**Запуск**

1. cd environments

Создадим виртуальное окружение:

python -m venv selenium\_env

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

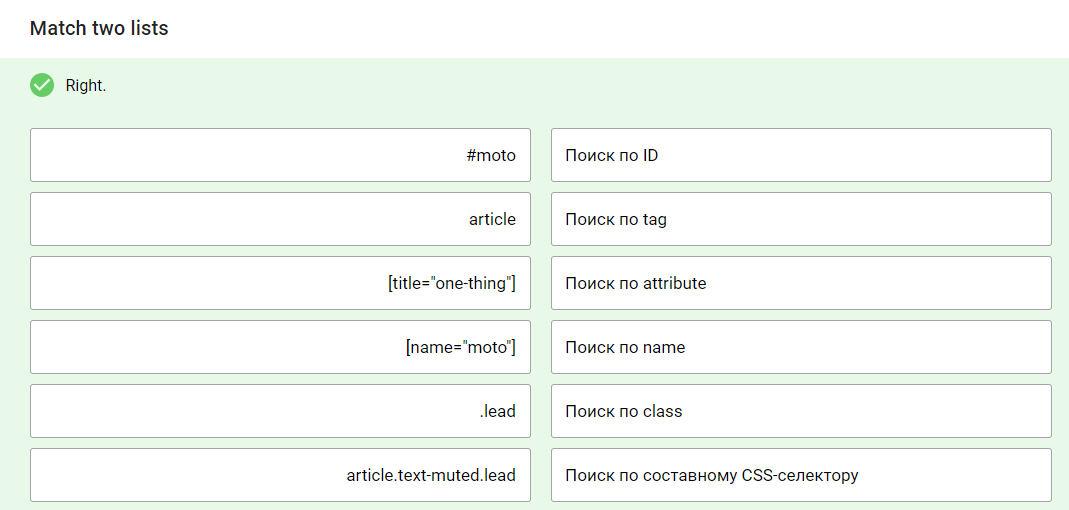
1. selenium\_env\Scripts\activate.bat

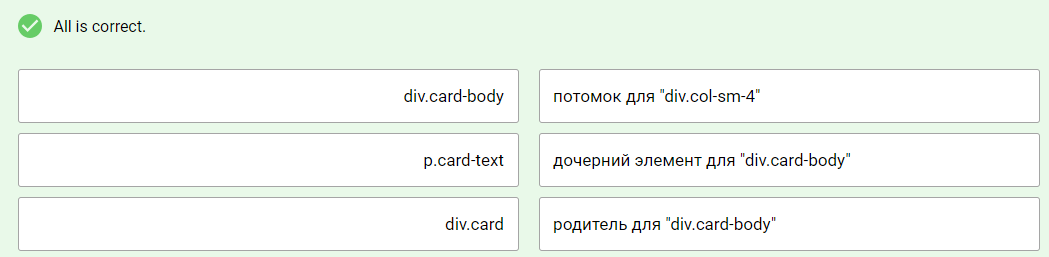
Если окружение активировано, то мы увидим в начале командной строки терминала название окружения в круглых скобках. Теперь мы можем устанавливать нужные нам пакеты и запускать скрипты для тестов, которые мы напишем:

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

#Запускаем из папки environments\ selenium\_course

3)python selenium\_course\lesson2.4\_step7.py





Откройте страницу <http://suninjuly.github.io/cats.html>. Откройте консоль разработчика и вкладку Elements в ней. Напишите **минимально** достаточный CSS-селектор, который найдет элемент с картинкой серьезного кота (Serious cat). Для поиска подходящего элемента в каталоге картинок используйте родительский элемент **div.col-sm-4**вместе с псевдо-классом**:nth-child(n)**, чтобы выбрать n-й по счету элемент, а также селектор для картинки по тегу **img**. При написании этого селектора не надо использовать символ **>**, т.к. это задача на поиск потомка без использования дочерних элементов. Для решения достаточно тех селекторов, которые уже указаны в задании.

