**Stepik**

**Автоматизация тестирования с помощью Selenium и Python**

**1 Модуль – Знакомимся с Selenium**

## Commenting and uncommenting lines of code﻿ PyCharm

* Press Ctrl+/

Установка Python в Windows

Во время установки убедитесь в том, что вы поставили галочку в разделе **Add Python 3.x to PATH**. В этом случае вызов интерпретатора Python будет доступен вам из командной строки.

Чтобы проверить правильность установки, откройте командную строку Windows (Пуск - Найти программу - cmd.exe и Запустить). В консоли введите **python --version**

**Внимание!**У консоли Windows специфичные отношения с операциями копировать/вставить. Чтобы разрешить выделение текста мышкой, нужно сделать следующие настройки:

* Правой кнопкой нажать на заголовок окна, и выбрать в меню пункт "Настройки" (Settings)
* В настройках поставить галочку "Разрешить выделение мышью" (QuickEdit Mode). Найти настройку вам также поможет скриншот приведенный ниже.
* Сохранить настройки.

Теперь вы сможете выбрать текст в консоли, зажав левую кнопку мыши. После выделения текста и нажатия Enter текст окажется в буфере обмена. Чтобы вставить текст в консоль, нужно нажать правую кнопку мыши.

Создадим папку, где будут храниться наши виртуальные окружения, и перейдем в неё:

mkdir environments

cd environments

Создадим виртуальное окружение:

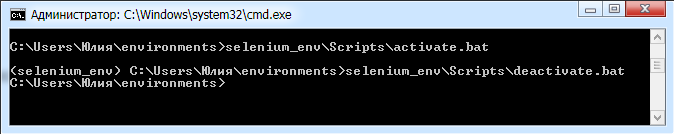
python -m venv selenium\_env

Запустим созданный для нас приложением venv файл activate.bat, чтобы активировать окружение:

selenium\_env\Scripts\activate.bat

Если окружение активировано, то мы увидим в начале командной строки терминала название окружения в круглых скобках. Теперь мы можем устанавливать нужные нам пакеты и запускать скрипты для тестов, которые мы напишем:

Если понадобится выйти из нашего окружения, достаточно выполнить команду deactivate:



Обратите внимание, что после перезапуска терминала необходимо снова активировать нужное окружение.

Проверим, что мы можем теперь использовать Python. Запустим интерпретатор Python и напишем собственную версию HelloWorld:

> selenium\_env\Scripts\activate.bat

(selenium\_env) С:\Users\Юлия\environments> python

>>> print("Hello, Selenium!")

В итоге вы должны увидеть вывод строки "Hello, Selenium! "

Выйдем из интерпретатора:

>>> exit()

Links:

# How to set the path and environment variables in Windows

<https://www.computerhope.com/issues/ch000549.htm>

# что такое переменная среды PATH?

<http://barancev.github.io/what-is-path-env-var/>

Установка Python в Ubunty

Если вы пользуетесь Ubuntu, то обычно Python2 и Python3 у вас уже установлены. Чтобы проверить, какая версия Python3 у вас в системе, откройте терминал и введите команду:

python3 –version

**Установка Python3.7**

Выполните в терминале:

sudo apt-get update && sudo apt-get upgrade

sudo apt-get install python3.7

Python будет установлен в папку /usr/bin/python3.7. Теперь вы можете запускать интерпретатор python командой python3.7, а также посмотреть более детально версию python:

python3.7 --version

Если вы хотите, чтобы данная версия Python была доступна по более короткой команде python3, то можно прописать alias, то есть короткое имя, следующим образом.

Открыть файл с настройками конфигураций для сеанса терминала .bashrc:

nano ~/.bashrc

Добавить в начало файла строчку:

alias python3=python3.7

Сохранить файл ctrl+o и закрыть его ctrl+x. В командной строке применить настройки:

source ~/.bashrc

Теперь данный alias должен стать постоянным.

**Создание виртуального окружения**

Для курса нам понадобятся дополнительные пакеты Python, для их дальнейшего использования установим систему управления пакетами pip:

python3 -m pip install pip

Чтобы не засорять наше основное Python-окружение, мы будем работать в виртуальном окружении, которое в дальнейшем можно будет легко удалить или изменить под ваши нужды. Установим модуль venv для управления виртуальными окружениями:

sudo apt-get install -y python3.7-venv

Установка Python в macOS

Чтобы проверить установлен ли у вас Python3 в системе, в консоли введите команду python3 --version:

python3 --version

Python 3.7.3

Для установки Python3 в macOS воспользуйтесь инструкцией [https://wsvincent.com/install-python3-mac/](https://wsvincent.com/install-python3-mac/" \t "_blank). Рекомендуем установить пакетный менеджер Homebrew, как указано в инструкции. С помощью него потом будет удобно устанавливать другие инструменты, нужные для разработки автотестов.

## Установка Selenium для Python

В нашем виртуальном окружении установим библиотеку Selenium. Проверьте, что вы указали для установки именно версию 3.14.0. Проект Selenium развивается очень быстро и в других версиях могут быть расхождения в наборе методов Selenium и некоторые примеры у вас могут не работать:

pip install selenium==3.14.0

Проверим, что библиотека действительно установлена:

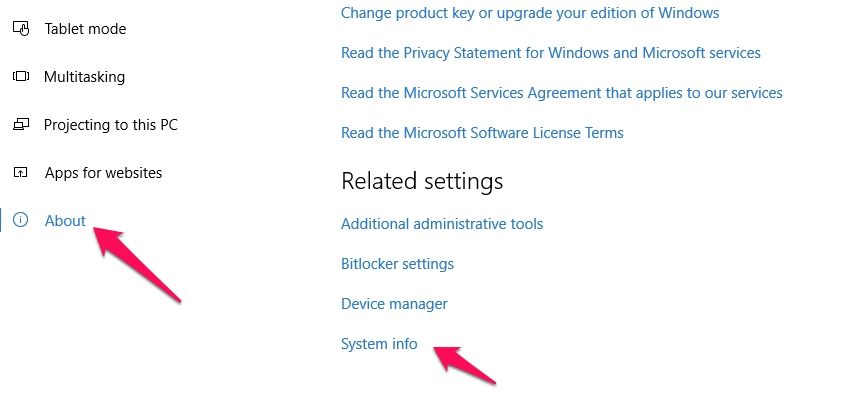
pip list

Чтобы узнать версию браузера, откройте новое окно в Chrome, в поисковой строке наберите: chrome://version/ и нажмите Enter. В верхней строчке вы увидите информацию про версию браузера.

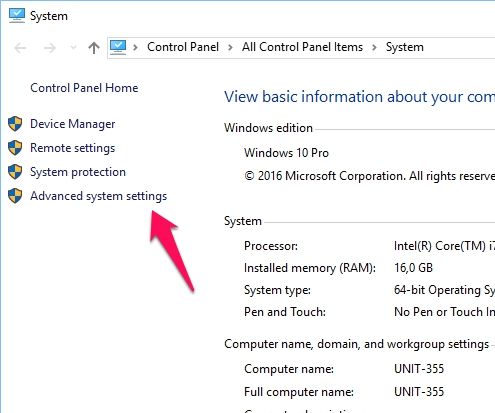
**Пример: как добавить путь в системную переменную PATH на Windows10**

1. Откройте настройки системы.

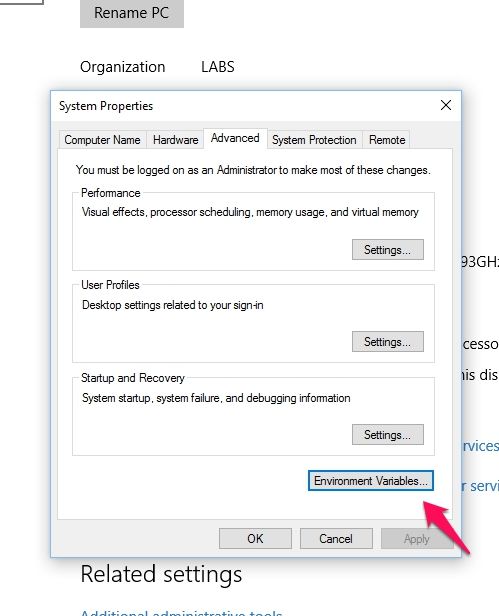
2. В настройках откройте вкладку About, затем System info:



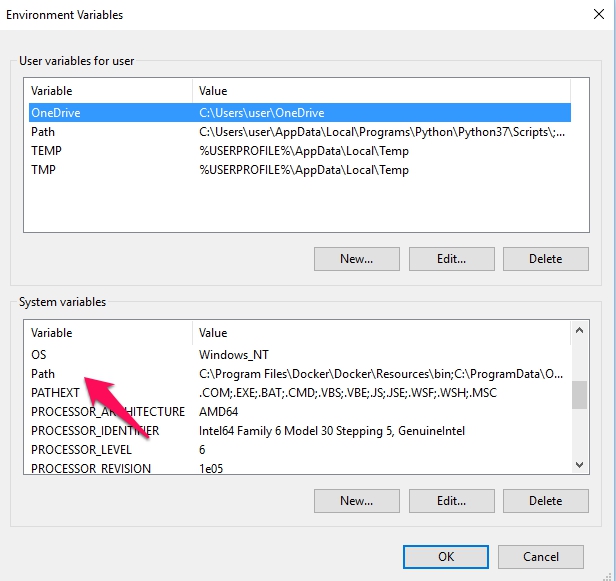
3. Выберите Advanced system settings:



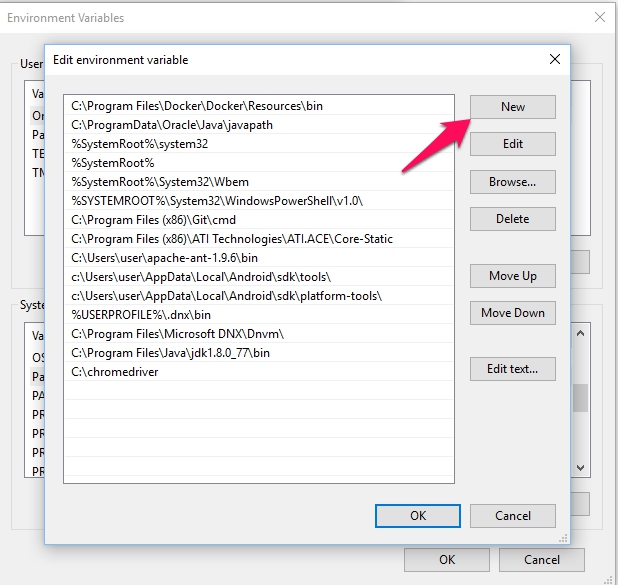
4. Выберите Environment Variables:



5. Кликните два раза на строчке Path в System variables:



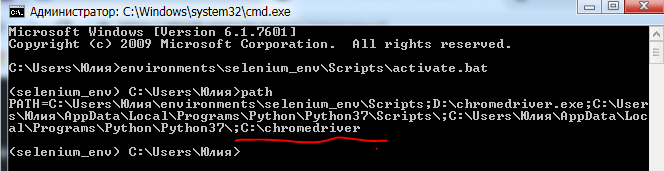
6. Нажмите кнопку New. Введите в новую строку путь к ChromeDriver — C:\chromedriver. Нажмите Enter. У вас должна появится строка с указанным путем:



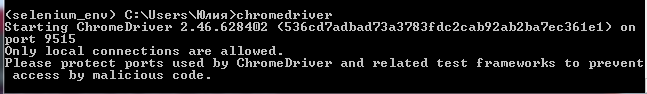
7. Если у вас была открыта командная строка Windows, не забудьте ее закрыть. Затем откройте новую командную строку, чтобы изменения переменной окружения стали доступны. Активируйте снова виртуальное окружение selenium\_env, которое мы создали в предыдущих шагах.

Давайте убедимся в том, что вебдрайвер установлен правильно.

Для начала проверим содержимое переменной path, для этого наберем в командной строке **Path:**



Ура, там есть папка с chromedriver! Попробуем вызвать его напрямую из командной строки:



Магия переменной path: хотя программа chromedriver находится где-то в другом каталоге, мы можем напрямую открывать её, используя имя chromedriver. Чтобы завершить процесс в консоли, нажмите Ctrl+C.

## Установка драйвера для браузера:Linux

Давайте установим и настроим ChromeDriver с помощью команд в терминале. Укажем нужную нам версию ChromeDriver для загрузки. Для получения ссылки перейдите в браузере на нужную вам версию драйвера по ссылке на [https://sites.google.com/a/chromium.org/chromedriver/downloads](https://sites.google.com/a/chromium.org/chromedriver/downloads" \t "_blank). На открывшейся странице нажмите на файле для Linux правой кнопкой и скопируйте путь к файлу. Замените в примере ниже путь к файлу для команды wget вашей ссылкой:

wget https://chromedriver.storage.googleapis.com/76.0.3809.126/chromedriver\_linux64.zip

unzip chromedriver\_linux64.zip

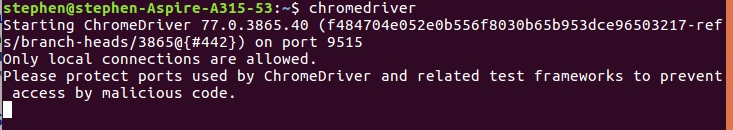
Переместите разархивированный файл с СhromeDriver в нужную папку и разрешите запускать chromedriver как исполняемый файл:

sudo mv chromedriver /usr/local/bin/chromedriver

sudo chown root:root /usr/local/bin/chromedriver

sudo chmod +x /usr/local/bin/chromedriver

Проверьте, что chromedriver доступен, выполнив команду chromedriver в терминале, вы должны получить сообщение о том, что процесс успешно запущен:



## Установка драйвера для браузера: macOS

При установке Python вы уже, скорее всего, установили пакетный менеджер Homebrew. Если нет, то рекомендуем сделать это сейчас, а затем с его помощью установить программу wget для загрузки файлов по сети.

brew install wget

Для установки драйвера откройте сайт [https://sites.google.com/a/chromium.org/chromedriver/downloads](https://sites.google.com/a/chromium.org/chromedriver/downloads" \t "_blank) и скопируйте ссылку на ту версию ChromeDriver, которая соответствует версии вашего браузера. Чтобы узнать версию браузера, откройте новое окно в Chrome, в поисковой строке наберите: chrome://version/ — и нажмите Enter. В верхней строчке вы увидите информацию про версию браузера.

cd ~/Downloads

wget https://chromedriver.storage.googleapis.com/76.0.3809.68/chromedriver\_mac64.zip

Разархивируйте скачанный файл и переместите его в папку /usr/local/bin, чтобы он был глобально доступен в вашей системе.

unzip chromedriver\_mac64.zip

sudo mv chromedriver /usr/local/bin

Проверим, что нужная версия chromedriver установлена.

chromedriver --version

Мы должны увидеть ответ системы:

ChromeDriver 76.0.3809.68 (420c9498db8ce8fcd190a954d51297672c1515d5-refs/branch-heads/3809@{#864})

В этом случае все хорошо, и можно переходить к следующему шагу.

Если мы вдруг видим что-то вроде:

-bash: chromedriver: command not found

то необходимо проверить папку /usr/local/bin на наличие файла chromedriver. Если файла там нет, то нужно повторить команды, описанные выше.

## Запуск браузера и первый скрипт

### ****Для всех ОС:****

В нашем виртуальном окружении запустим интерпретатор python:

(selenium\_env) alekspog@xenial:~/environments$ python

Теперь напишем скрипт, который откроет следующий шаг в данном уроке на Stepik и отправит правильное решение в задаче. Рекомендуем выполнять команды последовательно по одной, чтобы лучше понять, что происходит в каждой строчке. Не пугайтесь, если не понимаете, что делают те или иные команды, — подробнее мы их разберем в следующих уроках.

import time

# webdriver это и есть набор команд для управления браузером

from selenium import webdriver

# инициализируем драйвер браузера. После этой команды вы должны увидеть новое открытое окно браузера

driver = webdriver.Chrome()

# команда time.sleep устанавливает паузу в 5 секунд, чтобы мы успели увидеть, что происходит в браузере

time.sleep(5)

# Метод get сообщает браузеру, что нужно открыть сайт по указанной ссылке

driver.get("https://stepik.org/lesson/25969/step/12")

time.sleep(5)

# Метод find\_element\_by\_css\_selector позволяет найти нужный элемент на сайте, указав путь к нему. Способы поиска элементов мы обсудим позже

# Ищем поле для ввода текста

textarea = driver.find\_element\_by\_css\_selector(".textarea")

# Напишем текст ответа в найденное поле

textarea.send\_keys("get()")

time.sleep(5)

# Найдем кнопку, которая отправляет введенное решение

submit\_button = driver.find\_element\_by\_css\_selector(".submit-submission")

# Скажем драйверу, что нужно нажать на кнопку. После этой команды мы должны увидеть сообщение о правильном ответе

submit\_button.click()

time.sleep(5)

# После выполнения всех действий мы должны не забыть закрыть окно браузера

driver.quit()

Вы можете запустить этот же скрипт из файла. Скачайте файл [get\_method.py](https://stepik.org/media/attachments/lesson/222929/get_method.py) к себе в папку и запустите его. Для этого создадим папку selenium\_course и скопируем в нее скачанный файл. Затем запустим файл в нашем виртуальном окружении.

**Пример для Windows**

(Вместо user везде подставляйте имя пользователя в системе)

При выполнении следующих команд не забудьте изменить путь к вашей пользовательской папке вместо C:\Users\user и к папке с загруженным файлом.

Выполните следующие команды в командной строке Windows:

c:\Users\DEV> mkdir selenium\_course

C:\Users\DEV> copy C:\Users\DEV\Downloads\get\_method.py c:\Users\DEV\selenium\_course

C:\Users\DEV> environments\selenium\_env\Scripts\activate.bat

(selenium\_env) C:\Users\DEV>python c:\Users\DEV\selenium\_course\get\_method.py

**Пример для Ubuntu и macOS**

Выполните следующие команды в консоли:

mkdir ~/selenium\_course

mv ~/Downloads/get\_method.py ~/selenium\_course

python ~/selenium\_course/get\_method.py

Структура Веб страницы

**Атрибут name**

Свойство **name**, например, используется для задания якоря (закладки) в html-странице. Этот атрибут тоже можно использовать в качестве селектора для поиска элемента:

<a name="top"></a>

<p>Длинный текст</p>

...

