Часть 1 **Знакомимся с Selenium**

**Что такое Selenium?**

У термина довольно много разных значений, и много всего под ним подразумевают в разных контекстах.

Давайте разберемся и разложим все по полочкам!

Для начала: Selenium — это проект, который предназначен главным образом для автоматизации тестирования веб-приложений. Проект кстати, open-source, то есть с открытым исходным [кодом](https://github.com/SeleniumHQ/selenium). В любой момент можно заглянуть внутрь кода и посмотреть, как работает тот или иной метод, а также можно написать свои предложения разработчикам проекта, и, более того, сделать pull request с исправлениями и улучшениями кода.

Это собирательный, зонтичный проект*,*то есть он состоит из множества самых разных компонентов, например:

* Selenium WebDriver
* Selenium Grid
* Selenium Server
* Selenium IDE
* и не только...

А также проект содержит ряд конкретных драйверов браузера и клиентских библиотек для разных языков программирования.

Обычно из контекста понятно, о какой из частей Selenium идет речь. В данном курсе речь пойдет о Selenium WebDriver, и всякий раз, когда мы будем говорить Selenium, будем иметь в виду именно эту часть проекта. А подробнее в том, что это такое, мы разберемся на следующем шаге.

**Что такое Selenium WebDriver?**

Автотесты бывают очень разными, проверяют разные вещи и работают для разных уровней приложения. Наверняка вы не раз видели так называемую "пирамиду тестирования". В нашем курсе мы в основном будем говорить про end-to-end тесты, то есть такие, которые проверяют полную работу системы и имитируют действия пользователя.

Что нам нужно для того, чтобы написать такой тест для веб-приложения?

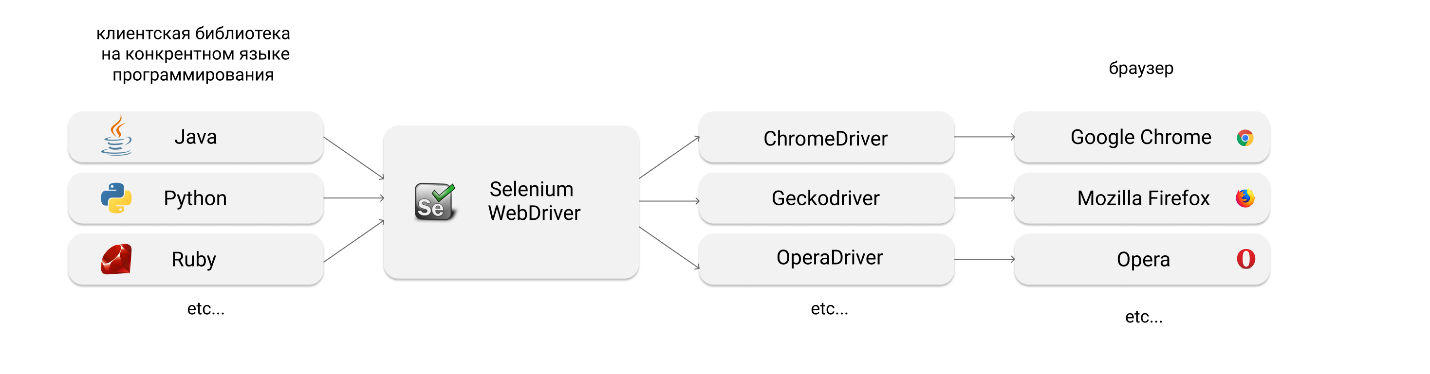
1. Очевидно, код на языке программирования. Там будет содержаться какая-то логика действий (например: залогинить пользователя, добавить товар в корзину) и проверки.
2. Браузер, чтобы имитировать действия пользователя. Желательно, какой-нибудь популярный. А еще лучше — несколько.

Эти две части нужно как-то связать воедино. И как раз для этого нужен Selenium WebDriver. По сути своей это — универсальный интерфейс, который позволяет манипулировать разными браузерами напрямую из кода на языке программирования. Его крутость как раз в его универсальности: за некоторыми исключениями, все наши тесты, написанные один раз, можно запускать для разных браузеров.

Верна универсальность и в другую сторону: браузер будет выполнять одинаковые действия, не важно из библиотеки какого именно языка программирования эти методы будут вызваны. Вообще-то, Selenium WebDriver поддерживает довольно много [языков](https://github.com/SeleniumHQ/selenium#documentation) программирования, хотя наибольшей популярностью в индустрии пользуются Java и Python.

В этом курсе мы будем изучать именно Selenium WebDriver, и в текстах будем использовать термин WebDriver. Однако, если вы будете искать в интернете какую-то информацию, связанную с Selenium WebDriver, то скорее всего увидите, что используется более общий термин Selenium. Вам нужно будет смотреть на контекст, чтобы определить, какая часть проекта Selenium имеется в виду.

