<https://paiml.com/docs/home/books/testing-in-python/chapter07-pytest-fixtures/>

<https://www.programcreek.com/python/example/81420/pytest.fixture>

<https://semaphoreci.com/community/tutorials/testing-python-applications-with-pytest>

<https://www.lambdatest.com/blog/end-to-end-tutorial-for-pytest-fixtures-with-examples/>

<https://www.codestudyblog.com/cnb11/1124181049.html>

<https://www.seanh.cc/2017/02/12/advanced-fixtures/>

<https://smarie.github.io/python-pytest-harvest/advanced_usage/>

<https://www.automationlaboratories.com/python/fixtures-in-pytest-automation-laboratories/>

[**https://metareal.blog/en/post/2020/04/25/useful-pytest-fixtures-to-help-you-improve-legacy-code/**](https://metareal.blog/en/post/2020/04/25/useful-pytest-fixtures-to-help-you-improve-legacy-code/)

[**https://martinheinz.dev/posts/**](https://martinheinz.dev/posts/)

**https://docs.python.org/3/library/functools.html#functools.lru\_cache**

[**https://github.com/pytest-dev/pytest/issues/6374**](https://github.com/pytest-dev/pytest/issues/6374)

[**https://miguendes.me/3-ways-to-test-api-client-applications-in-python**](https://miguendes.me/3-ways-to-test-api-client-applications-in-python)

[**https://towardsdatascience.com/testing-best-practices-for-machine-learning-libraries-41b7d0362c95**](https://towardsdatascience.com/testing-best-practices-for-machine-learning-libraries-41b7d0362c95)

[**https://www.programmersought.com/article/68147317398/**](https://www.programmersought.com/article/68147317398/)

1. **Pytest**

Pytest is a python based testing **framework**, which is used to write and execute test codes. In the present days of REST services, pytest is mainly used for API testing even though we can use pytest to write simple to complex tests, i.e., we can write codes to test API, database, UI, etc.

## Advantages of Pytest

* Pytest can **run multiple tests in parallel**, which reduces the execution time of the test suite.
* Pytest has its own way to **detect the test file and test functions** automatically, if not mentioned explicitly.
* Pytest allows us to **skip** a subset of the tests during execution.
* Pytest allows us to run a subset of the entire test suite.
* Pytest is free and open source.

Running pytest without mentioning a filename will run all files of format **test\_\*.py** or **\*\_test.py** in the current directory and subdirectories. Pytest automatically identifies those files as test files. We **can** make pytest run other filenames by explicitly mentioning them.

* Pytest requires the **test function names to start with test**. Function names which are not of format **test\*** are not considered as test functions by pytest. We **cannot** explicitly make pytest consider any function not starting with **test** as a test function.

pytest command will execute all the files of format **test\_\*** or **\*\_test** **in the current directory and subdirectories.**

To execute the tests from a specific file, use the following syntax −

pytest <filename> -v

1. **Fixture**

pytest fixtures are functions attached to the tests which **run before** the test function is executed. Fixtures are a set of resources that have to be set up before and cleaned up once the Selenium test automation execution is completed

A fixture function defined inside a test file has a scope within the test file only. We cannot use that fixture in another test file. To make a fixture available to multiple test files, we have to define the fixture function in a file called conftest.py. **conftest.py** is explained in the next chapter.

@pytest.fixture()  
def before():  
 print('\nbefore each test')  
  
  
def test\_1(before):  
 print('test\_1()')  
  
  
def test\_2(before):  
 print('test\_2()')

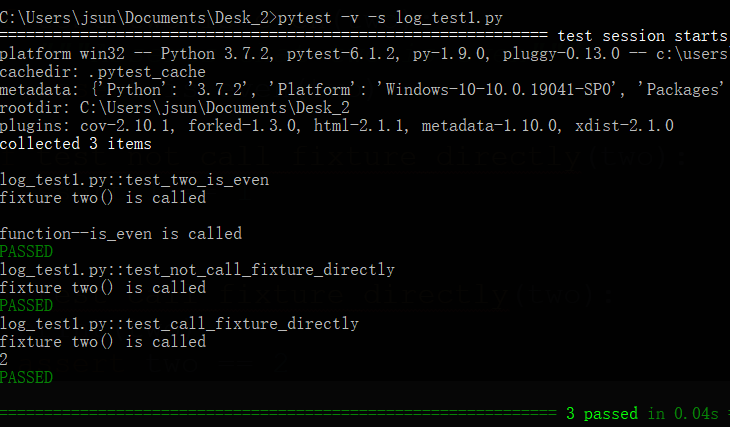
Output:

before each test  
test\_1()  
.  
before each test  
test\_2()

With the default parameters for ‘pytest.fixture()’, **the fixture is going to be called for every test that names the fixture in it’s parameter list**.

