```
py.test test_sample.py --collect-only             # collects information test suite

py.test test_sample.py -v                         # outputs verbose messages

py.test -q test_sampl                             # omit filename output

python -m pytest -q test_sample.py                # calling pytest through python

py.test --markers                                 # show available markers


# In order to create a reusable marker.

/*
# content of pytest.ini
[pytest]
markers =
    webtest: mark a test as a webtest.
*/



py.test -k "TestClass and not test_one"           # only run tests with names that match the "string
expression"

py.test test_server.py::TestClass::test_method    # cnly run tests that match the node ID

py.test -x                                        # stop after first failure

py.test --maxfail=2                               # stop after two failures

py.test --showlocals                              # show local variables in tracebacks

py.test -l                                        # (shortcut)


py.test --tb=long                                 # the default informative traceback formatting

py.test --tb=native                               # the Python standard library formatting

py.test --tb=short                                # a shorter traceback format

py.test --tb=line                                 # only one line per failure

py.test --tb=no                                   # no tracebak output

py.test -x --pdb                                  # drop to PDB on first failure, then end test session

py.test --durations=10                            # list of the slowest 10 test durations.
```

```
py.test --maxfail=2 -rf                    # exit after 2 failures, report fail info.

py.test -n 4                               # send tests to multiple CPUs



py.test -m slowest                         # run tests with decorator @pytest.mark.slowest or slowest = pytest.mark.slowest; @slowest



py.test --traceconfig   # find out which py.test plugins are active in your environment.

py.test --instafail          # if pytest-instafail is installed, show errors and failures instantly instead of waiting until the end of test suite.



# Test using parametrize
/*
    import pytest


    @pytest.mark.parametrize(
        ('n', 'expected'), [
            (1, 2),
            (2, 3),
            (3, 4),
            (4, 5),
            pytest.mark.xfail((1, 0)),
            pytest.mark.xfail(reason="some bug")((1, 0)),
            pytest.mark.skipif('sys.version_info >= (3,0)')((10, 11)),
        ]
    )

    def test_increment(n, expected):
        assert n + 1 == expected


***** *****

import unittest


class TestAbs(unittest.TestCase):
    def test_abs1(self):
        self.assertEqual(abs(-42), 42, "Should be absolute value of a number")

    def test_abs2(self):
        self.assertEqual(abs(-42), -42, "Should be absolute value of a number")
```

```
if __name__ == "__main__":
    unittest.main()
```

***** *****

```
def test_abs1():
assert abs(-42) == 42, "Should be absolute value of a number"

if __name__ == "__main__":
        test_abs1()
        print("All tests passed!")
```

***** *****

Классические фикстуры (fixtures)

Важной составляющей в использовании PyTest является концепция фикстур. Фикстуры в контексте PyTest — это вспомогательные функции для наших тестов, которые не являются частью тестового сценария.

Назначение фикстур может быть самым разным. Одно из распространенных применений фикстур — это подготовка тестового окружения и очистка тестового окружения и данных после завершения теста. Но, вообще говоря, фикстуры можно использовать для самых разных целей: для подключения к базе данных, с которой работают тесты, создания тестовых файлов или подготовки данных в текущем окружении с помощью API-методов. Более подробно про фикстуры в широком смысле вы можете прочитать в Википедии.

Классический способ работы с фикстурами — создание setup- и teardown-методов в файле с тестами (документация в PyTest).

Можно создавать фикстуры для модулей, классов и отдельных функций. Давайте попробуем написать фикстуру для инициализации браузера, который мы затем сможем использовать в наших тестах. После окончания тестов мы будем автоматически закрывать браузер с помощью команды browser.quit(), чтобы в нашей системе не оказалось множество открытых окон браузера. Вынесем инициализацию и закрытие браузера в фикстуры, чтобы не писать этот код для каждого теста.

Будем сразу объединять наши тесты в тест-сьюты, роль тест-сьюта будут играть классы, в которых мы будем хранить наши тесты.

Рассмотрим два примера: создание экземпляра браузера и его закрытие только один раз для всех тестов первого тест-сьюта и создание браузера для каждого теста во втором тест-сьюте. Сохраните следующий код в файл test_fixture1.py и запустите его с помощью PyTest. Не забудьте указать параметр -s, чтобы увидеть текст, который выводится командой print().

```
pytest -s test_fixture1.py
```

test_fixture1.py:

```
import time
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))

link = "http://selenium1py.pythonanywhere.com/"

'''
example fixtures
'''

class TestMainPage1():

    @classmethod
    def setup_class(self):
        print("\nstart browser for test suite..")
        self.browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))

    @classmethod
    def teardown_class(self):
        print("quit browser for test suite..")
        self.browser.quit()

    def test_guest_should_see_login_link(self):
        self.browser.get(link)
        self.browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self):
        self.browser.get(link)
        self.browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")


class TestMainPage2():

    def setup_method(self):
        print("start browser for test..")
        self.browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))

    def teardown_method(self):
        print("quit browser for test..")
        self.browser.quit()

    def test_guest_should_see_login_link(self):
        self.browser.get(link)
        self.browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self):
```

```
        self.browser.get(link)
        self.browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

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

collecting ... collected 4 items

test_fixture1.py::TestMainPage1::test_guest_should_see_login_link

test_fixture1.py::TestMainPage1::test_guest_should_see_basket_link_on_the_main_page

test_fixture1.py::TestMainPage2::test_guest_should_see_login_link

```
[vyafs-MacBook-Pro:section3 alekspog$ pytest -s test_fixture1.py
========== test session starts ==========
platform darwin -- Python 3.7.4, pytest-3.3.1, py-1.8.0, pluggy-0.6.0
rootdir: /Users/alekspog/PycharmProjects/course_solutions/section3, inifile:
collected 4 items

test_fixture1.py
start browser for test suite..
..quit browser for test suite..
start browser for test..
.quit browser for test..
start browser for test..
.                                                    [100%]quit browser for test.