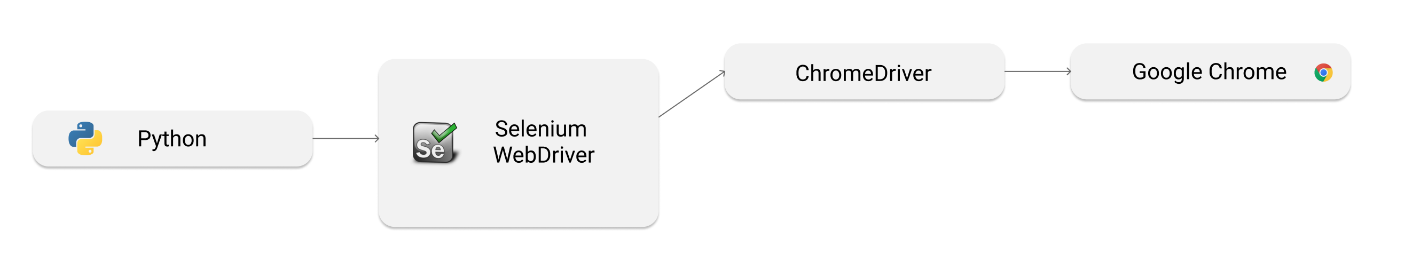
Итого получаем вот такую картинку:

[](https://ucarecdn.com/5cfd09d4-3291-4552-860e-e4d35f8fda0b/)

Это такое отношение "многие ко многим".

Что здесь еще важно? Так как все браузеры работают слегка по-разному, разработчики браузеров самостоятельно реализуют "драйвер" – программу, которая позволяет интерфейсу Selenium манипулировать кнопками, текстами, элементами и т.д.

В этом курсе мы будем работать со следующим набором инструментов:

[](https://ucarecdn.com/0420dc11-61ec-40de-88e3-f3edc600113e/)

Интересный факт! WebDriver стал настолько популярен, что его признали [стандартом](https://www.w3.org/TR/webdriver1/) протокола взаимодействия с браузером.

Подробнее можно почитать посты Алексея Баранцева: они хоть и старые, все еще хорошие. Он сейчас – один из тех разработчиков, кто вносит вклад в проект Selenium.

[Что такое Selenium?](https://habr.com/ru/post/152653/)

[Что такое Selenium WebDriver?](https://habr.com/ru/post/152971/)

1.2 Запускаем браузер с помощью Selenium WebDriver

**Содержание урока**

* [Установка Python3 в Windows](https://stepik.org/lesson/25969/step/2)
* [Установка Python3 в Ubuntu](https://stepik.org/lesson/25969/step/3)
* [Установка Python3 в macOS](https://stepik.org/lesson/25969/step/4)
* [Задание: интерпретатор Python](https://stepik.org/lesson/25969/step/5)
* [Установка Selenium для Python](https://stepik.org/lesson/25969/step/6)
* [Установка драйвера для браузера](https://stepik.org/lesson/25969/step/7)
* [Установка драйвера для браузера: Windows](https://stepik.org/lesson/25969/step/8)
* [Установка драйвера для браузера: Linux](https://stepik.org/lesson/25969/step/9)
* [Установка драйвера для браузера: macOS](https://stepik.org/lesson/25969/step/10)
* [Запуск браузера и первый скрипт](https://stepik.org/lesson/25969/step/11)
* [Задание](https://stepik.org/lesson/25969/step/12)

**Установка Selenium для Python**

Теперь мы почти готовы запустить браузер с помощью Python и Selenium, и выполнить в нём простые действия. Но предварительно нам еще надо установить библиотеку Selenium для Python и драйвер — программу, которая позволяет взаимодействовать с браузером при помощи Selenium.

В нашем виртуальном окружении установим библиотеку Selenium:

pip install selenium==4.\*

*Проект Selenium развивается очень быстро и в вашей установленной версии библиотеки могут быть расхождения в методах или в том, что библиотека выводит в терминал при выполнении скриптов. Если вы просите помощи в комментариях, пожалуйста, указывайте версии python-библиотек, которые вы используете. Версии библиотек в вашем виртуальном окружении можно узнать с помощью команды pip list.*

**Установка драйвера для браузера: Windows**

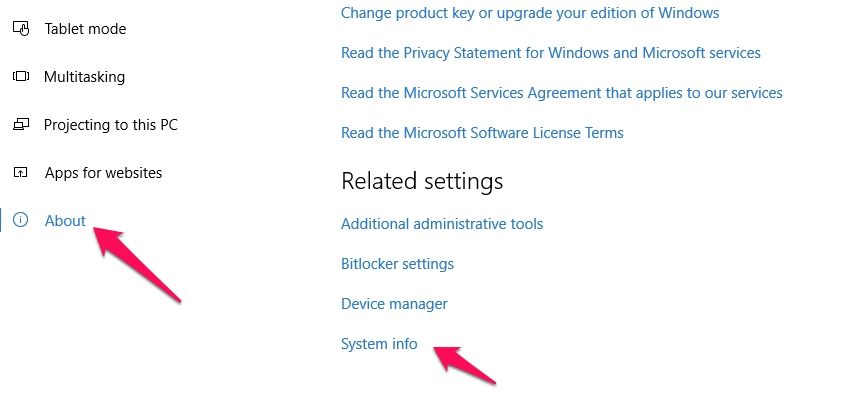
Для установки откройте сайт <https://sites.google.com/chromium.org/driver/> (старая версия сайта <https://sites.google.com/a/chromium.org/chromedriver/downloads>) и скачайте ту версию ChromeDriver, которая соответствует версии вашего браузера. Чтобы узнать версию браузера, откройте новое окно в Chrome, в поисковой строке наберите: chrome://version/ и нажмите Enter. В верхней строчке вы увидите информацию про версию браузера.

