**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. Для поиска Selenium предоставляет метод find\_element, который принимает два аргумента - тип локатора и значение локатора. Существуют следующие методы поиска элементов:

* **find\_element(By.ID, value)** — 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, value)** — 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, value)** — поиск с помощью языка запросов XPath, позволяет выполнять очень гибкий поиск элементов;
* **find\_element(By.NAME, value)** — поиск по атрибуту name элемента;
* **find\_element(By.TAG\_NAME, value)** — поиск элемента по названию тега элемента;
* **find\_element(By.CLASS\_NAME, value)** — поиск по значению атрибута class;
* **find\_element(By.LINK\_TEXT, value)**— поиск ссылки на странице по полному совпадению;
* **find\_element(By.PARTIAL\_LINK\_TEXT, value)**— 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

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")

Обратите внимание, что мы импортировали класс By, который содержит все возможные локаторы.

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

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")

**Поиск нескольких элементов**

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**, которые мы рассмотрим чуть позже.

Иногда в статьях про Selenium WebDriver вы также будете встречать термин "локаторы", под которым подразумеваются стратегии поиска и значения, по которым должен выполняться поиск. Например, можно искать по локатору By.ID со значением "send\_button".

## Работа с браузером в Selenium

Если вы уже пробовали запускать примеры скриптов, то могли заметить, что браузер не всегда закрывается после выполнения кода. Поэтому обратите внимание на то, что необходимо явно закрывать окно браузера в нашем коде при помощи команды **browser.quit().** Каждый раз при открытии браузера browser = webdriver.Chrome() в системе создается процесс, который останется висеть, если вы вручную закроете окно браузера. Чтобы не остаться без оперативной памяти после запуска нескольких скриптов, всегда добавляйте к своим скриптам команду закрытия:

from selenium import webdriver

from selenium.webdriver.common.by import By

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

browser = webdriver.Chrome()

browser.get(link)

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

button.click()

# закрываем браузер после всех манипуляций

browser.quit()

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" \t "_blank). 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()

Можете попробовать запустить оба примера и обратить внимание на разницу.

Подробно говорить об обработке исключений мы сейчас не будем, здесь важно понимать только то, что даже если в коде внутри блока **try** произойдет какая-то ошибка, то код внутри блока **finally** выполнится в любом случае. Советуем добавлять такую обработку ко всем своим скриптам при выполнении задач этого и следующего модулей, а в третьем модуле мы обсудим более лаконичные конструкции.

## Задание: поиск элементов с помощью Selenium

Вам нужно открыть страницу по [ссылке](http://suninjuly.github.io/simple_form_find_task.html) и заполнить форму на этой странице с помощью Selenium. Если всё сделано правильно, то вы увидите окно с проверочным кодом. Это число вам нужно ввести в качестве ответа в этой задаче.

Внимание: Файл с кодом находится в папке “code”, с названием lesson6\_step4.py

!Обратите внимание, что время для ввода данных ограничено. Однако благодаря Selenium вы сможете выполнить задачу до того, как время истечёт.

!Кроме того, ограничено время на ввод ответа. Постарайтесь вводить ответ сразу.

Для решения этой задачи мы подготовили для вас шаблон кода, в который нужно только подставить нужные значения для поиска вместо слов value1, value2 и т.д. Обратите внимание, что значения нужно заключать в кавычки, т.к. они должны передаваться в виде строки.

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

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

try:

browser = webdriver.Chrome()

browser.get(link)

input1 = browser.find\_element(By.TAG\_NAME, value1)

input1.send\_keys("Ivan")

input2 = browser.find\_element(By.NAME, value2)

input2.send\_keys("Petrov")

input3 = browser.find\_element(By.CLASS\_NAME, value3)

input3.send\_keys("Smolensk")

input4 = browser.find\_element(By.ID, "country")

input4.send\_keys("Russia")

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

finally:

# успеваем скопировать код за 30 секунд

time.sleep(30)

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

Системы UNIX/Linux ожидают пустую строку в конце файла, если в вашем скрипте ее не будет, то последняя строчка, содержащая код, может не выполниться.

Создайте файл **lesson6\_step4.py** (обратите внимание на расширение .py) и вставьте туда шаблон кода. Подберите селекторы и запустите из командной строки, так же, как в уроке 2:

**python** **lesson6\_step4.py**

Все последующие задачи с кодом запускайте по аналогии.

## Задание: поиск элемента по тексту в ссылке

В этой задаче мы попробуем искать элементы по тексту ссылки, для этого воспользуемся методом find\_element\_by\_link\_text:

link = browser.find\_element(By.LINK\_TEXT, text)

В качестве аргумента в метод передается такой текст, ссылку с которым мы хотим найти. Это тот самый текст, который содержится между открывающим и закрывающим тегом <a> вот тут </a>

Допустим, на странице <https://www.degreesymbol.net/> мы хотим найти ссылку с текстом "Degree symbol in Math" 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 in Math")

link.click()

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

link = browser.find\_element(By.PARTIAL\_LINK\_TEXT, "Math")

link.click()

Обычно поиск по подстроке чуть более удобный и гибкий, но с ним надо быть вдвойне аккуратными и проверять, что находится нужный элемент.