<p><a href="#top">Наверх</a></p>

Поиск элементов

Поиск по значению атрибута

**[value="Cat memes"]**.

Поиск по class

Давайте рассмотрим важную разницу между двумя способами поиска по классу. Допустим, у элемента **article** задано больше одного класса, как на странице [http://suninjuly.github.io/cats.html](http://suninjuly.github.io/cats.html" \t "_blank):

<article id="moto" class="lead text-muted" title="one-thing" name="moto">If there's one thing that the internet was made for, it's funny cat memes.</article>

Вариант **[class="lead"]** не найдет нам этот элемент, так как он ищет по точному совпадению. Чтобы найти элемент, нам нужно будет написать **[class="lead text-muted"]**, порядок классов при этом важен. **[class="text-muted lead"]** — уже не найдет искомый элемент.

Вариант **.lead** при этом позволит найти данный элемент, так как он ищет простое вхождение класса в элемент. Для уточнения селектора можно задать также оба класса, для этого нужно добавить второй класс к строке поиска без пробела и предварить его точкой: **.lead.text-muted**. Порядок классов в отличие от первого способа здесь не важен — **.text-muted.lead** так же найдет нужный элемент. Рекомендуем пользоваться вторым способом поиска классов, так как он является более гибким.

**Использование порядкового номера дочернего элемента**

Еще один способ найти этот элемент:

#posts > .item:nth-child(2) > .title

Псевдо-класс :nth-child(2) — позволяет найти второй по порядку элемент среди дочерних элементов для #posts. Затем с помощью конструкции > .title мы указываем, что нам нужен элемент .title, родителем которого является найденный ранее элемент .item.

**XPath** (XML Path Language) это язык запросов, который использует древовидную структуру документа

### XPath запрос всегда начинается с символа / или //

Символ / аналогичен символу > в CSS-селекторе, а символ // — пробелу. Их смысл:

* el1/el2 — выбирает элементы el2, являющиеся прямыми потомками el1;
* el1//el2 — выбирает элементы el2, являющиеся потомками el1 любой степени вложенности.

### 2. Символ [ ] — это команда фильтрации

Правил фильтрации очень много:

* по любому **атрибуту**, будь то id, class, title (или любой другой). Например, мы хотим найти картинку с летящим котом, для этого можно выполнить запрос //img[@id='bullet']
* по **порядковому номеру**. Допустим, мы хотим выбрать вторую по порядку карточку с котом. Для этого найдем элемент с классом "row" и возьмем его второго потомка: //div[@class="row"]/div[2]
* по **полному совпадению текста.** Да, XPath — это единственный способ найти элемент по внутреннему тексту. Если мы хотим найти блок текста с котом-Лениным, можно воспользоваться XPath селектором //p[text()="Lenin cat"]. Такой селектор вернет элемент, только если текст полностью совпадет. Здесь важно сказать, что не всегда поиск по тексту — это хорошая практика, особенно в случае мультиязычных сайтов.
* по **частичному совпадению** текста или атрибута. Для этого нужна функция contains. Запрос //p[contains(text(), "cat")] вернет нам все абзацы текста, которые содержат слово cat. Точно так же можно искать по частичному совпадению других атрибутов, это удобно, если у элемента несколько классов. Посмотрите на код навбара сайта с котами. Его можно найти селектором //div[contains(@class, "navbar")]
* в фильтрации еще можно использовать булевы операции (and, or, not) и некоторые простые арифметические выражения (но вообще не стоит, наверное). Допустим, мы хотим найти картинку обязательно с data-type "animal" и именем "bullet-cat", для этого подойдет запрос: //img[@name='bullet-cat' and @data-type='animal']

Поиск элементов с помощью Selenium WebDriver

Для поиска элементов на странице в Selenium WebDriver используются несколько стратегий, позволяющих искать по атрибутам элементов, текстам в ссылках, CSS-селекторам и XPath-селекторам. Существуют следующие методы поиска элементов:

* **find\_element\_by\_id** — поиск по уникальному атрибуту id элемента. Если ваши разработчики проставляют всем элементам в приложении уникальный id, то вам повезло, и вы чаще всего будет использовать этот метод, так как он наиболее стабильный;
* **find\_element\_by\_css\_selector** — поиск элемента с помощью правил на основе CSS. Это универсальный метод поиска, так как большинство веб-приложений использует CSS для вёрстки и задания оформления страницам. Если find\_element\_by\_id вам не подходит из-за отсутствия id у элементов, то скорее всего вы будете использовать именно этот метод в ваших тестах;
* **find\_element\_by\_xpath** — поиск с помощью языка запросов XPath, позволяет выполнять очень гибкий поиск элементов;
* **find\_element\_by\_name** — поиск по атрибуту name элемента;
* **find\_element\_by\_tag\_name** — поиск элемента по названию тега элемента;
* **find\_element\_by\_class\_name** — поиск по значению атрибута class;
* **find\_element\_by\_link\_text** — поиск ссылки на странице по полному совпадению;
* **find\_element\_by\_partial\_link\_text** — поиск ссылки на странице, если текст селектора совпадает с любой частью текста ссылки.

Например, мы хотим найти кнопку со значением id="submit\_button":

from selenium import webdriver

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/simple\_form\_find\_task.html")

button = browser.find\_element\_by\_id("submit")

Если страница у вас загрузилась, но дальше ничего не происходит, вернитесь обратно в консоль, в которой вы запускали ваш скрипт. Скорее всего, вы увидите там ошибку **NoSuchElementException**. Она будет выглядеть следующим образом:

selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate element: {"method":"id","selector":"submit"}

Ошибка очевидна: мы неправильно указали локатор — значит, кнопки с таким id на странице нет.

Исправим локатор, чтобы наш код проходил без ошибок:

from selenium import webdriver

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/simple\_form\_find\_task.html")

button = browser.find\_element\_by\_id("submit\_button")

Есть второй способ для поиска элементов с помощью универсального метода **find\_element**() и полей класса By из библиотеки selenium. Пример:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/simple\_form\_find\_task.html")

button = browser.find\_element(By.ID, "submit\_button")

Можно использовать те же стратегии поиска, что и в первом способе. Второй способ более удобен для оформления архитектуры тестовых сценариев с помощью подхода Page Object Model, о котором мы будем говорить далее. Пока же предлагаем пользоваться первым методом с явным указанием способа поиска, так как он кажется нам более удобным, но ничто не мешает вам пользоваться и тем, и другим. Поля класса By, которые можно использовать для поиска:

* **By.ID** – поиск по уникальному атрибуту id элемента;
* **By.CSS\_SELECTOR** – поиск элементов с помощью правил на основе CSS;
* **By.XPATH**– поиск элементов с помощью языка запросов XPath;
* **By.NAME** – поиск по атрибуту name элемента;
* **By.TAG\_NAME** – поиск по названию тега;
* **By.CLASS\_NAME** – поиск по атрибуту class элемента;
* **By.LINK\_TEXT** – поиск ссылки с указанным текстом. Текст ссылки должен быть точным совпадением;
* **By.PARTIAL\_LINK\_TEXT** – поиск ссылки по частичному совпадению текста.

!Важно. Вы можете столкнуться с ситуацией, когда на странице будет несколько элементов, подходящих под заданные вами параметры поиска. В этом случае WebDriver вернет вам только первый элемент, который встретит во время поиска по HTML. Если вам нужен не первый, а второй или следующие элементы, вам нужно либо задать более точный селектор для поиска, либо использовать методы **find\_elements\_by**, которые мы рассмотрим чуть позже.

Важно еще пояснить разницу между двумя командами:**browser.close()** и**browser.quit()**. Какая между ними разница, ведь на первый взгляд обе они осуществляют одно и то же?

На самом деле, **browser.close()**закрывает текущее окно браузера. Это значит, что если ваш скрипт вызвал всплывающее окно, или открыл что-то в новом окне или вкладке браузера, то закроется только текущее окно, а все остальные останутся висеть. В свою очередь **browser.quit()**закрывает все окна, вкладки, и процессы вебдрайвера, запущенные во время тестовой сессии. Подробнее можно посмотреть здесь: [Difference between webdriver.Dispose(), .Close() and .Quit()](https://stackoverflow.com/questions/15067107/difference-between-webdriver-dispose-close-and-quit). Будьте внимательны с этими методами и, в общем случае, всегда используйте **browser.quit().**

Но что будет, если скрипт не дойдет до выполнения этого финального шага, а упадет с ошибкой где-то раньше?

Для того чтобы гарантировать закрытие, даже если произошла ошибка в предыдущих строках, проще всего использовать конструкцию **try/finally**:

from selenium import webdriver

from selenium.webdriver.common.by import By

link = "http://suninjuly.github.io/simple\_form\_find\_task.html"

try:

browser = webdriver.Chrome()

browser.get(link)

button = browser.find\_element(By.ID, "submit\_button")

button.click()

finally:

# закрываем браузер после всех манипуляций

browser.quit()

даже если в коде внутри блока **try** произойдет какая-то ошибка, то код внутри блока **finally** выполнится в любом случае.

Допустим, на странице <https://www.degreesymbol.net/> мы хотим найти ссылку с текстом "Degree symbol examples" и перейти по ней. Если хотим найти элемент по полному соответствию текста, то нам подойдет такой код:

link = browser.find\_element\_by\_link\_text("» Degree symbol examples")

link.click()

А если хотим найти элемент со ссылкой по подстроке, то нужно написать следующий код:

link = browser.find\_element\_by\_partial\_link\_text("examples")

link.click()

## Поиск всех необходимых элементов с помощью find\_elements\_by

Мы уже упоминали, что метод find\_element\_by возвращает только первый из всех элементов, которые подходят под условия поиска. Иногда возникает ситуация, когда у нас есть несколько одинаковых по сути объектов на странице, например, иконки товаров в корзине интернет-магазина. В тесте нам нужно проверить, что отображаются все выбранные для покупки товары. Для этого существуют методы find\_element**s**\_by, которые в отличие от find\_element\_by вернут список всех найденных элементов по заданному условию. Проверив длину списка, мы можем удостовериться, что в корзине отобразилось правильное количество товаров. Пример кода (код приведен только для примера, сайта fake-shop.com скорее всего не существует):

# подготовка для теста

# открываем страницу первого товара

# данный сайт не существует, этот код приведен только для примера

browser.get("https://fake-shop.com/book1.html")

# добавляем товар в корзину

add\_button = browser.find\_element\_by\_css\_selector(".add")

add\_button.click()

# открываем страницу второго товара

browser.get("https://fake-shop.com/book2.html")

# добавляем товар в корзину

add\_button = browser.find\_element\_by\_css\_selector(".add")

add\_button.click()

# тестовый сценарий

# открываем корзину

browser.get("https://fake-shop.com/basket.html")

# ищем все добавленные товары

goods = browser.find\_elements\_by\_css\_selector(".good")

# проверяем, что количество товаров равно 2

assert len(goods) == 2

Также для поиска нескольких элементов мы можем использовать универсальный метод find\_element**s**вместе с атрибутами класса **By**:

from selenium.webdriver.common.by import By

browser.find\_elements(By.CSS\_SELECTOR, "button.submit")

!Важно. Обратите внимание на важную разницу в результатах, которые возвращают методы **find\_element** и **find\_elements**. Если первый метод не смог найти элемент на странице, то он вызовет ошибку NoSuchElementException, которая прервёт выполнение вашего кода. Второй же метод всегда возвращает валидный результат: если ничего не было найдено, то он вернёт пустой список и ваша программа перейдет к выполнению следующего шага в коде.

**Assert**

from selenium import webdriver  
import time  
  
try:  
 link = "http://suninjuly.github.io/registration1.html"  
 browser = webdriver.Chrome()  
 browser.get(link)  
  
 # Ваш код, который заполняет обязательные поля  
 input1 = browser.find\_element\_by\_xpath("/html/body/div/form/div[1]/div[1]/input")  
 input1.send\_keys("Ivan")  
 input2 = browser.find\_element\_by\_xpath("/html/body/div/form/div[1]/div[2]/input")  
 input2.send\_keys("Ivanov")  
 input3 = browser.find\_element\_by\_xpath("/html/body/div/form/div[1]/div[3]/input")  
 input3.send\_keys("ivanov@mail.ua")  
  
 # Отправляем заполненную форму  
 button = browser.find\_element\_by\_css\_selector("button.btn")  
 button.click()  
  
 # Проверяем, что смогли зарегистрироваться  
 # ждем загрузки страницы  
 time.sleep(1)  
  
 # находим элемент, содержащий текст  
 welcome\_text\_elt = browser.find\_element\_by\_tag\_name("h1")  
 # записываем в переменную welcome\_text текст из элемента welcome\_text\_elt  
 welcome\_text = welcome\_text\_elt.text  
  
 # с помощью assert проверяем, что ожидаемый текст совпадает с текстом на странице сайта  
 assert "Congratulations! You have successfully registered!" == welcome\_text  
  
finally:  
 # ожидание чтобы визуально оценить результаты прохождения скрипта  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()

**2 Модуль – Полезные методы Selenium**

from selenium import webdriver  
import time  
import math  
  
  
def calc(x):  
 return str(math.log(abs(12\*math.sin(int(x)))))  
  
  
try:  
# Открыть страницу http://suninjuly.github.io/math.html.  
 link = "http://suninjuly.github.io/math.html"  
 browser = webdriver.Chrome()  
 browser.get(link)  
#Считать значение для переменной x.  
#Посчитать математическую функцию от x (код для этого приведён ниже).  
 x\_element = browser.find\_element\_by\_xpath('//span[@id="input\_value"]')  
 x = x\_element.text  
 y = calc(x)  
#Ввести ответ в текстовое поле.  
 input1 = browser.find\_element\_by\_xpath('//input[@id="answer"]')  
 input1.send\_keys(y)  
  
  
#Отметить checkbox "I'm the robot".  
 input2 = browser.find\_element\_by\_xpath('//input[@id="robotCheckbox"]')  
 input2.click()  
#Выбрать radiobutton "Robots rule!".  
 input3 = browser.find\_element\_by\_xpath('//input[@id="robotsRule"]')  
 input3.click()  
  
# Нажать на кнопку Submit.  
 button = browser.find\_element\_by\_xpath('//button [contains(text(), "Submit")]')  
 button.click()  
  
finally:  
 # ожидание чтобы визуально оценить результаты прохождения скрипта  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()

## Метод get\_attribute

 В автотесте нам может понадобиться проверить, что для одного из radiobutton по умолчанию уже выбрано значение. Для этого мы можем проверить значение атрибута checked у этого элемента. Вот HTML-код элемента:

<input class="check-input" type="radio" name="ruler" id="peopleRule" value="people" checked>

Найдём этот элемент с помощью WebDriver:

people\_radio = browser.find\_element\_by\_id("peopleRule")

Найдём атрибут "checked" с помощью встроенного метода get\_attribute и проверим его значение:

people\_checked = people\_radio.get\_attribute("checked")

print("value of people radio: ", people\_checked)

assert people\_checked is not None, "People radio is not selected by default"

Т.к. у данного атрибута значение не указано явно, то метод get\_attribute вернёт "true". Возможно, вы заметили, что "true" написано с маленькой буквы, — все методы WebDriver взаимодействуют с браузером с помощью JavaScript, в котором булевые значения пишутся с маленькой буквы, а не с большой, как в Python.

Мы можем написать проверку другим способом, сравнив строки:

assert people\_checked == "true", "People radio is not selected by default"

Если атрибута нет, то метод get\_attribute вернёт значение **None**. Применим метод get\_attribute ко второму radiobutton, и убедимся, что атрибут отсутствует.

robots\_radio = browser.find\_element\_by\_id("robotsRule")

robots\_checked = robots\_radio.get\_attribute("checked")

assert robots\_checked is None

Так же мы можем проверять наличие атрибута disabled, который определяет, может ли пользователь взаимодействовать с элементом. Например, в предыдущем задании на странице с капчей для роботов JavaScript устанавливает атрибут disabled у кнопки **Submit**, когда истекает время, отведенное на решение задачи.

<button type="submit" class="btn btn-default" disabled>Submit</button>

from selenium import webdriver  
import time  
import math  
  
  
def calc(x):  
 return str(math.log(abs(12\*math.sin(int(x)))))  
  
try:  
#1-Открыть страницу http://suninjuly.github.io/get\_attribute.html.  
 link = "http://suninjuly.github.io/get\_attribute.html"  
 browser = webdriver.Chrome()  
 browser.get(link)  
#2-Найти на ней элемент-картинку, который является изображением сундука с сокровищами.  
#3-Взять у этого элемента значение атрибута valuex, которое является значением x для задачи.  
 x = browser.find\_element\_by\_xpath('//img[@id="treasure"]').get\_attribute("valuex")  
#4-Посчитать математическую функцию от x (сама функция остаётся неизменной).  
 y = calc(x)  
#5-Ввести ответ в текстовое поле.  
 input1 = browser.find\_element\_by\_xpath('//input[@id="answer"]').send\_keys(y)  
#6-Отметить checkbox "I'm the robot".  
 input2 = browser.find\_element\_by\_xpath('//input[@id="robotCheckbox"]').click()  
#7-Выбрать radiobutton "Robots rule!".  
 input3 = browser.find\_element\_by\_xpath('//input[@id="robotsRule"]').click()  
  
#8-Нажать на кнопку "Submit".  
 button = browser.find\_element\_by\_xpath('//button [contains(text(), "Submit")]').click()  
  
finally:  
 # ожидание чтобы визуально оценить результаты прохождения скрипта  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()

2.2 Работа с файлами, списками и js-скриптами

арианты ответа задаются тегом option, значение value может отсутствовать. Можно отмечать варианты с помощью обычного метода click(). Для этого сначала нужно применить метод click() для элемента с тегом select, чтобы список раскрылся, а затем кликнуть на нужный вариант ответа:

from selenium import webdriver

browser = webdriver.Chrome()

browser.get(link)

browser.find\_element\_by\_tag\_name("select").click()

browser.find\_element\_by\_css\_selector("option:nth-child(2)").click()

Последняя строчка может выглядеть и так:

browser.find\_element\_by\_css\_selector("[value='1']").click()

Это не самый удобный способ, так как нам приходится делать лишний клик для открытия списка.

