**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}/"

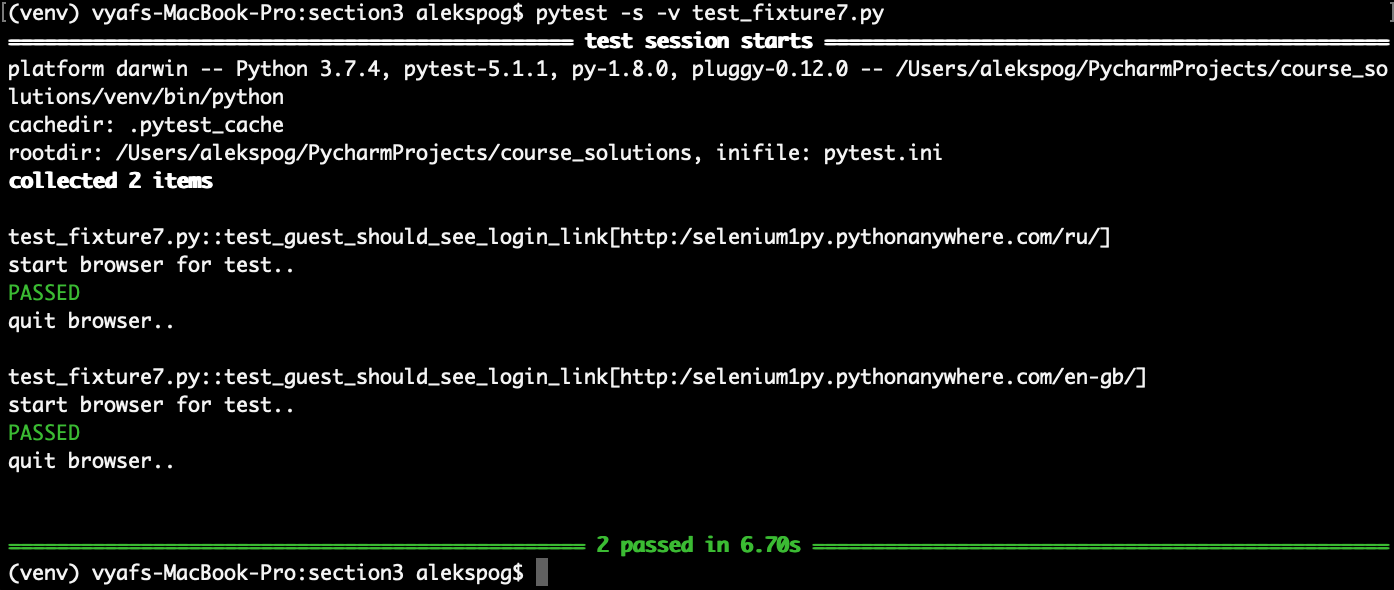
browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, "#login\_link")

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.



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

## 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..")

browser = webdriver.Chrome()

yield browser

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:**

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.

## 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](https://docs.pytest.org/en/latest/explanation/flaky.html?highlight=plugins#plugins), то есть ﻿дополнительных модулей, которые расширяют возможности этого фреймворка. Полный список доступных плагинов доступен [здесь](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](https://docs.pytest.org/en/stable/usage.html#modifying-python-traceback-printing):

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)

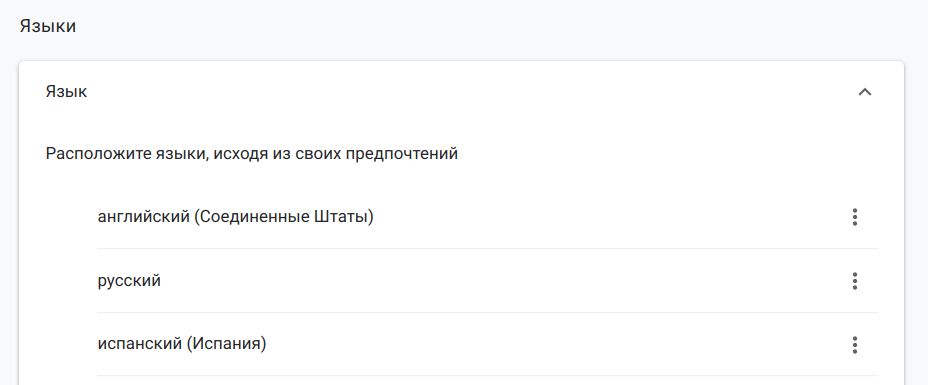
browser.find\_element(By.CSS\_SELECTOR, "#magic\_link")

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](https://stepik.org/lesson/237240/step/2), 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.