

**Fixtures**:

**Side Note**:

If want to understand and revise about fixtures quickly… glance through this example.

import pytest

@pytest.fixture

def order():

    return []

@pytest.fixture

def execute\_stage(order, setup\_stage):

    order.append("execute")

class TestOne:

    @pytest.fixture

    def setup\_stage(self, order):

        order.append("setup\_one")

    def test\_order(self, order, execute\_stage):

        assert order == ["setup\_one", "execute"]

class TestTwo:

    @pytest.fixture

    def setup\_stage(self, order):

        order.append("setup\_two")

    def test\_order(self, order, execute\_stage):

        assert order == ["setup\_two", "execute"]

The order fixture serves as a **foundation** for tracking the sequence in which the other fixtures are applied. It’s a simple list that gets modified by other fixtures to record the order of operations. Here’s a breakdown of its role:

* **Initialization**: The order fixture is initialized as an empty list. This is done to provide a clean slate for each test.
* **Recording**: As each fixture runs, it appends its identifier (like “setup\_one” or “execute”) to the order list. This way, you can track the execution order of fixtures.
* **Assertion**: In the test functions, the order list is used to assert that the fixtures ran in the expected sequence.

Think of order as a ledger or journal that keeps a record of the events (fixture executions) as they occur. This is crucial for tests where the order of setup steps matters, and you want to ensure that each step is performed correctly before moving on to the next one.

Just before we explain what fixtures are, let’s take a step backward here and look at the pattern of writing tests.

There are several different kind of test,

🡪 Unit test

🡪 Integration test

🡪 end -to-end test

🡪 functional test

In all these tests there is a common pattern of writing them



At the ***Arrange*** stage 🡪 We prepare our test state that we need to perform a specific action. (*Does the test require any objects / special settings / database / logging into a web app / connect to some api endpoint*). So the *arrange stage here is trying to prepare our test before performing an action*.

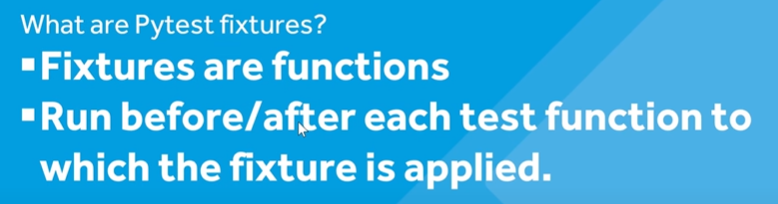
At the ***Act*** stage 🡪 This is the action stage, which could be *calling a function or a method, calling a rest api or interacting with a web page*. *Typically the act step should elicit some sort of response*.

It could be a http response(*some data that is returned from an api for example*).

At the ***Assert*** stage 🡪 Here we need to assert expected outcomes. In this *we take the response from the action and we can match it against expected outcome*. For example *if were meant to return some data in the act phase then the assert is going to test to see if that data was indeed returned*.

Of course if we do assert the action or the act and we have returned the expected outcome then we return true and therefore the test has passed.

Let’s see how *fixtures* fit into this paradigm.



So we can see that this *fixture will be applied in the arrange phase of the test because in the arrange phase where we will be preparing the data*(*for the action to be performed in act stage*), in this case fixtures which are just functions are going to run before each test.

Hence naturally fixtures in this paradigm (*Arrange 🡪 Act 🡪 Assert*) run before the action or the act.

So we can use fixtures for example to *collect data from database or to prepare a database connection, so that we can perform a function within the act to return some sort of result and then we can assert the result to check to see if we have returned the expected outcome*.

To summarize,



Let’s see this in practice,

import pytest

@pytest.fixture

def fixture\_1():

    print("run fixture-1")

    return 1

def test\_example1(fixture\_1):

    num = fixture\_1

    assert num == 1

🡪 To start writing a fixture use the @pytest.fixture decorator before a function. Below is a simple fixture.

@pytest.fixture

def fixture\_1():

    print("run fixture-1")

    return 1

🡪 If we want to use this fixture inside a test, we just need to pass this fixture as a parameter to our test in order to start utilizing it.

def test\_example1(fixture\_1):

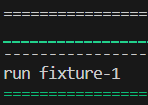
    num = fixture\_1

    assert num == 1

Here we bring our fixture as a variable and then assert it.

In this example *we are simply utilizing this fixture to prepare some data* (*in the form of a variable*) and *then asserting it*.

If we run pytest -rP, we see our print statement inside our fixture.



This clearly indicates that this fixture is being run by pytest. Let’s also add print in our test\_example1.