========== 4 passed in 10.60 seconds ==========
vyafs-MacBook-Pro:section3 alekspog$
```

Мы видим, что в первом тест-сьюте браузер запустился один раз, а во втором — два раза.

Данные и кэш, оставшиеся от запуска предыдущего теста, могут влиять на результаты выполнения следующего теста, поэтому лучше всего запускать отдельный браузер для каждого теста, чтобы тесты были стабильнее. К тому же если вдруг браузер зависнет в одном тесте, то другие тесты не пострадают, если они запускаются каждый в собственном браузере.

Минусы запуска браузера на каждый тест: каждый запуск и закрытие браузера занимают время, поэтому тесты будут идти дольше. Возможно, вы захотите оптимизировать время прогона тестов, но лучше это делать с помощью других инструментов, которые мы разберём в дальнейшем.

Обычно такие фикстуры переезжают вместе с тестами, написанными с помощью unittest, и приходится их поддерживать, но сейчас все пишут более гибкие фикстуры @pytest.fixture, которые мы рассмотрим в следующем шаге.

★★★★★★★★★★★

Фикстуры, возвращающие значение

Мы рассмотрели базовый подход к созданию фикстур, когда тестовые данные задаются и очищаются в setup и teardown методах. PyTest предлагает продвинутый подход к фикстурам, когда фикстуры можно задавать глобально, передавать их в тестовые методы как параметры, а также имеет набор встроенных фикстур. Это более гибкий и удобный способ работы со вспомогательными функциями, и сейчас вы сами увидите почему.

Возвращаемое значение

Фикстуры могут возвращать значение, которое затем можно использовать в тестах. Давайте перепишем наш предыдущий пример с использованием PyTest фикстур. Мы создадим фикстуру browser, которая будет создавать объект WebDriver. Этот объект мы сможем использовать в тестах для взаимодействия с браузером. Для этого мы напишем метод browser и укажем, что он является фикстурой с помощью декоратора @pytest.fixture. После этого мы можем вызывать фикстуру в тестах, передав ее как параметр. По умолчанию фикстура будет создаваться для каждого тестового метода, то есть для каждого теста запустится свой экземпляр браузера.

`pytest -s -v test_fixture2.py`

test_fixture2.py:

```
import pytest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service

link = "http://selenium1py.pythonanywhere.com/"

'''
example Фикстуры, возвращающие значение
'''


@pytest.fixture()
def browser():
  print("\nstart browser for test suite..")
  browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
  return browser


class TestMainPage1():
   # вызываем фикстуру в тесте, передав ее как параметр
```

```
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

★★★★★★★★★★★

Финализаторы — закрываем браузер

Вероятно, вы заметили, что мы не использовали в этом примере команду browser.quit().

Это привело к тому, что несколько окон браузера оставались открыты после окончания тестов, а закрылись только после завершения всех тестов.

Закрытие браузеров произошло благодаря встроенной фикстуре — сборщику мусора. Но если бы количество тестов насчитывало больше нескольких десятков, то открытые окна браузеров могли привести к тому, что оперативная память закончилась бы очень быстро.

Поэтому надо явно закрывать браузеры после каждого теста.

Для этого мы можем воспользоваться финализаторами.

Один из вариантов финализатора — использование ключевого слова Python: yield. После завершения теста, который вызывал фикстуру, выполнение фикстуры продолжится со строки, следующей за строкой со словом yield:

test_fixture3.py

```
import pytest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service

link = "http://selenium1py.pythonanywhere.com/"

'''
example Фикстуры, возвращающие значение
'''


@pytest.fixture()
```

```
def browser():

    print("\nstart browser for test suite..")
    browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    yield browser
    # этот код выполнится после завершения теста
    print("\nquit browser..")
    browser.quit()


class TestMainPage1():
    # вызываем фикстуру в тесте, передав ее как параметр
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

Есть альтернативный способ вызова teardown кода с помощью встроенной фикстуры request и ее метода addfinalizer. Можете изучить его сами по документации PyTest.

https://docs.pytest.org/en/latest/how-to/fixtures.html#adding-finalizers-directly

Рекомендуем также выносить очистку данных и памяти в фикстуру, вместо того чтобы писать это в шагах теста: финализатор выполнится даже в ситуации, когда тест упал с ошибкой.

★★★★★★★★★★★

Область видимости scope

Для фикстур можно задавать область покрытия фикстур. Допустимые значения: “function”, “class”, “module”, “session”. Соответственно, фикстура будет вызываться один раз для тестового метода, один раз для класса, один раз для модуля или один раз для всех тестов, запущенных в данной сессии.

Запустим все наши тесты из класса TestMainPage1 в одном браузере для экономии времени, задав scope="class" в фикстуре browser:

test_fixture4.py

```
import pytest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service

link = "http://selenium1py.pythonanywhere.com/"

'''
example: Область видимости scope
'''


@pytest.fixture(scope='class')
def browser():

    print("\nstart browser for test suite..")
    browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    yield browser
    # этот код выполнится после завершения теста
    print("\nquit browser..")
    browser.quit()


class TestMainPage1():
    # вызываем фикстуру в тесте, передав ее как параметр
    def test_guest_should_see_login_link(self, browser):
        print("start test1")
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")
        print("finish test1")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        print("start test2")
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
        print("finish test2")
```

Мы видим, что в данном примере браузер открылся один раз и тесты последовательно выполнились в этом браузере. Здесь мы проделали это в качестве примера, но мы крайне рекомендуем всё же запускать отдельный экземпляр браузера для каждого теста, чтобы повысить стабильность тестов.

Фикстуры, которые занимают много времени для запуска и ресурсов (обычно это работа с базами данных), можно вызывать и один раз за сессию запуска тестов.

Как работает yield
https://habr.com/ru/post/132554/

Олег Молчанов про yield https://www.youtube.com/watch?v=ZjaVrzOkpZk

★★★★★★★★★★★

Автоиспользование фикстур

При описании фикстуры можно указать дополнительный параметр autouse=True, который укажет, что фикстуру нужно запустить для каждого теста даже без явного вызова:

test_fixture_autouse.py