Есть более удобный способ, для которого используется специальный класс **Select** из библиотеки WebDriver. Вначале мы должны инициализировать новый объект, передав в него WebElement с тегом select. Далее можно найти любой вариант из списка с помощью метода **select\_by\_value(value):**

from selenium.webdriver.support.ui import Select

select = Select(browser.find\_element\_by\_tag\_name("select"))

select.select\_by\_value("1") # ищем элемент с текстом "Python"

Можно использовать еще два метода: **select.select\_by\_visible\_text("text")** и **select.select\_by\_index(index)**. Первый способ ищет элемент по видимому тексту, например,**select.select\_by\_visible\_text("Python")** найдёт "Python" для нашего примера.

Второй способ ищет элемент по его индексу или порядковому номеру. Индексация начинается с нуля. Для того чтобы найти элемент с текстом "Python", нужно использовать **select.select\_by\_index(1)**, так как опция с индексом 0 в данном примере имеет значение по умолчанию равное "--".

Cool task

from selenium import webdriver  
import time  
from selenium.webdriver.support.ui import Select  
  
try:  
#1-Открыть страницу http://suninjuly.github.io/get\_attribute.html.  
 link = "http://suninjuly.github.io/selects1.html"  
 browser = webdriver.Chrome()  
 browser.get(link)  
  
#2-Посчитать сумму заданных чисел  
 a = browser.find\_element\_by\_xpath('//span[@id="num1"]').text  
 b = browser.find\_element\_by\_xpath('//span[@id="num2"]').text  
 c = str(int(a) + int(b))  
  
# .text() - принимает в качестве значения строку! (Тип данных строка str)  
# для корректной суммы надо перевести два числа a и b в int; (Тип данных число int),  
# затем сумму (a + b) обратно в строку, т.к. значения в открывающемся списке находятся в строчном формате  
  
#3-Выбрать в выпадающем списке значение равное расчитанной сумме  
  
 select = Select(browser.find\_element\_by\_tag\_name("select"))  
 select.select\_by\_value(c)  
  
#4-Нажать кнопку "Submit"  
 button = browser.find\_element\_by\_xpath('//button [contains(text(), "Submit")]').click()  
  
finally:  
 # ожидание чтобы визуально оценить результаты прохождения скрипта  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()

## Метод execute\_script

**Метод execute\_script**

С помощью метода execute\_script можно выполнить программу, написанную на языке JavaScript, как часть сценария автотеста в запущенном браузере.

вызвать alert в браузере с помощью WebDriver. Пример сценария:

from selenium import webdriver

browser = webdriver.Chrome()

browser.execute\_script("alert('Robots at work');")

Обратите внимание, что исполняемый JavaScript нужно заключать в кавычки (двойные или одинарные).

Если кстати возникает дилемма о кавычках, можно все внутренние экранировать обратным слэшем вот так:

browser.execute\_script("alert(\"Robots at work\");") и не переступать свои убеждения использовать только двойные кавычки во что бы то ни стало.

Можно с помощью этого метода выполнить сразу несколько инструкций, перечислив их через точку с запятой. Изменим сначала заголовок страницы, а затем вызовем alert:

browser.execute\_script("document.title='Script executing';alert('Robots at work');")

## Пример задачи для execute\_script

Давайте теперь рассмотрим реальную ситуацию, когда пользователь должен кликнуть на элемент, который внезапно оказывается перекрыт другим элементом на странице.

Для клика в WebDriver мы используем метод click(). Если элемент оказывается перекрыт другим элементом, то наша программа вызовет следующую ошибку:

selenium.common.exceptions.WebDriverException: Message: unknown error: Element <button type="submit" class="btn btn-default" style="margin-bottom: 1000px;">...</button> is not clickable at point (87, 420). Other element would receive the click: <p>...</p>

Из описания ошибки можно понять, что указанный нами элемент нельзя кликнуть в данной точке, т.к. клик произойдёт на другом элементе с тегом <p>.

Чтобы увидеть пример данной ошибки, запустите следующий скрипт:

from selenium import webdriver

browser = webdriver.Chrome()

link = "https://SunInJuly.github.io/execute\_script.html"

browser.get(link)

button = browser.find\_element\_by\_tag\_name("button")

button.click()

Теперь вы можете сами посмотреть на эту [страницу](https://suninjuly.github.io/execute_script.html) и увидеть, что огромный футер действительно перекрывает нужную нам кнопку. Футером (footer) называется нижний блок, который обычно одинаков для всех страниц сайта. Чтобы понять, как решить эту проблему, нужно разобраться, как работает метод **click()**.

В первую очередь WebDriver проверит, что ширина и высота элемента больше 0, чтобы по нему можно было кликнуть.

Затем, если элемент находится за границей окна браузера, WebDriver автоматически проскроллит страницу, чтобы элемент попал в область видимости, то есть не находился за границей экрана. Но это не гарантирует того, что элемент не перекрыт другим элементом, который тоже находится в области видимости.

А в какую точку элемента будет происходить клик? Selenium рассчитывает координаты центра элемента и производит клик в вычисленную точку. Это тоже приведёт к ошибке, если часть элемента всё-таки видна, но элемент перекрыт больше чем на половину своей высоты или ширины.

Если мы столкнулись с такой ситуацией, мы можем заставить браузер дополнительно проскроллить нужный элемент, чтобы он точно стал видимым.  
Делается это с помощью следующего скрипта:

"return arguments[0].scrollIntoView(true);"

Мы дополнительно передали в метод scrollIntoView аргумент true, чтобы элемент после скролла оказался в области видимости. Другие возможные параметры метода можно посмотреть здесь: <https://developer.mozilla.org/ru/docs/Web/API/Element/scrollIntoView>

В итоге, чтобы кликнуть на перекрытую кнопку, нам нужно выполнить следующие команды в коде:

button = browser.find\_element\_by\_tag\_name("button")

browser.execute\_script("return arguments[0].scrollIntoView(true);", button)

button.click()

В метод execute\_script мы передали текст js-скрипта и найденный элемент button, к которому нужно будет проскроллить страницу. После выполнения кода элемент button должен оказаться в верхней части страницы. Подробнее о методе см <https://developer.mozilla.org/ru/docs/Web/API/Element/scrollIntoView> .

Также можно проскроллить всю страницу целиком на строго заданное количество пикселей. Эта команда проскроллит страницу на 100 пикселей вниз:

browser.execute\_script("window.scrollBy(0, 100);")

!Важно. Мы не будем в этом курсе изучать, как работает JavaScript, и обойдемся только приведенным выше примером скрипта с прокруткой страницы. Для сравнения приведем скрипт на этом языке, который делает то же, что приведенный выше пример для WebDriver:

// javascript

button = document.getElementsByTagName("button")[0];

button.scrollIntoView(true);

Можете попробовать исполнить его в консоли браузера на странице <http://suninjuly.github.io/execute_script.html>. Для этого откройте инструменты разработчика в браузере, перейдите на вкладку **консоль (console)**, скопируйте туда этот код и нажмите Enter. Таким образом можно протестировать кусочки js кода прежде чем внедрять его в свои тесты на python.

Обратите внимание, что в коде в WebDriver нужно использовать ключевое слово **return**. Также его нужно будет использовать, когда вы захотите получить какие-то данные после выполнения скрипта. При этом при тестировании скрипта в консоли браузера слово **return** использовать не надо.

## Задание на execute\_script

from selenium import webdriver  
import time  
import math  
  
  
def calc(x):  
 return str(math.log(abs(12\*math.sin(int(x)))))  
  
try:  
  
# 1-Открыть страницу http://SunInJuly.github.io/execute\_script.html.  
 link = "http://SunInJuly.github.io/execute\_script.html"  
  
 browser = webdriver.Chrome()  
 browser.get(link)  
  
# 2-Считать значение для переменной x.  
 x\_value = browser.find\_element\_by\_xpath("//span[@id='input\_value']").text  
  
# 3-Посчитать математическую функцию от x.  
 x = calc(x\_value)  
  
# 4-Проскроллить страницу вниз.  
# button = browser.find\_element\_by\_tag\_name("button")  
 button = browser.find\_element\_by\_xpath("//button[contains(text(),'Submit')]")  
 browser.execute\_script("return arguments[0].scrollIntoView(true);", button)  
 button.click()  
  
# 5-Ввести ответ в текстовое поле.  
 browser.find\_element\_by\_xpath("//input[@id='answer']").send\_keys(x)  
  
# 6-Выбрать checkbox "I'm the robot".  
 browser.find\_element\_by\_xpath("//input[@id='robotCheckbox']").click()  
  
# 7-Переключить radiobutton "Robots rule!".  
 browser.find\_element\_by\_xpath("//input[@id='robotsRule']").click()  
  
# 8-Нажать на кнопку "Submit".  
 browser.find\_element\_by\_xpath("//button[contains(text(),'Submit')]").click()  
  
finally:  
 # успеваем скопировать код за 30 секунд  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()  
  
# не забываем оставить пустую строку в конце файла

Как вариант еще можно скрывать ненужный элемент

browser.execute\_script(**'arguments[0].style.visibility =** \'**hidden**\'**'**, footer)

Еще в глобальном смысле мотнуть в самый верх или самый низ страницы можно и питоном для тега body

from selenium.webdriver.common.keys import Keys

browser.find\_element\_by\_tag\_name('body').send\_keys(Keys.END) #или Home если наверх

## Загрузка файлов

﻿Если нам понадобится загрузить файл на веб-странице, мы можем использовать уже знакомый нам метод send\_keys. Только теперь нам нужно в качестве аргумента передать путь к нужному файлу на диске вместо простого текста.

Чтобы указать путь к файлу, можно использовать стандартный модуль Python для работы с операционной системой — **os**. В этом случае ваш код не будет зависеть от операционной системы, которую вы используете. Добавление файла будет работать и на Windows, и на Linux, и даже на MaсOS.

Пример кода, который позволяет указать путь к файлу**'file.txt**', находящемуся в той же папке, что и скрипт, который вы запускаете:

import os

current\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_)) # получаем путь к директории текущего исполняемого файла

file\_path = os.path.join(current\_dir, 'file.txt') # добавляем к этому пути имя файла

element.send\_keys(file\_path)

Попробуйте добавить в файл отдельно команды **print(os.path.abspath(\_\_file\_\_))** и **print(os.path.abspath(os.path.dirname(\_\_file\_\_)))** и посмотрите на разницу

Обратите внимание, что это будет работать только при запуске кода из файла, в интерпретаторе не сработает.

os.path.abspath(os.path.dirname(\_\_file\_\_)) показывает абсолютный путь, т.е. от корня файловой системы (диска в Windows), а os.path.dirname(\_\_file\_\_) относительный, т.е от текущей директории (там где лежит пайтоновский файл, который исполняется)

Если совсем непонятно что происходит, пример:

Допустим, мы написали код скрипта и сохранили код в *lesson2\_step7.py* в свой локальной папке *D:\stepik\_homework.*Активируем виртуальное окружение и запускаем его **python lesson2\_step7.py.**В таком случае конструкция **os.path.abspath(os.path.dirname(\_\_file\_\_))**вернет нам путь до директории файла с кодом, то есть *D:\stepik\_homework***.**В эту же папку кладем файл, который хотим прикрепить, то есть *file.txt*. Тогда, после выполнения команды:

file\_path = os.path.join(current\_dir, 'file.txt')

В переменной *file\_path* будет полный путь к файлу '**D:\stepik\_homework\file.txt'**. Фишка в том, что если мы файлы *lesson2\_step7.py*вместе с*file.txt*перенесем в другую папку, или на компьютер с другой ОС, то такой код без правок заработает и там.

Элемент в форме, который выглядит, как кнопка добавления файла, имеет атрибут**type="file"**. Мы должны сначала найти этот элемент с помощью селектора, а затем применить к нему метод **send\_keys(file\_path)**.

## Задание: загрузка файла

import os  
from selenium import webdriver  
import time  
  
try:  
 # 1-Открыть страницу http://suninjuly.github.io/file\_input.html.  
 link = "http://suninjuly.github.io/file\_input.html"  
 browser = webdriver.Chrome()  
 browser.get(link)  
  
 # 2-Заполнить текстовые поля: имя, фамилия, email  
 browser.find\_element\_by\_xpath("//input[@name='firstname']").send\_keys("Pavel")  
 browser.find\_element\_by\_xpath("//input[@name='lastname']").send\_keys("Sazonov")  
 browser.find\_element\_by\_xpath("//input[@name='email']").send\_keys("pavel@mail.ua")  
  
 # 3-Загрузить файл. Файл должен иметь расширение .txt и может быть пустым  
 current\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_)) # получаем путь к директории текущего исполняемого файла  
 file\_path = os.path.join(current\_dir, 'file.txt') # добавляем к этому пути имя файла  
 element = browser.find\_element\_by\_xpath("//input[@id='file']").send\_keys(file\_path)  
  
 # 4-Нажать кнопку "Submit"  
 browser.find\_element\_by\_xpath("//button[@type='submit']").click()  
  
finally:  
 # успеваем скопировать код за 30 секунд  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()  
  
# не забываем оставить пустую строку в конце файла

**Работа с окнами**

## Alerts и как с ними жить

можно самостоятельно вызвать alert с помощью JavaScript:

alert('Hello!');

рассмотрим ситуацию, когда в сценарии теста возникает необходимость не только получить содержимое alert, но и нажать кнопку OK, чтобы закрыть alert. **Alert** является модальным окном: это означает, что пользователь не может взаимодействовать дальше с интерфейсом, пока не закроет alert. Для этого нужно сначала переключиться на окно с alert, а затем принять его с помощью команды **accept()**:

alert = browser.switch\_to.alert

alert.accept()

Чтобы получить текст из alert, используйте свойство text объекта alert:

alert = browser.switch\_to.alert

alert\_text = alert.text

Другой вариант модального окна, который предлагает пользователю выбор согласиться с сообщением или отказаться от него, называется **confirm**. Для переключения на окно confirmиспользуется та же команда, что и в случае с alert:

confirm = browser.switch\_to.alert

confirm.accept()

Для confirm**-**окон можно использовать следующий метод для отказа:

confirm.dismiss()

То же самое, что и при нажатии пользователем кнопки "Отмена".

Третий вариант модального окна — **prompt**— имеет дополнительное поле для ввода текста. Чтобы ввести текст, используйте метод **send\_keys()**:

prompt = browser.switch\_to.alert

prompt.send\_keys("My answer")

prompt.accept()

как распарсить строку:  split возвращает список, в качестве разделителя используем 'двоеточие пробел',  берём последний элемент, это нужное нам число. Осталось его буфер обмена втащить)) и далее копировать в поле для ответа в задании.

alert\_text.split(': ')[-1]

я не знаю как в буфер обмена питон заставить число поместить, с этим разобраться хочу.

заюзать эту библиотеку [https://pypi.org/project/pyperclip/](https://pypi.org/project/pyperclip/" \t "_blank)

pyperclip помог, спасибо!

alert = browser.switch\_to.alert

alert\_text = alert.text

addToClipBoard = alert\_text.split(': ')[-1]

pyperclip.copy(addToClipBoard)

и число в буфере обмена.

Use pyperclip copy in Clipboard and paste after it in other page

from selenium import webdriver  
import time  
import math  
import pyperclip  
# Terminal > pip install pyperclip  
  
def calc(x):  
 return str(math.log(abs(12\*math.sin(int(x)))))  
  
  
try:  
 # 1-Открыть страницу http://suninjuly.github.io/alert\_accept.html  
 link = "http://suninjuly.github.io/alert\_accept.html"  
 browser = webdriver.Chrome()  
 browser.get(link)  
  
 # 2-Нажать на кнопку  
 browser.find\_element\_by\_xpath('//button[@type="submit"]').click()  
  
 # 3-Принять confirm  
 confirm = browser.switch\_to.alert  
 confirm.accept()  
  
 # 4-На новой странице решить капчу для роботов, чтобы получить число с ответом  
 x = browser.find\_element\_by\_xpath('//span[@id="input\_value"]').text  
 y = calc(x)  
  
 browser.find\_element\_by\_xpath('//input[@id="answer"]').send\_keys(y)  
  
 browser.find\_element\_by\_xpath('//button[@type="submit"]').click()  
  
 # Cчитывать с окна алерта правильный результат и копировать его в буфер обмена  
  
 alert = browser.switch\_to.alert  
 alert\_text = alert.text  
 addToClipBoard = alert\_text.split(': ')[-1]  
 pyperclip.copy(addToClipBoard)  
  
# Вставить правильный результат в нужное место  
  
 link2 = "https://www.google.com/"  
 browser2 = webdriver.Chrome()  
 browser2.get(link2)  
 answer = pyperclip.paste()  
 browser2.find\_element\_by\_xpath('//input[@name="q"]').send\_keys(answer)  
  
  
  
 # answer = browser.find\_element\_by\_xpath('//input[@name="q"]')  
 # pyperclip.paste(answer)  
  
 # answer = browser.find\_element\_by\_xpath('//input[@name="q"]').pyperclip.paste  
  
 time.sleep(5)  
# >>> pyperclip.copy('The text to be copied to the clipboard.')  
# >>> pyperclip.paste()  
  
finally:  
 # успеваем скопировать код за 30 секунд  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()  
  
# не забываем оставить пустую строку в конце файла

Альтернативное решение

# получение ответа и автоматический его ввод на степике  
from selenium import webdriver  
import os  
import math  
import time  
  
def calc(x):  
 return str(math.log(abs(12\*math.sin(int(x)))))  
  
browser = webdriver.Chrome()  
browser.get('http://suninjuly.github.io/alert\_accept.html')  
  
button = browser.find\_element\_by\_class\_name('btn-primary').click()  
browser.switch\_to.alert.accept()  
  
x = browser.find\_element\_by\_id("input\_value").text  
y = calc(x)  
  
browser.find\_element\_by\_id('answer').send\_keys(y)  
browser.find\_element\_by\_class\_name('btn-primary').click()  
  
alert = browser.switch\_to.alert  
alert\_text = alert.text.split()  
alert.accept()  
answer = alert\_text[-1]  
  
browser.get('https://stepik.org/catalog?auth=login&language=ru')  
time.sleep(5)  
  
browser.find\_element\_by\_id('id\_login\_email').send\_keys('\*\*\*')# здесь вводится e-mail  
browser.find\_element\_by\_id('id\_login\_password').send\_keys('\*\*\*')# здесь вводится пароль  
  
browser.find\_element\_by\_class\_name('sign-form\_\_btn').click()  
time.sleep(3)  
browser.get('https://stepik.org/lesson/184253/step/4?unit=158843')  
time.sleep(3)  
  
answer\_input = browser.find\_element\_by\_css\_selector('textarea')  
browser.execute\_script("return arguments[0].scrollIntoView(true);", answer\_input)  
answer\_input.send\_keys(answer)  
  
button = browser.find\_element\_by\_class\_name('submit-submission')  
browser.execute\_script("return arguments[0].scrollIntoView(true);", button)  
time.sleep(1)  
button.click()

## Переход на новую вкладку браузера

При работе с веб-приложениями приходится переходить по ссылкам, которые открываются в новой вкладке браузера. WebDriver может работать только с одной вкладкой браузера. При открытии новой вкладки WebDriver продолжит работать со старой вкладкой. Для переключения на новую вкладку надо явно указать, на какую вкладку мы хотим перейти. Это делается с помощью команды switch\_to.window:

browser.switch\_to.window(window\_name)

Чтобы узнать имя новой вкладки, нужно использовать метод window\_handles, который возвращает массив имён всех вкладок. Зная, что в браузере теперь открыто две вкладки, выбираем вторую вкладку:

new\_window = browser.window\_handles[1]

Также мы можем запомнить имя текущей вкладки, чтобы иметь возможность потом к ней вернуться:

first\_window = browser.window\_handles[0]

После переключения на новую вкладку поиск и взаимодействие с элементами будут происходить уже на новой странице.

