* Create folder as (Python Package)
* Create file (test\_demo1.py)
* Keep in mind that both methods and test files need the test\_ prefix or \_test suffix to be recognized as a test file.
* Class name should also start with <Test as TestExample>

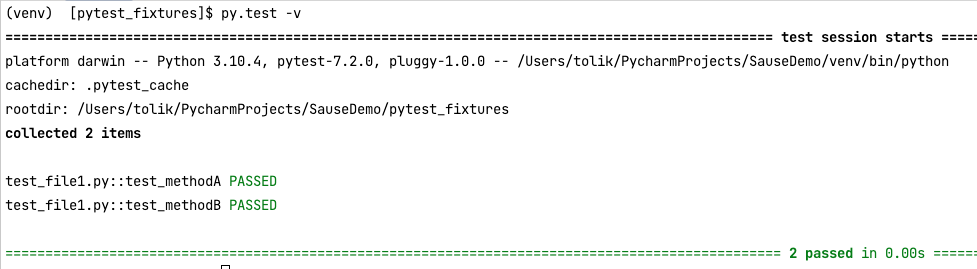
def test\_methodA():  
 print('Running method A')  
  
  
def test\_methodB():  
 print('Running method B')

Чтобы запустить файл: $ py.test



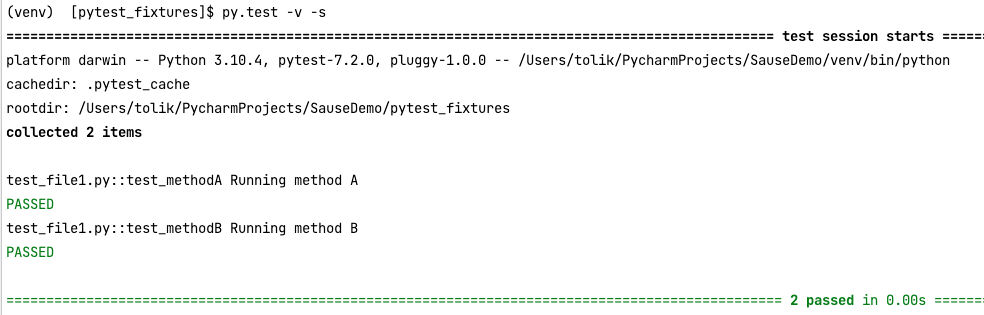
$ py.test -v

 -v gives the results more verbosity and detail to our tests. We can now see which specific tests have failed or passed.