* Скачайте с сайта <https://sites.google.com/chromium.org/driver/> (старая версия сайта <https://sites.google.com/a/chromium.org/chromedriver/downloads>) драйвер для вашей версии браузера. Разархивируйте скачанный файл.
* Создайте на диске C: папку chromedriver и положите разархивированный ранее файл chromedriver.exe в папку C:\chromedriver.
* Добавьте в системную переменную PATH папку C:\chromedriver. Как это сделать в разных версиях Windows, описано здесь: <https://www.computerhope.com/issues/ch000549.htm>.

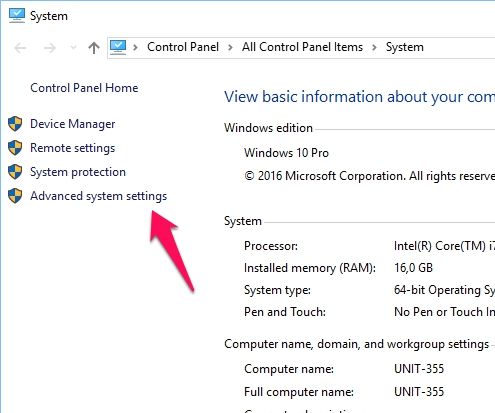
**Пример: как добавить путь в системную переменную PATH на Windows10**

1. Откройте настройки системы.

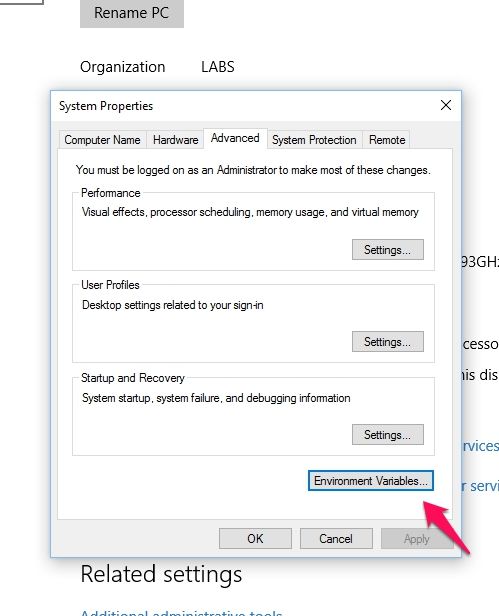
2. В настройках откройте вкладку About, затем System info:



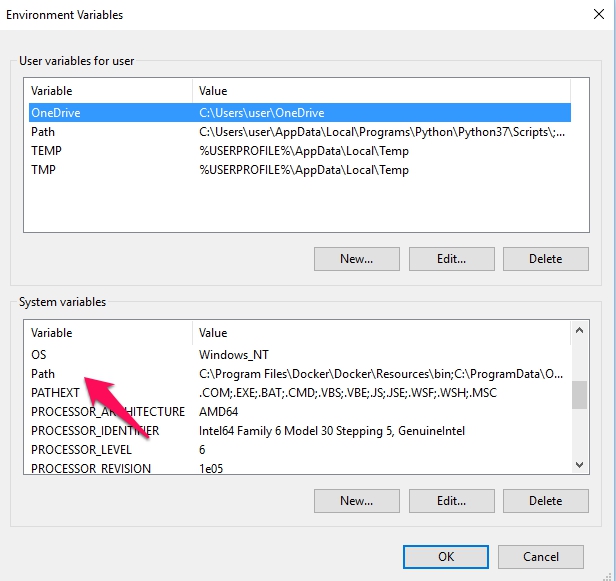
3. Выберите Advanced system settings:



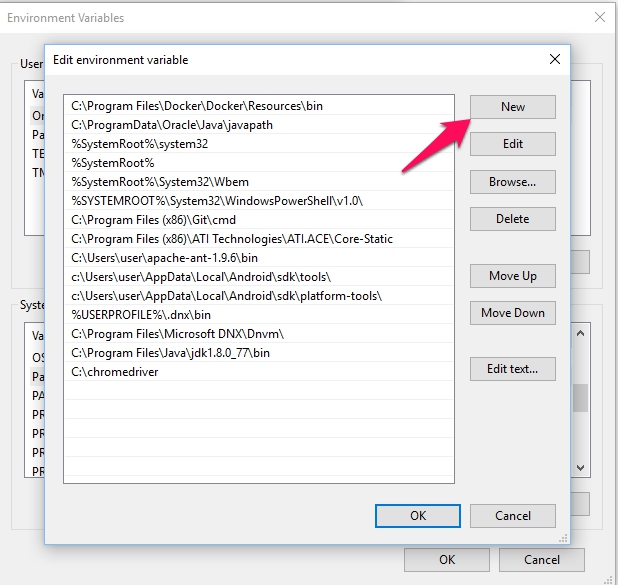
4. Выберите Environment Variables:



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



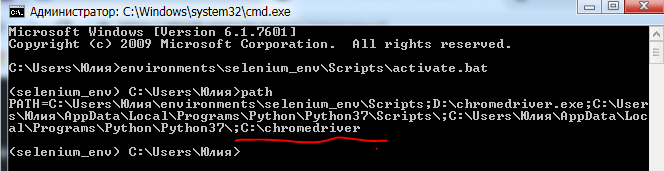
6. Нажмите кнопку New. Введите в новую строку путь к ChromeDriver — C:\chromedriver. Нажмите Enter. У вас должна появится строка с указанным путем:



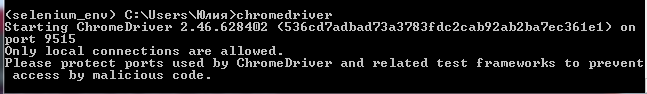
7. Если у вас была открыта командная строка Windows, не забудьте ее закрыть. Затем откройте новую командную строку, чтобы изменения переменной окружения стали доступны. Активируйте снова виртуальное окружение selenium\_env, которое мы создали в предыдущих шагах.

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.

Знак на этом этапе, что пошло что-то не так:



В таком случае попробуйте перезапустить консоль, перезапустить компьютер, перепроверить и добавить заново по инструкциям папку с chromedriver в переменную path.

Когда все получится, переходите к шагу <https://stepik.org/lesson/25969/step/11>, в котором мы запустим браузер с помощью Selenium WebDriver и выполним простые команды.

1.3 Структура веб-страницы

**Что представляет собой веб-страница?**

В данном уроке мы подробнее разберёмся с самой важной для нас возможностью Selenium WebDriver — умению взаимодействовать с элементами веб-страницы. Любая страница в интернете представляет собой **html**-файл, в котором с помощью языка разметки HTML описана её структура.

Еще практически все сайты используют язык **JavaScript**, который позволяет сделать веб-страницу интерактивной, т.е. реагировать на действия пользователя, запрашивать у пользователя данные и возвращать их. WebDriver тоже позволяет выполнять javascript-команды в браузере, но это требуется не очень часто, поэтому мы рассмотрим эту возможность в дальнейших уроках.

Для полноты картины нужно упомянуть еще **CSS (**Cascading Style Sheets — *каскадные таблицы стилей*), который используется для вёрстки (красивого оформления страниц). Наверное, вы уже сталкивались с тем, что поломанная вёрстка страниц может приводить к очень неприятным багам на сайте. Благодаря WebDriver мы также можем поймать какую-то неожиданную проблему в оформлении — например, когда нужная пользователю кнопка оказывается перекрыта второстепенным элементом.

Сейчас нас будет интересовать в первую очередь структура страницы, т.е. ее описание на языке HTML. Умея описывать путь к элементу на странице, мы можем найти такой элемент и выполнить с ним необходимые действия, например, отправить текст в текстовое поле или нажать на правильную кнопку.

Далее мы рассмотрим несколько способов поиска элементов внутри страницы:

1. Поиск с помощью CSS-селекторов, когда путь к элементу описывается через синтаксис CSS. Селектор — это описание пути к элементу на странице.
2. Поиск с помощью указания значений тегов или атрибутов элементов: ID, class, и т.д.
3. Поиск с помощью языка запросов XPath.

Поиск с помощью CSS-селекторов, с нашей точки зрения, является наиболее удобным способом, т.к. он покрывает практически все возможные ситуации, и CSS-селекторы выглядят более читабельными. В реальности в разных случаях может понадобиться использовать и другие методы поиска.

Учим селекторы в игровой форме <https://flukeout.github.io/>

## Структура веб-страницы

Как мы говорили ранее, язык CSS используется для описания внешнего вида страницы, которая написана с помощью языка разметки HTML. Если мы хотим сказать браузеру, что какую-то кнопку мы хотим сделать зелёной, мы сначала должны задать правило, которое позволяет выделить эту кнопку среди других элементов. Например, это можно сделать с помощью CSS-селектора. Селектор однозначно выделит кнопку среди других элементов html-страницы, а стиль color задаст кнопке цвет green. Синтаксис такого выражения в CSS-файле выглядит так:

button {

color: green;

}

button - это селектор, выражение color: green задает цвет элементу button.

Знание правил написания CSS-селекторов понадобится и нам, чтобы искать элементы на HTML-странице и взаимодействовать с ними.

Также нам понадобится знание трёх особенностей языка HTML:

1. Страница на языке HTML состоит из элементов, начало и конец элемента задаются с помощью специальных пометок – **тегов** (**tag**);
2. У тегов есть **атрибуты**, которые определяют свойства элементов;
3. Страница на языке HTML имеет **иерархическую** структуру.