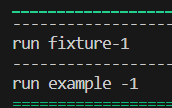
def test\_example1(fixture\_1):

    print("run example -1")

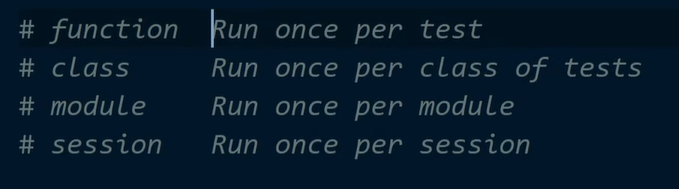
    num = fixture\_1

    assert num == 1

Both fixture and test is running,



*By default a fixture is supposed to run once per test, but it might happen that we need to run a fixture over multiple tests*.



We can do this by defining a fixture in a different way for example, defining a fixture as a module or a session.

To see this in action, Let’s use the same fixture in two different tests.

@pytest.fixture

def fixture\_1():

    print("run fixture-1")

    return 1

def test\_example1(fixture\_1):

    print("run example -1")

    num = fixture\_1

    assert num == 1

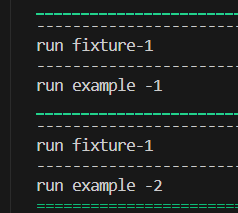
def test\_example2(fixture\_1):

    print("run example -2")

    num = fixture\_1

    assert num == 1

if we run pytest -rP



We can see that every time a test run, our fixture is running as well.

Now let’s change the scope of fixture to ‘*session’* and run the tests again.

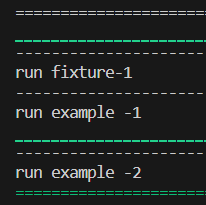
@pytest.fixture(scope="session")

def fixture\_1():

    print("run fixture-1")

    return 1

We can see our fixture runs only once.



It was mentioned before that fixtures can be run at the start and the end of a test as well, Let’s see this in action,

@pytest.fixture(scope="session")

def yield\_fixture():

    print("start test phase")

    yield 6

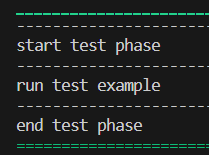
    print("end test phase")

def test\_example1(yield\_fixture):

    print("run test example")

    assert yield\_fixture == 6

and we get this output,



We can see here that we have start of a test, running it and then end of test phase.

This is about sessions where we have to setup the test and then perform tear down at the end of test.

*What are some practical examples of fixtures in Django*?

Its very likely that when we test our django application at some point we need to bring in our database to our tests.

For this demo, we will just utilize the User table that we get from Django by default.

import pytest

from django.contrib.auth.models import User

Now in order for us to access a database with pytest, there are a few different ways. One of them is by utilizing *django\_db* decorator.

@pytest.mark.django\_db

def test\_user\_create():

Its used whenever we want to utilize our data in the test function.

Now we need to test whether we can create a new user.

Note: We don’t have an actual database for our application as of this point but when we use *pytest.mark.django\_db* to test our app, a test database is created and also same database tables won’t persist for different test cases as an example,

@pytest.mark.django\_db

def test\_user\_create():

    User.objects.create\_user("test", "test@test.com", "test")

    print(User.objects.count()) 🡪 1

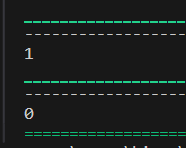
    assert User.objects.count() == 1

@pytest.mark.django\_db

def test\_user\_create1():

    print(User.objects.count()) 🡪 0

    assert User.objects.count() == 0



So we can see that User count in both tests is different, as they both use separate database tables.

***Databases with fixtures***:

We have seen interaction between database and tests, now we will see how fixtures interact with database.

In fixture instead of using *django\_db* decorator we will use *db* parameter in the fixture function like this.

@pytest.fixture

def user\_1(db): 🡪 *db as parameter*

Then we can simply run a query to our database.

@pytest.fixture

def user\_1(db):

    return User.objects.create\_user("test\_user")

So this fixture here returns a User object called *test\_user*, lets use this inside a test.

@pytest.mark.django\_db

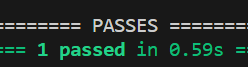
def test\_set\_check\_password(user\_1):

    user\_1.set\_password("new-password")

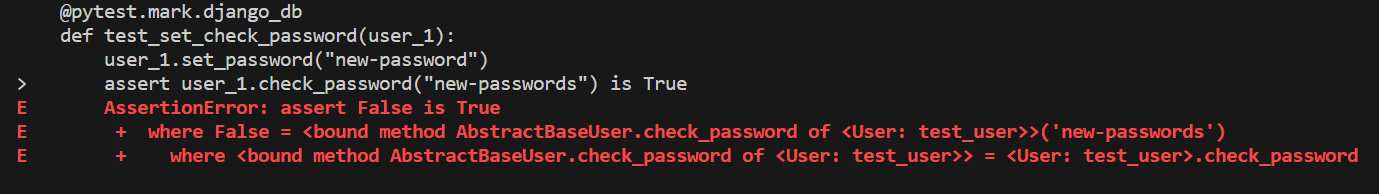
    assert user\_1.check\_password("new-password") is True

Since we will utilize database in this test, therefore we needed to mark it with *django\_db* decorator, then for this *test\_user* we *set\_password* and check if its same with *check\_password*.

