Fixtures let you move that out of your test, into a callable which returns what you need.

Sounds simple enough, but pytest adds a bunch of facilities tailored to the kinds of things you run into when writing a big pile of tests:

- Simply put the name of the fixture in your test function's arguments and pytest will find it and pass it in
- Fixtures can be located from various places: local file, a **conftest.py** in the current (or any parent) directory, any imported code that has a **"@pytest.fixture"** decorator, and pytest built-in fixtures
- Fixtures can do a return or a yield, the latter leading to useful teardown-like patterns
- You can speed up your tests by flagging how often a fixture should be computed
- Interesting ways to parameterize fixtures for reuse
    - The purpose of test fixtures is to provide a fixed baseline upon which tests can reliably and repeatedly execute.

Why do you want fixtures?

If your tests need to work on data you typically need to set them up. This is often a process that has to be repeated and independent for each test. This often leads to duplicate code which is "number one in the stink parade"

The `@pytest.fixture` decorator provides an easy yet powerful way to setup and teardown resources. You can then pass these defined fixture objects into your test functions as input arguments.

You want each test to be independent, something that you can enforce by running your tests in random order.

Fixtures are also referred to as dependency injections which you can read more about here. Let's look at some actual code next.

To mark a fixture function:

@pytest.fixture(*scope='function'*, *params=None*, *autouse=False*, *ids=None*)

- 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 functions can directly use fixture names as input arguments in which case the fixture instance returned from the fixture function will be injected.

Parameters:	**scope** – the scope for which this fixture is shared, one of “function” (default), “class”, “module”, “session”. **params** – an optional list of parameters which will cause multiple invocations of the fixture function and all of the tests using it. **autouse** – if True, the fixture function is activated for all tests that can see it. If False (the default) then an explicit reference is needed to activate the fixture. **ids** – list of string ids each corresponding to the params so that they are part of the test id. If no ids are provided, they will be generated automatically from the params.

How to use the Fixture

Fixture can be used in 3 different ways,

1. By passing fixture directly to test functions.
2. By calling a fixture using the use_fixture marker.
3. By using the “autouse”.

Passing the fixture to test functions,

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pyt
============================================== test sess
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, plug
ython\python37-32\python.exe
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2
collected 4 items

test_fixture.py::test_script_one Running before the test
PASSED
test_fixture.py::test_script_two Running before the test
PASSED
test_fixture.py::test_script_three Running before the test
PASSED
test_fixture.py::test_script_four Running before the test
PASSED
```

Note: fixture named 'setup', called before every test.

Calling a fixture using "use_fixture" marker

```
import pytest
pytestmark = pytest.mark.usefixtures("setup")

@pytest.fixture
def setup():
    print("Running before the test")


def test_script_one():
    pass

def test_script_two():
    pass


def test_script_three():
    pass


def test_script_four():
    pass
```

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pyt
============================================== test sess
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, plug
ython\python37-32\python.exe
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2
collected 4 items

test_fixture.py::test_script_one Running before the test
PASSED
test_fixture.py::test_script_two Running before the test
PASSED
test_fixture.py::test_script_three Running before the test
PASSED
test_fixture.py::test_script_four Running before the test
PASSED
```

Using autouse:

```
import pytest

@pytest.fixture(autouse=True)
def setup():
    print("Running before the test")


def test_script_one():
    pass

def test_script_two():
    pass


def test_script_three():
    pass


def test_script_four():
    pass
```

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pytest -vs
============================================= test session starts ======
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- c:\u
ython\python37-32\python.exe
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile: pytest.ini
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2, ordering-0.6, x
collected 4 items

test_fixture.py::test_script_one Running before the test
PASSED
test_fixture.py::test_script_two Running before the test
PASSED
test_fixture.py::test_script_three Running before the test
PASSED
test_fixture.py::test_script_four Running before the test
PASSED

========================================== 4 passed in 0.17 seconds ====
```

Note: fixture “setup” will execute before every test function, to execute after every test, we can use the same fixture to run after the test also.

Fixture finalization / executing teardown code

By using,

1. Add finalizer
2. Yield

Using add finalizer:

```
import pytest

@pytest.fixture(autouse=True)
def setup(request):
    print("\n Running before the test")
    def teardown():
        print("\n Running after the test")
    request.addfinalizer(teardown)

def test_script_one():
    pass

def test_script_two():
    pass
```

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pytest -vs
============================================== test session start
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, pluggy-0.12.0
ython\python37-32\python.exe
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile: pytest.
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2, orderin
collected 4 items

test_fixture.py::test_script_one
 Running before the test
PASSED
 Running after the test

test_fixture.py::test_script_two
 Running before the test
PASSED
 Running after the test
```

Using Yield

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

```
import pytest

@pytest.fixture(autouse=True)
def setup():
    print("\n Running before the test")
    yield
    print("\n Running after the test")

def test_script_one():
    pass

def test_script_two():
    pass
```

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pytest -vs
============================================== test session starts ==
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 --
ython\python37-32\python.exe
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile: pytest.ini
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2, ordering-0
collected 4 items

test_fixture.py::test_script_one
 Running before the test
PASSED
 Running after the test

test_fixture.py::test_script_two
 Running before the test
PASSED
 Running after the test
```

Note: Fixture named “setup”, executes before every test method and after every test functions.

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

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

Scope: sharing a fixture instance across tests in a class, module or session

But if we want a fixture to executes before the class, before the module and before the session. Then we need to use the "scope" attribute and assign "class", "module" and "session" as an argument to fixture.

Within a function request for features, fixture of higher-scopes (such as session) are instantiated first than lower-scoped fixtures (such as function or class). The relative order of fixtures of same scope follows the declared order in the test function and honours dependencies between fixtures.

Consider the code below:

```
import pytest

@pytest.fixture(scope='function', autouse=True)
def function_setup():
    print("\n Running before the test")
    yield
    print("\n Running after the test")

@pytest.fixture(scope='class', autouse=True)
def class_setup():
    print("\n Running before the class")
    yield
    print("\n Running after the class")

@pytest.fixture(scope='module', autouse=True)
def module_setup():
    print("\n Running before the module")
    yield
    print("\n Running after the module")

@pytest.fixture(scope='session', autouse=True)
def session_setup():
    print("\n Running before the session")
    yield
    print("\n Running after the session")

def test_script_one():
    pass

def test_script_two():
    pass


class TestSample:
    def test_script_three(self):
        pass

    def test_script_four(self):
        pass
```

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pytest -vs
============================================= test session starts ==================
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- c:\users\jayapriy
ython\python37-32\python.exe
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile: pytest.ini
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2, ordering-0.6, xdist-1.29.0
collected 4 items

test_fixture.py::test_script_one
 Running before the session

 Running before the module

 Running before the class

 Running before the test
PASSED
 Running after the test

 Running after the class

test_fixture.py::test_script_two
 Running before the class

 Running before the test
PASSED
 Running after the test

 Running after the class

test_fixture.py::TestSample::test_script_three
 Running before the class

 Running before the test
PASSED
 Running after the test

test_fixture.py::TestSample::test_script_four
 Running before the test
PASSED
 Running after the test

 Running after the class

 Running after the module

 Running after the session


========================================== 4 passed in 0.22 seconds ================
```

Re-use fixtures in various test files

The second and last feature I want to highlight. You can add fixtures to a predefined file called `conftest.py`. Fixtures in this file will be automatically discovered upon running pytest, no import needed.

conftest

```python
import pytest

@pytest.fixture(scope='function', autouse=True)
def function_setup():
    print("\n Running before the test")
    yield
    print("\n Running after the test")

@pytest.fixture(scope='class', autouse=True)
def class_setup():
    print("\n Running before the class")
    yield
    print("\n Running after the class")

@pytest.fixture(scope='module', autouse=True)
def module_setup():
    print("\n Running before the module")
    yield
    print("\n Running after the module")

@pytest.fixture(scope='session', autouse=True)
def session_setup():
    print("\n Running before the session")
    yield
    print("\n Running after the session")
```

Note: All the common fixtures have to be developed in the conftest.py file

conftest.py: sharing fixture functions

If during implementing your tests you realize that you want to use a fixture function from multiple test files you can move it to a conftest.py file. You don't need to import the fixture you want to use in a test, it automatically gets discovered by pytest.

Parametrizing fixtures

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

The fixture function gets access to each parameter through the special **request** object:

```
import pytest

@pytest.fixture(scope='function', params=["Chrome","Firefox"], autouse=True)
def function_setup(request):
    print("\n Launching "+request.param +" browser")
    yield
    print("\n Closing "+request.param +" browser")
```

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:

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pytest -vs
============================================= test session starts =============================================
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- c:\users\jayapriyapramod\appdata\local\programs\p
ython\python37-32\python.exe
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile: pytest.ini
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2, ordering-0.6, xdist-1.29.0
collected 4 items

test_fixture.py::test_script_one[Chrome]
 Launching Chrome browser
PASSED
 Closing Chrome browser

test_fixture.py::test_script_one[Firefox]
 Launching Firefox browser
PASSED
 Closing Firefox browser

test_fixture.py::test_script_two[Chrome]
 Launching Chrome browser
PASSED
 Closing Chrome browser

test_fixture.py::test_script_two[Firefox]
 Launching Firefox browser
PASSED
 Closing Firefox browser
```

We see that our two test functions each ran twice, against the different browser's instances.

Sharing the objects from fixture to test functions:

In order to share the objects,

we use,

request -> A request object gives access to the requesting test context

Node -> underlying collection node (depends on current request scope)

Below is the code,

```python
import pytest

@pytest.fixture(scope='function', params=["Chrome","Firefox"], autouse=True)
def function_setup(request):
    request.node.browsername = request.param
    yield
    pass
```

```
D:\Training\PythonSelenium\PyTest_Practice\test_fixtures>pytest -vs
============================================================================ test session starts =====
platform win32 -- Python 3.7.4, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- c:\users\jayapriyapramod\a
cachedir: .pytest_cache
rootdir: D:\Training\PythonSelenium\PyTest_Practice, inifile: pytest.ini
plugins: allure-pytest-2.7.1, dependency-0.4.0, forked-1.0.2, ordering-0.6, xdist-1.29.0
collected 4 items

test_fixture.py::test_script_one[Chrome] Browser name:  Chrome
PASSED
test_fixture.py::test_script_one[Firefox] Browser name:  Firefox
PASSED
test_fixture.py::test_script_two[Chrome] Browser name:  Chrome
PASSED
test_fixture.py::test_script_two[Firefox] Browser name:  Firefox
PASSED

============================================================================ 4 passed in 0.12 seconds ====
```