### 1. **Tag**–элемент или тег

Начало и конец элементов обрамляются специальными словами – **тегами (tag)**. Задача тегов – обозначить, какой именно тип информации на странице они представляют (картинка, текст, блок, ссылка и так далее). Например:

<p>Здесь написан текст.</p>

В первом примере элемент обозначен открывающим (<p>) и закрывающим (</p>) тегами, между тегами помещён текст, который увидит пользователь. Некоторым тегам закрывающий тег не нужен, достаточно только открывающего. Например, тег:

<br>

Это тег для переноса строки, внутри него никакого текста или вложенных элементов быть не может.

### 2. **Attribute**–**атрибуты или свойства элемента**

Элементам можно задавать различные стандартные свойства, а также создавать собственные свойства или атрибуты. Атрибуты обычно имеют вид: имя="значение". Иногда знак "=" и значение могут быть опущены или значение может быть пустой строкой: имя="".

В примере ниже помимо тегов и текста добавлен еще атрибут**href**, который задаёт ссылку для тега <a>, на которую будет вести текст внутри тега.

<a href="/about">Посмотреть подробнее</a>

**Атрибут class**

В третьем примере появляется еще один важный атрибут – **класс (class)**. Элементу задано два класса: **simple-text** и **main**. Классы чаще всего используют для задания правил вёрстки с помощью CSS:

<div class="simple-text main">Здесь написан текст.</div>

**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>

**Атрибут id**

В этом примере появился еще атрибут **id**– уникальный указатель на элемент. Значение id в общем случае не должно повторяться в пределах страницы:

<button id="save\_button">Сохранить</button>

!Важно. В современных JavaScript-фреймворках id-атрибуты чаще всего генерируются динамически самим фреймворком, поэтому они изменяются каждый раз при перезагрузке страницы и совершенно нечитабельны, например: вы увидите что-то вроде id="u\_ps\_0\_0\_n" или id="avadspffd". В таких случаях вам придется пользоваться другими селекторами или использовать собственные data-атрибуты, о которых мы расскажем далее. Названия классов также могут генерироваться автоматически. Поэтому предлагаем вам простое правило: если увидите нечеловекочитаемое значение атрибута или если значение атрибута меняется при перезагрузке страницы, то не используйте его. В наших примерах мы используем статические страницы, поэтому вы смело можете использовать описанные здесь CSS-селекторы.

### 3. **Иерархия** в HTML-документах

HTML-документ часто сравнивают с моделью семейного древа, в котором есть родители, дети, братья, предки и потомки. Такое сравнение допустимо благодаря возможности вкладывать одни элементы в другие. Вложенные (дочерние) элементы могут, в свою очередь, содержать свои дочерние элементы, следовательно, глубина вложения элементов может быть любой.

Давайте разберёмся с терминологией данного древа:

* потомок элемента X – элемент любой степени вложенности внутри элемента X;
* ребёнок или дочерний элемент — прямой потомок (т.е. элемент на первом уровне вложенности);
* предок элемента Y – любой элемент X, который включает в себя элемент Y;
* родитель — это прямой предок (т.е. элемент, который расположен выше строго на 1 уровень);
* братский или соседний элемент – элемент X, который расположен на одном уровне иерархии с элементом Y. Элементы X и Y имеют одного родителя.

Теперь попробуем закрепить терминологию на примере. Возьмём такой пример HTML кода:

<div class="news">

<div class="title">

<p class="title\_text">Selenium 4alpha is out now</p>

</div>

<div class="text\_block">

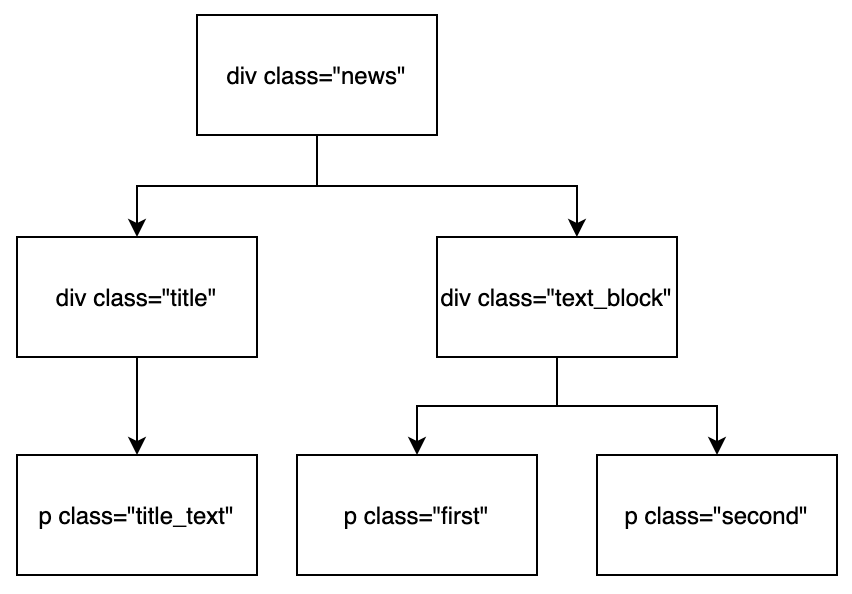
<p class="first">After 3 years from it’s a major release, now Selenium has put out its first alpha version of Selenium 4 on Apr 24, 2019.</p>

<p class="second">If you want you can make your hands dirty with Selenium 4 and it’s feature by downloading the dependencies from Maven Repository.</p>

</div>

</div>

Теперь нарисуем блок-схему этого HTML кода, где каждый блок будет соответствовать одному HTML элементу.