```
import pytest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service

link = "http://selenium1py.pythonanywhere.com/"

'''
example: Автоиспользование фикстур
'''


@pytest.fixture(scope='class')
def browser():
    print("\nstart browser for test suite..")
    browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    yield browser
    # этот код выполнится после завершения теста
    print("\nquit browser..")
    browser.quit()


@pytest.fixture(autouse=True)
def prepare_data():
    print()
    print("Preparing some critical data for every test")


class TestMainPage1():
    # вызываем фикстуру в тесте, передав ее как параметр
    def test_guest_should_see_login_link(self, browser):
        print("start test1")
```

```
    browser.get(link)
    browser.find_element(By.CSS_SELECTOR, "#login_link")
    print("finish test1")


def test_guest_should_see_basket_link_on_the_main_page(self, browser):
    print("start test2")
    browser.get(link)
    browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
    print("finish test2")
```

```
[vyafs-MacBook-Pro:section3 alekspog$ pytest -s test_fixture_autouse.py
============================= test session starts ==============================
platform darwin -- Python 3.7.4, pytest-3.3.1, py-1.8.0, pluggy-0.6.0
rootdir: /Users/alekspog/PycharmProjects/course_solutions/section3, inifile:
collected 2 items

test_fixture_autouse.py
preparing some critical data for every test

start browser for test..
.
quit browser..

preparing some critical data for every test

start browser for test..
.                                                                    [100%]
quit browser..


========================== 2 passed in 6.48 seconds ===========================
```

Попробуйте запустить этот код и увидите, что для каждого теста фикстура подготовки данных выполнилась без явного вызова. Нужно быть аккуратнее с этим параметром, потому что фикстура выполняется для всех тестов. Без явной необходимости автоиспользованием фикстур лучше не пользоваться.

Итог

Вспомогательные функции — это очень мощная штука, которая решает много проблем при работе с автотестами. Основной плюс в том, что их удобно использовать в любых тестах без дублирования лишнего кода.

Дополнительные материалы про фикстуры, которые мы настоятельно советуем почитать, приведены ниже:

https://habr.com/ru/company/yandex/blog/242795/

https://docs.pytest.org/en/stable/fixture.html

★★★★★★★★★★★

Задание: область видимости фикстур

У нас есть набор тестов, который использует несколько фикстур. Посчитайте, сколько смайликов будет напечатано при выполнении этого тестового класса?

```
import pytest

@pytest.fixture(scope="class")
def prepare_faces():
    print("^_^", "\n")
    yield
    print(":3", "\n")


@pytest.fixture()
def very_important_fixture():
    print(":)", "\n")


@pytest.fixture(autouse=True)
def print_smiling_faces():
    print(":-P", "\n")


class TestPrintSmilingFaces():
    def test_first_smiling_faces(self, prepare_faces, very_important_fixture):
        # какие-то проверки

    def test_second_smiling_faces(self, prepare_faces):
        # какие-то проверки
```

 ★★★★★★★★★★ 

3.5 PyTest — маркировка

https://stepik.org/lesson/236918/step/1?unit=209305

Маркировка тестов часть 1
- Маркировка тестов часть 2
- Пропуск тестов
- XFail: помечать тест как ожидаемо падающий
- Задание: пропуск тестов
- Задание: запуск тестов

Маркировка тестов часть 1

Когда тестов становится много, хорошо иметь способ разделять тесты не только по названиям, но также по каким-нибудь заданным нами категориям. Например, мы можем выбрать небольшое количество критичных тестов (smoke), которые нужно запускать на каждый коммит разработчиков, а остальные тесты обозначить как регрессионные (regression) и запускать их только перед релизом. Или у нас могут быть тесты, специфичные для конкретного браузера (internet explorer 11), и мы хотим запускать эти тесты только под данный браузер. Для выборочного запуска таких тестов в PyTest используется маркировка тестов или метки (marks). Для маркировки теста нужно написать декоратор вида @pytest.mark.mark_name, где mark_name — произвольная строка.

Давайте разделим тесты в одном из предыдущих примеров на smoke и regression.

test_fixture8.py:

```
import pytest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service

link = "http://selenium1py.pythonanywhere.com/"

'''
example: Маркировка тестов

'''


@pytest.fixture(scope='function')
def browser():
    print("\nstart browser for test suite..")
    browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
```

```
    yield browser
    # этот код выполнится после завершения теста
    print("\nquit browser..")
    browser.quit()


class TestMainPage1():

    @pytest.mark.smoke
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    @pytest.mark.regression
    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

Чтобы запустить тест с нужной маркировкой, нужно передать в командной строке параметр -m и нужную метку:

```
pytest -s -v -m smoke test_fixture8.py
```

Если всё сделано правильно, то должен запуститься только тест с маркировкой smoke.

При этом вы увидите warning, то есть предупреждение:

```
PytestUnknownMarkWarning: Unknown pytest.mark.smoke - is this a typo?  You can register custom marks to avoid this warning - for details, see https://docs.pytest.org/en/latest/mark.html

    PytestUnknownMarkWarning,
```

Это предупреждение появилось потому, что в последних версиях PyTest настоятельно рекомендуется регистрировать метки явно перед использованием. Это, например, позволяет избегать опечаток, когда вы можете ошибочно пометить ваш тест несуществующей меткой, и он будет пропускаться при прогоне тестов.

Как же регистрировать метки?