Текущую вкладку можно узнать так:  
  
current\_window = browser.current\_window\_handle

browser.switch\_to.iframe

ничего особо сложного)

from selenium import webdriver  
import time  
import math  
import pyperclip  
  
  
def calc(x):  
 return str(math.log(abs(12\*math.sin(int(x)))))  
  
  
try:  
 # 1-Открыть страницу http://suninjuly.github.io/redirect\_accept.html  
 link = "http://suninjuly.github.io/redirect\_accept.html"  
 browser = webdriver.Chrome()  
 browser.get(link)  
  
 # 2-Нажать на кнопку  
 browser.find\_element\_by\_xpath('//button[@type="submit"]').click()  
  
 # 3-Переключиться на новую вкладку  
 new\_window = browser.window\_handles[1]  
 browser.switch\_to.window(new\_window)  
  
 # 4-Пройти капчу для робота и получить число-ответ  
 x = browser.find\_element\_by\_xpath('//span[@id="input\_value"]').text  
 y = calc(x)  
  
 browser.find\_element\_by\_xpath('//input[@id="answer"]').send\_keys(y)  
  
 browser.find\_element\_by\_xpath('//button[@type="submit"]').click()  
  
 # Cчитывать с окна алерта правильный результат и копировать его в буфер обмена  
  
 alert = browser.switch\_to.alert  
 alert\_text = alert.text  
 addToClipBoard = alert\_text.split(': ')[-1]  
 pyperclip.copy(addToClipBoard)  
  
finally:  
 # успеваем скопировать код за 30 секунд  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()  
  
# не забываем оставить пустую строку в конце файла

2.4 Настройка ожиданий

A **single-page application** (**SPA**) is a [web application](https://en.wikipedia.org/wiki/Web_application) or [website](https://en.wikipedia.org/wiki/Website) that interacts with the user by dynamically rewriting the current [web page](https://en.wikipedia.org/wiki/Web_design) with new data from the [web server](https://en.wikipedia.org/wiki/Web_server), instead of the default method of a [web browser](https://en.wikipedia.org/wiki/Web_browser) loading entire new pages. The goal is faster transitions that make the website [feel](https://en.wikipedia.org/wiki/User_experience) more like a [native](https://en.wikipedia.org/wiki/Native_(computing)) [app](https://en.wikipedia.org/wiki/Application_software).

## Selenium Waits (Implicit Waits)

В Selenium WebDriver есть специальный способ организации такого ожидания, который позволяет задать ожидание при инициализации драйвера, чтобы применить его ко всем тестам. Ожидание называется **неявным** (**Implicit wait**), так как его не надо явно указывать каждый раз, когда мы выполняем поиск элементов, оно автоматически будет применяться при вызове каждой последующей команды.

from selenium import webdriver  
  
browser = webdriver.Chrome()  
# говорим WebDriver искать каждый элемент в течение 5 секунд  
browser.implicitly\_wait(5)  
  
browser.get("http://suninjuly.github.io/wait1.html")  
  
button = browser.find\_element\_by\_id("verify")  
button.click()  
message = browser.find\_element\_by\_id("verify\_message")  
  
assert "successful" in message.text

Теперь мы можем быть уверены, что при небольших задержках в работе сайта наши тесты продолжат работать стабильно. На каждый вызов команды **find\_element** WebDriver будет ждать 5 секунд до появления элемента на странице прежде, чем выбросить исключение **NoSuchElementException**.

Теперь мы знаем, как настроить ожидание поиска элемента. Во время поиска WebDriver каждые 0.5 секунды проверяет, появился ли нужный элемент в DOM-модели браузера (Document Object Model — «объектная модель документа», интерфейс для доступа к HTML-содержимому сайта). Если произойдет ошибка, то WebDriver выбросит одно из следующих исключений (**exceptions**):

* Если элемент не был найден за отведенное время, то мы получим **NoSuchElementException.**
* Если элемент был найден в момент поиска, но при последующем обращении к элементу DOM изменился, то получим **StaleElementReferenceException**. Например, мы нашли элемент **Кнопка** и через какое-то время решили выполнить с ним уже известный нам метод click. Если кнопка за это время была скрыта скриптом, то метод применять уже бесполезно — элемент "устарел" (stale) и мы увидим исключение.
* Если элемент был найден в момент поиска, но сам элемент невидим (например, имеет нулевые размеры), и реальный пользователь не смог бы с ним взаимодействовать, то получим **ElementNotVisibleException**.

Знание причин появления исключений помогает отлаживать тесты и понимать, где находится баг в случае его возникновения.

## Explicit Waits (WebDriverWait и expected\_conditions)

Чтобы тест был надежным, нам нужно не только найти кнопку на странице, но и дождаться, когда кнопка станет кликабельной. Для реализации подобных ожиданий в Selenium WebDriver существует понятие **явных** ожиданий (**Explicit Waits**), которые позволяют задать специальное ожидание для конкретного элемента. Задание явных ожиданий реализуется с помощью инструментов WebDriverWait и **expected\_conditions**

from selenium.webdriver.common.by import By  
from selenium.webdriver.support.ui import WebDriverWait  
from selenium.webdriver.support import expected\_conditions as EC  
from selenium import webdriver  
  
browser = webdriver.Chrome()  
  
browser.get("http://suninjuly.github.io/wait2.html")  
  
# говорим Selenium проверять в течение 5 секунд, пока кнопка не станет кликабельной  
button = WebDriverWait(browser, 5).until(  
 EC.element\_to\_be\_clickable((By.ID, "verify"))  
 )  
button.click()  
message = browser.find\_element\_by\_id("verify\_message")  
  
assert "successful" in message.text

Как вы видите, в этом случае нужно использовать поиск элементов с помощью класса **By**, который мы рассмотрели [ранее](https://stepik.org/lesson/138920/step/2?unit=196194). **element\_to\_be\_clickable**вернет элемент, когда он станет кликабельным, или вернет **False**в ином случае.

Обратите внимание, что в объекте WebDriverWait используется функция **until**, в которую передается правило ожидания, элемент, а также значение, по которому мы будем искать элемент. В модуле **expected\_conditions** есть много других правил, которые позволяют реализовать необходимые ожидания:

* title\_is
* title\_contains
* presence\_of\_element\_located
* visibility\_of\_element\_located
* visibility\_of
* presence\_of\_all\_elements\_located
* text\_to\_be\_present\_in\_element
* text\_to\_be\_present\_in\_element\_value
* frame\_to\_be\_available\_and\_switch\_to\_it
* invisibility\_of\_element\_located
* element\_to\_be\_clickable
* staleness\_of
* element\_to\_be\_selected
* element\_located\_to\_be\_selected
* element\_selection\_state\_to\_be
* element\_located\_selection\_state\_to\_be
* alert\_is\_present

Если мы захотим проверять, что кнопка становится неактивной после отправки данных, то можно задать негативное правило с помощью метода **until\_not**:

# говорим Selenium проверять в течение 5 секунд пока кнопка станет неактивной

button = WebDriverWait(browser, 5).until\_not(

EC.element\_to\_be\_clickable((By.ID, "verify"))

)

# Задание: ждем нужный текст на странице  
  
# Попробуем теперь написать программу, которая будет бронировать нам дом для отдыха по строго заданной цене.  
# Более высокая цена нас не устраивает, а по более низкой цене объект успеет забронировать кто-то другой.  
  
import math  
import time  
import pyperclip  
from selenium import webdriver  
from selenium.webdriver.common.by import By  
from selenium.webdriver.support.ui import WebDriverWait  
from selenium.webdriver.support import expected\_conditions as EC  
  
def calc(x):  
 return str(math.log(abs(12\*math.sin(int(x)))))  
  
  
try:  
  
# 1 - Открыть страницу http://suninjuly.github.io/explicit\_wait2.html  
 browser = webdriver.Chrome()  
 browser.get("http://suninjuly.github.io/explicit\_wait2.html")  
# 2 - Дождаться, когда цена дома уменьшится до $100 (ожидание нужно установить не меньше 12 секунд)  
# Чтобы определить момент, когда цена аренды уменьшится до $100, используйте метод text\_to\_be\_present\_in\_element  
# из библиотеки expected\_conditions.  
 WebDriverWait(browser, 12).until(EC.text\_to\_be\_present\_in\_element((By.ID, "price"), str(100)))  
# 3 - Нажать на кнопку "Book"  
 browser.find\_element\_by\_xpath("//button[@id = 'book']").click()  
# 4 - Решить уже известную нам математическую задачу (используйте ранее написанный код) и отправить решение  
  
 x = browser.find\_element\_by\_xpath('//span[@id="input\_value"]').text  
 y = calc(x)  
  
 browser.find\_element\_by\_xpath('//input[@id="answer"]').send\_keys(y)  
  
 browser.find\_element\_by\_xpath('//button[@type="submit"]').click()  
  
# Если все сделано правильно и быстро, то вы увидите окно с числом. Отправьте его в качестве ответа на это задание.  
  
# Cчитывать с окна алерта правильный результат и копировать его в буфер обмена  
  
 alert = browser.switch\_to.alert  
 alert\_text = alert.text  
 addToClipBoard = alert\_text.split(': ')[-1]  
 pyperclip.copy(addToClipBoard)  
  
  
finally:  
 # успеваем скопировать код за 30 секунд  
 time.sleep(10)  
 # закрываем браузер после всех манипуляций  
 browser.quit()  
  
# не забываем оставить пустую строку в конце файла

Alternative

from selenium.webdriver.common.by import By  
from selenium.webdriver.support.ui import WebDriverWait  
from selenium.webdriver.support import expected\_conditions as EC  
from selenium import webdriver  
import math  
  
  
def calc(x):  
 return str(math.log(abs(12 \* math.sin(int(x)))))  
  
  
browser = webdriver.Chrome()  
link = 'http://suninjuly.github.io/explicit\_wait2.html'  
browser.implicitly\_wait(5) # ожидание действия в 5 секунд, если не нашелся selector  
try:  
 browser.get(link)  
 WebDriverWait(browser, 15).until(EC.text\_to\_be\_present\_in\_element((By.ID, "price"), '100'))  
 browser.find\_element\_by\_css\_selector('#book').click()  
 button = browser.find\_element\_by\_css\_selector('#solve')  
 browser.execute\_script("return arguments[0].scrollIntoView(true);", button)  
 x1 = browser.find\_element\_by\_css\_selector('#input\_value').text  
 y = calc(x1)  
 browser.find\_element\_by\_css\_selector('#answer').send\_keys(y)  
 button.click()  
 alert = browser.switch\_to.alert  
 answer = alert.text.split()[-1]  
 print(answer)  
 alert.accept()  
 # авторизуемся на Степике  
 browser.get('https://stepik.org/catalog?auth=login&language=ru')  
  
 browser.find\_element\_by\_id('id\_login\_email').send\_keys('login') # здесь вводится e-mail  
 browser.find\_element\_by\_id('id\_login\_password').send\_keys('password') # здесь вводится пароль  
  
 browser.find\_element\_by\_class\_name('sign-form\_\_btn').click()  
 browser.get('https://stepik.org/lesson/181384/step/8?unit=156009')  
  
 answer\_input = browser.find\_element\_by\_css\_selector('textarea')  
 browser.execute\_script("return arguments[0].scrollIntoView(true);", answer\_input)  
 answer\_input.send\_keys(answer)  
  
 button = browser.find\_element\_by\_class\_name('submit-submission')  
 browser.execute\_script("return arguments[0].scrollIntoView(true);", button)  
 button.click()  
finally:  
 browser.quit()

**3 Модуль – Тестовые фреймворки**

**GIT**

в PyCharm, в нем есть очень удобные инструменты для работы напрямую с GitHub, можно связать, например новый проект в PyCharm с репозиторием на GitHub, для этого, копируйте путь к репозиторию, естественно, создав его предварительно, и переходите во вкладку VCS - Git - Clone... затем, в поле URL вставляете адрес, который берете из GitHub, а в поле Directory указываете путь до папки, в которой будет лежать ваш проект.

После этого скорее всего вас попросят залогиниться на GitHub из pyCharm (но это не точно, возможно, попросят на другой стадии, просто не забывайте логин и пароль от аккаунта), следующая работа очень проста, по right-click в поле проекта у вас появляется дополнительная вкладка в меню: "**Git**", в ней:

* **Git - Add**-- добавить файл (физически этот файл уже может быть создан у вас локально).При создании нового файла, система сразу предлагает использовать эту команду;
* ***Git - Commit*** -- сделать коммит, команда выдает форму с возможностью ввести комментарий;
* **Git - Repository - Push**-- отправить все изменения текущие на сервер.

## Клонируем репозиторий

**Windows:**

[**https://gitforwindows.org/**](https://gitforwindows.org/)—  скачать и установить клиент.

Для начала нужно скопировать к себе только что созданный репозиторий. Для этого применяется термин "склонировать".

На страничке своего проекта нажмите на зеленую кнопку Clone or download и скопируйте адрес из строки:

Изображение выглядит как текст

Автоматически созданное описание

в Windows нужно запустить установленный **git bash**

Чтобы склонировать к себе введите в консоли:

**git clone \*адрес вашего репозитория\***

Чтобы перейти в каталог репозитория, введите:

**cd \*имя репозитория \***

Изображение выглядит как текст

Автоматически созданное описание

В итоге, у вас на компьютере оказался ваш удаленный репозиторий. Давайте сделаем там парочку изменений!

Для тех, кто не смог или не понял -**можно сделать еще проще**. Скачав и установив Git, вы просто заходите в искомую деректорию (где лежать будет проект) и жмете ПКМ > Git Bash Here.

Введите в консоль ключевое слово **git**, чтобы увидеть популярные команды

Если вы хотите посмотреть подсказку по какой-то конкретной команде, то можете ввести **git help <команда>** или **git <команда> --help**. Вы увидите подробную документацию по этой команде из руководства пользователя.

Чтобы Git отслеживал изменения в ваших файлах, нужно их "добавить". Это специальная команда, которая говорит Git, на какие файлы нужно смотреть и записывать их изменения, а все остальные файлы будут игнорироваться.

Для того чтобы добавить файлы под бдительный взор Git, нужно выполнить команду:

**git add README.md**

имена файлов зависят от регистра (case sensitive)!

Попробуйте выполнить **git status**и посмотрите, что изменилось в выводе.

Для того чтобы зафиксировать и сохранить свою работу нужно выполнить "коммит". Коммит — это небольшой кусочек вашей работы. Хорошей практикой считается делать коммиты не слишком маленькими (не на каждую новую написанную строку), но и не слишком большими (всю недельную работу сливать одним коммитом — не лучшая идея). Старайтесь добавлять в коммит одну небольшую функцию или класс.

Еще к коммитам пишут короткие сообщения, описывающие изменения — постарайтесь писать их как можно более осмысленными. Просто представьте, что их будут читать ваши коллеги  (или вы сами через год) в попытках разобраться, что вы сделали. Сообщение добавляется с помощью флага **-m**.

Чтобы сделать коммит, нужно ввести команду:

**git commit -m "тут ваше сообщение о коммите"**

(обратите внимание на кавычки!)

Если все прошло как надо, в ответе вы увидите, сколько строк и файлов изменилось:

Изображение выглядит как текст

Автоматически созданное описание

Ура, вы сделали первый коммит!

Попробуйте снова выполнить **git status**  и посмотрите на вывод.

Чтобы посмотреть историю коммитов, можно вывести **git log**:

Изображение выглядит как текст

Автоматически созданное описание

Перед началом работы, лучше сразу же сконфигурировать имя и e-mail:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Чтобы в жизни не париться по поводу добавления файлов , можно просто делать git commit -a -m "",чтобы файлы сразу добавлялись

добавлять нужно каждый раз - так мы сообщаем какие именно измнения (в каких именнно файлах) мы хотим зафиксировать. Я рекомендую делать это руками на первых порах, потом когда освоитесь и поймете что нужно добавлять и как игнорировать файлы, можно использовать **git commit -a :**этот флаг автоматически добавляет все изменения в коммит.

## Добавление изменений на сервер (push)

Сейчас у вашего репозитория есть две разные копии — одна локальная, которая уже содержит изменения в файле, и удаленная — на гитхабе. Необходимо наши локальные коммиты положить в удаленный репозиторий. Для этого есть специальная команда **git push <репозиторий><название ветки>**.

Сейчас мы не будем вдаваться подробно в тему ветвления. Достаточно знать, что основная ветка, на которой вы находитесь по умолчанию — это **master**. Мы будем пушить в удаленный репозиторий **origin**— оригинальный репозиторий, откуда мы скопировали к себе на компьютер локальную версию.  
Обратите внимание, что основная ветка в репозитории будет называться **master**, если он был создан до осени 2020 года. Увидеть это можно в Git Bash после перехода в папку проекта.  
  
В репозитиориях, которые были созданы позднее, основная ветка по умолчанию может называться **main**.  


Выполним команду:

Выполним команду:

**git push origin master**

Или **git push origin main** для новых репозиториев.  
  