In case password is same in both places,



But if different then,



In our tests we may want to extend from our fixture or add more data to perform tests, so *we might need our fixture to perform a function and just return some data from database*.

Another example,

@pytest.fixture

def user\_1(db):

    user = User.objects.create\_user("test\_user")

    return user

def test\_check\_username(user\_1):

    assert user\_1.username == "test\_user"

Here we simply created and returned a user object from user\_1 fixture and then used it in our test to check its username attribute.

With our fixtures and test examples, we can see our AAA paradigm,



We are using the fixture to arrange our data (*creating user*), then acting on our data (*setting this user password*) and finally assert (*checking the password*).

Note: Let’s consider this code as an example, where we have a single fixture being used by two tests.

@pytest.fixture

def user\_1(db):

    user = User.objects.create\_user("test\_user")

    print("create user")

    return user

def test\_check\_username1(user\_1):

    print("Check\_user1")

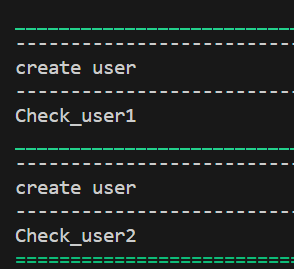
    assert user\_1.username == "test\_user"

def test\_check\_username2(user\_1):

    print("Check\_user2")

    assert user\_1.username == "test\_user"

If we run this, we see the fixture being run for every test.



If we change the fixture scope to ‘*session*’(running only once).

@pytest.fixture(scope="session")

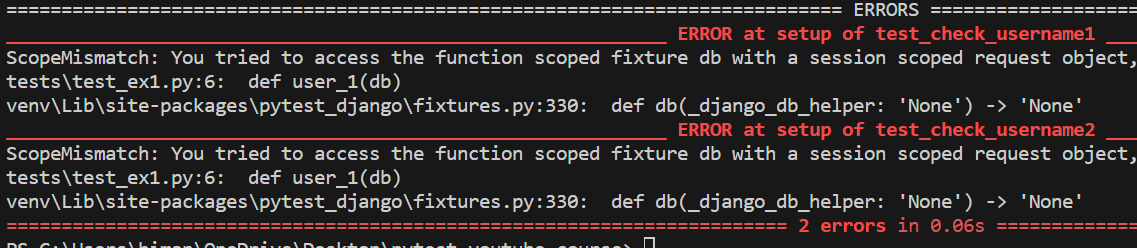
def user\_1(db):

    user = User.objects.create\_user("test\_user")

    print("create user")

    return user

We get errors in both tests.



ScopeMismatch: You tried to access the function scoped fixture db with a session scoped request object, involved factories:

tests\test\_ex1.py:6: def user\_1(db)

venv\Lib\site-packages\pytest\_django\fixtures.py:330: def db(\_django\_db\_helper: 'None') -> 'None'.

So it means that *we can’t use a fixture once per session and run it from two different tests*.

When we run pytest, it looks for a file called *conftest.py* which can be used to utilize our fixtures. So let’s move our fixture code into this file.

# conftest.py

import pytest

from django.contrib.auth.models import User

@pytest.fixture 🡪 *Not a session scope*

def user\_1(db):

    user = User.objects.create\_user("test\_user")

    print("create user")

    return user

and our test file has only tests now (*no fixtures created here*).

# our test file

def test\_check\_username1(user\_1):

    print("Check\_user1")

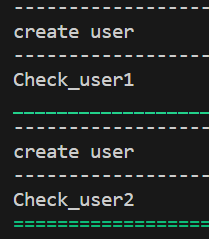
    assert user\_1.username == "test\_user"

def test\_check\_username2(user\_1):

    print("Check\_user2")

    assert user\_1.username == "test\_user"

Our tests pass now,



***Factories***:

There can be different kinds of users accessing our database (*admin, authenticated, anonymous*) so its not a smart strategy to build 3 different fixtures for these types. Instead we use *factories* as a fixture.

This *factory as a fixture generates different type of users for different type of setup*.

In our conftest.py file, we will create a new fixture which we are going to use as a factory.

@pytest.fixture

def new\_user\_factory(db):

    def create\_app\_user( 🡪 *a nested function*

        username: str,

        password: str = None,

        first\_name: str = "firstName",

        last\_name: str = "lastName",

        email: str = "test@test.com",

        is\_staff: str = False,

        is\_superuser: str = False,

        is\_active: str = True,

    ):

        user = User.objects.create\_user()

        return user

    return create\_app\_user

These are pretty much all the fields you can define inside the user table.