Создайте файл pytest.ini в корневой директории вашего тестового проекта и добавьте в файл следующие строки:

```
[pytest]

markers =

    smoke: marker for smoke tests

    regression: marker for regression tests
```

Текст после знака ":" является поясняющим — его можно не писать.

Снова запустите тесты:

```
pytest -s -v -m smoke test_fixture8.py
```

Теперь предупреждений быть не должно.

Так же можно маркировать целый тестовый класс. В этом случае маркировка будет применена ко всем тестовым методам, входящим в класс.

★★★★★★★★★★

Маркировка тестов часть 2

Инверсия

Чтобы запустить все тесты, не имеющие заданную маркировку, можно использовать инверсию. Для запуска всех тестов, не отмеченных как smoke, нужно выполнить команду:

```
pytest -s -v -m "not smoke" test_fixture8.py
```

Объединение тестов с разными маркировками

Для запуска тестов с разными метками можно использовать логическое ИЛИ. Запустим smoke и regression-тесты:

```
pytest -s -v -m "smoke or regression" test_fixture8.py
```

Выбор тестов, имеющих несколько маркировок

Предположим, у нас есть smoke-тесты, которые нужно запускать только для определенной операционной системы, например, для Windows 10. Зарегистрируем метку win10 в файле pytest.ini, а также добавим к одному из тестов эту метку.

pytest.ini:

```
[pytest]
addopts = --strict-markers
markers =
    smoke: marker for smoke tests
    regression: marker for regression tests
    win 10: only for win10 OS
```

test_fixture81.py:

```
import pytest
```

```
from selenium import webdriver
from selenium.webdriver.common.by import By
link = "http://selenium1py.pythonanywhere.com/"

@pytest.fixture(scope="function")
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    print("\nquit browser..")
    browser.quit()


class TestMainPage1:

    @pytest.mark.smoke
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    @pytest.mark.smoke
    @pytest.mark.win10
    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

Чтобы запустить только smoke-тесты для Windows 10, нужно использовать логическое И:

pytest -s -v -m "smoke and win10" test_fixture81.py

Должен выполнится тест test_guest_should_see_basket_link_on_the_main_page.

★★★★★★★★★★

Пропуск тестов

В PyTest есть стандартные метки, которые позволяют пропустить тест при сборе тестов для запуска (то есть не запускать тест) или запустить, но отметить особенным статусом тот тест, который ожидаемо упадёт из-за наличия бага, чтобы он не влиял на результаты прогона всех тестов. Эти метки не требуют дополнительного объявления в pytest.ini.

Пропустить тест

Итак, чтобы пропустить тест, его отмечают в коде как @pytest.mark.skip:

А еще есть марк `skipif` — пропуск запуска по условию.

```
import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By

link = "http://selenium1py.pythonanywhere.com/"

@pytest.fixture(scope="function")
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    print("\nquit browser..")
    browser.quit()
```

```
class TestMainPage1():


    @pytest.mark.skip

    def test_guest_should_see_login_link(self, browser):

        browser.get(link)

        browser.find_element(By.CSS_SELECTOR, "#login_link")


    def test_guest_should_see_basket_link_on_the_main_page(self, browser):

        browser.get(link)

        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

В результатах теста мы увидим, что один тест был пропущен, а другой успешно прошёл: "1 passed, 1 skipped".

```
import pytest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service

link = "http://selenium1py.pythonanywhere.com/"

'''
example: Пропустить тест
чтобы пропустить тест, его отмечают в коде как @pytest.mark.skip:
'''


@pytest.fixture(scope='class')
def browser():
    print("\nstart browser for test suite..")
    browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    yield browser
    # этот код выполнится после завершения теста
```

```
    print("\nquit browser..")
    browser.quit()


class TestMainPage1():

    @pytest.mark.skip
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")


    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

★★★★★★★★★★★

XFail: помечать тест как ожидаемо падающий

Отметить тест как падающий

Теперь добавим в наш тестовый класс тест, который проверяет наличие кнопки "Избранное":

```
def test_guest_should_see_search_button_on_the_main_page(self, browser):

    browser.get(link)

    browser.find_element(By.CSS_SELECTOR, "button.favorite")
```

Предположим, что такая кнопка должна быть, но из-за изменений в коде она пропала. Пока разработчики исправляют баг, мы хотим, чтобы результат прогона всех наших тестов был успешен, но падающий тест помечался соответствующим образом, чтобы про него не забыть.

Добавим маркировку @pytest.mark.xfail для падающего теста.

test_fixture10.py:

```
import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By


link = "http://selenium1py.pythonanywhere.com/"
```

```
@pytest.fixture(scope="function")
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    print("\nquit browser..")
    browser.quit()



class TestMainPage1():
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")

    @pytest.mark.xfail
    def test_guest_should_see_search_button_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "button.favorite")
```

Запустим наши тесты:

```
pytest -v test_fixture10.py
```

Наш упавший тест теперь отмечен как xfail, но результат прогона тестов помечен как успешный:

```
(venv) vyafs-MacBook-Pro:section3 alekspog$ pytest -v test_fixture10.py
============================= test session starts ==============================
platform darwin -- Python 3.7.4, pytest-5.1.1, py-1.8.0, pluggy-0.12.0 -- /Users/alekspog/PycharmProjects/course_so
lutions/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/alekspog/PycharmProjects/course_solutions, inifile: pytest.ini
collected 3 items

test_fixture10.py::TestMainPage1::test_guest_should_see_login_link PASSED        [ 33%]
test_fixture10.py::TestMainPage1::test_guest_should_see_basket_link_on_the_main_page PASSED [ 66%]
test_fixture10.py::TestMainPage1::test_guest_should_see_search_button_on_the_main_page XFAIL [100%]