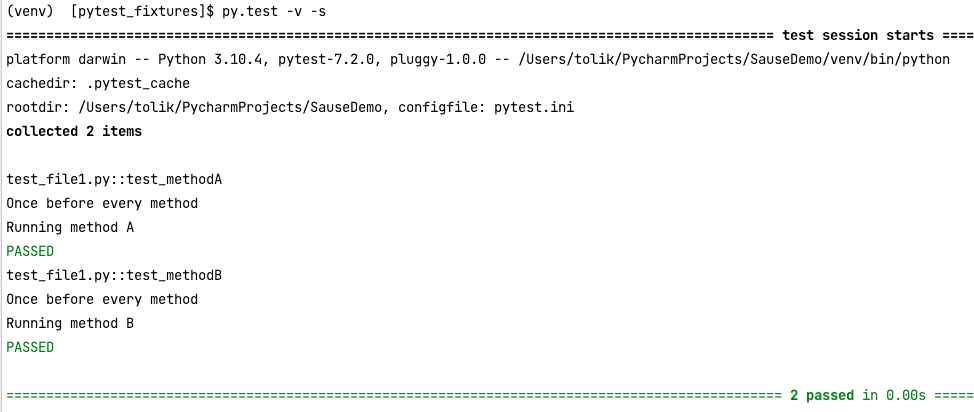
$ py.test -v -s

-s prints statements

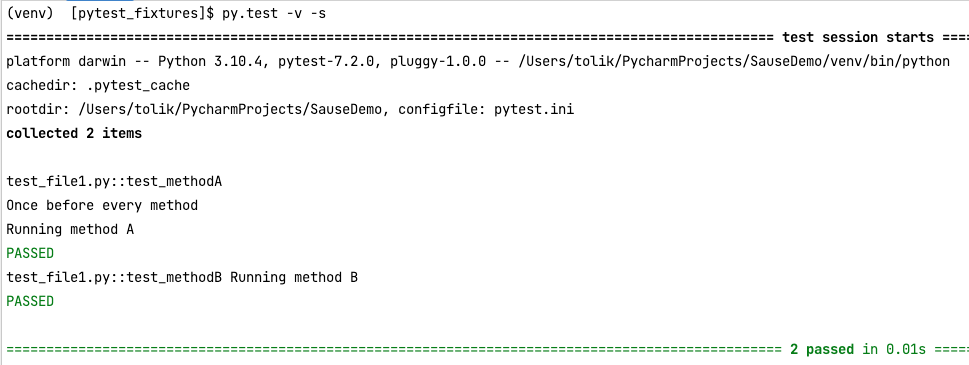


* Import the pytest module in all Python files you want to test, as well as in any associated configuration files for the fixtures.
* Function is a fixture with @pytest.fixture. These specific Python decorations let us know that the next method is a pytest fixture.
* @pytest.fixture => Implementing the simplest pytest fixture can just return an object, like an integer.

import pytest  
  
@pytest.fixture()  
def setUp():  
 print("Once before every method")  
  
  
def test\_methodA(setUp):  
 print('Running method A')  
  
  
def test\_methodB(setUp):  
 print('Running method B')



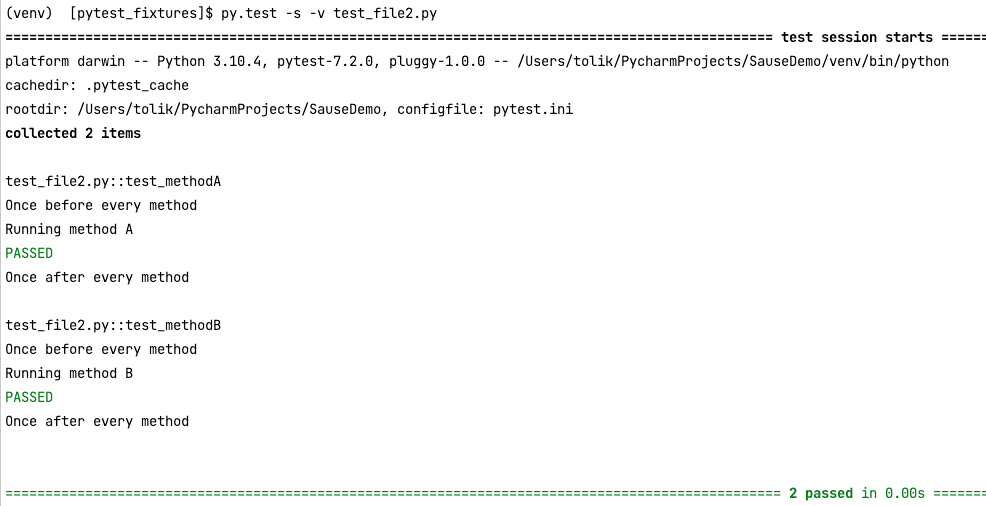
import pytest  
  
@pytest.fixture()  
def setUp():  
 print("\nOnce before every method")  
  
  
def test\_methodA(setUp):  
 print('Running method A')  
  
  
def test\_methodB():  
 print('Running method B')



* Create one more test file (e.g. test\_file2.py)

import pytest  
  
  
@pytest.fixture()  
def set\_up():  
 print("\nOnce before every method")  
 yield  
 print('\nOnce after every method')  
  
  
def test\_methodA(set\_up):  
 print('Running method A')  
  
  
def test\_methodB(set\_up):  
 print('Running method B')

$ py.test -s -v test\_file2.py => run one file from the package folder



Multiple Ways to Run Test Cases

test\_file1.py

import pytest  
  
@pytest.fixture()  
def set\_up():  
 print("\nfile1 : Once before every method")  
  
  
def test\_methodA(set\_up):  
 print('Running method A file1')  
  
  
def test\_methodB(set\_up):  
 print('Running method B file1')