* Какие элементы являются потомками элемента, имеющего class="news"? Это все остальные элементы в этом html-коде: элемент с class="title", элемент с class="title\_text" и другие.
* Какие элементы можно назвать дочерними (child) элементами для элемента с class="news"? Только элементы с class= "title" и class="text\_block".
* Какие элементы являются предками элемента с class="first"? Это элементы с class="text\_block" и class="news".
* Какой элемент является родителем элемента с class="first"? Только элемент с class="text\_block".
* Какой элемент мы можем назвать братским или соседним элементом для элемента, имеющего class="first"? Это элемент с class="second".

Эти знания помогут нам применять мощный механизм поиска элементов с помощью CSS-селекторов.

## Атрибуты, которые влияют на отображение элемента

Сейчас же мы рассмотрим ту часть языка HTML, которaя позволит вам писать код для автотестов.

Поговорим чуть подробнее про атрибуты элементов. Некоторые атрибуты влияют на отрисовку элемента на странице, а другие не влияют напрямую, но могут использоваться в JavaScript-коде или быть нужными только для локации элемента в тестах.

Примеры атрибутов, которые повлияют на отображение и поведение элемента на странице:

<h1 style="color: blue;"> Заголовок будет синим, т.к. цвет задан в атрибуте style </h1>

<p hidden> Атрибут hidden скрывает элемент на странице, элемент не будет показываться </p>

<button disabled> Кнопка с атрибутом disabled будет заблокирована </button>

Чуть ниже вы найдете интерактивную консоль, попробуйте добавить вышеуказанные атрибуты элементам в HTML и посмотрите, как изменится отображение элементов. Не забудьте отправить решение кнопкой "Отправить" ("Submit"), после того как все пункты чек-листа будут отмечены как верные.

## Подробнее об атрибутах (data-атрибуты)

Как мы уже говорили, некоторые атрибуты напрямую не влияют на отображение элемента на страницах. О некоторых таких важных атрибутах мы уже поговорили в шаге 3 (например, **id**).

А еще список атрибутов можно расширять: это значит, что разработчик может создать свой собственный атрибут и присвоить ему любые значения. Что это значит для тестировщика? Это значит, что можно договориться с разработчиками о специальном атрибуте, который вы будете использовать в своих тестах для поиска нужных элементов и который не будет изменяться при исправлении верстки сайта. Это добавит стабильности вашим тестам. Правда, есть несколько ограничений:

* веб-сайт должен использовать стандарт HTML5 (большинство современных сайтов соответствует этому требованию)
* использовать можно только латинские буквы, и символы дефис (-), двоеточие (:) и подчёркивание (\_)

Также принято названия таких атрибутов начинать со слова: "**data"**, например, "data-button".

Что еще важно знать про атрибуты элементов?

Некоторые атрибуты являются универсальными, они могут относиться к любому тегу и любому типу элементов. Например, **hidden** (т.е. скрытым) можно сделать любой элемент. Некоторые же атрибуты ассоциированы строго с определенным тегом, например, для картинки, которая задается тегом **img**, обязательно нужно указывать атрибут **src**.

Если вы собираетесь в дальнейшем работать с автоматизацией тестирования веб-продуктов, то вам будет очень полезно изучить HTML более детально. Вы сможете быстро подбирать нужные селекторы, с первого взгляда на HTML-разметку будете видеть что <a> - это ссылка, <p> - текст, а <ul> - ненумерованный (маркированный) список. Но это большая и широкая тема, которая заслуживает отдельного курса (можно проходить такие курсы самостоятельно, например, <https://stepik.org/course/38218/>).

Образовательная рекомендация: если вы не совсем понимаете, что означает тег или атрибут элемента, попробуйте погуглить. Каждый кусочек этих знаний так или иначе поможет вам в будущем.

Для закрепления знаний про атрибуты предлагаем вам следующее задание: сопоставьте атрибут с его описанием. Не все из этих атрибутов были упомянуты ранее, так что не стесняйтесь пользоваться любым поисковиком :)

1.4 Поиск элементов

* [Поиск элементов с помощью CSS-селекторов](https://stepik.org/lesson/102555/step/2)
* [Задание: поиск по тегу](https://stepik.org/lesson/102555/step/3)
* [Поиск элементов с помощью составных CSS-селекторов](https://stepik.org/lesson/102555/step/4)
* [Задание: подбор селекторов 1](https://stepik.org/lesson/102555/step/5)
* [Задание: иерархия элементов в веб-странице](https://stepik.org/lesson/102555/step/6)
* [Задание: поиск потомка](https://stepik.org/lesson/102555/step/7)
* [Задание: подбор селекторов 2](https://stepik.org/lesson/102555/step/8)
* [Задание: подбор селекторов 3](https://stepik.org/lesson/102555/step/9)
* [Поиск элементов с помощью XPath](https://stepik.org/lesson/102555/step/10)
* [Задание: поиск по XPath](https://stepik.org/lesson/102555/step/11)