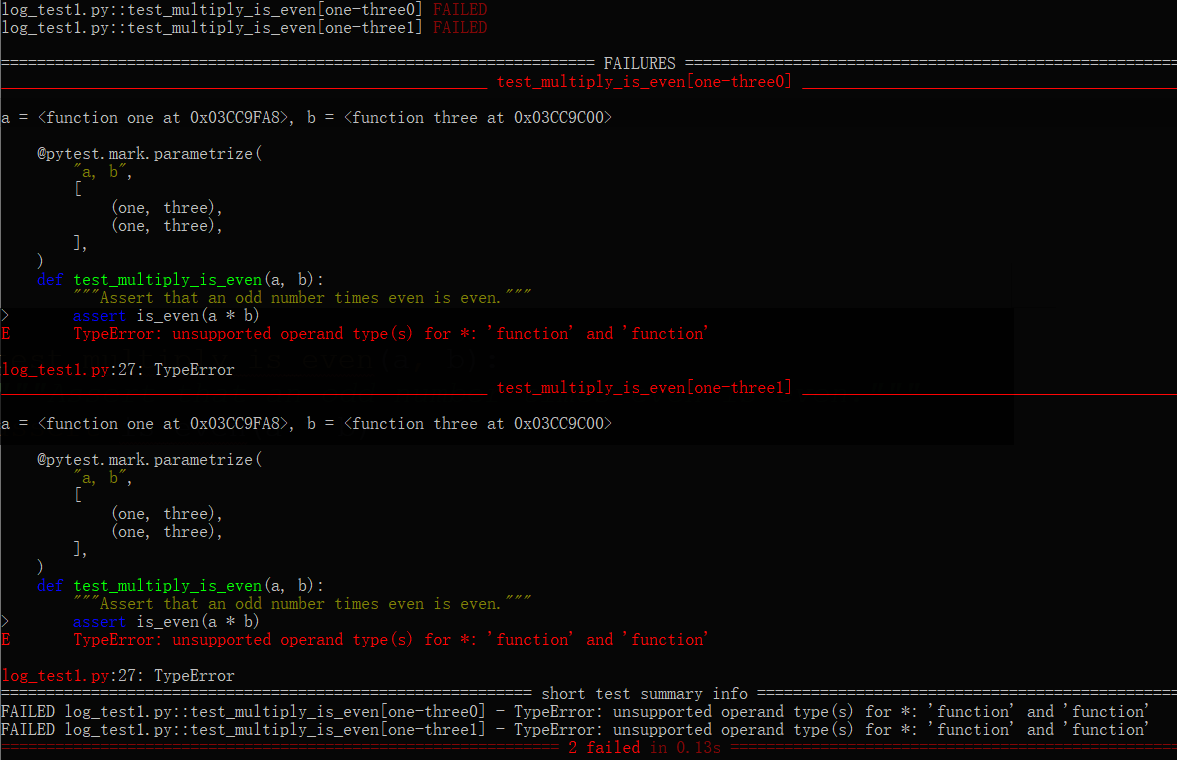
1. **Fixture example**

import pytest  
  
def is\_even(n: int) -> bool:  
 *"""Returns True if n is even."""* print("\nfunction--is\_even is called")  
 return n % 2 == 0  
  
  
@pytest.fixture()  
def two():  
 print("\nfixture two() is called")  
 return 2  
  
@pytest.fixture()  
def four():  
 return 4  
  
# def test\_four\_is\_even(four):  
# """Asserts that four is even"""  
# assert is\_even(four)  
#  
def test\_two\_is\_even(two):  
 *"""Asserts that two is even"""* assert is\_even(two)  
  
def test\_not\_call\_fixture\_directly(two):  
 assert 1 == 1  
  
  
def test\_call\_fixture\_directly(two):  
 print(two)  
 assert two == 2

****

**passing a fixture as argument in a parameterized test doesn't work**

import pytest  
  
def is\_even(n: int) -> bool:  
 *"""Returns True if n is even."""* return n % 2 == 0  
  
  
@pytest.fixture()  
def one():  
 return 1  
  
@pytest.fixture()  
def three():  
 return 3  
  
  
#passing a fixture as argument in a parameterized test doesn't work  
@pytest.mark.parametrize(  
 "a, b",  
 [  
 (one, four),  
 (two, three),  
 ],  
)  
def test\_multiply\_is\_even(a, b):  
 *"""Assert that an odd number times even is even."""* assert is\_even(a \* b)

****

1. **Ex 2**

To make that possible, we have two alternatives. The first one is using request.getfixturevalue, which is available on pytest. This function dynamically runs a named fixture function.

import pytest  
  
@pytest.fixture(params=[1, 2])  
def one(request):  
 return request.param  
  
@pytest.mark.parametrize('arg1,arg2', [  
 ('val1', pytest.lazy\_fixture('one')),  
])  
def test\_func(arg1, arg2):  
 assert arg2 in [1, 2]

import pytest  
import pytest-lazy-fixture  
  
def is\_even(n: int) -> bool:  
 *"""Returns True if n is even."""* return n % 2 == 0  
  
  
@pytest.fixture()  
def one():  
 return 1  
  
@pytest.fixture()  
def three():  
 return 3  
  
# @pytest.mark.parametrize(  
# "a, b",  
# [  
# (one, three),  
# (one, three),  
# ]  
# )  
# # def test\_multiply\_is\_even(a, b):  
# # """Assert that an odd number times even is even."""  
# # assert is\_even(a \* b)  
# def test\_multiply\_is\_even\_request(a, b, request):  
# """Assert that an odd number times even is even."""  
# a = request.getfixturevalue(a)  
# b = request.getfixturevalue(b)  
# assert is\_even(a \* b)  
  
  
# pip install pytest-lazy-fixture  
@pytest.mark.parametrize(  
 "a, b",  
 [  
 (pytest.lazy\_fixture(("one", "four"))),  
 # same as (pytest.lazy\_fixture(("two", "three")))  
 (pytest.lazy\_fixture("two"), pytest.lazy\_fixture("three")),  
 ],  
)  
def test\_multiply(a, b):  
 *"""Assert that an odd number times even is even."""* assert is\_even(a \* b)

1. **Three ways to use a fixture**
2. **name it** from the test.

@pytest.fixture()  
def **before**():  
 print('\nbefore each test')  
  
  
def test\_1(**before**):  
 print('test\_1()')  
  
  
def test\_2(**before**):  
 print('test\_2()')

1. **usefixtures decorator**  
   You can mark a test or a test class with ‘pytest.mark.usefixtures()’ and include a list of fixtures to be used with the test or class of tests.  
   This is especially convenient when dealing with test classes.  
   It also is useful when converting unittest classes to use pytest fixtures.  
   I’ll give an example shortly.

@pytest.fixture()  
def before():  
 print('\nbefore each test')  
  
  
@pytest.**mark.usefixtures("before")**   
def test\_1():  
 print('test\_1()')  
  
@pytest.mark.usefixtures("before")   
def test\_2():  
 print('test\_2()')

class Test:  
 @pytest.mark.usefixtures("before")  
 def test\_1(self):  
 print('test\_1()')  
  
 @pytest.mark.usefixtures("before")  
 def test\_2(self):  
 print('test\_2()')

@pytest.fixture()  
def before():  
 print('\nbefore each test')  
  
  
@pytest.mark.usefixtures("before")  
class Test:  
 def test\_1(self):  
 print('test\_1()')  
  
 def test\_2(self):  
 print('test\_2()')

1. **autouse**

@pytest.fixture(**scope**='function', **params**=None, **autouse**=False)  
def before(**request**):  
 print('\nbefore()')  
 return None

1. **Return value**
2. Returning some data from a fixture

**@pytest.fixture()  
def some\_data():  
 data = {'foo':1, 'bar':2, 'baz':3}  
 return data**  
  
def test\_foo(**some\_data**):  
 assert **some\_data['foo']** == 1

1. Returning a database object

@pytest.fixture()  
def cheese\_db(request):  
 print('\n[setup] cheese\_db, connect to db')  
 # code to connect to your db   
 a\_dictionary\_for\_now = {'Brie': 'No.', 'Camenbert': 'Ah! We have Camenbert, yessir.'}  
 def fin():  
 print('\n[teardown] cheese\_db finalizer, disconnect from db')  
 request.addfinalizer(fin)  
 return a\_dictionary\_for\_now  
  
def test\_cheese\_database(cheese\_db):  
 print('in test\_cheese\_database()')  
 for variety in cheese\_db.keys():  
 print('%s : %s' % (variety, cheese\_db[variety]))  
  
def test\_brie(cheese\_db):  
 print('in test\_brie()')  
 assert cheese\_db['Brie'] == 'No.'   
  
def test\_camenbert(cheese\_db):  
 print('in test\_camenbert()')  
 assert cheese\_db['Camenbert'] != 'No.'