test\_file2.py

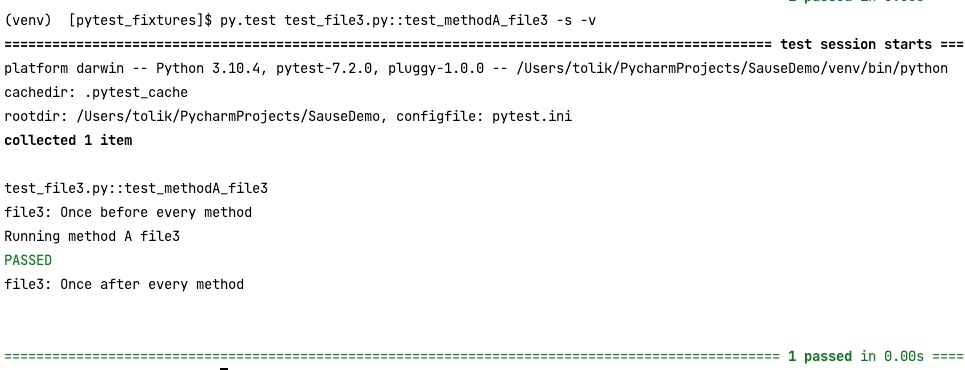
import pytest  
  
  
@pytest.fixture()  
def set\_up():  
 print("\nfile2: Once before every method")  
 yield  
 print('\nfile2: Once after every method')  
  
  
def test\_methodA\_file2(set\_up):  
 print('Running method A file2')  
  
  
def test\_methodB\_file2(set\_up):  
 print('Running method B file2')

test\_file3.py

import pytest  
  
  
@pytest.fixture()  
def set\_up():  
 print("\nfile3: Once before every method")  
 yield  
 print('\nfile3: Once after every method')  
  
  
def test\_methodA\_file3(set\_up):  
 print('Running method A file3')  
  
  
def test\_methodB\_file3(set\_up):  
 print('Running method B file3')

* To run only one method from specific file:

$ py.test test\_file3.py::test\_methodA\_file3 -s -v



conftest.py => Common Fixtures to Multiple Modules

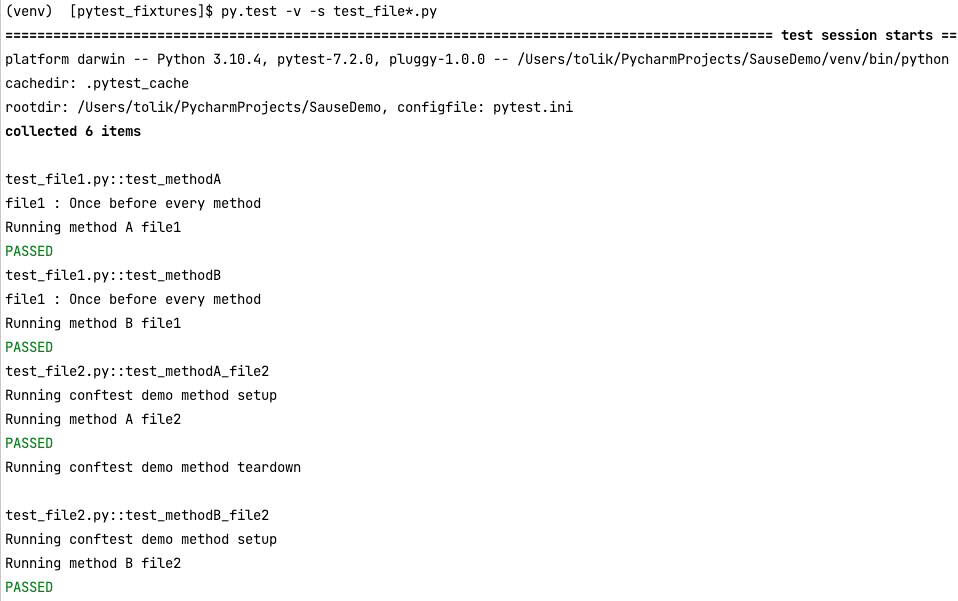
Создаем файл conftest.py

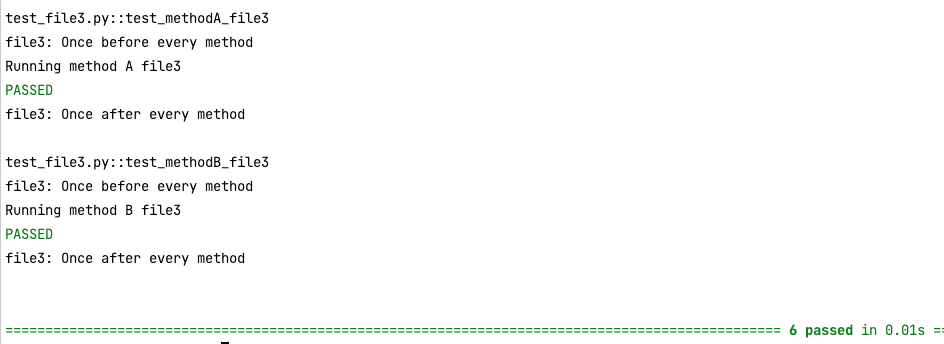
import pytest  
  
  
@pytest.fixture()  
def set\_up():  
 print("\nRunning conftest demo method setup")  
 yield  
 print('\nRunning conftest demo method teardown')