Пожалуйста, не используйте селекторы которые генерирует браузер по кнопке "скопировать css селектор" или расширения. Зачастую они строят полный путь начиная от **body** - а это очень нестабильный селектор, писать такие в своем коде это плохая практика. При малейшем изменении структуры страницы все ваши селекторы потеряют актуальность.

Не используйте в синтаксисе квадратные скобки - конструкции вида *[class="stepic"].*

Введите найденный селектор в качестве ответа на это задание.

**.col-sm-4:nth-child(2) img**

**Поиск элементов с помощью Selenium**

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.

ink = 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()

## checkbox и radiobutton

option1 = browser.find\_element\_by\_css\_selector("[value='python']") option1.click()

**Поиск элементов с помощью XPath**

В работе с веб-страницами не всегда получается найти селектор, однозначно описывающий путь к нужному элементу. В такой ситуации для тестировщика лучшим решением проблемы будет пойти к фронтенд-разработчику проекта и договориться о специальном атрибуте, который будет использоваться в автотестах. Таким образом можно повысить тестируемость приложения. Увы, проекты бывают разные, и не всегда это возможно. И когда другого выхода больше нет, а автоматизировать как-то надо, можно обратиться к помощи языка запросов **XPath**.

На тему XPath мнения расходятся, но, как бы то ни было, это мощный и гибкий инструмент, который позволяет писать сложные запросы для поиска элементов.

Прежде всего, XPath (XML Path Language) это язык запросов, который использует древовидную структуру документа. Проверять XPath-запросы можно точно так же как и CSS-селекторы - в консоли разработчика. Откройте консоль на странице с котиками <http://suninjuly.github.io/cats.html>, и давайте на её примере разберемся в основах синтаксиса. Попробуйте вбить каждый из запросов-примеров в строку поиска, чтобы увидеть, что именно находит поисковый запрос.

**1. 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.

Разница состоит в том, что в XPath, когда мы начинаем запрос с символа /,  мы должны указать элемент, являющийся корнем нашего документа. Корнем всегда будет элемент с тегом <html>. Пример: /html/body/header

Мы можем начинать запрос и с символа //. Это будет означать, что мы хотим найти всех потомков корневого элемента без указания корневого элемента. В этом случае, для поиска того же хедера, мы можем выполнить запрос //header, так как других заголовков у нас нет.

Важно! Такой поиск может быть неоднозначным. Например, запрос //div вернет вам все элементы с тегом <div>. Избегайте неоднозначных ситуаций, они плохо влияют на здоровье ваших автотестов.

**2. Символ [ ] - это команда фильтрации**

Если по запросу найдено несколько элементов, то будет произведена фильтрация по правилу, указанному в скобках.

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']

**3. Символ \* - команда выбора всех элементов**

* Например можем найти текст в заголовке запросом //div/\*[@class="jumbotron-heading"]. Это может быть удобно, когда мы не знаем точно тег элемента, который ищем.

**4. Поиск по классу в XPath регистрозависим**

Также как и в случае поиска по CSS-селектором будьте внимательными к регистру при поиске по классам:

**//div/\*[@class="Jumbotron-heading"]** не найдет элемент на нашей странице.

Что важно знать про XPath, чтобы пользоваться им безболезненно:

* Не используйте селекторы вида //div[1]/div[2]/div[3] без крайней нужды: по такому селектору невозможно с первого раза понять, что за элемент вы ищете. А когда структура страницы хоть немного изменится, то ваш селектор с большой вероятностью перестанет работать;
* Если есть возможность использовать CSS-селекторы: сlass, id или name - лучше использовать их вместо поиска по XPath;
* Можно искать по полному или частичному совпадению текста или любого атрибута;
* Можно использовать булевы операции и простую арифметику;
* Можно удобно перемещаться по структуре документа (переходить к потомкам и к родителям);
* Подойдет, когда у сайта всё плохо с атрибутами и нет возможности достучаться до разработчиков;
* Есть мнение, что поиск по XPath в среднем медленнее, чем по css. Но достоверно это неизвестно;
* Не стоит использовать разные расширения для браузеров по поиску XPath: они подбирают нечитабельные и переусложненные селекторы. Лучше потратить немного времени и разобраться в синтаксисе самостоятельно, тем более, что он не очень сложный.