Git попросит ввести ваш логин и пароль на GitHub, и покажет примерно следующее:

Изображение выглядит как текст

Автоматически созданное описание

Это сообщение о том, сколько данных откуда и куда отправились.

Теперь откройте свой репозиторий на гитхабе в браузере. Если вы все сделали правильно, то гитхаб подтянет описание проекта из файла и красиво отобразит на страничке:

## Добавьте в репозиторий задачи из предыдущего модуля

Чтобы добавить много файлов за раз испльзуйте:

git add lesson\*.py

 lesson\*.py - название файлов у меня, запись означает, что файлы должны начинаться с "lesson" и заканчиваться на ".py", а \* означает, что в том месте может быть что угодно

<https://desktop.github.com/>

## **Тестирование web-приложений и тестовые фреймворки**

Любой тест должен содержать:

1. Входные данные.
2. Тестовый сценарий, то есть набор шагов, которые надо выполнить для получения результата.
3. Проверка ожидаемого результата.

## Проверка ожидаемого результата

Как можно проверить ожидаемый результат? Для этого используется встроенная в Python инструкция **assert**, которая проверяет истинность утверждений. **assert True** не приводит к выводу дополнительных сообщений, а вот **assert False** вызовет исключение **AssertionError**.

Рассмотрим работу assert на примере встроенной функции **abs**(), которая возвращает абсолютное значение числа по модулю. Для этого активируйте созданное ранее виртуальное окружение и запустите интерпретатор Python. Например, для Linux выполните:

source selenium\_env/bin/activate

python

Теперь будем вводить приведенные ниже команды и смотреть на результат их выполнения.

Если значение выражения истинно, то в консоли не должно появиться дополнительных сообщений. Выполним:

>>> assert abs(-42) == 42

Если условие не выполнено, то в консоли выводится лог ошибки с названием файла и номером строчки, в которой произошла ошибка, а также тип ошибки **AssertionError**:

>>> assert abs(-42) == -42

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AssertionError

Простое сообщение **AssertionError** не очень информативно. Когда тестов становится много, бывает сложно вспомнить, что именно мы проверяем в данном тесте. Для добавления дополнительного сообщения можно при вызове assert через запятую написать нужное сообщение, которое будет выведено в случае ошибки проверки результата:

>>> assert abs(-42) == -42, "Should be absolute value of a number"

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AssertionError: Should be absolute value of a number

Читаемость имеет значение. Не злоупотребляйте DRY (правило «не повторяйся»). Повторение хорошо, если улучшает читаемость. Попробуйте найти баланс между кодом [DRY и DAMP](https://stackoverflow.com/questions/6453235/what-does-damp-not-dry-mean-when-talking-about-unit-tests) (DAMP — Descriptive And Meaningful Phrases, содержательные и осмысленные фразы).

## **Составные сообщения об ошибках**

Если элемент встречается на нескольких страницах приложения, не лишним будет указать, где именно произошла ошибка:

assert self.is\_element\_present('new\_announcement\_button', timeout=30), "No new announcement button on profile page"

### Форматирование строк с помощью конкатенации

В питоне такое можно провернуть с помощью конкатенации строк, например:

actual\_result = "abrakadabra"

print("Wrong text, got " + actual\_result + ", something wrong")

Но из-за обилия кавычек, знаков сложения и вот этого всего этот способ не самый удобный и читается тоже плохо.

### Форматирование строк с помощью str.format

Гораздо лучше воспользоваться возможностью python для форматирования строк. Дополнительно можно почитать здесь: [https://realpython.com/python-string-formatting/#2-new-style-string-formatting-strformat](https://realpython.com/python-string-formatting/" \l "2-new-style-string-formatting-strformat" \t "_blank)

Если вкратце, то python умеет подставлять пользовательские значения в строки с помощью функции **.format()**. Синтаксис выглядит примерно так:

"Let's count together: {}, then goes {}, and then {}".format("one", "two", "three")

Попробуйте запустить её в интерпретаторе:

print("Let's count together: {}, then goes {}, and then {}".format("one", "two", "three"))

Такая строка при исполнении кода превратится в:

Let's count together: one, then goes two, and then three

Таким образом мы можем удобно компоновать ожидаемое и фактическое значение в одну строку.

### Форматирование строк с помощью f-strings

И наконец наиболее современный способ форматирования строк, который появился в Python3.6, носит название f-strings. Он позволяет исполнять выражения на Python прямо внутри строк, обладает еще большей лаконичностью и удобством использования. Для использования возможностей f-strings нужно указывать символ f перед строкой в таком формате: f"ваша строка {my\_var}". В фигурных скобках указывается имя переменной, значение которой надо подставить в строку, или выражение, результат исполнения которого также требуется подставить в вашу строку.

Подробнее про f-strings можно почитать здесь: [https://realpython.com/python-string-formatting/#3-string-interpolation-f-strings-python-36](https://realpython.com/python-string-formatting/" \l "3-string-interpolation-f-strings-python-36" \t "_blank). Так как мы предполагаем, что вы используете последнюю версию Python, то предлагаем вам применять именно этот подход в данном курсе.

Пример 1:

str1 = "one"

str2 = "two"

str3 = "three"

print(f"Let's count together: {str1}, then goes {str2}, and then {str3}")

Итог выполнения выражений в интерпретаторе:

Let's count together: one, then goes two, and then three

Пример 2:

actual\_result = "abrakadabra"

f"Wrong text, got {actual\_result}, something wrong"

Итог выполнения выражений в интерпретаторе:

Wrong text, got abrakadabra, something wrong

Пример 3:

>>> f"{2+3}"

'5'

Еще один важный момент: когда вы работаете с текстом элементов на странице или любым другим контентом, который может измениться, всегда записывайте его в отдельную переменную для сравнения.

**неправильно:**

assert self.catalog\_link.text == "Каталог", \

f"Wrong language, got {self.catalog\_link.text} instead of 'Каталог'"

Дважды считывать атрибут — это плохая практика, потому что при повторном считывании текст на странице может измениться, и вы получите неактуальный текст об ошибке. Результат выполнения такого теста сложно анализировать:

"Wrong language, got 'Каталог' instead of 'Каталог'"

**правильно:**

catalog\_text = self.catalog\_link.text # считываем текст и записываем его в переменную

assert catalog\_text == "Каталог", \

f"Wrong language, got {catalog\_text} instead of 'Каталог'"

## Задание: составные сообщения об ошибках

Для закрепления материала реализуйте проверку самостоятельно.

Вам дана функция **test\_input\_text**,  которая принимает два значения: **expected\_result** — ожидаемый результат, и **actual\_result** — фактический результат. Обратите внимание, *input* использовать **не нужно**!

Функция должна проверить совпадение значений с помощью оператора **assert** и, в случае несовпадения, предоставить исчерпывающее сообщение об ошибке.

**Важно!** Формат ошибки должен точно совпадать с приведенным в примере, чтобы его засчитала проверяющая система!

Маленький совет: попробуйте воспользоваться кнопкой "Запустить код" и протестируйте ваш код на разных введенных значениях, проверьте вывод вашей функции на разных парах. Обрабатывать ситуацию с пустым или невалидным вводом не нужно.

**Sample Input 1:**

8 11

**Sample Output 1:**

expected 8, got 11

**Sample Input 2:**

11 11

**Sample Output 2:**

**Sample Input 3:**

11 15

**Sample Output 3:**

expected 11, got 15

expected\_result = 8

actual\_result = 11

def test\_input\_text(expected\_result, actual\_result):

# ваша реализация, напишите assert и сообщение об ошибке

assert expected\_result == actual\_result, \

f"expected {expected\_result}, got {actual\_result}"

## Задание: составные сообщения об ошибках и поиск подстроки

Иногда при работе с текстами не нужны жёсткие проверки на полное совпадение, и требуется проверить, что некий текст является подстрокой другого текста. Это можно сделать либо с помощью ключевого слова **in**, либо с помощью функции **find**:

s = 'My Name is Julia'

if 'Name' in s:

print('Substring found')

index = s.find('Name')

if index != -1:

print(f'Substring found at index {index}')

Попробуйте запустить этот код в интерпретаторе, чтобы понять разницу в подходах.

Конструкция **'Name' in s** возвращает просто **True** или **False**, a **find()** возвращает индекс первого вхождения подстроки в строку и -1, если подстрока не найдена. Обычно в автотестах достаточно использовать **in**, потому что это более читабельный вариант.

Например, для проверки того, что в текущем url содержится строка login:

assert "login" in browser.current\_url, # сообщение об ошибке

Реализуйте подобную проверку самостоятельно.

Вам дан шаблон для функции **test\_substring**, которая принимает два значения: **full\_string** и **substring**.

Функция должна проверить вхождение строки **substring** в строку **full\_string**с помощью оператора **assert** и, в случае несовпадения, предоставить исчерпывающее сообщение об ошибке.

**Важно!** Формат ошибки должен точно совпадать с приведенным в примере, чтобы его засчитала проверяющая система!

Маленький совет: попробуйте воспользоваться кнопкой "Запустить код" и протестируйте ваш код на разных введенных значениях, проверьте вывод вашей функции на разных парах. Обрабатывать ситуацию с пустым или невалидным вводом не нужно.

**Sample Input 1:**

fulltext some\_value

**Sample Output 1:**

expected 'some\_value' to be substring of 'fulltext'

**Sample Input 2:**

1 1

**Sample Output 2:**

**Sample Input 3:**

some\_text some

**Sample Output 3:**

def test\_substring(full\_string, substring):

# ваша реализация, напишите assert и сообщение об ошибке

assert substring in full\_string, \

f"expected '{substring}' to be substring of '{full\_string}'"

## Тестовые сценарии

def test\_abs1():

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

if \_\_name\_\_ == "\_\_main\_\_":

test\_abs1()

print("All tests passed!")

Конструкция if \_\_name\_\_ == "\_\_main\_\_" служит для подтверждения того что данный скрипт был запущен напрямую, а не вызван внутри другого файла в качестве модуля. Весь код написанный в теле этого условия будет выполнен только если пользователь запустил файл самостоятельно.

## Выбор test runner

## unittest

Тест-раннеры сами находят тестовые методы в указанных при запуске файлах, но для этого нужно следовать общепринятым правилам. Общее правило для всех фреймворков: название тестового метода должно начинаться со слова "test\_".  Дальше может идти любой текст, который является уникальным названием для теста:

def test\_name\_for\_your\_test():

Для unittest существуют собственные дополнительные правила:

* Тесты обязательно должны находиться в специальном тестовом классе.
* Вместо assert должны использоваться специальные assertion методы.

Давайте теперь изменим наши предыдущие тесты, чтобы их можно было запустить с помощью unittest. Для этого нам понадобится выполнить следующие шаги:

1. Импортировать unittest в файл: **import unittest**
2. Создать класс, который должен наследоваться от класса TestCase: **class TestAbs(unittest.TestCase):**
3. Превратить тестовые функции в методы, добавив ссылку на экземпляр класса self в качестве первого аргумента функции: **def test\_abs1(self):**
4. Изменить assert на **self.assertEqual()**
5. Заменить строку запуска программы на **unittest.main()**

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()

После изменений запустим наш файл с тестами всё так же с помощью Python:

python test\_abs\_project.py

.F

======================================================================

FAIL: test\_abs2 (\_\_main\_\_.TestAbs)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "test\_abs\_project.py", line 9, in test\_abs2

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

AssertionError: Should be absolute value of a number

----------------------------------------------------------------------

Ran 2 tests in 0.000s

FAILED (failures=1)

Теперь мы видим более подробную информацию о результатах запуска: было запущено два теста, один тест выполнился с ошибкой. Место ошибки и пояснение к ней отображаются в логе.

**3.3 Тестирование с помощью PyTest**

## PyTest — преимущества и недостатки

Рассмотрим преимущества использования PyTest:

1) PyTest полностью обратно совместим с фреймворками unittest и nosetest. Это означает, что если изначально вы писали тесты, используя unittest, то перейти на PyTest можно буквально в ту же минуту. Для этого в вашем виртуальном окружении должен быть установлен пакет PyTest. Не забудьте активировать ваше виртуальное окружение и установите PyTest.

**Для Windows:**

> selenium\_env\Scripts\activate.bat

(selenium\_env) С:\Users\user\environments> pip install pytest==5.1.1

**Для Linux и macOS:**

​​​​​​​$ source selenium\_env/bin/activate

(selenium\_env) $ pip install pytest==5.1.1

Теперь мы можем запустить тесты в нашем файле *test\_abs\_project.py*с помощью PyTest, не изменяя сам файл. PyTest сам найдёт тесты в папке, в которой вы их запускаете, и выполнит их:

pytest test\_abs\_project.py

2) Подробный отчёт с поддержкой цветовых схем из коробки.

3) PyTest не требует написания дополнительных специфических конструкций в тестах, как того требует unittest (no boilerplate).

4) Для проверок используется стандартный assert из Python.

5) Возможность создания динамических фикстур (специальных функций, которые настраивают тестовые окружения и готовят тестовые данные).

6) Дополнительные возможности по настройке фикстур.

7) Параметризация тестов — для одного теста можно задать разные параметры (тест запустится несколько раз с разными тестовыми данными).

8) Наличие маркировок (marks), которые позволяют маркировать тесты для их выборочного запуска.

9) Возможность передавать дополнительные параметры через командную строку для настройки тестовых окружений.

10) Большое количество плагинов, которые расширяют возможности PyTest и позволяют решать узкоспециализированные проблемы, что может сэкономить много времени.

Рассмотрим минусы PyTest:

1) PyTest требуется устанавливать дополнительно, так как он не входит в стандартный пакет библиотек Python, в отличие от unittest. Нужно не забывать об этом, когда вы будете настраивать автоматический запуск тестов с помощью CI-сервера.

2) Использование PyTest требует более глубокого понимания языка Python, чтобы разобраться, как применять фикстуры, параметризацию и другие возможности PyTest.

## Фиксируем пакеты в requirements.txt

Откройте терминал, перейдите в директорию, в которой вы работаете с автотестами, и активируйте виртуальное окружение.

После чего выполните в терминале команду:

pip freeze > requirements.txt

Эта команда сохранит все версии пакетов в специальный файл **requirements.txt.**

Как их оттуда достать? Попробуйте создать новое виртуальное окружение и активировать. После чего выполните команду:

pip install -r requirements.txt

В свежем окружении все пакеты установлены одной командой!

## **PyTest: правила запуска тестов**

Bажные особенности запуска тестов с помощью PyTest. Когда мы выполняем команду **pytest**, тест-раннер собирает все тесты для запуска по определенным правилам:

* если мы не передали никакого аргумента в команду, а написали просто pytest, тест-раннер начнёт поиск в текущей директории
* как аргумент можно передать файл, путь к директории или любую комбинацию директорий и файлов, например:

pytest scripts/selenium\_scripts

# найти все тесты в директории scripts/selenium\_scripts

pytest test\_user\_interface.py

# найти и выполнить все тесты в файле

pytest scripts/drafts.py::test\_register\_new\_user\_parametrized

# найти тест с именем test\_register\_new\_user\_parametrized в указанном файле в указанной директории и выполнить

* дальше происходит рекурсивный поиск: то есть PyTest обойдет все вложенные директории
* во всех директориях PyTest ищет файлы, которые удовлетворяют правилу  **test\_\*.py** или **\*\_test.py** (то есть начинаются на test\_ или заканчиваются \_test и имеют расширение .py)
* внутри всех этих файлов находит тестовые функции по следующему правилу:
  + все тесты, название которых начинается с **test**, которые находятся вне классов
  + все тесты, название которых начинается с **test** внутри классов, имя которых начинается с **Test** (и без метода \_\_init\_\_ внутри класса)

## PyTest — отчёты

Если запустить PyTest с параметром **-v** (**verbose**, то есть подробный), то в отчёт добавится дополнительная информация со списком тестов и статусом их прохождения:

pytest -v test\_3.py

## PyTest — как пишут тесты

PyTest не требует написания дополнительных специфических конструкций в тестах, как того требует unittest.

Мы уже увидели, что PyTest может запускать тесты, написанные в unittest-стиле. Перепишем наши тесты из **test\_abs\_project.py** в более простом формате, который также понимает PyTest. Назовём новый файл test\_abs.py:

def test\_abs1():

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

def test\_abs2():

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

Запустим тесты в этом файле:

pytest test\_abs.py

Код тестов стал короче и читабельнее.

pytest -v --tb=line test\_abs.py

## PyTest — проверка ожидаемого результата (assert)

Если вы используете unittest, то для проверки ожидаемых результатов в тестах вам нужно знать и использовать большой набор assert-методов, например, таких: assertEqual, assertNotEqual, assertTrue, assertFalse и [другие](https://docs.python.org/3/library/unittest.html" \l "assert-methods%EF%BB%BF" \t "_blank).

В PyTest используется стандартный assert метод из языка Python, что делает код более очевидным.

Давайте сравним два подхода. Проверим, что две переменные равны друг другу.

**unittest:**

self.assertEqual(a, b, msg="Значения разные")

**PyTest:**

assert a == b, "Значения разные"

С помощью assert можно проверять любую конструкцию, которая возвращает True/False. Это может быть проверка равенства, неравенства, содержания подстроки в строке или любая другая вспомогательная функция, которую вы опишете самостоятельно. Все это делает код проверок приятным и понятным для чтения:

assert user\_is\_authorised(), "User is guest"

Если нужно проверить, что тест вызывает ожидаемое исключение (довольно редкая ситуация для UI-тестов, и вам этот способ, скорее всего, никогда не пригодится), мы можем использовать специальную конструкцию **with pytest.raises()**. Например, можно проверить, что на странице сайта не должен отображаться какой-то элемент:

import pytest

from selenium import webdriver

from selenium.common.exceptions import NoSuchElementException

def test\_exception1():

try:

browser = webdriver.Chrome()

browser.get("http://selenium1py.pythonanywhere.com/")

with pytest.raises(NoSuchElementException):

browser.find\_element\_by\_css\_selector("button.btn")

pytest.fail("Не должно быть кнопки Отправить")

finally:

browser.quit()

def test\_exception2():

try:

browser = webdriver.Chrome()

browser.get("http://selenium1py.pythonanywhere.com/")

with pytest.raises(NoSuchElementException):

browser.find\_element\_by\_css\_selector("no\_such\_button.btn")

pytest.fail("Не должно быть кнопки Отправить")

finally:

browser.quit()

В первом тесте элемент будет найден, поэтому ошибка **NoSuchElementException**, которую ожидает контекстный менеджер pytest.raises, не возникнет, и тест упадёт.

test\_3\_3\_9\_pytest\_raises.py:8 (test\_exception1)

E Failed: Не должно быть кнопки Отправить

Во втором тесте, как мы и ожидали, кнопка не будет найдена, и тест пройдет.