The ‘fin’ function is acting as the ‘teardown’ for the fixture.  
There’s nothing special about the name.

**The finalizer is called after all of the tests that use the fixture????**.  
If you’ve used [parameterized fixtures](https://pythontesting.net/framework/pytest/pytest-fixtures-nuts-bolts/#params), the finalizer is called between instances of the parameterized fixture changes.

Since the default scope is "function", the [cheese db example](https://pythontesting.net/framework/pytest/pytest-fixtures-nuts-bolts/#cheese_example) will open and close the db for every test.

**[setup] cheese\_db, connect to db**  
in test\_cheese\_database()  
Camenbert : Ah! We have Camenbert, yessir.  
Brie : No.  
.  
**[teardown]** **cheese\_db finalizer, disconnect from db**  
  
[setup] cheese\_db, connect to db  
in test\_brie()  
.  
**[teardown]** cheese\_db finalizer, disconnect from db  
  
[setup] cheese\_db, connect to db  
in test\_camenbert()  
.  
**[teardown]** cheese\_db finalizer, disconnect from db

Change the scope to “module”, the [teardown] will only run once for all the tests.

import pytest  
  
  
@pytest.fixture(scope="module")  
def cheese\_db(request):  
 print('\n[setup] cheese\_db, connect to db')  
 # code to connect to your db   
 a\_dictionary\_for\_now = {'Brie': 'No.', 'Camenbert': 'Ah! We have Camenbert, yessir.'}  
 def fin():  
 print('\n[teardown] cheese\_db finalizer, disconnect from db')  
 request.addfinalizer(fin)  
 return a\_dictionary\_for\_now  
  
def test\_cheese\_database(cheese\_db):  
 print('in test\_cheese\_database()')  
 for variety in cheese\_db.keys():  
 print('%s : %s' % (variety, cheese\_db[variety]))  
  
def test\_brie(cheese\_db):  
 print('in test\_brie()')  
 assert cheese\_db['Brie'] == 'No.'   
  
def test\_camenbert(cheese\_db):  
 print('in test\_camenbert()')  
 assert cheese\_db['Camenbert'] != 'No.'

C:\Users\jsun\Documents\Desk\_2>python -m pytest log\_test.py -v -s

collected 3 items  
  
log\_test.py::test\_cheese\_database  
**[setup]** **cheese\_db, connect to db**  
in test\_cheese\_database()  
Brie : No.  
Camenbert : Ah! We have Camenbert, yessir.  
**PASSED**  
log\_test.py::test\_brie in test\_brie()  
**PASSED**  
log\_test.py::test\_camenbert in test\_camenbert()  
**PASSED**  
**[teardown]** **cheese\_db finalizer, disconnect from db**

### **Scope**

Scope controls how often a fixture gets called. The default is "function".  
Here are the options for scope:

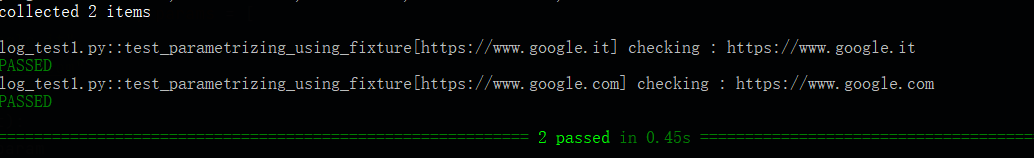
|  |  |
| --- | --- |
| **function** | Run once per test |
| **class** | Run once per class of tests |
| **module** | Run once per module |
| **session** | Run once per session |

1. **Parameterizing Tests**

import pytest  
  
@pytest.mark.parametrize("num,output",[(1,11),(2,22),(3,35),(4,44)])  
def test\_multiplication\_11(num, output):  
 assert 11\*num == output

import pytest  
import requests  
  
@pytest.mark.parametrize('endpoint', [  
 "https://www.google.it",  
 "https://www.google.com"  
])  
def test\_parametrized(endpoint):  
 *"""  
 this will create two tests, one for every value of the endpoint parameter  
 """* print("checking : " + endpoint)  
 assert requests.get(endpoint).status\_code == 200  
  
@pytest.fixture(scope="session")  
def get\_endpoints():  
 return [  
 "https://www.google.it",  
 "https://www.google.com"  
 ]  
  
def test\_using\_fixture(get\_endpoints):  
 *"""this will assert the result of the fixture"""* assert get\_endpoints == [  
 "https://www.google.it",  
 "https://www.google.com"  
 ]  
  
#ERROR: parametrize can't pass fixture  
# @pytest.mark.parametrize('endpoint', get\_endpoints)  
# def test\_parametrizing\_using\_fixture(endpoint):  
# """  
# this should load data from get\_endpoints fixture and pass every value of  
# get\_endpoints as endpoint value  
# """  
# print("checking : " + endpoint)  
# assert requests.get(endpoint).status\_code == 200

import pytest  
import requests  
  
  
  
@pytest.fixture(scope = "session", params = [  
 "https://www.google.it",  
 "https://www.google.com"  
])  
def endpoint(request):  
 return request.param  
  
  
def test\_parametrizing\_using\_fixture(endpoint):  
 print("checking : " + endpoint)  
 assert requests.get(endpoint).status\_code == 200

****

1. **Grouping the test**

Pytest allows us to use markers on test functions. Markers are used to set various features/attributes to test functions. Pytest provides many inbuilt markers such as xfail, skip and parametrize. Apart from that, users can create their own marker names. Markers are applied on the tests using the syntax given below −

@pytest.mark.<markername>

**pytest -m <markername> -v**

**pytest -m great -v**

import pytest  
@pytest.mark.great  
def test\_greater():  
 num = 100  
 assert num > 100  
  
@pytest.mark.great  
def test\_greater\_equal():  
 num = 100  
 assert num >= 100  
  
@pytest.mark.others  
def test\_less():  
 num = 100  
 assert num < 200

# **Stop Test Suite after N Test Failures**

1. **Stop the test suite after N test failures.**

In a real scenario, once a new version of the code is ready to deploy, it is first deployed into **pre-prod/staging environment**. Then a test suite runs on it.

The code is qualified for deploying to production only if the test suite passes. If there is test failure, whether it is one or many, the code is not production ready.

Therefore, what if we want to stop the execution of test suite soon after n number of test fails. This can be done in **pytest** using **maxfail**.