В курсе мы не будем работать с XPath-селекторами, и в основном будем использовать CSS. В случае необходимости можно познакомиться с XPath подробнее по следующим ссылкам:

<https://www.w3schools.com/xml/xpath_syntax.asp>

<https://msdn.microsoft.com/ru-ru/library/ms256086(v=vs.120).aspx>

<https://msiter.ru/tutorials/xpath/syntax>

<https://habr.com/post/114772/>

<http://internetka.in.ua/xpath-start-part1/>

<http://internetka.in.ua/xpath-start-part2/>

<http://internetka.in.ua/xpath-start-part3/>

<http://internetka.in.ua/xpath-start4/>﻿

<https://testerslittlehelper.wordpress.com/2016/07/10/real-xpath/>

<https://drive.google.com/file/d/0B_IyyodHL4rXcU1BY1R1TzFXbmc/view>

**АТРИБУТЫ ЭЛЕМЕНТА**

from selenium import webdriver

import time

link = "http://suninjuly.github.io/math.html"

try:

browser = webdriver.Chrome()

browser.get(link)

#проверяем значение атрибута checked у people\_radio

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

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"

#проверяем значение атрибута checked у robots\_radio

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

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

print("value of robots\_radio: ", robots\_checked)

assert robots\_checked is None

#проверяем значение атрибута disabled у кнопки Submit

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

button\_disabled = button.get\_attribute("disabled")

print("value of button Submit: ", button\_disabled)

assert button\_disabled is None

#проверяем значение атрибута disabled у кнопки Submit после таймаута

time.sleep(10)

button\_disabled = button.get\_attribute("disabled")

print("value of button Submit after 10sec: ", button\_disabled)

assert button\_disabled is not None

finally:

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

**ВЫТАЩИТЬ ЗНАЧЕНИЕ ИЗ АРИБУТА VALUE**

x\_element = browser.find\_element\_by\_id("treasure")

x = x\_element.get\_attribute("valuex")

print(x)

## Работа со списками

Посмотрим, как выглядит html для списка:

<label for="dropdown">Выберите язык программирования:</label>

<select id="dropdown" class="custom-select">

 <option selected>--</option>

 <option value="1">Python</option>

 <option value="2">Java</option>

 <option value="3">JavaScript</option>

</select>

Варианты 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 "--".

## 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()

assert True

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 проскролить нужный элемент, чтобы он точно стал видимым.  
Делается это с помощью следующего скрипта:

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

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 проскролить страницу. После выполнения кода элемент button должен оказаться в верхней части страницы.

Также можно проскролить 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();

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.

## 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.

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. Подробнее о методах модуля **os** можете почитать самостоятельно в документации: <https://docs.python.org/3/library/os.path.html>. Обратите внимание, что это будет работать только при запуске кода из файла, в интерпретаторе не сработает.

from selenium import webdriver

import time

import os

try:

link = "http://suninjuly.github.io/file\_input.html"

browser = webdriver.Chrome()

browser.get(link)

#заполняем поля

firstname = browser.find\_element\_by\_name("firstname")

firstname.send\_keys("firstname")

lastname = browser.find\_element\_by\_name("lastname")

lastname.send\_keys("lastname")

email = browser.find\_element\_by\_name("email")

email.send\_keys("email")

#выбираем файл

file1 = browser.find\_element\_by\_id("file")

# получаем путь к директории текущего исполняемого файла

current\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))

# добавляем к этому пути имя файла

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

file1.send\_keys(file\_path)