## Поиск элементов с помощью CSS-селекторов

Ниже приведены части элементов HTML-страницы, по которым можно найти элемент:

* id
* tag
* значение атрибута
* name
* class

Давайте откроем страницу <http://suninjuly.github.io/cats.html> и попробуем найти элемент, который содержит картинку с Котом-пулей (Bullet cat). Ниже приведён упрощенный кусок html-кода страницы:

<div class="col-sm-4">

<div class="card mb-4 box-shadow">

  <img id="bullet" name="bullet-cat" data-type="animal" class="card-img-top" src="images/bullet\_cat.jpg">

</div>

</div>

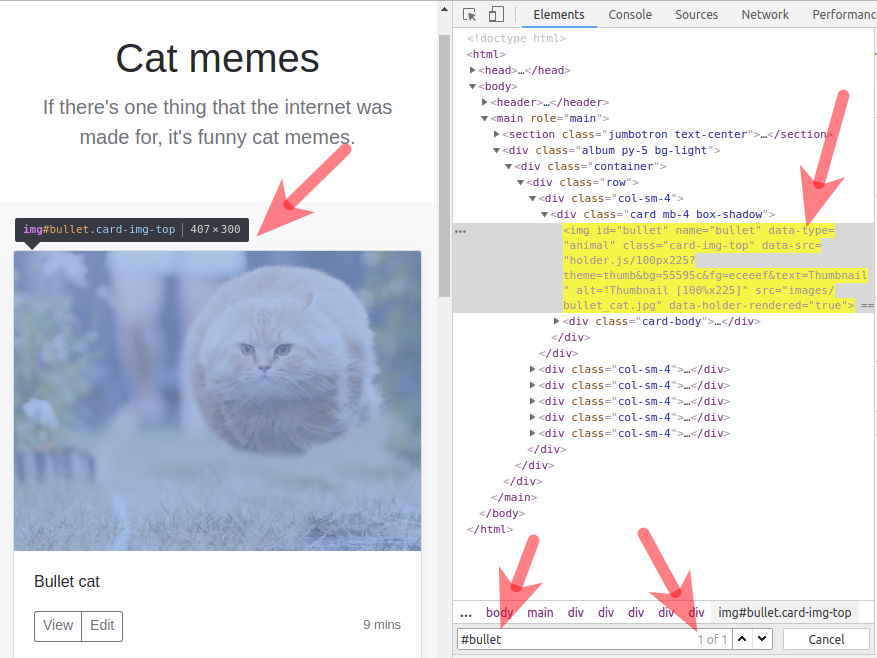
Для начала мы попробуем искать элементы вручную с помощью консоли браузера, а в следующем уроке научимся писать код, который выполняет ту же задачу поиска.

**Поиск по id**

Какое везение! У элемента с нашей картинкой есть атрибут id="bullet", а значит, мы однозначно можем найти её с помощью селектора **#bullet**(знак **#** означает, что мы ищем по **id** со значением **bullet**).

Можно проверить правильность подобранного селектора непосредственно в браузере в консоли разработчика. Откройте консоль разработчика и перейдите в ней на вкладку Elements. Затем нажмите ctrl+F и в открывшейся внизу поисковой строке введите селектор. Если селектор написан правильно, то вы увидите подсвеченный элемент на веб-странице, а также элемент будет подсвечен жёлтым цветом в html-коде. Еще в поисковой строке вы увидите количество найденных элементов. Желательно писать точные селекторы, которые позволяют найти ровно один элемент. В написании таких селекторов мы потренируемся в одной из следующих задач.

Еще один способ открыть консоль разработчика в браузере: нажать правой кнопкой мыши на любой элемент страницы и выбрать пункт меню "Посмотреть код" (англ. "Inspect") в контекстном меню. При этом на вкладке Elements сразу будет подсвечен кусок HTML-кода, описывающий данный элемент.



**Поиск по tag**

Чтобы найти элемент по тегу, просто напишите название тега в поисковой строке, как мы делали это при поиске по id (только без знака **#**), например, **h1**. Поиск по **h1** найдёт для нас элемент с названием страницы. Поиск по тегам не очень удобен, т.к. разработчики используют небольшое количество тегов для разметки страниц, и скорее всего, одному тегу будет соответствовать множество элементов.

**Поиск по значению атрибута**

Можно найти элемент, указав название атрибута и его значение. Например, можно переписать поиск по id в следующем виде **[id="bullet"]**вместо **#bullet**.

Лучше использовать вариант с квадратными скобками при поиске значения атрибута для тех атрибутов, у которых нет собственных коротких команд поиска. Например, давайте найдем элемент h1 по значению его атрибута value: **[value="Cat memes"]**.

**Поиск по name**

Этот вариант поиска является разновидностью поиска по значению атрибута и записывается так же: **[name="bullet-cat"]**. Мы выделяем этот вариант потому что он довольно часто используется, а также выделяется как отдельный вид поиска элементов в Selenium WebDriver.