Now we can pass value to these arguments in our *User.objects.create\_user* method in order to create a new user *(if we don’t provide any value to these arguments they will use default values*).

@pytest.fixture

def new\_user\_factory(db):

    def create\_app\_user(

        username: str,

        password: str = None,

        first\_name: str = "firstName",

        last\_name: str = "lastName",

        email: str = "test@test.com",

        is\_staff: str = False,

        is\_superuser: str = False,

        is\_active: str = True,

    ):

        user = User.objects.create\_user(

            username=username,

            password=password,

            first\_name=first\_name,

            last\_name=last\_name,

            email=email,

            is\_staff=is\_staff,

            is\_superuser=is\_superuser,

            is\_active=is\_active,

        )

        return user

    return create\_app\_user

We are trying to build flexibility here this way.

Now we will build another fixture that is going to utilize this factory.

@pytest.fixture

def new\_user(db, new\_user\_factory):

    return new\_user\_factory("Test\_user", "password", "MyName")

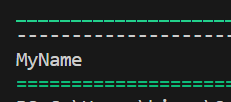
Inside this fixture we pass into the factory the username, password and first\_name as arguments and get a user object.

To use this user object inside our test,

def test\_new\_user(new\_user):

    print(new\_user.first\_name)

    assert new\_user.first\_name == "MyName"



This is the username we have generated from factory fixture that utilize the factory.

If we want to create a new fixture for a staff member,

@pytest.fixture

def new\_user2(db, new\_user\_factory):

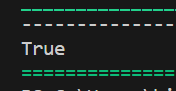
    return new\_user\_factory("Test\_user", "password", "MyName", is\_staff=True)

And use it in our test,

def test\_new\_user(new\_user2):

    print(new\_user2.is\_staff)

    assert new\_user2.is\_staff



Q> **what do most production grade projects that use pytest use their scope as functional/module/class/session and why?**

In production-grade projects, the choice of scope for pytest fixtures is often determined by the specific needs of the tests and the resources they use. Here’s a general guideline:

* **Function Scope**: This is the default and most commonly used scope. It’s ideal for resources that need to be fresh for every test, like database connections or temporary files. It helps prevent side effects between tests.
* **Class Scope**: Used when multiple tests in the same class can share the setup, like when they are testing different aspects of the same object.
* **Module Scope**: Chosen for resources that are expensive to set up and tear down and can be safely shared across all tests in a module, like a web server or a database schema.
* **Session Scope**: Reserved for resources that are very expensive to set up and can be shared across the entire test session, like a database filled with test data.

The choice of scope is often a balance between test isolation (ensuring one test’s setup doesn’t affect another) and performance (minimizing setup and teardown time). Function scope provides the highest isolation but can be slow if the setup is expensive. Session scope has the best performance but can lead to issues with test interdependence if not managed carefully.

Q> Explain fixture instantiation order using laymen terms?  
Let’s use an everyday example to understand the fixture instantiation order in pytest:

Imagine you’re hosting a dinner party (the test function). To prepare for this, you need to set up various things like the dining table, the food, and the music system. These setups are like fixtures in pytest—they need to be ready before the party starts.

**Scope**: This is like the scale of your preparations. Some preparations are just for this party (function scope), some are for all parties this week (module scope), and some are for all parties this season (session scope). In pytest, the scope defines how often a fixture is called: once per function, module, or session.

**Dependencies**: This is the order in which you prepare. You can’t put food on the table if the table isn’t set up yet. In pytest, some fixtures depend on others, and pytest ensures that dependent fixtures are executed first.

**Autouse**: These are the things you always do for a party, like cleaning the house. You don’t decide on this each time; it’s automatic. In pytest, autouse fixtures are automatically used by tests without needing to be explicitly requested.

So, when pytest runs a test, it’s like planning the party. It looks at what needs to be done (scope), the order things should be done in (dependencies), and what’s always done (autouse). Just like you wouldn’t rely on the order you wrote your to-do list to plan the party, pytest doesn’t rely on the order fixtures are defined or requested. It relies on scope, dependencies, and autouse to determine the correct order to set everything up for the test (party).

This ensures that everything is prepared correctly and in the right order, so the party (test) can go off without a hitch!

Q> The order of fixture scopes from broadest to least broad?

Certainly! In pytest, the scopes of fixtures are defined from the broadest to the least broad as follows:

1. **Session**: The fixture is executed once per test session.
2. **Module**: The fixture is executed once per test module (a test file).
3. **Class**: If you’re using classes to group tests, the fixture is executed once per class of tests.
4. **Function**: The fixture is executed once per test function.

So, the order from broadest to least broad is: **Session**, **Module**, **Class**, and **Function**. These scopes help manage the setup and teardown processes efficiently by determining how often fixtures should be invoked.