3.4 Использование фикстур в PyTest

## 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](https://en.wikipedia.org/wiki/Test_fixture" \l "Software" \t "_blank).

Классический способ работы с фикстурами — создание setup- и teardown-методов в файле с тестами ([документация в PyTest](https://docs.pytest.org/en/latest/how-to/xunit_setup.html?highlight=teardown" \t "_blank)).

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:**

from selenium import webdriver

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

class TestMainPage1():

@classmethod

def setup\_class(self):

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

self.browser = webdriver.Chrome()

@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()

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")

В консоли видим:

Изображение выглядит как текст

Автоматически созданное описание

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.

флаг -s является кратким вариантом опции --capture=no. По умолчанию стандартный ввод/вывод и вывод ошибок перехватывается pytest, и не отображается в выводе интерпретатора. Благодаря --capture=no мы можем увидеть в данном примере результат работы функции print.

1) фикстуры - это просто функции, которые выполняются перед\после  запуском тест кейса(очищение кэша, регистрация и т.п.).  
2) @classmethod - ну вообще говоря, это декоратор. Просто это зарезервированный декоратор, чтобы сделать метод методом класса, т.е. результат работы этого метода не зависит от того какой объект его вызвал(в качестве житейского примера, это могут быть функции для генерации какого-нибудь рандомного значения, переходы кодировок, переводы и т.п.)

Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода. Вот почему декораторы можно рассматривать как практику метапрограммирования, когда программы могут работать с другими программами как со своими данными.

## 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

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

@pytest.fixture

def browser():

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

browser = webdriver.Chrome()

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")

Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.

обновление pytest на более новую версию. Для этого в виртуальном окружении выполнить команду:

pip install --upgrade pytest --user

## 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

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

@pytest.fixture

def browser():

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

browser = webdriver.Chrome()

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" \l "adding-finalizers-directly" \t "_blank).

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\_fixture5.py

import pytest

from selenium import webdriver

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

@pytest.fixture(scope="class")

def browser():

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

browser = webdriver.Chrome()

yield browser

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.

## 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

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

@pytest.fixture

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

@pytest.fixture(autouse=True)

def prepare\_data():

print()

print("preparing some critical data for every test")

class TestMainPage1():

def test\_guest\_should\_see\_login\_link(self, browser):

# не передаём как параметр фикстуру prepare\_data, но она все равно выполняется

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")

Изображение выглядит как текст

Автоматически созданное описание

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.

3.5 PyTest — маркировка

## 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

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.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]

markers =

smoke: marker for smoke tests

regression: marker for regression tests

win10

**test\_fixture81.py:**

import pytest

from selenium import webdriver

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**:

import pytest

from selenium import webdriver

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)

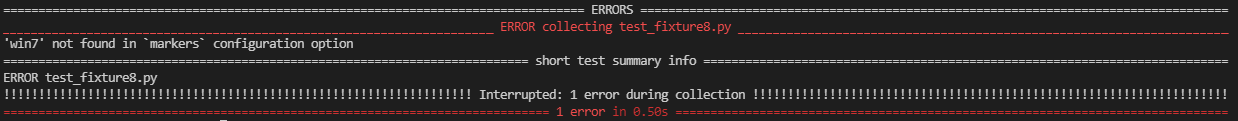
browser.find\_element\_by\_css\_selector(".basket-mini .btn-group > a")

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

\_\_\_\_\_

Если маркировка skip добавляется к функции, где уже есть другие маркировки, то skip должен быть последним маркером, иначе пропускаться не будет.   
@pytest.mark.regression

@pytest.mark.win10

@pytest.mark.skip   
  
Также можно добавить строку "addopts = --strict-markers"  в pytest.ini, благодаря ей при указании маркера отсутствующего в файле мы будем проинформированы ошибкой.

[pytest]

**addopts = --strict-markers**

markers =

    smoke: marker for smoke tests

    regression: marker for regression tests

    win 10: only for win10 OS

\_\_\_

Еще для пропущенного теста можно оставлять комментарий. Например, @pytest.mark.skip(reason="Reason to skip test"). Получим:  
  
test\_fixture\_marks\_skip.py::TestMainPage1::test\_guest\_should\_see\_login\_link SKIPPED (Reason to skip test)

\_\_\_\_\_

## 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

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:

Изображение выглядит как текст

Автоматически созданное описание

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

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.

Изображение выглядит как текст

Автоматически созданное описание

**XPASS-тесты**

Поменяем селектор в последнем тесте, чтобы тест начал проходить.

**test\_fixture10b.py:**

import pytest

from selenium import webdriver

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:

Изображение выглядит как текст

Автоматически созданное описание

Дополнительно об использовании этих меток можно почитать в документации: [Skip and xfail: dealing with tests that cannot succeed](https://pytest.org/en/stable/skipping.html" \t "_blank).  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.

## Задание: пропуск тестов

Изучите самостоятельно документацию про маркировку [xfail](https://docs.pytest.org/en/latest/reference/reference.html?highlight=xfail" \l "pytest.mark.xfail" \t "_blank). Найдите там параметр, который в случае неожиданного прохождения теста, помеченного как xfail, отметит в отчете этот тест как упавший. Пометьте таким образом первый тест из этого тестового набора.

**test\_xfail.py:**

import pytest  
  
@pytest.mark.xfail(strict=bool)  
def test\_succeed():  
 assert True  
  
  
@pytest.mark.xfail  
def test\_not\_succeed():  
 assert False  
  
  
@pytest.mark.skip  
def test\_skipped():  
 assert False

3.6 PyTest — параметризация, конфигурирование, плагины

## 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

@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

Для тех, кто к этому шагу успел забыть ключи, как я:

*-v - выводить отчет более подробно*

-s - про него тут https://stepik.org/lesson/237240/step/2?discussion=1061080&unit=209628

если просто, то -s показывает в консоли выводы print'ов.

 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.

Изображение выглядит как текст

Автоматически созданное описание

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](https://docs.pytest.org/en/latest/how-to/parametrize.html?highlight=parametrize" \t "_blank)

---

Например, это может выглядеть вот так:

languages = [

("ru", "русский"),

("de" "немецкий"),

("ua", "украинский"),

("en-gb", "английский")

]

@pytest.mark.parametrize("code, lang", languages)

def test\_guest\_should\_see\_login\_link(browser, code, lang)

link = "http://selenium1py.pythonanywhere.com/{}/".format(code)

print("Проверяемый язык %s" % lang)

---

## Задание: параметризация тестов

Инопланетяне оставляют загадочные сообщения на Stepik в фидбеке задач на правильное решение. Мы смогли локализовать несколько url-адресов задач, где появляются кусочки сообщений. Ваша задача — реализовать автотест со следующим сценарием действий:

* открыть страницу
* ввести правильный ответ
* нажать кнопку "Отправить"
* дождаться фидбека о том, что ответ правильный
* проверить, что текст в опциональном фидбеке полностью совпадает с **"Correct!"**

Опциональный фидбек — это текст в черном поле, как показано на скриншоте:

Изображение выглядит как текст

Автоматически созданное описание

Правильным ответом на задачу в заданных шагах является число:

import time

import math

answer = math.log(int(time.time()))

Используйте маркировку **pytest** для параметризации и передайте в тест список ссылок в качестве параметров:

https://stepik.org/lesson/236895/step/1  
https://stepik.org/lesson/236896/step/1  
https://stepik.org/lesson/236897/step/1  
https://stepik.org/lesson/236898/step/1  
https://stepik.org/lesson/236899/step/1  
https://stepik.org/lesson/236903/step/1  
https://stepik.org/lesson/236904/step/1  
https://stepik.org/lesson/236905/step/1

Используйте осмысленное сообщение об ошибке в проверке текста, а также настройте нужные ожидания, чтобы тесты работали стабильно.

В упавших тестах найдите кусочки послания. Тест должен падать, если текст в опциональном фидбеке не совпадает со строкой **"Correct!"** Соберите кусочки текста в одно предложение и отправьте в качестве ответа на это задание.

**Важно!**Чтобы пройти это задание, дополнительно убедитесь в том, что у вас установлено правильное локальное время ([https://time.is/ru/](https://time.is/ru/" \t "_blank)). Ответ для каждой задачи нужно пересчитывать отдельно, иначе они устаревают.

Answers

from selenium import webdriver  
import pytest  
import time  
import math  
  
final = ''  
  
  
@pytest.fixture(scope="session")  
def browser():  
 br = webdriver.Chrome()  
 yield br  
 br.quit()  
 print(final) # напечатать ответ про Сов в конце всей сессии  
  
  
@pytest.mark.parametrize('lesson', ['236895', '236896', '236897', '236898', '236899', '236903', '236904', '236905'])  
def test\_find\_hidden\_text(browser, lesson):  
 global final  
 link = f'https://stepik.org/lesson/{lesson}/step/1'  
 browser.implicitly\_wait(10)  
 browser.get(link)  
 answer = math.log(int(time.time()))  
 browser.find\_element\_by\_css\_selector('textarea').send\_keys(str(answer))  
 browser.find\_element\_by\_css\_selector('.submit-submission ').click()  
 check\_text = browser.find\_element\_by\_css\_selector('.smart-hints\_\_hint').text  
 try:  
 assert 'Correct!' == check\_text  
 except AssertionError:  
 final += check\_text # собираем ответ про Сов с каждой ошибкой

Other

import pytest  
from selenium import webdriver  
import time  
import math  
from selenium.webdriver.support import expected\_conditions as EC  
from selenium.webdriver.support.ui import WebDriverWait  
from selenium.webdriver.common.by import By  
massiv = ['236895', '236896','236897', '236898', '236899', '236903', '236904', '236905']  
final = []  
@pytest.fixture(scope="function")  
def browser():  
 browser=webdriver.Chrome()  
 yield browser  
 browser.quit()  
 print(final)  
@pytest.mark.parametrize('number', massiv)  
def test(browser, number):  
 answer = math.log(int(time.time()))  
 browser.get(f'https://stepik.org/lesson/{number}/step/1')  
 wait = WebDriverWait(browser, 10)  
 wait.until(EC.element\_to\_be\_clickable((By.CSS\_SELECTOR, 'textarea')))  
 browser.find\_element\_by\_css\_selector('textarea').send\_keys(str(answer))  
 browser.find\_element\_by\_class\_name('submit-submission').click()  
 wait.until(EC.presence\_of\_element\_located((By.CLASS\_NAME, 'smart-hints\_\_feedback')))  
 correct\_checker = browser.find\_element\_by\_class\_name('smart-hints\_\_feedback').text  
 assert correct\_checker == 'Correct!', f"Correct checker fails you, words was {correct\_checker}"

## Conftest.py — конфигурация тестов

## 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

@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.

подскажите, почему браузер не стартует и тест как-будто не ранится через пишарм, а через консоль всё ок ?

ReplyHide replies

[](https://stepik.org/users/60210280)

[Павел Бобров](https://stepik.org/users/60210280)

2 years ago[Link](https://stepik.org/lesson/237240/step/4?discussion=1529245&reply=1530074&unit=209628" \o "Link to this reply)

**@Anastasia\_Silaeva**, вероятно, нужно поменять rest runner в настройках pycharm.  
File -> Settings - > Tools - > Python Integrated tools -> строка Testing  
  
Также попробуй удалить все "пребилды": Run -> Edit Configuration -> Python tests -> удалить все записи из вкладки

 2

Reply

[](https://stepik.org/users/21925106)

[Anastasia Silaeva](https://stepik.org/users/21925106)

2 years ago[Link](https://stepik.org/lesson/237240/step/4?discussion=1529245&reply=1530146&unit=209628" \o "Link to this reply)

**@Павел\_Бобров**, я уже разобралась , спасибо.  
там нужно было делать Run 'pytest in имя\_файла.py'

## Установка Firefox и Selenium-драйвера geckodriver

До этого момента мы запускали наши тесты только в браузере Chrome, но что делать, если нужно тестировать наше веб-приложение и в других браузерах? При этом мы будем запускать те же тесты, но при запуске тестов указывать, на каком браузере нужно запускать тесты. Возьмем в качестве второго браузера Firefox, так как он является вторым по популярности браузером, и его можно запустить на любой платформе. Запускать тесты мы хотим, указывая при запуске параметр browser\_name, такой командой:

pytest -s -v --browser\_name=firefox test\_cmd.py

Сейчас нам придется вспомнить муки установки chromedriver из урока <https://stepik.org/lesson/25969/> и повторить похожий сценарий установки браузера Firefox и Selenium-драйвера для него.

Для установки Firefox скачайте его с официального сайта и установите в вашей ОС: [https://www.mozilla.org/firefox/new/](https://www.mozilla.org/firefox/new/" \t "_blank).

Selenium-драйвер для Firefox носит название geckodriver. Скачайте последнюю версию geckodriver с сайта [https://github.com/mozilla/geckodriver/releases](https://github.com/mozilla/geckodriver/releases" \t "_blank) и распакуйте его в папку C:\geckodriver на Windows, /usr/local/bin на Ubuntu и macOS. Для более подробной инструкции по установке geckodriver смотрите [https://selenium-python.com/install-geckodriver](https://selenium-python.com/install-geckodriver" \t "_blank). Для Windows не забудьте добавить в системную переменную PATH папку C:\geckodriver и перезапустить командную строку, чтобы путь стал доступен.

Чтобы проверить правильность установки geckodriver, выполните в интерпретаторе Python команды:

from selenium import webdriver

# инициализируем драйвер браузера. После этой команды вы должны увидеть новое открытое окно браузера

driver = webdriver.Firefox()

driver.get("https://stepik.org/lesson/25969/step/8")

Если вы увидели, как запустилось новое окно браузера Firefox и открылась указанная ссылка, то можете переходить к следующему шагу.

Если при попытке выполнения кода вы увидели подобное сообщение:

selenium.common.exceptions.WebDriverException: Message: 'geckodriver' executable needs to be in PATH.

значит, geckodriver не установлен или к нему не прописан путь в системе. Повторите заново действия по установке. Если Firefox всё равно не запускается, то напишите в комментариях последовательность ваших действий и подробный лог ошибки, чтобы мы могли вам помочь.

[Danylo Kholodov](https://stepik.org/users/18039284)

2 years ago[Link](https://stepik.org/lesson/237240/step/5?discussion=1238424&unit=209628" \o "Link to this discussion)

Чтобы избавить себя от мучений, очень советую использовать библиотеку **[webdriver\_manager](https://github.com/SergeyPirogov/webdriver_manager" \t "_blank),**и не нужно вручную мучиться с установкой драйверов, все будет сделано за Вас.

Вот пример:

pip install webdriver\_manager

from webdriver\_manager.firefox import GeckoDriverManager

driver = webdriver.Firefox(executable\_path=GeckoDriverManager().install())

## 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](https://docs.pytest.org/en/latest/example/simple.html?highlight=addoption" \t "_blank)

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.

Немного по своему переписал, чтобы упростить добавление новых браузеров, мб кому пригодится.

import pytest

from selenium import webdriver

supported\_browsers = {

'chrome': webdriver.Chrome,

'firefox': webdriver.Firefox

}

def pytest\_addoption(parser):

parser.addoption('--browser\_name', action='store', default='chrome',

help="Choose browser: chrome or firefox")

@pytest.fixture(scope="function")

def browser(request):

browser\_name = request.config.getoption("browser\_name")

if browser\_name in supported\_browsers:

browser = supported\_browsers.get(browser\_name)()

print(f"\nstart {browser\_name} browser for test..")

else:

joined\_browsers = ', '.join(supported\_browsers.keys())

raise pytest.UsageError(f"--browser\_name is invalid, supported browsers: {joined\_browsers}")

yield browser

print("\nquit browser..")

browser.quit()

## Плагины и перезапуск тестов

Для PyTest написано большое количество [плагинов](https://docs.pytest.org/en/latest/explanation/flaky.html?highlight=plugins" \l "plugins" \t "_blank" \o "Link: https://docs.pytest.org/en/latest/plugins.html), 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" \t "_blank).

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" \l "modifying-python-traceback-printing" \t "_blank):

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:

Graphical user interface, text, application, email

Description automatically generated

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.

Conftest

import pytest  
from selenium import webdriver  
from selenium.webdriver.chrome.options import Options  
  
  
def pytest\_addoption(parser):  
 parser.addoption('--browser\_name', action='store', default=None,  
 help="Choose browser: chrome or firefox")  
 parser.addoption('--language', action='store', default=None,  
 help="Choose language: ru, en, ... (etc.)")  
  
  
@pytest.fixture(scope="function")  
def browser(request):  
 browser\_name = request.config.getoption("browser\_name")  
 user\_language = request.config.getoption("language")  
 browser = None  
 if browser\_name == "chrome":  
 options = Options()  
 options.add\_experimental\_option('prefs', {'intl.accept\_languages': user\_language})  
 print("\nstart chrome browser for test..")  
 browser = webdriver.Chrome(options=options)  
 elif browser\_name == "firefox":  
 fp = webdriver.FirefoxProfile()  
 fp.set\_preference("intl.accept\_languages", user\_language)  
 print("\nstart firefox browser for test..")  
 browser = webdriver.Firefox(firefox\_profile=fp)  
 else:  
 raise pytest.UsageError("--browser\_name should be chrome or firefox")  
  
 yield browser  
 print("\nquit browser..")  
 browser.quit()

## Задание: запуск автотестов для разных языков интерфейса

Мы хотим, чтобы разрабатываемый нами интернет-магазин работал одинаково хорошо для пользователей из любой страны. Чтобы убедиться в работоспособности решения с поддержкой разных языков, мы планируем запускать набор автотестов для каждого языка. Вам как разработчику автотестов нужно реализовать решение, которое позволит запускать автотесты для разных языков пользователей, передавая нужный язык в командной строке.