The syntax to stop the execution of test suite soon **after n number of test fails** is as follows –

pytest **--maxfail=<num>**

import pytest  
import math  
  
def test\_sqrt\_failure():  
 num = 25  
 assert math.sqrt(num) == 6  
  
def test\_square\_failure():  
 num = 7  
 assert 7\*7 == 40  
  
def test\_equality\_failure():  
 assert 10 == 11

C:\Users\jsun\Documents\Desk\_2>python -m pytest log\_test1.py -v -s **--maxfail=2**

========================================================== test session starts ===========================================================

platform win32 -- Python 3.7.2, pytest-6.1.2, py-1.9.0, pluggy-0.13.0 -- C:\Users\jsun\AppData\Local\Programs\Python\Python37-32\python.exe

cachedir: .pytest\_cache

metadata: {'Python': '3.7.2', 'Platform': 'Windows-10-10.0.19041-SP0', 'Packages': {'pytest': '6.1.2', 'py': '1.9.0', 'pluggy': '0.13.0'}, 'Plugins': {'cov': '2.10.1', 'forked': '1.3.0', 'html': '2.1.1', 'metadata': '1.10.0', 'xdist': '2.1.0'}}

rootdir: C:\Users\jsun\Documents\Desk\_2

plugins: cov-2.10.1, forked-1.3.0, html-2.1.1, metadata-1.10.0, xdist-2.1.0

collected 3 items

log\_test1.py::test\_sqrt\_failure FAILED

log\_test1.py::test\_square\_failure FAILED

================================================================ FAILURES ================================================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_sqrt\_failure \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_sqrt\_failure():

num = 25

> assert math.sqrt(num) == 6

E assert 5.0 == 6

E +5.0

E -6

log\_test1.py:6: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_square\_failure \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_square\_failure():

num = 7

> assert 7\*7 == 40

E assert 49 == 40

E +49

E -40

log\_test1.py:10: AssertionError

======================================================== short test summary info =========================================================

FAILED log\_test1.py::test\_sqrt\_failure - assert 5.0 == 6

FAILED log\_test1.py::test\_square\_failure - assert 49 == 40

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! **stopping after 2 failures** !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

=========================================================== 2 failed in 0.13s ============================================================

1. **Xfail/Skip Tests**

Now, consider the below situations −

* A test is not relevant for some time due to some reasons.
* **A new feature is being implemented** and we already added a test for that feature.

In these situations, we have the option to **xfail** the test or skip the tests.

Pytest will execute the xfailed test, **but it will not be considered as part failed or passed tests.** Details of these tests will not be printed even if the test fails (remember pytest usually prints the failed test details). We can xfail tests using the following marker –

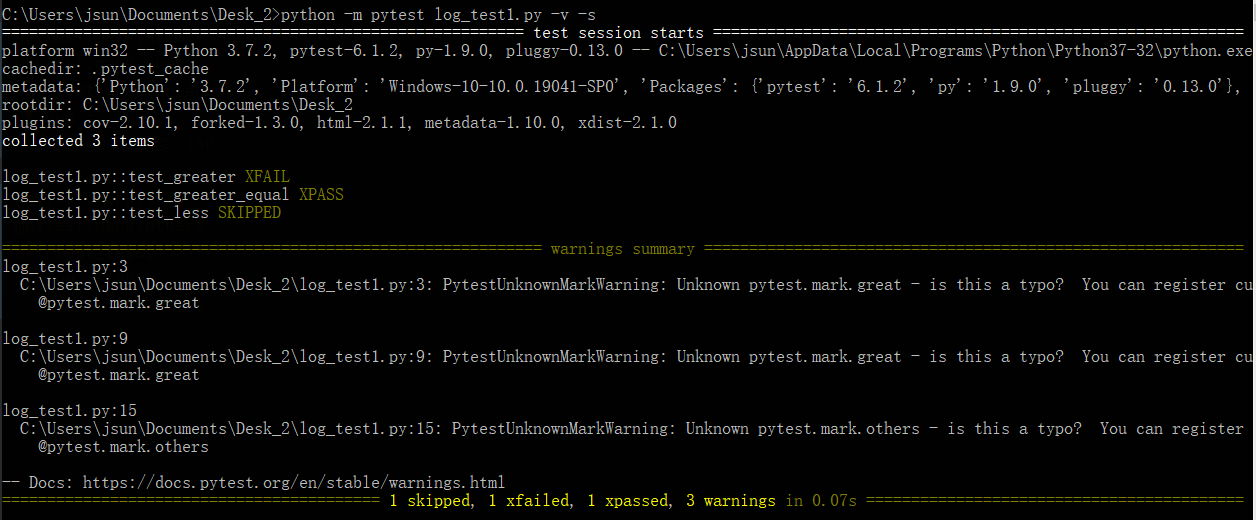
**@pytest.mark.xfail**

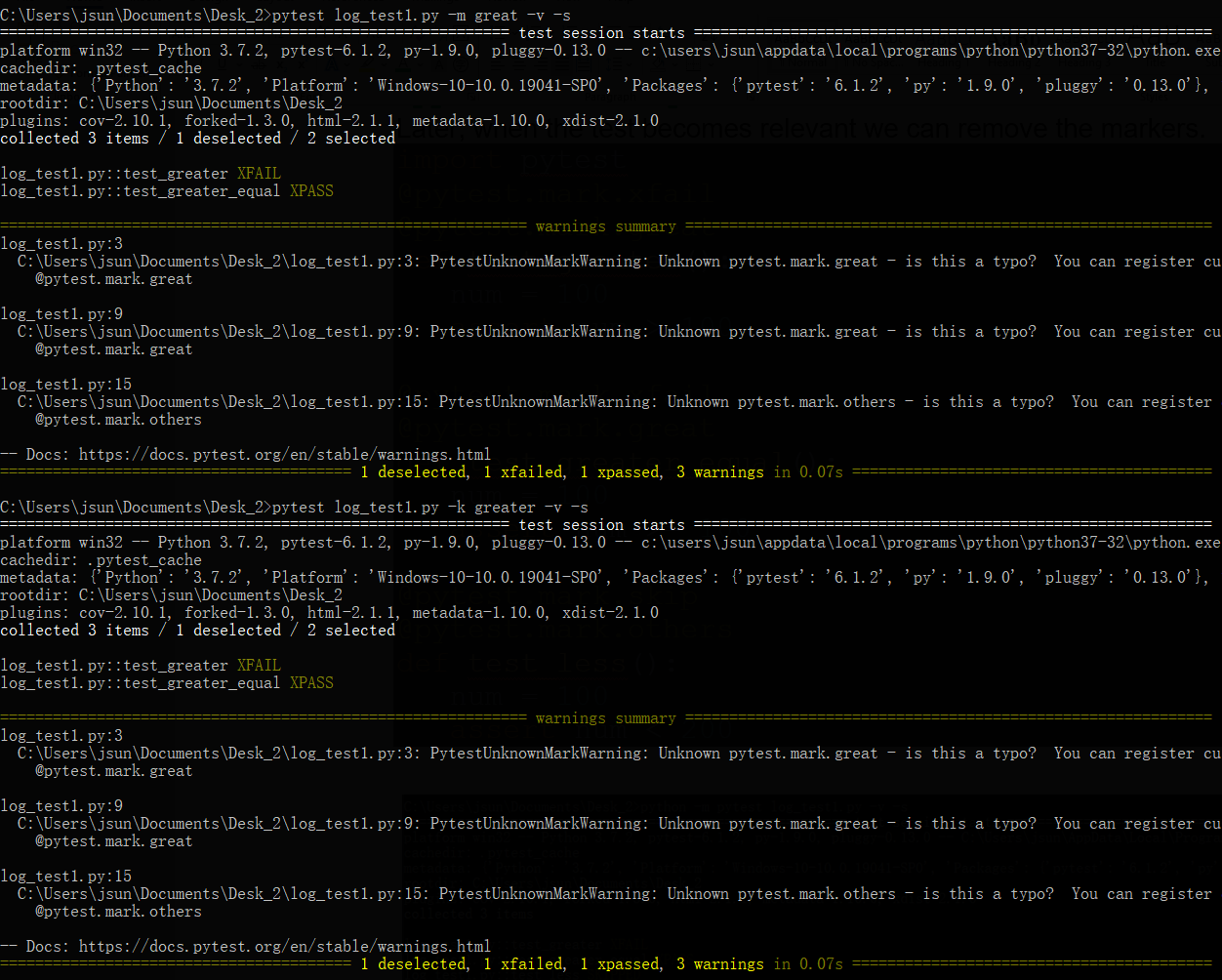
Skipping a test means that the test will not be executed. We can skip tests using the following marker −