========================== 2 passed, 1 xfailed in 11.11s ===========================
(venv) vyafs-MacBook-Pro:section3 alekspog$
```

Когда баг починят, мы это узнаем, так как теперь тест будет отмечен как XPASS (“unexpectedly passing” — неожиданно проходит).

После этого маркировку xfail для теста можно удалить.

Кстати, к маркировке xfail можно добавлять параметр reason. Чтобы увидеть это сообщение в консоли, при запуске нужно добавлять параметр pytest -rx.

test_fixture10a.py:

```
import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By


link = http://selenium1py.pythonanywhere.com/


@pytest.fixture(scope="function")

def browser():

    print("\nstart browser for test..")

    browser = webdriver.Chrome()

    yield browser

    print("\nquit browser..")

    browser.quit()
```

```
class TestMainPage1():


    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")


    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")


    @pytest.mark.xfail(reason="fixing this bug right now")
    def test_guest_should_see_search_button_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "button.favorite")
```

Запустим наши тесты:

```
$ pytest -rx -v test_fixture10a.py
```

Сравните вывод в первом и во втором случае.

```
(venv) vyafs-MacBook-Pro:section3 alekspog$ pytest -rx -v test_fixture10a.py
============================= test session starts ==============================
platform darwin -- Python 3.7.4, pytest-5.1.1, py-1.8.0, pluggy-0.12.0 -- /Users/alekspog/PycharmProjects/course_so
lutions/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/alekspog/PycharmProjects/course_solutions, inifile: pytest.ini
collected 3 items

test_fixture10a.py::TestMainPage1::test_guest_should_see_login_link PASSED                [ 33%]
test_fixture10a.py::TestMainPage1::test_guest_should_see_basket_link_on_the_main_page PASSED [ 66%]
test_fixture10a.py::TestMainPage1::test_guest_should_see_search_button_on_the_main_page XFAIL [100%]

=========================== short test summary info ============================
XFAIL test_fixture10a.py::TestMainPage1::test_guest_should_see_search_button_on_the_main_page
  fixing this bug right now
==================== 2 passed, 1 xfailed in 9.87s ======================
(venv) vyafs-MacBook-Pro:section3 alekspog$
```

XPASS-тесты

Поменяем селектор в последнем тесте, чтобы тест начал проходить.

test_fixture10b.py:

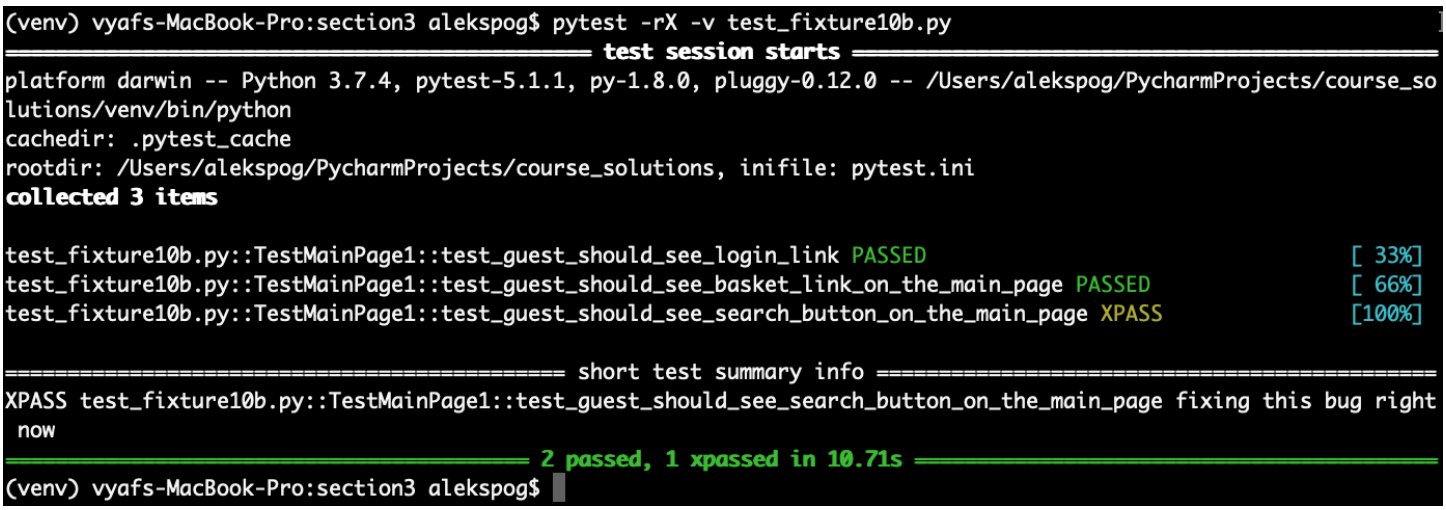
```
import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By

link = "http://selenium1py.pythonanywhere.com/"

@pytest.fixture(scope="function")
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    print("\nquit browser..")
    browser.quit()

class TestMainPage1():
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

```
    @pytest.mark.xfail(reason="fixing this bug right now")
    def test_guest_should_see_search_button_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "input.btn.btn-default")
```

Запустите тесты. Здесь мы добавили символ X в параметр -r, чтобы получить подробную информацию по XPASS-тестам:

```
$ pytest -rX -v test_fixture10b.py
```

И изучите отчёт:



Дополнительно об использовании этих меток можно почитать в документации: [Skip and xfail: dealing with tests that cannot succeed](). Там есть много разных интересных особенностей, например, как пропускать тест только при выполнении условия, как сделать так, чтобы внезапно прошедший xfailed тест в отчете стал красным, и так далее.

-v - режим verbous (многословный). Детально расскажет о прохождении.

-rx - report on XFAIL (отчитаться о наличии метки XFAIL). В целом, даже без (remark = "") покажет в каком тесте была метка.