Из файла test\_file2.py удаляем функцию def set\_up():

import pytest  
  
  
def test\_methodA\_file2(set\_up):  
 print('Running method A file2')  
  
  
def test\_methodB\_file2(set\_up):  
 print('Running method B file2')

$ py.test -v -s test\_file\*.py



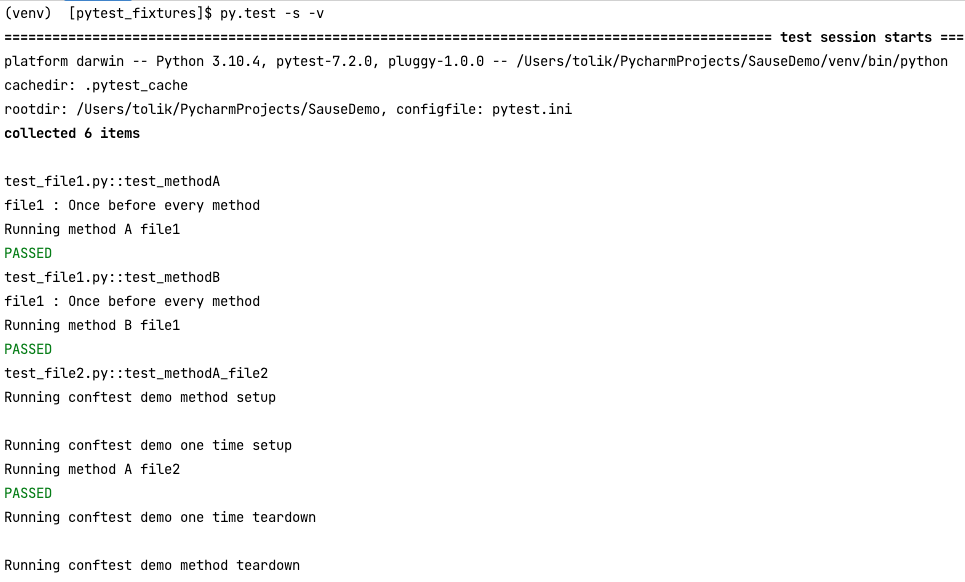


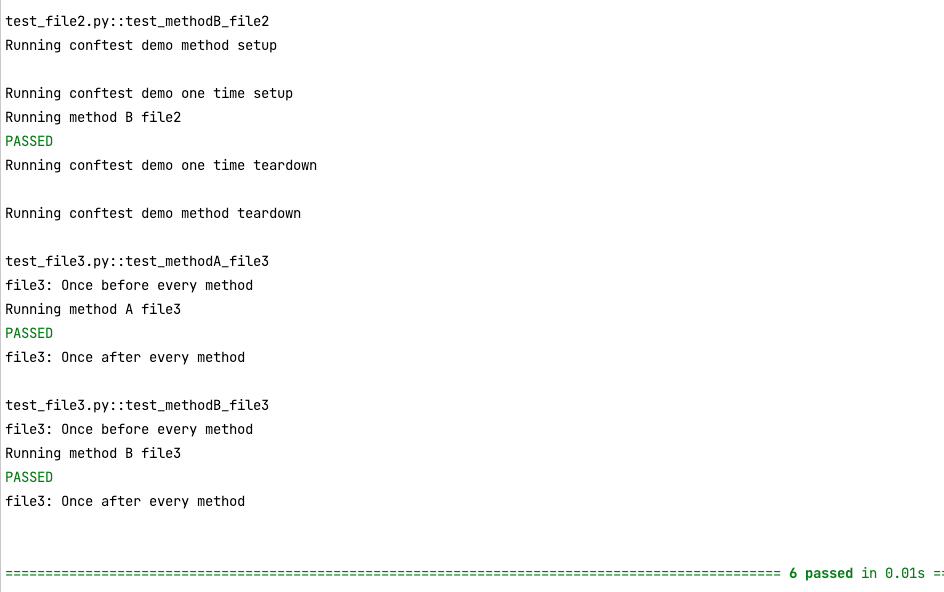
Добавляем еще одну функцию в conftest file

import pytest  
  
  
@pytest.fixture()  
def set\_up():  
 print("\nRunning conftest demo method setup")  
 yield  
 print('\nRunning conftest demo method teardown')  
  
  
@pytest.fixture()  
def oneTimeSetup(scope='module'):  
 print("\nRunning conftest demo one time setup")  
 yield  
 print('\nRunning conftest demo one time teardown')

By default, scope set as a function, and it applies to every function in the module.