**@pytest.mark.skip**

Later, when the test becomes relevant we can remove the markers.

import pytest  
@pytest.mark.xfail  
@pytest.mark.great  
def test\_greater():  
 num = 100  
 assert num > 100  
  
@pytest.mark.xfail  
@pytest.mark.great  
def test\_greater\_equal():  
 num = 100  
 assert num >= 100  
  
@pytest.mark.skip  
@pytest.mark.others  
def test\_less():  
 num = 100  
 assert num < 200



****

1. **Run test in Parallel**

By default, pytest runs tests in sequential order. In a real scenario, a test suite will have a number of test files and each file will have a bunch of tests. This will lead to a large execution time. To overcome this, pytest provides us with an option to run tests in parallel.

For this, we need to first install the pytest-xdist plugin.

Install pytest-xdist by running the following command −

**pip install pytest-xdist**

**pytest -n 3**

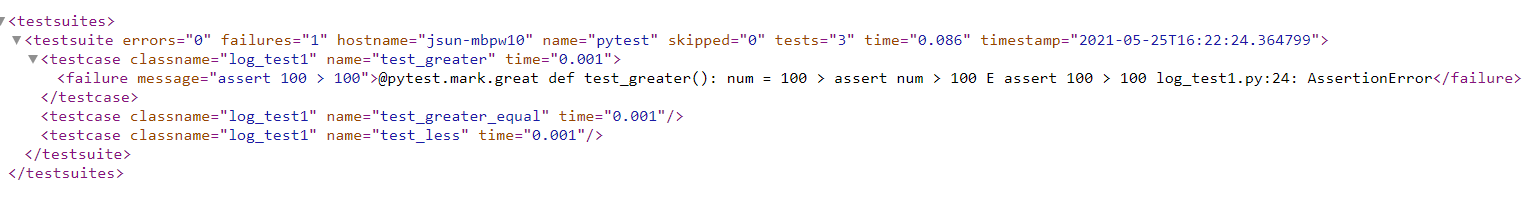
-n <num> runs the tests by using multiple workers, here it is 3.

1. **Test Execution Results in XML**

We can generate the details of the test execution in an xml file. This xml file is mainly useful in cases where we have a dashboard that projects the test results. In such cases, the xml can be parsed to get the details of the execution.

We will now execute the tests from test\_multiplcation.py and generate the xml by running

pytest test\_multiplication.py -v **--junitxml="result.xml"**



1. **Warnings and disable-warnings**

Disabling warnings summary

Although not recommended, you can use the --disable-warnings command-line option to suppress the warning summary entirely from the test run output.

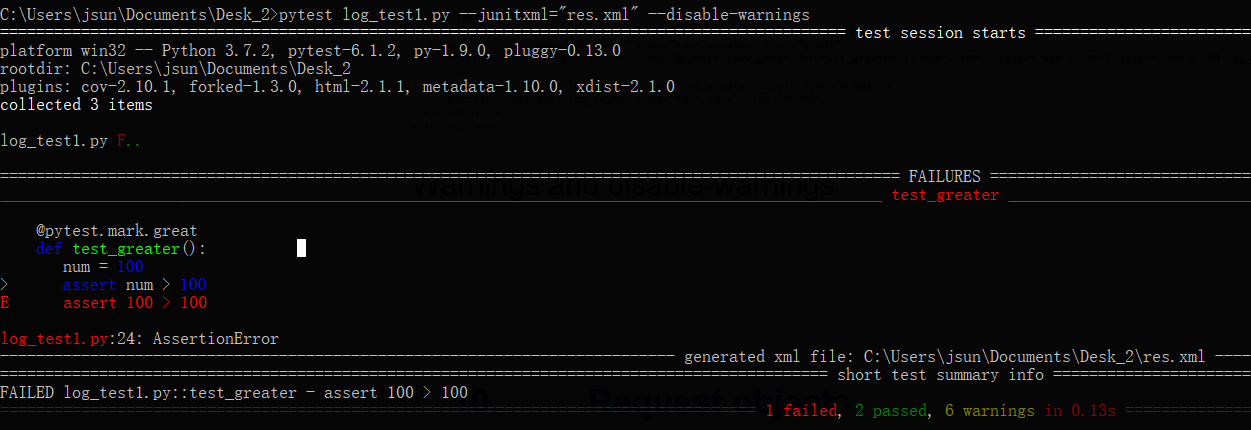
Disabling warning capture entirely[¶](https://docs.pytest.org/en/6.2.x/warnings.html#disabling-warning-capture-entirely)

This plugin is enabled by default but can be disabled entirely in your pytest.ini file with:

**[pytest]**

addopts = -p no:warnings

Or passing -p no:warnings in the command-line. This might be useful if your test suites handles warnings using an external system.