Увидела у коллег, что на практике практике в причину обычно пишут еще и ссылку на тикет, что бы можно было перейти и посмотреть пофиксили баг или еще нет. Так просто и так гениально

```
@pytest.mark.xfail(reason="вот тут ссылка на Тикет в джире или другом таск трекере")
```

нужно ли добавлять skip и xfail в список pytest.ini или это забронированные значения?

нет, не нужно.

🏅 ★★★★★★★★★★ 🏅

3.6 PyTest — параметризация, конфигурирование, плагины

Содержание урока

★★★★★★★★★★

Conftest.py — конфигурация тестов

Ранее мы добавили фикстуру browser, которая создает нам экземпляр браузера для тестов в данном файле. Когда файлов с тестами становится больше одного, приходится в каждом файле с тестами описывать данную фикстуру. Это очень неудобно.

Для хранения часто употребимых фикстур и хранения глобальных настроек нужно использовать файл conftest.py, который должен лежать в директории верхнего уровня в вашем проекте с тестами.

Можно создавать дополнительные файлы conftest.py в других директориях, но тогда настройки в этих файлах будут применяться только к тестам в под-директориях.

Создадим файл conftest.py в корневом каталоге нашего тестового проекта и перенесем туда фикстуру browser.

Заметьте, насколько лаконичнее стал выглядеть файл с тестами.

conftest.py:

```
import pytest
```

```
from selenium import webdriver

from selenium.webdriver.common.by import By



@pytest.fixture(scope="function")

def browser():
    print("\nstart browser for test suite..")
    browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    yield browser
    # этот код выполнится после завершения теста
    print("\nquit browser..")
    browser.quit()
```

Теперь, сколько бы файлов с тестами мы ни создали, у тестов будет доступ к фикстуре browser.

Фикстура передается в тестовый метод в качестве аргумента. Таким образом можно удобно переиспользовать одни и те же вспомогательные функции в разных частях проекта.

test_conftest.py:

```
from selenium.webdriver.common.by import By

link = "http://selenium1py.pythonanywhere.com/"



def test_guest_should_see_login_link(browser):

    browser.get(link)

    browser.find_element(By.CSS_SELECTOR, "#login_link")
```

ОЧЕНЬ ВАЖНО!

Есть одна важная особенность поведения конфигурационных файлов, о которой вы обязательно должны знать. PyTest автоматически находит и подгружает файлы conftest.py, которые находятся в директории с тестами. Если вы храните все свои скрипты для курса в одной директории, будьте аккуратны и следите, чтобы не возникало ситуации, когда вы запускаете тесты из папки tests:

```
tests/

├── conftest.py

├── subfolder
```

```
│   └── conftest.py
│   └── test_abs.py
```

следует избегать!

В таком случае применяется ОБА файла conftest.py, что может вести к непредсказуемым ошибкам и конфликтам.

Таким образом можно переопределять разные фикстуры, но мы в рамках курса рекомендуем придерживаться одного файла на проект/задачу и держать их горизонтально, как-нибудь так:

```
selenium_course_solutions/
├── section3
│   └── conftest.py
│   └── test_languages.py
├── section4
│   └── conftest.py
│   └── test_main_page.py
```

правильно!

Будьте внимательны и следите, чтобы не было разных conftest во вложенных друг в друга директориях, особенно, когда будете скачивать и проверять задания сокурсников.

Override a fixture on a folder (conftest) level

https://docs.pytest.org/en/7.1.x/how-to/fixtures.html?highlight=fixture%20folder#override-a-fixture-on-a-folder-conftest-level

★★★★★★★★★★★

Параметризация тестов

PyTest позволяет запустить один и тот же тест с разными входными параметрами. Для этого используется декоратор @pytest.mark.parametrize(). Наш сайт доступен для разных языков. Напишем тест, который проверит, что для сайта с русским и английским языком будет

отображаться ссылка на форму логина. Передадим в наш тест ссылки на русскую и английскую версию главной страницы сайта.

В @pytest.mark.parametrize() нужно передать параметр, который должен изменяться, и список значений параметра. В самом тесте наш параметр тоже нужно передавать в качестве аргумента. Обратите внимание, что внутри декоратора имя параметра оборачивается в кавычки, а в списке аргументов теста кавычки не нужны.

test_fixture7.py:

```
import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By


@pytest.fixture(scope="function")

def browser():

    print("\nstart browser for test..")

    browser = webdriver.Chrome()

    yield browser

    print("\nquit browser..")

    browser.quit()


@pytest.mark.parametrize('language', ["ru", "en-gb"])

def test_guest_should_see_login_link(browser, language):

    link = f"http://selenium1py.pythonanywhere.com/{language}/"

    browser.get(link)

    browser.find_element(By.CSS_SELECTOR, "#login_link")
```

*****

```
import pytest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
```

```
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service

link = "http://selenium1py.pythonanywhere.com/"

'''
example: Параметризация тестов
'''


@pytest.fixture(scope='class')
def browser():
    print("\nstart browser for test suite..")
    browser = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    yield browser
    # этот код выполнится после завершения теста
    print("\nquit browser..")
    browser.quit()


@pytest.mark.parametrize('language', ["ru", "en-gb"])
def test_guest_should_see_login_link(browser, language):
    link = f"http://selenium1py.pythonanywhere.com/{language}/"
    browser.get(link)
    browser.find_element(By.CSS_SELECTOR, "#login_link")
```

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

collecting ... collected 2 items


test_fixture9_parametrize.py::test_guest_should_see_login_link[ru]

test_fixture9_parametrize.py::test_guest_should_see_login_link[en-gb]
```

Запустите тест:

```
pytest -s -v test_fixture7.py
```

Вы увидите, что запустятся два теста.

В названии каждого теста в квадратных скобках будет написан параметр, с которым он был запущен.

Таким образом мы можем быстро и без дублирования кода увеличить количество проверок для похожих сценариев.