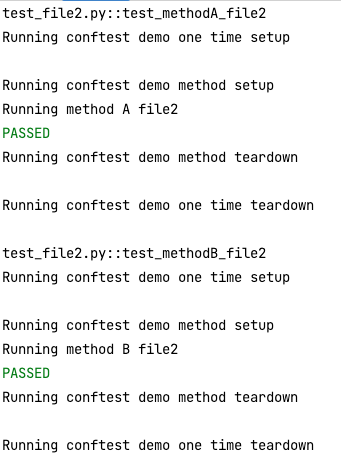
import pytest  
  
  
def test\_methodA\_file2(set\_up, oneTimeSetup):  
 print('Running method A file2')  
  
  
def test\_methodB\_file2(set\_up, oneTimeSetup):  
 print('Running method B file2')





Изменяем порядок аргументов в функции. И первым аргументом будет, oneTimeSetup, а потом set\_up

import pytest  
  
  
def test\_methodA\_file2(oneTimeSetup, set\_up ):  
 print('Running method A file2')  
  
  
def test\_methodB\_file2(oneTimeSetup, set\_up):  
 print('Running method B file2')



How to maintain run order of tests

Install new package pytest-ordering

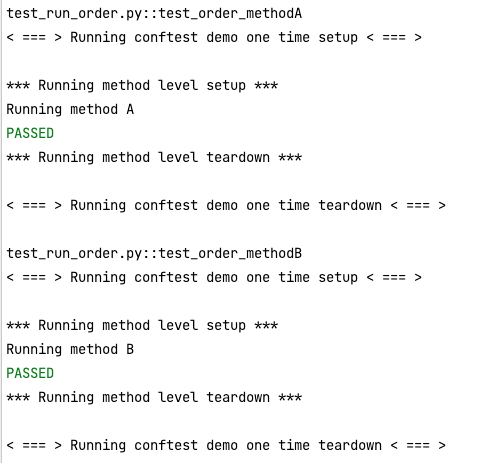
Create file test\_run\_ordering.py

conftest.py

import pytest  
  
  
@pytest.fixture()  
def set\_up():  
 print("\n\*\*\* Running method level setup \*\*\*")  
 yield  
 print('\n\*\*\* Running method level teardown \*\*\*')  
  
  
@pytest.fixture()  
def oneTimeSetup(scope='module'):  
 print("\n< === > Running conftest demo one time setup < === >")  
 yield  
 print('\n< === > Running conftest demo one time teardown < === >')

test\_run\_ordering.py

import pytest  
  
  
def test\_order\_methodA(oneTimeSetup, set\_up):  
 print('Running method A')  
  
  
def test\_order\_methodB(oneTimeSetup, set\_up):  
 print('Running method B')  
  
  
def test\_order\_methodC(oneTimeSetup, set\_up):  
 print('Running method C')  
  
  
def test\_order\_methodD(oneTimeSetup, set\_up):  
 print('Running method D')  
  
  
def test\_order\_methodE(oneTimeSetup, set\_up):  
 print('Running method E')  
  
  
def test\_order\_methodF(oneTimeSetup, set\_up):  
 print('Running method F')



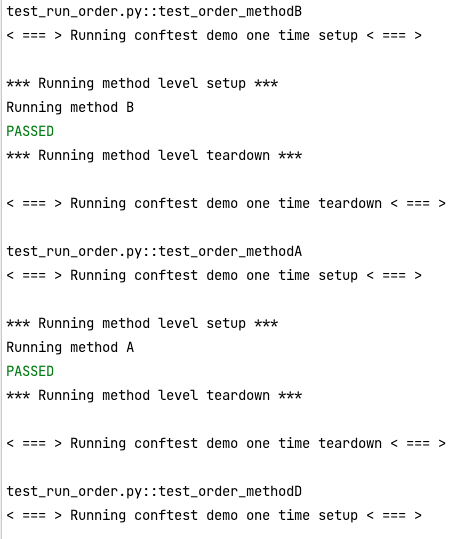
import pytest  
  
  
@pytest.mark.run(order=2)  
def test\_order\_methodA(oneTimeSetup, set\_up):  
 print('Running method A')  
  