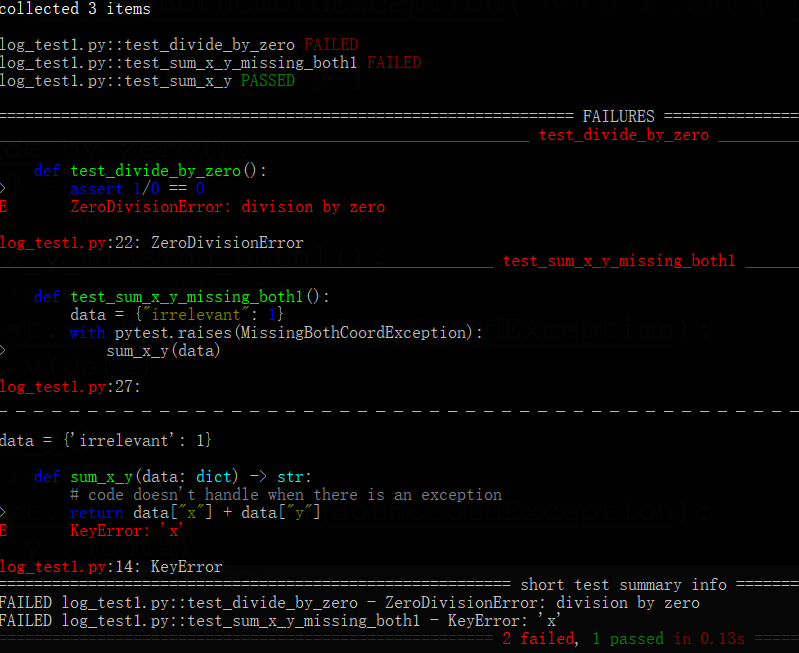


1. **Pytest Exceptions**

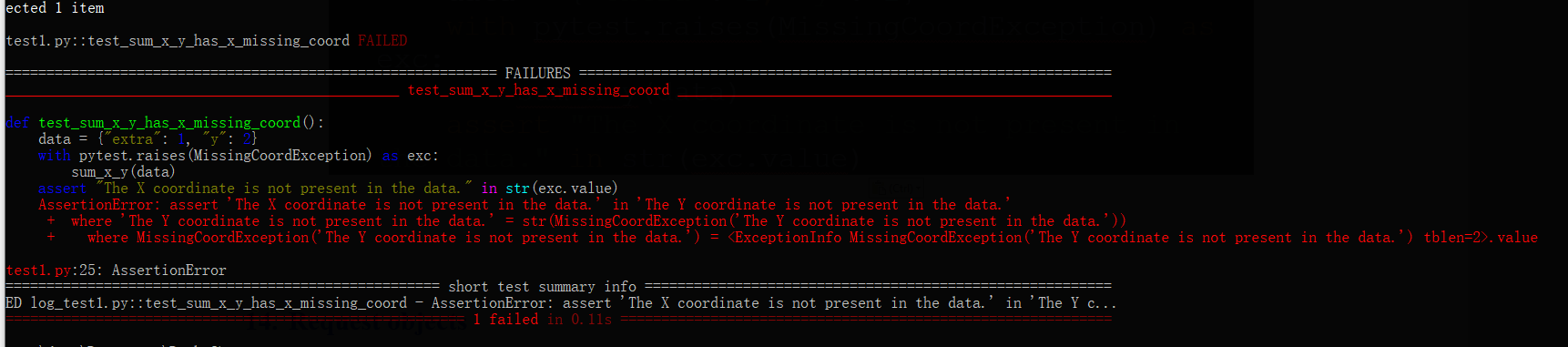
Let’s imagine that we have a function that checks for some keys in a dictionary. If a key is not present, it should raise a KeyError. As you can see, this is very generic and doesn’t tell the users much about the error. We can make it cleaner by raising custom exceptions, with different messages depending on the field.

* Check if an exception is raised in pytest.

import pytest  
  
  
class MissingCoordException(Exception):  
 *"""Exception raised when X or Y is not present in the data."""*class MissingBothCoordException(Exception):  
 *"""Exception raised when both X and Y are not present in the data."""*def sum\_x\_y(data: dict) -> str:  
 # code doesn't handle when there is an exception  
 return data["x"] + data["y"]  
  
def sum\_x\_y\_2(data: dict) -> str:  
 if "x" not in data and "y" not in data:  
 raise MissingBothCoordException("Both x and y coord missing.")  
 return data["x"] + data["y"]  
  
def test\_divide\_by\_zero():  
 assert 1/0 == 0  
  
def test\_sum\_x\_y\_missing\_both1():  
 data = {"irrelevant": 1}  
 with pytest.raises(MissingBothCoordException):  
 sum\_x\_y(data)  
  
def test\_sum\_x\_y():  
 data = {"irrelevant": 1}  
 with pytest.raises(MissingBothCoordException):  
 sum\_x\_y\_2(data)



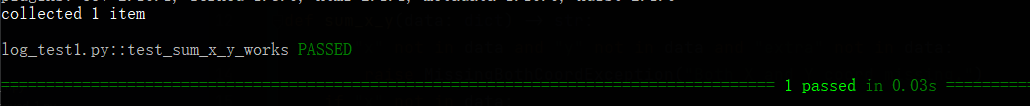
* **Assert Exception error message and type**
* import pytest  
    
    
  class MissingCoordException(Exception):  
   *"""Exception raised when X or Y is not present in the data."""*class MissingBothCoordException(Exception):  
   *"""Exception raised when both X and Y are not present in the data."""*def sum\_x\_y(data: dict) -> str:  
   if "x" not in data and "y" not in data and "extra" not in data:  
   raise MissingBothCoordException("Both X and Y coord missing.")  
   if "x" not in data:  
   raise MissingCoordException("The Y coordinate is not present in the data.")  
   if "y" not in data:  
   raise MissingCoordException("The Y coordinate is not present in the data.")  
   return data["x"] + data["y"]  
    
  def test\_sum\_x\_y\_has\_x\_missing\_coord():  
   data = {"extra": 1, "y": 2}  
   with pytest.raises(MissingCoordException) as exc:  
   sum\_x\_y(data)  
   assert "The X coordinate is not present in the data." in str(exc.value)

****

* **How to Assert That NO Exception Is Raised**

The last section in this tutorial is about yet another common use case: how to assert that no exception is thrown. One way we can do that is by using a try / except. If it raises an exception, we catch it and assert False.

import pytest  
  
  
class MissingCoordException(Exception):  
 *"""Exception raised when X or Y is not present in the data."""*class MissingBothCoordException(Exception):  
 *"""Exception raised when both X and Y are not present in the data."""*def sum\_x\_y(data: dict) -> str:  
 if "x" not in data and "y" not in data and "extra" not in data:  
 raise MissingBothCoordException("Both X and Y coord missing.")  
 if "x" not in data:  
 raise MissingCoordException("The Y coordinate is not present in the data.")  
 if "y" not in data:  
 raise MissingCoordException("The Y coordinate is not present in the data.")  
 return data["x"] + data["y"]  
  
def test\_sum\_x\_y\_works():  
 data = {"extra": 1, "y": 2, "x": 1}  
  
 try:  
 sum\_x\_y(data)  
 except Exception as exc:  
 assert False, f"'sum\_x\_y' raised an exception {exc}"