Внимание: Файл с кодом находится в папке “code”, с названием lesson6\_step5.py

### Задание

На указанной ниже странице вам нужно найти зашифрованную ссылку и кликнуть по ней:

1. Добавьте в самый верх своего кода import math
2. Добавьте в код команду, которая откроет страницу: <http://suninjuly.github.io/find_link_text>
3. Добавьте команду, которая найдет ссылку с текстом. Текст ссылки, который нужно найти, зашифрован формулой:

str(math.ceil(math.pow(math.pi, math.e)\*10000))

(можно вставить данное выражение в свой код, а можно выполнить в интерпретаторе, скопировать оттуда результат и уже его использовать в вашем коде)

1. Добавьте команду для клика по найденной ссылке: она перенесет вас на форму регистрации
2. Заполните скриптом форму так же как вы делали в предыдущем шаге урока
3. После успешного заполнения вы получите код - отправьте его в качестве ответа на это задание

**Важно!**Поиск по тексту ссылки бывает очень удобным, так как часто тексты меняются реже, чем атрибуты элементов. Но лучше избегать такого метода поиска. Например, если приложение имеет несколько языков интерфейса, ваши тесты будут проходить только с определенным языком интерфейса. Применяйте этот метод с осторожностью и помните про возможные ограничения.

Читать больше:

<https://selenium-python.readthedocs.io/locating-elements.html#locating-hyperlinks-by-link-text>

## Поиск всех необходимых элементов с помощью find\_elements

Мы уже упоминали, что метод **find\_element** 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 существует метод **find\_elements**, которые в отличие от **find\_element** вернёт 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

!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.

## Задание: использование метода find\_element****s****

В этой задаче вам нужно заполнить форму (<http://suninjuly.github.io/huge_form.html>). С помощью неё отдел маркетинга компании N захотел собрать подробную информацию о пользователях своего продукта. В награду за заполнение формы становится доступен код на скидку. Но маркетологи явно переусердствовали, добавив в форму 100 обязательных полей и ограничив время на ее заполнение. Теперь эта задача под силу только роботам ﻿🤖﻿.

Используйте WebDriver, метод **find\_elements**, нужные локатор и его значение. Введите полученный код в качестве ответа к этой задаче.

Используйте приведенный ниже шаблон: в цикле for мы можем последовательно взять каждый элемент из найденного списка текстовых полей и отправить произвольный текст в каждое поле. Если скрипт не успевает заполнить форму, выберите текст покороче.

Внимание: Файл с кодом находится в папке “code”, с названием lesson6\_step7.py

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

try:

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/huge\_form.html")

elements = browser.find\_elements("вставьте сюда нужный локатор", "вставьте сюда нужное значение")

for element in elements:

element.send\_keys("Мой ответ")

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

finally:

# успеваем скопировать код за 30 секунд

time.sleep(30)

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

## Задание: поиск элемента по XPath

На этот раз воспользуемся возможностью искать элементы по XPath.

На странице <http://suninjuly.github.io/find_xpath_form> вы найдете такую же форму регистрации, как в шаге 3, при этом в нее добавилась куча одинаковых кнопок отправки. Но сработает только кнопка с текстом "Submit", и наша задача нажать в коде именно на неё.

Ваши шаги:

1. В коде из [шага 4](https://stepik.org/lesson/138920/step/4) замените ссылку на  <http://suninjuly.github.io/find_xpath_form>.
2. Подберите уникальный XPath-селектор так, чтобы он находил только кнопку с текстом **Submit**. XPath можете формулировать как угодно (по тексту, по структуре, по атрибутам) - главное, чтобы он работал.
3. Модифицируйте код из шага 3 таким образом, чтобы поиск кнопки происходил с помощью XPath.
4. Запустите ваш код.

**Уникальность селекторов: часть 1**

Мы уже упоминали, что идеальный селектор — это такой селектор, который позволяет найти только один искомый элемент на странице. Благодаря уникальным селекторам наши тесты становятся стабильнее и меньше зависят от изменений в вёрстке страницы. Небольшие изменения разработчики делают достаточно часто, а мы бы не хотели постоянно исправлять наши тесты.

Другое важное замечание: хороший тест проверяет только маленькую, атомарную часть функциональности. Простые тесты, которые проверяют небольшой сценарий, лучше, чем один большой тест, проверяющий сразу много сценариев. Благодаря простым тестам мы быстрее локализуем место в продукте, где появился баг, а также можем найти одновременно несколько новых багов. Упавший большой автотест укажет только на первую встреченную проблему, так как он заканчивает работу при первой же найденной ошибке. В этом их отличие от ручных тестов, в которых мы, проверяя функциональность продукта по тест-кейсу, можем гибко обойти встречающиеся проблемы и пройти тест-кейс до конца, найдя все баги.