  
@pytest.mark.run(order=1)  
def test\_order\_methodB(oneTimeSetup, set\_up):  
 print('Running method B')  
  
@pytest.mark.run(order=4)  
def test\_order\_methodC(oneTimeSetup, set\_up):  
 print('Running method C')  
  
  
@pytest.mark.run(order=6)  
def test\_order\_methodD(oneTimeSetup, set\_up):  
 print('Running method D')  
  
  
@pytest.mark.run(order=3)  
def test\_order\_methodE(oneTimeSetup, set\_up):  
 print('Running method E')  
  
  
@pytest.mark.run(order=5)  
def test\_order\_methodF(oneTimeSetup, set\_up):  
 print('Running method F')

Install new package pytest-order

Create new file test\_run\_order.py

https://pypi.org/project/pytest-order/

import pytest  
  
  
@pytest.mark.order(2)  
def test\_order\_methodA(oneTimeSetup, set\_up):  
 print('Running method A')  
  
  
@pytest.mark.order(1)  
def test\_order\_methodB(oneTimeSetup, set\_up):  
 print('Running method B')  
  
@pytest.mark.order(4)  
def test\_order\_methodC(oneTimeSetup, set\_up):  
 print('Running method C')  
  
  
@pytest.mark.order(3)  
def test\_order\_methodD(oneTimeSetup, set\_up):  
 print('Running method D')  
  
  
@pytest.mark.order(6)  
def test\_order\_methodE(oneTimeSetup, set\_up):  
 print('Running method E')  
  
  
@pytest.mark.order(5)  
def test\_order\_methodF(oneTimeSetup, set\_up):  
 print('Running method F')



Running Tests Based on Command Line Arguments

conftest.py

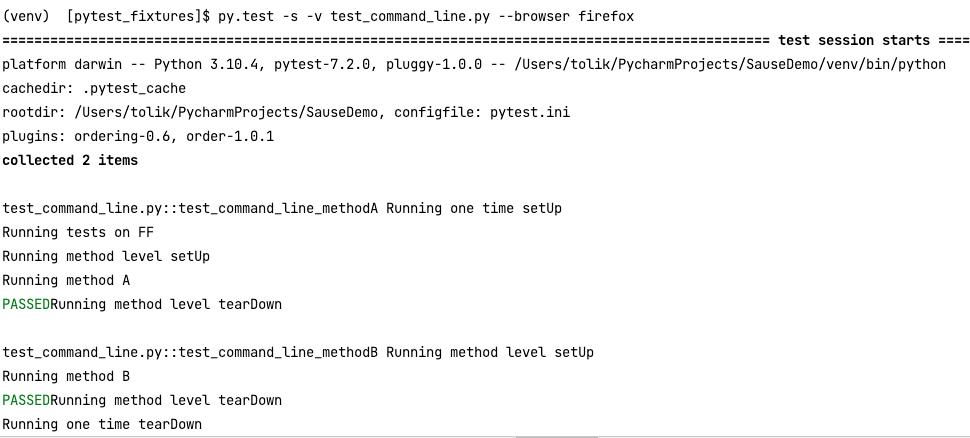
import pytest  
  
  
@pytest.fixture()  
def setUp():  
 print("Running method level setUp")  
 yield  
 print("Running method level tearDown")  
  
  
@pytest.fixture(scope="module")  
def oneTimeSetUp(browser, osType):  
 print("Running one time setUp")  
 if browser == 'firefox':  
 print("Running tests on FF")  
 else:  
 print("Running tests on chrome")  
 yield  
 print("Running one time tearDown")  
  
  
def pytest\_addoption(parser):  
 parser.addoption("--browser")  
 parser.addoption("--osType", help="Type of operating system")  
  
  
@pytest.fixture(scope="session")  
def browser(request):  
 return request.config.getoption("--browser")  
  
  
@pytest.fixture(scope="session")  
def osType(request):  
 return request.config.getoption("--osType")

test\_command\_line.py

import pytest  
  
  
def test\_command\_line\_methodA(oneTimeSetUp, setUp):  
 print("Running method A")  
  
  
def test\_command\_line\_methodB(oneTimeSetUp, setUp):  
 print("Running method B")