****

import pytest  
  
  
class MissingCoordException(Exception):  
 *"""Exception raised when X or Y is not present in the data."""*class MissingBothCoordException(Exception):  
 *"""Exception raised when both X and Y are not present in the data."""*# def sum\_x\_y(data: dict) -> str:  
# if "x" not in data and "y" not in data and "extra" not in data:  
# raise MissingBothCoordException("Both X and Y coord missing.")  
# if "x" not in data:  
# raise MissingCoordException("The Y coordinate is not present in the data.")  
# if "y" not in data:  
# raise MissingCoordException("The Y coordinate is not present in the data.")  
# return data["x"] + data["y"]  
  
def sum\_x\_y(data: dict) -> str:  
 if "x" not in data and "y" not in data and "extra" not in data:  
 raise MissingBothCoordException("'extra field and x / y coord missing.")  
 if "x" not in data:  
 raise MissingCoordException("The X coordinate is not present in the data.")  
 if "y" not in data:  
 raise MissingCoordException("The Y coordinate is not present in the data.")  
 raise ValueError("Oh no, this shouldn't have happened.")  
 return data["x"] + data["y"]  
  
def test\_sum\_x\_y\_works():  
 data = {"extra": 1, "y": 2, "x": 1}  
  
 try:  
 sum\_x\_y(data)  
 except Exception as exc:  
 assert False, f"'sum\_x\_y' raised an exception {exc}"

****

1. **Request objects**

### Pytest fixture request parameter

In the [cheese db example](https://pythontesting.net/framework/pytest/pytest-fixtures-nuts-bolts/#cheese_example), the fixture includes a request parameter. You need the request parameter to a fixture to add a finilizer.

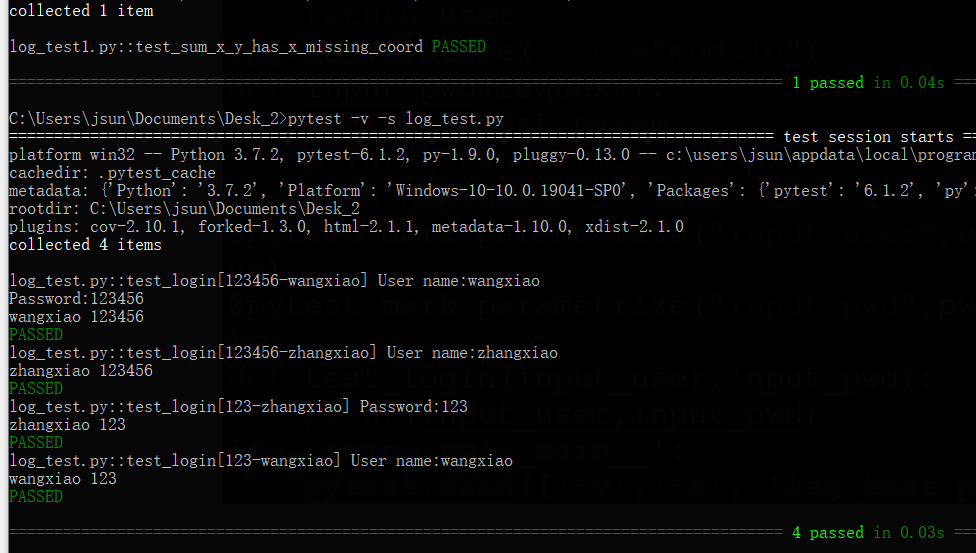
However, it has other uses too.

In the example below, I’m showing the use (well, printing stuff) of some of the items

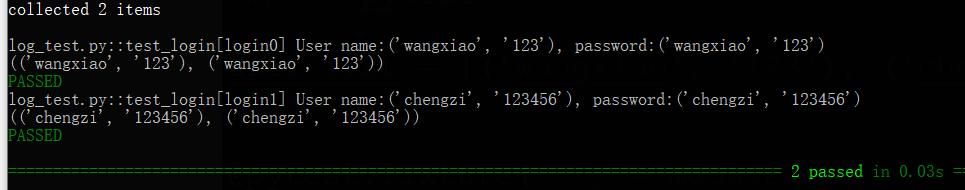
@pytest.fixture()  
def my\_fixture(request):  
 print('\n-----------------')  
 print('fixturename : %s' % request.fixturename)  
 print('scope : %s' % request.scope)  
 print('function : %s' % request.function.\_\_name\_\_)  
 print('cls : %s' % request.cls)  
 print('module : %s' % request.module.\_\_name\_\_)  
 print('fspath : %s' % request.fspath)  
 print('-----------------')  
  
 if request.function.\_\_name\_\_ == 'test\_three':  
 request.applymarker(pytest.mark.xfail)  
  
def test\_one(my\_fixture):  
 print('test\_one():')  
  
class TestClass():  
 def test\_two(self, my\_fixture):  
 print('test\_two()')  
  
def test\_three(my\_fixture):  
 print('test\_three()')  
 assert False

**#**

import pytest  
user\_list=['wangxiao','zhangxiao']  
pwd\_list=['123456','123']  
@pytest.fixture(scope="module")  
def input\_user(request):  
 user=request.param  
 print ("User name:% s"% user)  
 return user  
@pytest.fixture(scope="module")  
def input\_pwd(request):  
 pwd = request.param  
 print ("Password:% s"% pwd)  
 return pwd  
@pytest.mark.parametrize("input\_user",user\_list,indirect=True)  
@pytest.mark.parametrize("input\_pwd",pwd\_list,indirect=True)  
def test\_login(input\_user,input\_pwd):  
 print(input\_user,input\_pwd)  
**if \_\_name\_\_=='\_\_main\_\_':  
 pytest.main(['-v','-s', 'log\_test.py'])**

****

import pytest  
  
user\_list = [('wangxiao', '123'), ('chengzi', '123456')]  
  
  
@pytest.fixture(scope="module")  
def login(request):  
 user = request.param  
 pwd = request.param  
 print("User name:% s, password:% s" % (user, pwd))  
 return user, pwd  
  
  
@pytest.mark.parametrize("login", user\_list, indirect=True)  
def test\_login(login):  
 print(login)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 pytest.main(['-s', 'log\_test.py'])