```
(venv) vyafs-MacBook-Pro:section3 alekspog$ pytest -s -v test_fixture7.py
============================= test session starts ==============================
platform darwin -- Python 3.7.4, pytest-5.1.1, py-1.8.0, pluggy-0.12.0 -- /Users/alekspog/PycharmProjects/course_so
lutions/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/alekspog/PycharmProjects/course_solutions, inifile: pytest.ini
collected 2 items

test_fixture7.py::test_guest_should_see_login_link[http:/selenium1py.pythonanywhere.com/ru/]
start browser for test..
PASSED
quit browser..

test_fixture7.py::test_guest_should_see_login_link[http:/selenium1py.pythonanywhere.com/en-gb/]
start browser for test..
PASSED
quit browser..


============================== 2 passed in 6.70s ===============================
(venv) vyafs-MacBook-Pro:section3 alekspog$
```

Можно задавать параметризацию также для всего тестового класса, чтобы все тесты в классе запустились с заданными параметрами.

В таком случае отметка о параметризации должна быть перед объявлением класса:

```
@pytest.mark.parametrize('language', ["ru", "en-gb"])
class TestLogin:
    def test_guest_should_see_login_link(self, browser, language):
        link = f"http://selenium1py.pythonanywhere.com/{language}/"
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")
        # этот тест запустится 2 раза

    def test_guest_should_see_navbar_element(self, browser, language):
        # этот тест тоже запустится дважды
```

Дополнительно, полезный туториал из документации: [Parametrizing fixtures and test functions]

★★★★★★★★★★

Conftest.py и передача параметров в командной строке

Встроенная фикстура request может получать данные о текущем запущенном тесте, что позволяет, например, сохранять дополнительные данные в отчёт, а также делать многие другие интересные вещи. В этом шаге мы хотим показать, как можно настраивать тестовые окружения с помощью передачи параметров через командную строку.

Это делается с помощью встроенной функции pytest_addoption и фикстуры request.

Сначала добавляем в файле conftest обработчик опции в функции pytest_addoption, затем напишем фикстуру, которая будет обрабатывать переданные в опции данные.

Подробнее можно ознакомиться

здесь: https://docs.pytest.org/en/latest/example/simple.html?highlight=addoption

Добавим логику обработки командной строки в conftest.py.

Для запроса значения параметра мы можем вызвать команду:

```
browser_name = request.config.getoption("browser_name")
```

conftest.py:

```
import pytest

from selenium import webdriver


def pytest_addoption(parser):
    parser.addoption('--browser_name', action='store', default=None,
                     help="Choose browser: chrome or firefox")



@pytest.fixture(scope="function")
def browser(request):
    browser_name = request.config.getoption("browser_name")
    browser = None
```

```
    if browser_name == "chrome":
        print("\nstart chrome browser for test..")
        browser = webdriver.Chrome()
    elif browser_name == "firefox":
        print("\nstart firefox browser for test..")
        browser = webdriver.Firefox()
    else:
        raise pytest.UsageError("--browser_name should be chrome or firefox")
    yield browser
    print("\nquit browser..")
    browser.quit()
```

test_parser.py:

```
link = "http://selenium1py.pythonanywhere.com/"



def test_guest_should_see_login_link(browser):
    browser.get(link)
    browser.find_element(By.CSS_SELECTOR, "#login_link")
```

Если вы теперь запустите тесты без параметра, то получите ошибку:

```
pytest -s -v test_parser.py
```

```
_pytest.config.UsageError: --browser_name should be chrome or firefox
```

Можно задать значение параметра по умолчанию, чтобы в командной строке не обязательно было указывать параметр --browser_name, например, так:

```
parser.addoption('--browser_name', action='store', default="chrome",
                 help="Choose browser: chrome or firefox")
```

Давайте укажем параметр:

```
pytest -s -v --browser_name=chrome test_parser.py
```

А теперь запустим тесты на Firefox:

```
pytest -s -v --browser_name=firefox test_parser.py
```

Вы должны увидеть, как сначала тесты запустятся в браузере Chrome, а затем — в Firefox.

★★★★★★★★★★★

Плагины и перезапуск тестов

Для PyTest написано большое количество плагинов, то есть дополнительных модулей, которые расширяют возможности этого фреймворка. Полный список доступных плагинов доступен здесь.

Plugin List =>

https://docs.pytest.org/en/latest/reference/plugin_list.html

Рассмотрим еще одну проблему, с которой вы обязательно столкнетесь, когда будете писать end-to-end тесты на Selenium. Flaky-тесты или "мигающие" авто-тесты, т.е. такие тесты, которые по независящим от нас внешним обстоятельствам или из-за трудновоспроизводимых багов, могут иногда падать, хотя всё остальное время они проходят успешно. Это может происходить в момент прохождения тестов из-за одновременного обновления сайта, из-за сетевых проблем или странных стечений обстоятельств. Конечно, надо стараться исключать такие проблемы и искать причины возникновения багов, но в реальном мире бывает, что это требует слишком много усилий. Поэтому мы будем перезапускать упавший тест, чтобы еще раз убедиться, что он действительно нашел баг, а не упал случайно.

Это сделать очень просто. Для этого мы будем использовать плагин pytest-rerunfailures.

Сначала установим плагин в нашем виртуальном окружении. После установки плагин будет автоматически найден PyTest, и можно будет пользоваться его функциональностью без дополнительных изменений кода:

```
pip install pytest-rerunfailures
```

Чтобы указать количество перезапусков для каждого из упавших тестов, нужно добавить в командную строку параметр:

"--reruns n", где n — это количество перезапусков.

Если при повторных запусках тесты пройдут успешно, то и прогон тестов будет считаться успешным.

Количество перезапусков отображается в отчёте, благодаря чему можно позже анализировать проблемные тесты.

Дополнительно мы указали параметр "--tb=line", чтобы сократить лог с результатами теста. Можете почитать подробнее про настройку вывода в документации PyTest:

```
pytest -v --tb=line --reruns 1 --browser_name=chrome test_rerun.py
```

Давайте напишем два теста: один из них будет проходить, а другой — нет. Посмотрим, как выглядит перезапуск.

test_rerun.py:

```
link = "http://selenium1py.pythonanywhere.com/"


def test_guest_should_see_login_link_pass(browser):
    browser.get(link)
    browser.find_element(By.CSS_SELECTOR, "#login_link")


def test_guest_should_see_login_link_fail(browser):
```

```
    browser.get(link)

    browser.find_element(By.CSS_SELECTOR, "#magic_link")
```

Мы увидим сообщение: "1 failed, 1 passed, 1 rerun in 9.20s", то есть упавший тест был перезапущен, но при втором запуске тоже упал.

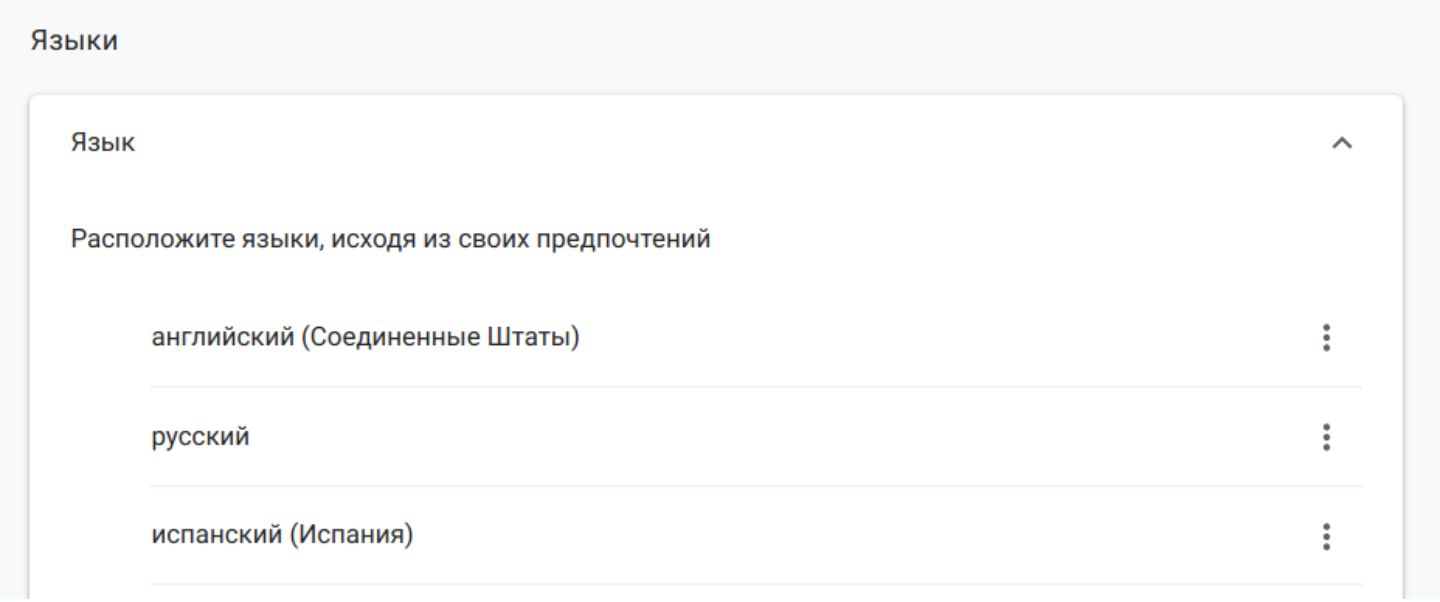
Если бы во второй раз мигающий тест все-таки прошёл успешно, то мы бы увидели сообщение: "2 passed, 1 rerun in 9.20s", и итоговый результат запуска всех тестов считался бы успешным.

★★★★★★★★★★★

Запуск автотестов для разных языков интерфейса

Цель: научиться запускать автотесты для разных локалей, т.е. для разных языков интерфейсов.

Мы уже запускали автотесты для разных языков в одном из предыдущих шагов, используя параметризацию с помощью разных ссылок, но такой подход сложно масштабировать на большое количество тестов. Давайте сделаем так, чтобы сервер сам решал, какой язык интерфейса нужно отобразить, основываясь на данных браузера. Браузер передает данные о языке пользователя через запросы к серверу, указывая в Headers (заголовке запроса) параметр accept-language. Если сервер получит запрос с заголовком {accept-language: ru, en}, то он отобразит пользователю русскоязычный интерфейс сайта. Если русский язык не поддерживается, то будет показан следующий язык из списка, в данном случае пользователь увидит англоязычный интерфейс. Это, кстати, примерно то же самое, что и выставить предпочтительный язык в настройках своего браузера:



Чтобы указать язык браузера с помощью WebDriver, используйте класс Options и метод add_experimental_option, как указано в примере ниже:

```
from selenium.webdriver.chrome.options import Options


options = Options()

options.add_experimental_option('prefs', {'intl.accept_languages': user_language})

browser = webdriver.Chrome(options=options)
```

Для Firefox объявление нужного языка будет выглядеть немного иначе:

```
fp = webdriver.FirefoxProfile()

fp.set_preference("intl.accept_languages", user_language)

browser = webdriver.Firefox(firefox_profile=fp)
```

В конструктор webdriver.Chrome или webdriver.Firefox вы можете добавлять разные аргументы, расширяя возможности тестирования ваших веб-приложений: можно указывать прокси-сервер для контроля сетевого трафика или запускать разные версии браузера, указывая локальный путь к файлу браузера. Предполагаем, что эти возможности вам понадобятся позже и вы сами сможете найти настройки для этих задач.