# Отправляем заполненную форму

submit = browser.find\_element\_by\_class\_name("btn.btn-primary")

submit.click()

except Exception as error:

print(f'{error}')

#print(welcome\_text)

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

## Alerts и как с ними жить

Мы уже встречали alert в нашем курсе, когда получали число-ответ в задачах. Также мы узнали, что можно самостоятельно вызвать alert с помощью JavaScript:

alert('Hello!');

Теперь 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()

## 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.

**СОСТОЯНИЕ ЭЛЕМЕНТОВ**

## Explicit Waits (WebDriverWait и expected\_conditions)

В предыдущем шаге мы решили проблему с ожиданием элементов на странице. Однако методы **find\_element** проверяют только то, что элемент появился на странице. В то же время элемент может иметь дополнительные свойства, которые могут быть важны для наших тестов. Рассмотрим пример с кнопкой, которая отправляет данные:

* Кнопка может быть неактивной, то есть её нельзя кликнуть;
* Кнопка может содержать текст, который меняется в зависимости от действий пользователя. Например, текст "Отправить" после нажатия кнопки поменяется на "Отправлено";
* Кнопка может быть перекрыта каким-то другим элементом или быть невидимой.

Если мы хотим в тесте кликнуть на кнопку, а она в этот момент неактивна, то WebDriver все равно проэмулирует действие нажатия на кнопку, но данные не будут отправлены.

Давайте попробуем запустить следующий тест:

from selenium import webdriver

browser = webdriver.Chrome()

# говорим WebDriver ждать все элементы в течение 5 секунд

browser.implicitly\_wait(5)

browser.get("http://suninjuly.github.io/wait2.html")

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

button.click()

message = browser.find\_element\_by\_id("verify\_message")

assert "successful" in message.text

Мы видим, что WebDriver смог найти кнопку с**id="verify"**и кликнуть по ней, но тест упал на поиске элемента "**verify\_message**" с итоговым сообщением:

no such element: Unable to locate element: {"method":"id","selector":"verify\_message"}

Это произошло из-за того, что WebDriver быстро нашел кнопку и кликнул по ней, хотя кнопка была еще неактивной. На странице мы специально задали программно паузу в 1 секунду после загрузки сайта перед активированием кнопки, но неактивная кнопка в момент загрузки — обычное дело для реального сайта.

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**. Улучшим наш тест:

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 так же 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

Описание каждого правила можно найти на [сайте](https://selenium-python.readthedocs.io/api.html#module-selenium.webdriver.support.expected_conditions).

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"))

)

Ссылки полезные:

В этом уроке мы постарались собрать ссылки на ресурсы, где вы сможете найти дополнительную информацию по использованию Selenium и о тонкостях при работе с ним:

**Общее**

* <https://selenium-python.com/>
* [http://selenium-python.readthedocs.io](http://selenium-python.readthedocs.io/)
* <http://chromedriver.chromium.org/getting-started>﻿
* [﻿https://www.guru99.com/selenium-tutorial.html](https://www.guru99.com/selenium-tutorial.html) - ﻿Туториал на английском, ориентирован на Java.﻿
* <https://www.guru99.com/live-selenium-project.html> - ﻿Можно попробовать писать автотесты для демо-сайта ﻿банка. Тоже Java.
* <http://barancev.github.io/good-locators/> - что такое хорошие селекторы
* <http://barancev.github.io/what-is-path-env-var/> - что за PATH переменная?

**Ожидания в Selenium WebDriver**

* <https://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp>
* <https://selenium-python.readthedocs.io/waits.html>﻿
* [https://selenium-python.readthedocs.io/api.html#module-selenium.webdriver.support.expected\_condition...](https://selenium-python.readthedocs.io/api.html#module-selenium.webdriver.support.expected_conditions)﻿
* <https://stackoverflow.com/questions/15122864/selenium-wait-until-document-is-ready>
* <https://blog.codeship.com/get-selenium-to-wait-for-page-load/>