Рассмотрим следующий пример: у нас есть форма регистрации, в которой есть обязательные и необязательные поля для заполнения. Нужно проверить, что можно успешно зарегистрироваться на сайте.

**Сценарий плохого автотеста:**

* Открыть страницу с формой
* Заполнить все поля
* Нажать кнопку "Регистрация"
* Проверить, что есть сообщение об успешной регистрации

**Лучше разбить предыдущий тест на набор более простых автотестов:**

**1**

* Открыть страницу с формой
* Заполнить только обязательные поля
* Нажать кнопку "Регистрация"
* Проверить, что есть сообщение об успешной регистрации

2

* Открыть страницу с формой
* Заполнить все обязательные поля
* Заполнить все необязательные поля
* Нажать кнопку "Регистрация"
* Проверить, что есть сообщение об успешной регистрации

3

* Открыть страницу с формой
* Заполнить только необязательные поля
* Проверить, что кнопка "Регистрация" неактивна

## Уникальность селекторов: часть 2

Попробуем реализовать один из автотестов из предыдущего шага. Вам дана [страница](http://suninjuly.github.io/registration1.html) с формой регистрации. Проверьте, что можно зарегистрироваться на сайте, заполнив только обязательные поля, отмеченные символом \*: First name, last name, email. Текст для полей может быть любым. Успешность регистрации проверяется сравнением ожидаемого текста "Congratulations! You have successfully registered!" с текстом на странице, которая открывается после регистрации. Для сравнения воспользуемся стандартной конструкцией assert из языка Python.

Ниже дан шаблон кода, который вам нужно использовать для своего теста. Не забывайте, что селекторы должны быть уникальными.

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

try:

link = "http://suninjuly.github.io/registration1.html"

browser = webdriver.Chrome()

browser.get(link)

# Ваш код, который заполняет обязательные поля

...

# 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()

Углубимся немного в использовании конструкции assert из данного примера. Если результат проверки "Поздравляем! Вы успешно зарегистрировались!" == welcome\_text вернет значение False, то далее выполнится код **assert False**.Он бросит исключение AssertionError и номер строки, в которой произошла ошибка. Если код написан правильно и работал ранее, то такой результат равносилен тому, что наш автотест обнаружил баг в тестируемом веб-приложении. Если результат проверки вернет True, то выполнится выражение **assert True**. В этом случае код завершится без ошибок — тест прошел успешно. Подробнее про использование assert в коде мы поговорим в третьем модуле этого курса.

В этом задании нет автоматических проверок вашего кода. Просто убедитесь, что ваш тест проходит успешно, и вы не видите AssertionError в результатах работы вашего кода.

**Замечание**

В этом примере мы использовали метод **time.sleep(1)**, чтобы дождаться загрузки следующей страницы, прежде чем выполнять проверки. Если вы будете запускать код без этого метода, ваш код может внезапно упасть, хотя проходил ранее. Без использования такой паузы WebDriver может перейти к поиску тега h1 слишком рано, когда новая страница еще не загрузилась. В таком случаем будем видеть в терминале ошибку:

NoSuchElementException... Unable to locate element: {"method":"tag name","selector":"h1"}

Метод time.sleep(1) говорит Python подождать 1 секунду, прежде чем выполнять следующую строчку кода. Если вы всё равно видите эту ошибку, просто увеличьте количество секунд ожидания.

Проблема со своевременным поиском элемента — одна из самых больших проблем, которую приходится решать при разработке автотестов для UI. В условиях постоянно изменяющейся скорости сетевого соединения и неравномерности нагрузки на серверы скорость загрузки страницы может сильно варьироваться. Еще одним фактором, влияющим на стабильность работы тестов, является принцип асинхронности выполнения кода JavaScript. На простых страницах вы можете этого и не заметить, но на функционально богатых страницах время появления элементов страницы может быть непредсказуемо. Хорошо было бы организовать тесты так, чтобы не сложилось ситуации, когда они не проходят по причине нестабильной скорости интернета или других причин, которые от нас не зависят.

Решать эту проблему с помощью time.sleep() считается плохой практикой, так как заранее трудно указать нужное время ожидания. Если выставить слишком большое время ожидания, то тесты будут идти неоправданно долго. В дальнейших уроках мы рассмотрим более красивые и эффективные способы решения этой проблемы, а пока будем использовать time.sleep() из-за его простоты и наглядности.

## Уникальность селекторов: часть 3

В предыдущем примере мы хотели показать, что, выбирая селектор с классом ".second" для автотеста для первой формы, вы можете попасть в следующую ситуацию: обязательное поле "Last name:" вдруг пропадёт при изменениях в продукте, как это произошло во второй версии формы, и данные будут введены во второстепенное поле с тем же классом. В этом случае тест пройдёт успешно, и мы не будем знать о возникшей проблеме. Уникальный селектор может спасти нас в этой ситуации — тест упадёт именно в тот момент, когда не обнаружит нужный элемент на странице.