$ py.test -s -v test\_command\_line.py --browser firefox

$ py.test -s -v test\_command\_line.py --browser chrome



Structure Tests in a Test Class

* How to use test class to wrap methods under one class
* Learn about the autouse keywords in fixtures
* Assert the result to create a real test scenario
* @pytest.fixture(scope="class") in the conftest.py file change the scope to be equal to class

Create file < class\_to\_test.py >

class SomeClassToTest:  
  
 def \_\_init\_\_(self, value):  
 self.value = value  
  
 def sumNumbers(self, a, b):  
 return a + b + self.value

conftest.py file

import pytest  
  
  
@pytest.fixture()  
def set\_up():  
 print("Running method level setUp")  
 yield  
 print("Running method level tearDown")  
  
  
@pytest.fixture(scope="class")  
def oneTimeSetUp(browser, osType):  
 print("Running one time setUp")  
 if browser == 'firefox':  
 print("Running tests on FF")  
 else:  
 print("Running tests on chrome")  
 yield  
 print("Running one time tearDown")  
  
  
def pytest\_addoption(parser):  
 parser.addoption("--browser")  
 parser.addoption("--osType", help="Type of operating system")  
  
  
@pytest.fixture(scope="session")  
def browser(request):  
 return request.config.getoption("--browser")  
  
  
@pytest.fixture(scope="session")  
def osType(request):  
 return request.config.getoption("--osType")

* Create the file < test\_class.py >
* import pytest  
  from pytest\_fixtures.class\_to\_test import SomeClassToTest  
    
    
  class TestClaseDemo:  
     
   def test\_methodA(self):  
   print('Running method A')  
     
   def test\_methodB(self):  
   print('Running method B')

Импортируем pytest, также импортируем из файла class\_to\_test => class SomeClassToTest

Если мы хотим импортировать некоторые декораторы из fixtures и у нас этих методов около сотни, то должен ли я прописывать их всех в качестве аргументов?

def test\_methodA(self, oneTimeSetup, set\_up, method1, method2, args):  
 print('Running method A')

Легче добавить fixture и перечислить все те функции, которые мы хотим использовать из conftest файла. Это добавляется над строкой класс и указываем, какие функции мы хотим использовать. Таким образом все эти функции доступны для нашего класса.

Пример:

@pytest.mark.usefixtures("метод1", "метод2", "метод3", ..args)  
class TestClaseDemo:

import pytest  
from pytest\_fixtures.class\_to\_test import SomeClassToTest  
  
  
@pytest.mark.usefixtures("oneTimeSetUp", "set\_up")  
class TestClaseDemo:  
  
 @pytest.fixture(autouse=True)  
 def classSetup(self):  
 self.abc = SomeClassToTest(10)  
  
 def test\_methodA(self):  
 result = self.abc.sumNumbers(2, 8)  
 assert result == 20  
 print("Running method A")  
  
 def test\_methodB(self):  
 print("Running method B")