1. Создайте GitHub-репозиторий, в котором будут лежать файлы conftest.py и test\_items.py.
2. Добавьте в файл conftest.py обработчик, который считывает из командной строки параметр language.
3. Реализуйте в файле conftest.py логику запуска браузера с указанным языком пользователя. Браузер должен объявляться в фикстуре browser и передаваться в тест как параметр.
4. В файл test\_items.py напишите тест, который проверяет, что страница товара на сайте содержит кнопку добавления в корзину. Например, можно проверять товар, доступный по [http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/](http://selenium1py.pythonanywhere.com/catalogue/coders-at-work_207/" \t "_blank).
5. Тест должен запускаться с параметром language следующей командой:

pytest --language=es test\_items.py

и проходить успешно. Достаточно, чтобы код работал только для браузера Сhrome.

1. Отправить ссылку на данный репозиторий в качестве ответа на данное задание.
2. Отправить решение на рецензирование другим учащимся. Не забудьте, что решение на рецензирование можно отправить только один раз.
3. Проверьте решения минимум трех других учащихся, чтобы получить баллы за задание.

Это задание с peer-review, поэтому кроме формальных критериев другие учащиеся могут проверять корректность написания вашего кода.

Важно! Если при рецензировании чужого решения вы получаете ошибку вроде:

raise ValueError("option names %s already added" % conflict)

ValueError: option names {'--language'} already added

Перепроверьте, что у вас нет своего conftest.py в директории выше, смотри [шаг 4](https://stepik.org/lesson/237240/step/4?unit=209628).

Ваше решение будет проверяться по следующим критериям:

1. Тест в репозитории можно запустить командой pytest --language=es, тест успешно проходит.
2. Проверка работоспособности кода для разных языков. Добавьте в файл с тестом команду**time.sleep(30)** сразу после открытия ссылки. Запустите тест с параметром **--language=fr**и визуально проверьте, что фраза на кнопке добавления в корзину выглядит так: **"Ajouter au panier"**.
3. Браузер должен объявляться в фикстуре **browser** и передаваться в тест как параметр.
4. В тесте проверяется наличие кнопки добавления в корзину. Селектор кнопки является уникальным для проверяемой страницы. Есть **assert**.
5. Название тестового метода внутри файла test\_items.py соответствует задаче. Название test\_something не удовлетворяет требованиям.

Conftest

import pytest  
from selenium import webdriver  
from selenium.webdriver.chrome.options import Options  
  
  
def pytest\_addoption(parser):  
 parser.addoption('--browser\_name', action='store', default="chrome",  
 help="Choose browser: chrome or firefox")  
 parser.addoption('--language', action='store', default=None,  
 help="Choose language: ru, en, ... (etc.)")  
  
  
@pytest.fixture(scope="function")  
def browser(request):  
 browser\_name = request.config.getoption("browser\_name")  
 user\_language = request.config.getoption("language")  
 browser = None  
 if browser\_name == "chrome":  
 options = Options()  
 options.add\_experimental\_option('prefs', {'intl.accept\_languages': user\_language})  
 print("\nstart chrome browser for test..")  
 browser = webdriver.Chrome(options=options)  
 elif browser\_name == "firefox":  
 fp = webdriver.FirefoxProfile()  
 fp.set\_preference("intl.accept\_languages", user\_language)  
 print("\nstart firefox browser for test..")  
 browser = webdriver.Firefox(firefox\_profile=fp)  
 else:  
 raise pytest.UsageError("--browser\_name should be chrome or firefox")  
  
 yield browser  
 print("\nquit browser..")  
 browser.quit()

test file

import time  
from selenium.webdriver.support import expected\_conditions as EC  
from selenium.webdriver.support.ui import WebDriverWait  
from selenium.webdriver.common.by import By  
  
  
def test\_presence\_of\_button(browser):  
 browser.get(" http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/")  
 # time.sleep(30)  
 wait = WebDriverWait(browser, 10)  
 button = wait.until(EC.presence\_of\_element\_located((By.CLASS\_NAME, 'btn-add-to-basket')))  
  
 assert button is not None, "this is wrong selector"

## Полезные ссылки

### Git

* <https://learngitbranching.js.org/> — отличный интерактивный туториал
* <https://git-scm.com/book/ru/v2/> — лучшая книга вообще
* <https://hyperskill.org/learn/topic/257/>﻿
* <https://stepik.org/course/4138/>﻿
* <http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/index.html>
* <https://habr.com/company/intel/blog/344962/>
* <https://githowto.com/ru>

### Тестирование веб-приложений

* <https://realpython.com/python-testing/> — инструменты для тестирования кода в Python
* [Пирамида тестов на практике](https://habr.com/ru/post/358950/)
* [unittest — документация](https://docs.python.org/3/library/unittest.html)

### Тестирование с помощью PyTest

* [Статья про PyTest](https://habr.com/ru/post/269759/)
* [Введение в PyTest](https://medium.com/@dmrlx/%D0%B2%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-pytest-cc6175c7d0dc)
* [Документация PyTest](https://docs.pytest.org/en/latest/)
* [Conventions for Python test discovery](https://docs.pytest.org/en/stable/goodpractices.html)
* [Полезные флаги pytest](https://gist.github.com/amatellanes/12136508b816469678c2)

### Использование фикстур в PyTest

* [Фикстуры — определение](https://en.wikipedia.org/wiki/Test_fixture#Software)
* [Фикстуры в PyTest](https://docs.pytest.org/en/stable/fixture.html)
* [setup и teardown методы](https://docs.pytest.org/en/stable/xunit_setup.html)
* <https://habr.com/ru/company/yandex/blog/242795/>
* <https://medium.com/@dmrlx/%D0%B2%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-pytest-cc6175c7d0dc>
* [Skip and xfail: dealing with tests that cannot succeed](https://pytest.org/en/stable/skipping.html)

### Параметризация, конфигурирование, плагины

* [Parametrizing fixtures and test functions](https://docs.pytest.org/en/stable/parametrize.html)
* [Инструкции по установке geckodriver](https://selenium-python.com/install-geckodriver)
* [Передача параметров в PyTest из командной строки](https://docs.pytest.org/en/stable/example/simple.html?highlight=addoption)
* [Список плагинов PyTest](http://docs.pytest.org/en/stable/plugins.html)
* [Полный список доступных плагинов с описаниями](https://plugincompat.herokuapp.com/)
* [Настройка вывода PyTest](https://docs.pytest.org/en/stable/usage.html#modifying-python-traceback-printing)

**4 Модуль – Применение паттерна Page Object Model**

## Немного о Code Style

Самое важное для нас в будущих шагах, что все функции внутри класса так же должны быть отделены отступом:

Один отступ — это четыре пробела. Табуляции использовать нежелательно (исключение составляют ситуации, когда вы поддерживаете какой-то уже существующий код с табуляциями, в таком случае смешивать табуляции с пробелами не следует).

## Code Style: базовые принципы

### Имена переменных и функций

Одним из самых важных аспектов читаемого кода является именование: будь то объявление переменных, описание функций, названия классов и так далее. Очень важно, чтобы все имена, которые вы присваивали сущностям, были осмысленными и отражали реальную суть этого объекта. Избегайте однобуквенных и бессмысленных названий типа var1, x, y, my\_function, class2 и так далее. Идеальный код — самодокументируемый, к которому не нужны дополнительные пояснения. Если вы чувствуете, что вам хочется написать поясняющий комментарий, это повод переписать код так, чтобы комментарий не понадобился.

Обычно внутри каждой компании есть дополнительные внутренние соглашения о том, как именовать переменные, но общие правила в индустрии примерно одинаковые.

Функции пишутся через\_нижнее\_подчеркивание:

def test\_guest\_can\_see\_lesson\_name\_in\_lesson\_without\_course(self, driver):

Классы пишут с помощью CamelCase:

class TestLessonNameWithoutCourseForGuest():

Константы пишут в стиле UPPERCASE:

MAIN\_PAGE = "/catalog"

### Максимальная простота кода

Здесь нам на помощь приходят известные принципы написания кода [DRY](https://en.wikipedia.org/wiki/Don't_repeat_yourself" \t "_blank) (Don't repeat yourself) и [KISS](https://en.wikipedia.org/wiki/KISS_principle" \t "_blank) (Keep it simple, stupid).

* Пишите максимально простой код везде, где это возможно.
* Не используйте переусложненных конструкций без большой необходимости (поменьше лямбда-выражений, map и разной другой магии). Если кусок кода можно заменить конструкцией более простой для понимания — замените.
* Пишите максимально линейный код, где это возможно, это проще для восприятия.
* Избегайте большой вложенности блоков кода, такие конструкции тяжело читать.
* Если можно вынести повторяющуюся логику куда-то, выносите, не повторяйтесь.
* По возможности пишите явный код вместо неявного. Чем меньше магии "под капотом", тем лучше.

## Code Style в автотестах

Здесь мы попытались собрать важные принципы написания автотестов:

* Стремитесь к максимальной линейности кода тестов: избегайте ветвления и циклов в тест-кейсах. Если хочется добавить в тесте if, значит, нужно разбить этот тест на два теста или подготовить тестовое окружение так, чтобы не было необходимости использовать ветвление.
* Избегайте зависимых тестов, которые нужно запускать строго в определенном порядке. При росте количества автотестов вы захотите запускать их в несколько потоков параллельно, что будет невозможно при наличии зависимых тестов. А еще зависимые тесты очень не надежны. Подробнее: [http://barancev.github.io/test-deps-are-evil/](http://barancev.github.io/test-deps-are-evil/" \t "_blank)
* Стремитесь к тому, чтобы тест не полагался на контент, а готовил данные самостоятельно (и удалял после завершения). Используйте чистые браузеры и новых пользователей для лучшей воспроизводимости.
* Абсолютная линейность проверок. Когда вы пишете assert-утверждения в функциях, не следует использовать ветвления и циклы. Логика проверок должна быть линейна, иначе разбор багов и починка автотестов будут стоить очень дорого.
* Именуйте проверки в одинаковом стиле, чтобы по первому взгляду можно было понять, что это именно проверка. Например, можно именовать функции по шаблону should\_be\_smth:

def should\_be\_reply\_comment()

* Тесты именуются в одинаковом стиле. Имена тестов должны хорошо отражать различия в похожих сценариях. Можно использовать те же подходы, что и при добавлении имен к тест-кейсам в тестовой документации. Например, test\_guest\_can\_see\_teach\_button() — обратите внимание на явное указание на роль пользователя.
* Одинаковые тесты, которые отличаются только каким-то контентом (например, языком интерфейса), следует не копировать, а параметризовать.
* Пишите максимально атомарные и неделимые тесты. Не нужно писать один мега-тест, который проверяет вообще всё, напишите лучше десяток маленьких — проще будет локализовать проблему, когда она возникнет.

## Подготовка окружения

В этом модуле мы создадим с нуля полноценный тестовый проект, который будет являться вашим финальным заданием. Для этого будем использовать популярные в индустрии инструменты Git и GitHub, с которыми в общих чертах мы познакомились в предыдущем модуле.

Добавлять изменения мы будем постепенно, чтобы в вашем репозитории была красивая история коммитов. Потому что именно так происходит написание промышленного кода, а наша задача в этом курсе — максимально приблизиться к этому процессу.

Итак:

1. Создайте отдельный **публичный** репозиторий с осмысленным названием на GitHub.
2. Склонируйте его к себе на локальную машину.
3. Добавьте туда файл conftest.py из предыдущего модуля. Убедитесь дополнительно, что там есть параметр для задания языка интерфейса, по умолчанию равный "**en**".
4. Убедитесь что ни во вложенных папках, ни во внешних папках нет других файлов conftest.py, почему это важно смотри здесь: [Conftest.py — конфигурация тестов](https://stepik.org/lesson/237240/step/4).
5. Добавьте в репозиторий файл requirements.txt из предыдущего модуля.
6. Создайте пустой файл \_\_init\_\_.py, чтобы работали относительные импорты.
7. Создайте файл test\_main\_page.py и добавьте в него тест из предыдущего модуля:

def test\_guest\_can\_go\_to\_login\_page(browser): link = "http://selenium1py.pythonanywhere.com/" browser.get(link) login\_link = browser.find\_element\_by\_css\_selector("#login\_link") login\_link.click()

1. Не забудьте активировать окружение, которое мы создали ранее. Опционально, можно создать для этого проекта новое виртуальное окружение для удобства. В таком случае убедитесь что вы установили туда все необходимые пакеты из *requirements.txt.*А еще не стоит добавлять файлы окружения в репозиторий и вообще в отслеживаемые — лишние файлы на GitHub это моветон.
2. Убедитесь, что тест работает, с помощью следующей команды: pytest -v --tb=line --language=en test\_main\_page.py. Здесь и далее мы будем использовать эту команду для запуска. В этой команде мы использовали опцию PyTest **--tb=line**, которая указывает, что нужно выводить только одну строку из лога каждого упавшего теста. Так вам будет проще разобраться в том, как выглядят сообщения об ошибках.
3. Добавьте все новые файлы в Git командой **git add \***
4. Проверьте, что нужные файлы попали в планируемый коммит: **git status**
5. Зафиксируйте изменения коммитом с осмысленным сообщением: **git commit -m "write your message".**
6. По желанию добавьте описание репозитория с описанием вашего тестового проекта.

если вы хотите удалить файлы из репозитория гитхаб, то так (например папку .idea):

git rm -r --cached .idea

## Что такое Page Object Model?

**Page Object Model** или кратко Page Object — это паттерн программирования, который очень популярен в автоматизации тестирования и является одним из стандартов при автоматизации тестирования веб-продуктов. Это также один из удобных способов структурировать свой код таким образом, чтобы его было удобно поддерживать, менять и работать с ним.

Основная идея состоит в том, что каждую страницу веб-приложения можно описать в виде объекта класса. Способы взаимодействия пользователя со страницей можно описать с помощью методов класса. В идеале тест, который будет использовать Page Object, должен описывать бизнес-логику тестового сценария и скрывать Selenium-методы взаимодействия с браузером и страницей. При изменениях в верстке страницы не придется исправлять тесты, связанные с этой страницей. Вместо этого нужно будет поправить только класс, описывающий страницу.

То есть здесь применяются те же принципы, что и в разработке: мы хотим повысить читаемость кода и вынести в абстрактные методы все детали. Тесты должны быть просто и понятно написаны, а повторяющиеся куски кода выделены в отдельные функции. В Page Object мы отделяем логику действий, например, авторизовать пользователя, от конкретной реализации (найти поле почты, ввести туда данные, найти поле пароля, ввести туда данные, найти кнопку и т.д.).

Рассмотрим такой простой тест-кейс:

1. Открыть главную страницу
2. Перейти на страницу логина

Ожидаемый результат:

Открыта страница логина

Давайте посмотрим на кусочек кода теста из предыдущего модуля, который реализует первую часть этого теста:

test\_main\_page.py:

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

def test\_guest\_can\_go\_to\_login\_page(browser):

browser.get(link)

login\_link = browser.find\_element\_by\_css\_selector("#login\_link")

login\_link.click()

Что здесь происходит?

Мы открываем ссылку, находим элемент с определенным селектором и нажимаем на этот элемент.

Что мы на самом деле имеем в виду?

Мы хотим открыть страницу логина. Давайте выделим это действие в отдельную функцию с понятным названием, пока все в том же файле test\_main\_page.py :

def go\_to\_login\_page(browser):

login\_link = browser.find\_element\_by\_css\_selector("#login\_link")

login\_link.click()

и наш тест упрощается:

def test\_guest\_can\_go\_to\_login\_page(browser):

browser.get(link)

go\_to\_login\_page(browser)

При написании следующих тестов, когда нам понадобится перейти к странице логина с главной страницы, нам не нужно будет копировать этот кусочек кода или писать заново — мы сможем переиспользовать уже написанный метод.

## Почему именно Page Object?

Можно, конечно, хранить всю логику наших тестов в одном файле под каким-нибудь условным названием steps.py, и для начала это уже неплохо. Но если мы тестируем большой веб-продукт с множеством разных состояний и переходов, этот файл может разрастись до огромных размеров, и найти в нем нужный метод будет непросто. Еще бывают ситуации, когда на разных страницах логически один и тот же метод имеет разную реализацию. Например, у нашего интернет-магазина может быть метод "добавить в корзину". Но пользователь может добавлять товар в корзину как со страницы каталога, так и со страницы самого товара.

Было бы удобно выделить все методы, которые логически относятся к одной веб-странице в нашем продукте, в отдельный класс в нашем коде. Отсюда и название Page Object — это абстрактный объект, который содержит в себе методы для работы с конкретной веб-страницей.

**Важно!**Обычно методы у Page Object бывают двух типов: сделать что-то и проверить что-то.

Рассмотрим страницу товара в интернет магазине [http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook\_209/](http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209/" \t "_blank).

Какие могут быть методы у Page Object, ассоциированного с такой страницей? Запишем основные сценарии:

* добавить в корзину;
* проверить, что есть сообщение об успешном добавлении в корзину;
* перейти к написанию отзыва;
* проверить, что есть название, цена, описание товара;
* вернуться на главную.

Обратите внимание, что все проверки у нас тоже становятся отдельными методами. В самом тест-кейсе не остается никаких вспомогательных слов типа assert, только описание шагов. Прямо как в нашей тестовой документации.

Тесты будут выглядеть примерно так:

def test\_add\_to\_cart(browser):

page = ProductPage(url="", browser) # инициализируем объект Page Object

page.open() # открываем страницу в браузере

page.should\_be\_add\_to\_cart\_button() # проверяем что есть кнопка добавления в корзину

page.add\_product\_to\_cart() # жмем кнопку добавить в корзину

page.should\_be\_success\_message() # проверяем что есть сообщение с нужным текстом

4.2 Пишем первые тесты с помощью Page Object

## Базовая страница для проекта: BasePage

Давайте перепишем тест из файла test\_main\_page.py с помощью паттерна Page Object, который мы добавили на этапе [подготовки окружения.](https://stepik.org/lesson/199980/step/6?unit=174035) Мы будем работать с главной страницей нашего приложения, поэтому дадим классу говорящее название MainPage.

**Важно!**В этом уроке мы напишем самостоятельно простую реализацию паттерна Page Object. А в следующих уроках уже рассмотрим существующие фреймворки и то, как они могут облегчить нам жизнь. Сейчас самая главная задача — осознать принципы работы.

1. Создайте в своем проекте папку pages, там мы будем хранить все наши Page Object

2. В папке создайте два файла: base\_page.py и main\_page.py

Для начала сделаем базовую страницу, от которой будут унаследованы все остальные классы. В ней мы опишем вспомогательные методы для работы с драйвером.

3. В файле base\_page.py создайте класс с названием BasePage.