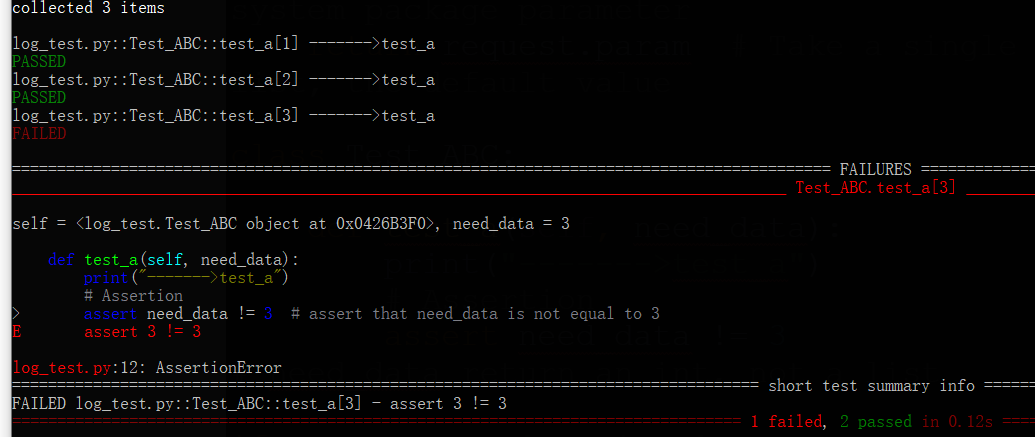


#

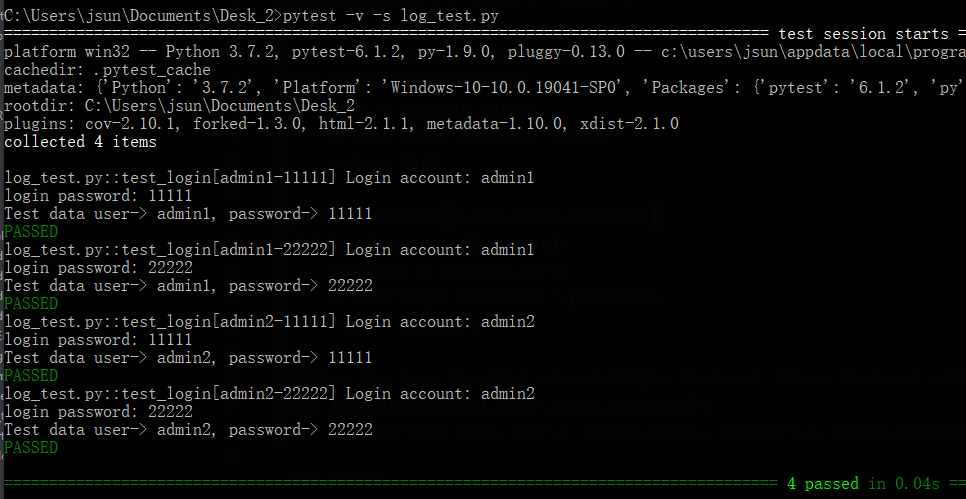
List of [1,2,3] is called one at a time

import pytest  
  
@pytest.fixture(params=[1, 2, 3])  
def need\_data(request): # incoming parameter request system package parameter  
 return request.param # Take a single value in the list, the default value  
  
class Test\_ABC:  
  
 def test\_a(self, need\_data):  
 print("------->test\_a")  
 # Assertion  
 assert need\_data != 3

# need\_data return an int, not a list  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 pytest.main(["-s", "-v", "log\_test.py"])



import pytest  
# Test account data  
test\_user = ["admin1", "admin2"]  
test\_password = ["11111", "22222"]  
  
@pytest.fixture(params=test\_user)  
def input\_user(request):  
 user = request.param  
 print('Login account:',user)  
 return user  
  
@pytest.fixture(params=test\_password)  
def input\_password(request):  
 password = request.param  
 print('login password:',password)  
 return password  
  
# When the same function uses multiple fixtures, these fixtures will be combined to test each situation  
def test\_login(input\_user,input\_password):  
 print(f"Test data user-> {input\_user}, password-> {input\_password}")



### Implement setup and teardown

We can use conftest.py to implement setup for test cases, then we can also implement the similar function of teardown.

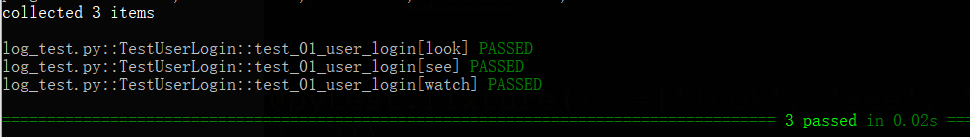
We can use**yield** to fulfill

# conftest.py  
  
  
import pytest  
  
info = {"username": "root", "password": "mysql"}  
  
@pytest.fixture(scope='module')  
def login\_ini():  
 # Establish a database link  
 db = pymysql.connect(host="127.0.0.1", user=info['username'], password=info['password'], database='school')  
 cursor = db.cursor()  
  
 # Use execute() method to execute SQL query  
 cursor.execute("select s\_name from student")  
 # Get all the data  
 data = cursor.fetchall()  
 print(data)  
  
 # Get disconnected database connection  
 # Use yield to achieve the similar function of teardown. That is, execute the content behind yield after the test function is executed  
 # yield login\_ini  
 # You can write a yield directly without writing others  
 yield  
 cursor.close()  
 db.close()  
 print("Disconnected")  
  
def test\_login(login\_ini):  
 print('Query the database successfully')  
  
def test\_2(login\_ini):  
 print('test\*\*\*\*\*')  
  
  
# collected 1 item  
  
# test\_fix\_setup.py (('Zhao Lei',), ('Money and electricity',), ('Sun Feng',), ('Li Yun',), ('Zhou Mei',), ('Wu Lan',), ('Zheng Zhu',), ('Wang Ju',))  
# Query the database successfully  
# .Test\*\*\*\*\*  
# .Disconnect  
#  
# ============================================================ 2 passed in 0.04s ========

# id

* ids: A list of string ids. Each id corresponds to a parameter and is part of the test id. If no id is provided, they will automatically generate an identifier from the parameter.

# conftest.py  
  
import pytest  
  
  
@pytest.fixture(ids=['look', 'see', 'watch'], params=[1, 2, 3])  
def common\_login(request):  
 yield request.param  
  
# test\_account\_login.py  
  
import pytest  
  
@pytest.mark.usefixtures('common\_login')  
class TestUserLogin:  
  
 def test\_01\_user\_login(self):  
 assert 1 == 1  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 pytest.main(['-v', '-s', 'log\_test.py'])



* Give fixture a name
* # conftest.py  
    
  import pytest  
    
  @pytest.fixture(name='XXX', params=[1, 2, 3])  
  def common\_login(request):  
   yield request.param  
    
  # test\_account\_login.py  
    
  import pytest  
  @pytest.mark.usefixtures('XXX')  
  class TestUserLogin:  
    
   def test\_01\_user\_login(self):  
   assert 1 == 1  
    
    
  if \_\_name\_\_ == '\_\_main\_\_':  
   pytest.main(['-v', '-s', 'log\_test.py'])