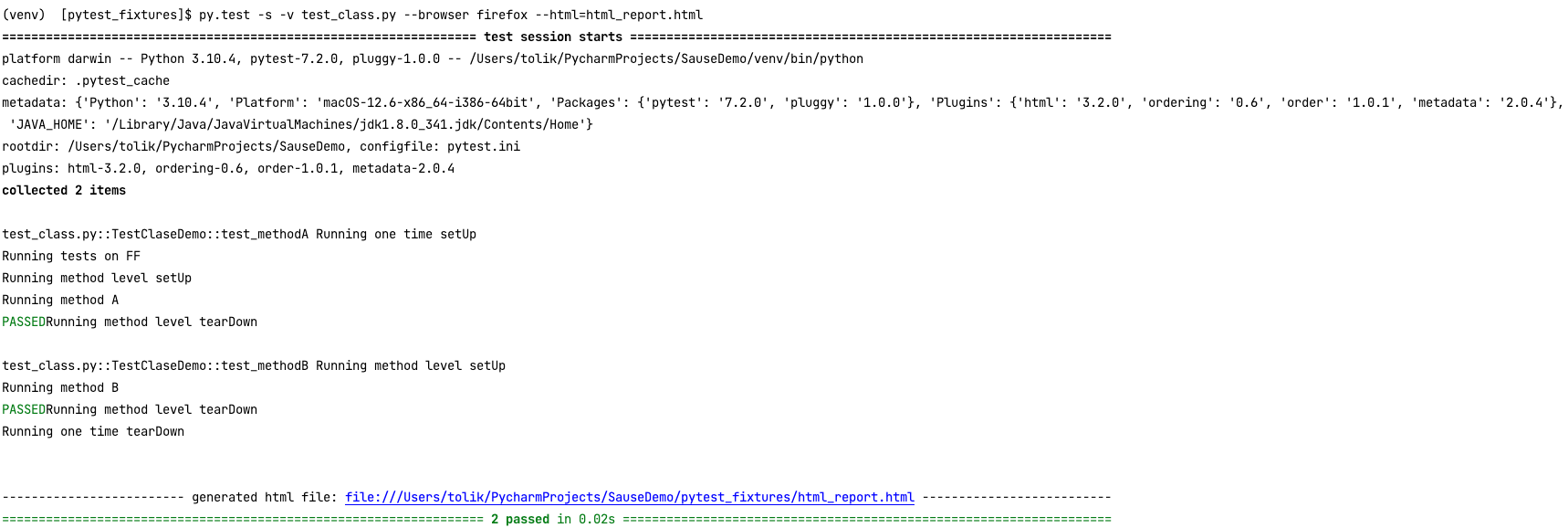
How to Generate HTML Test Report

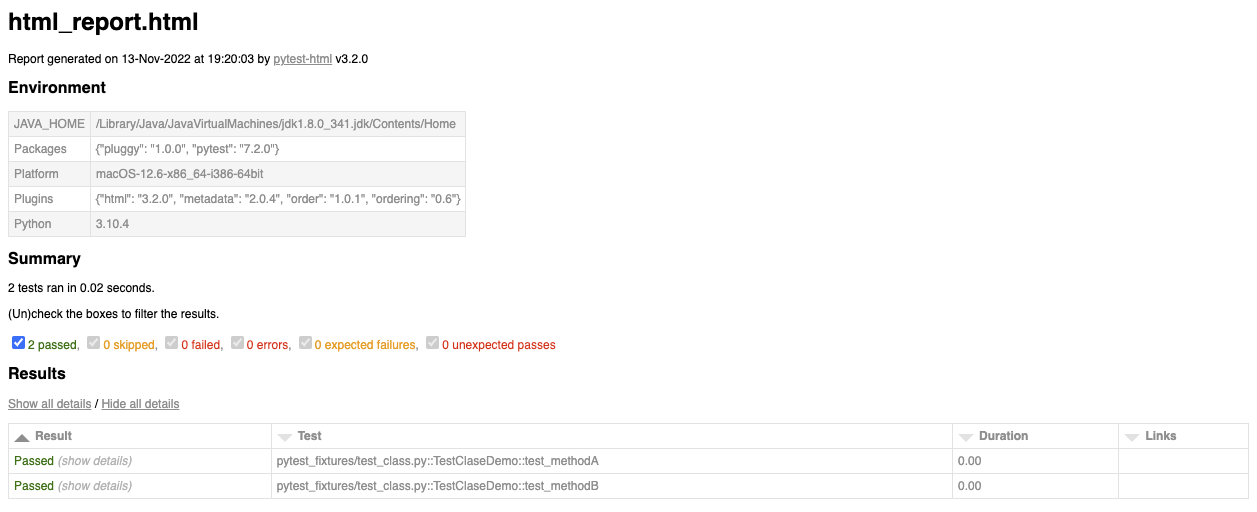
Install package pytest-html

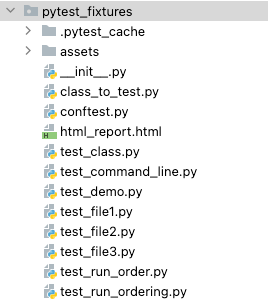
$ pip3 install pytest-html

$ py.test -s -v test\_class.py --browser firefox --html=html\_report.html

Если вы хотите получить репорт то в конце строки команды введите --html=имя\_файла.html







Taking a screenshot method

def screenShot(self, resultMessage):

"""

Takes screenshot of the current open web page

"""

fileName = resultMessage + "." + str(round(time.time() \* 1000)) + ".png"

screenshotDirectory = "../screenshots/"

relativeFileName = screenshotDirectory + fileName

currentDirectory = os.path.dirname(\_\_file\_\_)

destinationFile = os.path.join(currentDirectory, relativeFileName)

destinationDirectory = os.path.join(currentDirectory, screenshotDirectory)

try:

if not os.path.exists(destinationDirectory):

os.makedirs(destinationDirectory)

self.driver.save\_screenshot(destinationFile)

self.log.info("Screenshot save to directory: " + destinationFile)

except:

self.log.error("### Exception Occurred when taking screenshot")

print\_stack()