В Python такие вещи делаются с помощью следующей конструкции:

class BasePage():

4. Теперь в наш класс нужно добавить методы. Первым делом добавим конструктор — метод, который вызывается, когда мы создаем объект. Конструктор объявляется ключевым словом \_\_init\_\_. В него в качестве параметров мы передаем экземпляр драйвера и url адрес. Внутри конструктора сохраняем эти данные как аттрибуты нашего класса. Получается примерно так:

def \_\_init\_\_(self, browser, url):

self.browser = browser

self.url = url

5. Теперь добавим еще один метод open. Он должен открывать нужную страницу в браузере, используя метод get().

Объявите ниже в том же классе:

def open(self):

и реализуйте этот метод: нужна всего одна строка. Эту строку нужно отправить в качестве ответа на это задание, без отступов.

6. После того как Stepik принял ваш ответ как правильный, добавьте новые файлы в Git и зафиксируйте изменения коммитом (не забудьте осмысленное сообщение).

В итоге у вас должен следующий код в файле base\_page.py:

class BasePage():  
 def \_\_init\_\_(self, browser, url):  
 self.browser = browser  
 self.url = url  
  
 def open(self):  
 self.browser.get(self.url)

## Page Object для главной страницы сайта

Теперь реализуем Page Object, который будет связан с главной страницей интернет-магазина.

1. Откройте файл main\_page.py

2. В нем нужно сделать импорт базового класса BasePage:

from .base\_page import BasePage

3. В нем создайте класс  MainPage. Его нужно сделать наследником класса BasePage. Класс-предок в Python указывается в скобках:

class MainPage(BasePage):

таким образом, класс MainPage будет иметь доступ ко всем атрибутам и методам своего класса-предка.

4. Перенесите метод из предыдущего урока в класс MainPage:

def go\_to\_login\_page(browser):

   login\_link = browser.find\_element\_by\_css\_selector("#login\_link")

login\_link.click()

Чтобы все работало, надо слегка видоизменить его. В аргументы больше не надо передавать экземпляр браузера, мы его передаем и сохраняем на этапе создания Page Object. Вместо него нужно указать аргумент self , чтобы иметь доступ к атрибутам и методам класса:

def go\_to\_login\_page(self):

Так как браузер у нас хранится как аргумент класса BasePage, обращаться к нему нужно соответствующим образом с помощью self:

self.browser.find\_element\_by\_css\_selector("#login\_link")

Заодно заменим find на более универсальный:

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

Итого, файл *main\_page.py:*

from .base\_page import BasePage  
from selenium.webdriver.common.by import By  
  
class MainPage(BasePage):  
 def go\_to\_login\_page(self):  
 login\_link = self.browser.find\_element(By.CSS\_SELECTOR, "#login\_link")  
 login\_link.click()

## Первый тест на основе Page Object

Ура, первый прототип страницы мы уже реализовали! Давайте теперь перепишем тест с помощью Page Object:

1. Откройте файл с вашим тестом *test\_main\_page.py*

2. В самом верху файла нужно импортировать класс, описывающий главную страницу:

from .pages.main\_page import MainPage

3. Теперь преобразуем сам тест в *test\_main\_page.py*:

from .pages.main\_page import MainPage  
  
  
def test\_guest\_can\_go\_to\_login\_page(browser):  
 link = "http://selenium1py.pythonanywhere.com/"  
 page = MainPage(browser, link) # инициализируем Page Object, передаем в конструктор экземпляр драйвера и url адрес  
 page.open() # открываем страницу  
 page.go\_to\_login\_page() # выполняем метод страницы — переходим на страницу логина

4. Убедитесь, что тест проходит, запустив его все той же командой:

pytest -v --tb=line --language=en test\_main\_page.py

5. Добавьте изменения и сделайте коммит (с осмысленным сообщением!)

Теперь наш тест **почти**полностью написан в модном стиле Page Object! Почему **почти**— узнаете в следующих шагах.

Обратите внимание на  структуру файлов на данном этапе:

Text

Description automatically generated

## Методы-проверки в Page Object

Давайте теперь автоматизируем другой тест-кейс и посмотрим на его примере, как делать методы-проверки.

Допустим, нам нужно проверять такой сценарий:

1. Открыть главную страницу
2. Проверить, что есть ссылка, которая ведет на логин

Для этого в классе MainPage нужно реализовать метод, который будет проверять наличие ссылки. Обычно все такие методы-проверки называются похожим образом, мы будем называть их should\_be\_(название элемента).

Итак, в классе MainPage создайте метод should\_be\_login\_link.

Для первой пробы можно реализовать его самым примитивным образом:

def should\_be\_login\_link(self):

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

Сейчас мы намеренно сделали селектор неправильным, чтобы посмотреть, что именно выдаст тест, если поймает баг. Это хорошая практика: писать сначала красные тесты и только потом делать их зелеными.

Добавляем в файл с тест-кейсами новый тест:

def test\_guest\_should\_see\_login\_link(browser):

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

page = MainPage(browser, link)

page.open()

page.should\_be\_login\_link()

Запустите получившийся тест:

pytest -v --tb=line --language=en test\_main\_page.py

Вывод об ошибке не очень понятный, правда? Разобраться, что именно пошло не так, довольно тяжело. Поэтому в следующем шаге нам нужно будет обработать исключение, которое выбрасывает WebDriver.

В качестве ответа на данное задание напишите название исключения, которое вы получили в результате запуска теста.

Познакомиться подробнее с работой с исключениями в Python вы можете в данной статье: <http://pythonworld.ru/tipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html>

## Проверка элемента на странице

Чтобы выводить адекватное сообщение об ошибке, мы будем все проверки осуществлять с помощью assert и перехватывать исключения.

Для этого напишем вспомогательный метод поиска элемента в нашей базовой странице BasePage, который будет возвращать нам True или False. Можно сделать это по-разному (с настройкой явных или неявных ожиданий). Сейчас воспользуемся неявным ожиданием.

1. В конструктор BasePage добавим команду для неявного ожидания со значением по умолчанию в 10:

def \_\_init\_\_(self, browser, url, timeout=10):

self.browser = browser

self.url = url

self.browser.implicitly\_wait(timeout)

2. Теперь в этом же классе реализуем метод is\_element\_present, в котором будем перехватывать исключение. В него будем передавать два аргумента: *как*искать (css, id, xpath и тд) и собственно *что*искать (строку-селектор).

Чтобы перехватывать исключение, нужна конструкция try/except:

def is\_element\_present(self, how, what):

try:

self.browser.find\_element(how, what)

except (имя исключения):

return False

return True

Чтобы импортировать нужное нам исключение, в самом верху файла нужно указать:

from selenium.common.exceptions import имя\_исключения

Отлично! Теперь для всех проверок, что элемент действительно присутствует на странице, мы можем использовать этот метод.

3. Теперь модифицируем метод проверки ссылки на логин так, чтобы он выдавал адекватное сообщение об ошибке:

def should\_be\_login\_link(self):

assert self.is\_element\_present(By.CSS\_SELECTOR, "#login\_link\_invalid"), "Login link is not presented"

Запустите тесты и посмотрите, что вывод об ошибке стал более понятным:

pytest -v --tb=line --language=en test\_main\_page.py

**И не забудьте заменить селектор на правильный, чтобы тест снова проходил!**

4. Сделайте коммит изменений (с осмысленным сообщением).

## Элементы страниц в паттерне Page Object

Помните, мы говорили о том, что тесты почти соответствуют подходу Page Object?

Сейчас разберемся, почему **почти**на примере короткой и поучительной истории.

У нас уже есть два тест-кейса, которые так или иначе взаимодействуют со ссылкой на логин. Представим себе ситуацию, что у нас модный быстрый agile: разработчики постоянно вносят изменения в продукт. В какой-то прекрасный момент изменения коснулись и шапки сайта. Вот приходит к вам разработчик с новой ссылкой и говорит протестировать.

Замените линк, на котором запускаются тесты на <http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209?promo=midsummer>

Запустите тесты командой:

pytest -v --tb=line --language=en test\_main\_page.py

Тесты упали, и теперь нам нужно их поддерживать, то есть чинить. Подберите новый селектор к ссылке на логин.

Нам придется поправить в файле main\_page.py несколько мест, где используется измененный селектор. Посчитайте, сколько строк вам нужно будет отредактировать, чтобы починить ваши тесты, и внесите полученное число в первое поле ответа ниже.

Чтобы этого избежать, при проектировании тестов (да и вообще кода) хорошей практикой является выносить селектор во внешнюю переменную.

Давайте этим и займемся:

1. В папке pages создайте новый файл locators.py

2. Внутри создайте новый класс. Каждый класс будет соответствовать каждому классу PageObject:

from selenium.webdriver.common.by import By

class MainPageLocators():

LOGIN\_LINK = (By.CSS\_SELECTOR, "#login\_link")

теперь каждый селектор — это пара: как искать и что искать.

3. В файле main\_page.py импортируйте новый класс с локаторами

from .locators import MainPageLocators

4. Теперь в классе MainPage замените все строки, где содержится "**#login\_link**" таким образом:

def should\_be\_login\_link(self):

assert self.is\_element\_present(\*MainPageLocators.LOGIN\_LINK), "Login link is not presented"

Обратите внимание здесь на символ**\***, он указывает на то, что мы передали именно пару, и этот кортеж нужно распаковать.

5. Запустите тесты с помощью той же самой команды:

pytest -v --tb=line --language=en test\_main\_page.py

Они, конечно, снова упадут. Но теперь посчитайте, сколько строк вам нужно будет отредактировать, когда тесты написаны в такой конфигурации? Внесите число во второе поле ответа.

**Итак, PageObject — это не только методы, но и элементы.**

Исправлять руками сломанные селекторы во всем проекте — долго и муторно, и есть большой риск забыть и оставить старый селектор. Когда мы выносим селекторы в отдельную сущность, мы уменьшаем время на поддержку тестов и сильно упрощаем себе жизнь в долгосрочной перспективе.

А ещё спринт спустя промоакция закончилась, и фичу с изменением шапки откатили назад. Теперь ссылка работает так же, как раньше. Удалите ссылку с промоакцией, и верните обычную ссылку для запуска тестов:

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

Не забудьте вернуть старый селектор **#login\_link**, так чтобы тесты снова проходили. Они нам еще пригодятся!

## Реализация LoginPage

Если вы хорошо ориентируетесь в тест-дизайне, скорее всего вас немного коробит тест с переходом к логину — там ведь нет никаких проверок. Давайте проверим, что мы действительно перешли на страницу логина. Для этого нам будет нужен новый Page Object. Заодно разберемся, как между ними переключаться в ходе теста.

Скачайте [файл](https://stepik.org/media/attachments/lesson/199980/login_page.py) с шаблоном для LoginPage. Добавьте его в папку pages. Внутри есть заглушки для методов проверок:

should\_be\_login\_url

should\_be\_login\_form

should\_be\_register\_form

Реализуйте их самостоятельно:

1. В файле locators.py создайте класс LoginPageLocators

2. Подберите селекторы к формам регистрации и логина, добавьте их в класс LoginPageLocators

3. Напишите проверки, используя эти селекторы. Не забудьте через запятую указать адекватное сообщение об ошибке. Напишите сначала красный тест, чтобы убедиться в понятности вывода.

4. В методе should\_be\_login\_url реализуйте проверку, что подстрока "login" есть в текущем url браузера. Для этого используйте соответствующее [свойство Webdriver](https://selenium-python.readthedocs.io/api.html#selenium.webdriver.remote.webdriver.WebDriver.current_url).

5. Добавьте изменения в коммит с осмысленным сообщением

Теперь посмотрим, как можно осуществлять переход между страницами.

from .base\_page import BasePage  
from .locators import LoginPageLocators  
  
class LoginPage(BasePage):  
 def should\_be\_login\_page(self):  
 self.should\_be\_login\_url()  
 self.should\_be\_login\_form()  
 self.should\_be\_register\_form()  
  
 def should\_be\_login\_url(self):  
 assert LoginPageLocators.LOGIN\_URL in self.browser.current\_url, "Login url is not presented"  
  
 def should\_be\_login\_form(self):  
 # реализуйте проверку, что есть форма логина  
 assert self.is\_element\_present(\*LoginPageLocators.LOGIN\_FORM), "Login form is not presented"  
  
 def should\_be\_register\_form(self):  
 # реализуйте проверку, что есть форма регистрации на странице  
 assert self.is\_element\_present(\*LoginPageLocators.REGISTER\_FORM), "Register form is not presented"

LOCATORS

from selenium.webdriver.common.by import By  
  
  
class MainPageLocators():  
 LOGIN\_LINK = (By.CSS\_SELECTOR, "#login\_link")  
  
  
class LoginPageLocators():  
 LOGIN\_URL = "login"  
 LOGIN\_FORM = (By.CSS\_SELECTOR, "#login\_form")  
 REGISTER\_FORM = (By.CSS\_SELECTOR, "#register\_form")

TESTS

from .pages.main\_page import MainPage  
from .pages.login\_page import LoginPage  
  
link = "http://selenium1py.pythonanywhere.com/"  
link\_login = "http://selenium1py.pythonanywhere.com/en-gb/accounts/login/"  
  
def test\_guest\_can\_go\_to\_login\_page(browser):  
 page = MainPage(browser, link) # инициализируем Page Object, передаем в конструктор экземпляр драйвера и url адрес  
 page.open() # открываем страницу  
 page.go\_to\_login\_page() # выполняем метод страницы — переходим на страницу логина  
  
def test\_guest\_should\_see\_login\_link(browser):  
 page = MainPage(browser, link)  
 page.open()  
 page.should\_be\_login\_link()  
  
def test\_guest\_should\_see\_login\_url(browser):  
 page = LoginPage(browser, link\_login)  
 page.open()  
 page.should\_be\_login\_url()  
  
def test\_guest\_should\_see\_login\_form(browser):  
 page = LoginPage(browser, link\_login)  
 page.open()  
 page.should\_be\_login\_form()  
  
def test\_guest\_should\_see\_register\_form(browser):  
 page = LoginPage(browser, link\_login)  
 page.open()  
 page.should\_be\_register\_form()  
  
  
# pytest -v --tb=line --language=en test\_main\_page.py

## Переходы между страницами

Переход можно реализовать двумя разными способами.

Первый способ: возвращать нужный Page Object.

Для этого в файле main\_page.py нужно сделать импорт страницы с логином:

from .login\_page import LoginPage

Затем в методе, который осуществляет переход к странице логина, проинициализировать новый объект Page и вернуть его:

def go\_to\_login\_page(self):

link = self.browser.find\_element(\*MainPageLocators.LOGIN\_LINK)

link.click()

return LoginPage(browser=self.browser, url=self.browser.current\_url)

Обратите внимание! При создании объекта мы обязательно передаем ему тот же самый объект драйвера для работы с браузером, а в качестве url передаем текущий адрес.

Теперь в тесте нам не нужно думать про инициализацию страницы: она уже создана. Сохранив возвращаемое значение в переменную, мы можем использовать методы новой страницы в тесте:

def test\_guest\_can\_go\_to\_login\_page(browser):

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

page = MainPage(browser, link)

page.open()

login\_page = page.go\_to\_login\_page()

login\_page.should\_be\_login\_page()

Плюсы такого подхода:

* тест выглядит аккуратнее — не нужно инициализировать страницу в теле теста;
* явно возвращаем страницу — тип страницы ассоциирован с методом;
* не нужно каждый раз думать в разных тестах про инициализацию страницы — уменьшаем дублирование кода;

минусы:

* если у нас копится большое количество страниц и переходов — образуется много перекрестных импортов;
* большая связность кода — при изменении логики придется менять возвращаемое значение;
* сложнее понимать код, так как страница инициализируется неявно;
* образуются циклические зависимости, что часто приводит к ошибкам.

Второй подход: переход происходит неявно, страницу инициализируем в теле теста:

1. Закомментируйте строку с возвращаемым значением

def go\_to\_login\_page(self):

link = self.browser.find\_element(\*MainPageLocators.LOGIN\_LINK)

link.click()

# return LoginPage(browser=self.browser, url=self.browser.current\_url)

2. Инициализируем LoginPage в теле теста (не забудьте импортировать в файл нужный класс):

from .pages.login\_page import LoginPage

def test\_guest\_can\_go\_to\_login\_page(browser):

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

page = MainPage(browser, link)

page.open()

page.go\_to\_login\_page()

login\_page = LoginPage(browser, browser.current\_url)

login\_page.should\_be\_login\_page()

Плюсы:

* меньше связность кода;
* меньше импортов, нет перекрестных импортов;
* больше гибкость;
* в тесте понятнее что происходит, т.к. явно инициализируем страницу.

Минусы:

* появляется лишний шаг в тест-кейсе;
* каждый раз при написании теста нужно думать про корректные переходы;
* дублируется код.

И тот и другой подход можно успешно применять в своих проектах, главное делать это с умом. Сейчас оставьте второй вариант с явной инициализацией страниц в теле теста, чтобы избежать лишних сложностей с циклическими зависимостями.

Уберите лишний закомментированный код, и зафиксируйте изменения в коммите с осмысленным сообщением.

## Удобство поддержки тестов — инкапсуляция бизнес-логики в методах

Что делать, если изменилась логика взаимодействия со страницей, которая используется у нас в нескольких тестах? Например, нам нужно проверить возможность перехода на страницу логина по ссылке в навбаре для каждой из страниц сайта. Предположим, что таких страниц 20, и, значит, у нас есть 20 тестов, использующих метод **go\_to\_login\_page** класса MainPage. Затем разработчики добавили alert, который вызывается при клике на нужную нам ссылку. Мы увидим, что все 20 тестов упали, так как в методе go\_to\_login\_page нет шага с обработкой alert, следовательно, метод should\_be\_login\_page не сработает. Добавив обработку alert в метод **go\_to\_login\_page**, мы восстановим работоспособность всех тестов, не меняя самих тестов:

def go\_to\_login\_page(self):

   link = self.browser.find\_element\_by\_css\_selector("#login\_link")

link.click()

alert = self.browser.switch\_to.alert

alert.accept()

Это еще одно преимущество использования паттерна Page Object — мы разделяем сам тест и логику взаимодействия со страницей. Тест становится более читабельным, и его легче поддерживать при изменениях в коде приложения.

4.3 Улучшаем дизайн тестов

## Задание: добавление в корзину со страницы товара