**Поиск по class**

Поиск по классу можно записать в виде **[class="jumbotron-heading"]**, так как class тоже является атрибутом элемента. Но раз уж классы используются практически в каждой странице при задании стилей страниц, то для них также имеется свой короткий вариант поиска: **.jumbotron-heading**. То есть мы пишем значение класса и предваряем его точкой.

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>:

<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.

Еще одно важное замечание. Поиск по классу чувствителен к регистру, то есть **.Lead** уже не найдет нужный элемент.

В консоли браузера вы также можете искать по простому совпадению текста в HTML, например, запрос **lead** подсветит текст **lead**. Однако, не стоит пользоваться таким поиском для выбора элементов, так как он слишком общий и не может использоваться в качестве селектора.

Мы рассмотрели разные варианты написания пути к элементу на странице, используя синтаксис CSS, т.е. научились писать CSS-селекторы. Слово "селектор" является буквальным переводом от английского слова selector. Selector в свою очередь происходит от глагола select, что переводится как "выбирать".

Далее в этом уроке мы научимся искать элементы, комбинируя способы составления селекторов, рассмотренные в данном шаге.

## Поиск элементов с помощью составных CSS-селекторов

Теперь предположим, что не можем найти элемент на странице, используя простой селектор, так как такой селектор находит сразу несколько элементов. Ниже мы привели часть кода простой HTML-страницы, описывающей блог. Саму страницу вы можете посмотреть по [ссылке](http://suninjuly.github.io/blog_example.html).

Вопрос: как нам найти селектор для подписи у второй картинки? Вот здесь нам поможет иерархическая структура страницы и возможность комбинировать CSS-селекторы. CSS-селекторы позволяют использовать одновременно любые селекторы, рассмотренные ранее, а также имеют некоторые дополнительные возможности для уточнения поиска.

<div id="posts" class="post-list">

  <div id="post1" class="item">

    <div class="title">Как я провел лето</div>

    <img src="./images/summer.png">

  </div>

  <div id="post2" class="item">

    <div class="title second">Ходили купаться</div>

    <img src="./images/bad\_dog.jpg">

  </div>

  <div id="post3" class="item">

    <div class="title">С друзьями</div>

    <img src="./images/friends.jpg">

  </div>

</div>

**Использование потомков**

Попробуем найти элемент с текстом "Ходили купаться". Для решения этой задачи мы можем взять элемент, стоящий выше в иерархии нужного нам элемента, и написать следующий селектор:

#post2 .title

Здесь символ **#** означает, что надо искать элемент с id post2, пробел - что также нужно найти элемент-потомок, а **.**, что элемент-потомок должен иметь класс со значением title.

Элемент .title называется **потомком** (англ. **descendant**) элемента #post2. Потомок может находиться на любом уровне вложенности, все элементы с селектором .title также являются и потомками элемента #posts, хотя и расположены от него на два уровня ниже. #posts .title найдет все 3 элемента с классом title.

!Внимание. Символ пробела " " является значащим в CSS-селекторах. Это важный символ, который разделяет описание предка и потомка. Если бы мы записали селектор #post2.title без пробела, то в данном примере не было найдено ни одного элемента. Такая запись означала бы, что мы хотим найти элемент, который одновременно содержит id "post2" и класс "title". Таким образом #post2 .title и #post2.title — это разные селекторы**.**

**Использование дочерних элементов**

Другой способ найти этот элемент:

#post2 > div.title

Здесь мы указали еще тег элемента divи уточнили, что нужно взять элемент с тегом и классом: div.title, который находится строго на один уровень иерархии ниже чем элемент #post2. Для этого используется символ >.

Элемент #post2 в этом случае называется **родителем** (англ. **parent**) для элементаdiv.title, а элемент div.title называется **дочерним элементом** (англ. **child**) для элемента #post2. Если символа > нет, то будет выполнен поиск всех элементов div.title на любом уровне ниже первого элемента.

!Внимание. В данном случае символы пробела вокруг символа ">" не несут важного значения в отличие от предыдущего примера, и могут быть опущены. Запись #post2>div.titleаналогична записи #post2 > div.title.

**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.

**Использование нескольких классов**

Также мы можем использовать сразу несколько классов элемента, чтобы его найти. Для этого классы записываются подряд через точку: .title.second

Мы рассмотрели базовые селекторы, которых будет достаточно для написания простых UI-тестов. Если вы захотите разобраться подробнее в css-селекторах, то мы рекомендуем вам посмотреть следующие статьи:

<https://learn.javascript.ru/css-selectors>

<https://www.w3schools.com/cssref/css_selectors.asp>

<https://developer.mozilla.org/ru/docs/Web/CSS/CSS_Selectors>

## 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/>

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

1.6 Поиск элементов с помощью Selenium WebDriver

## Содержание урока

* [Поиск элементов с помощью Selenium WebDriver](https://stepik.org/lesson/138920/step/2)
* [Работа с браузером в Selenium](https://stepik.org/lesson/138920/step/3)
* [Задание: поиск элементов с помощью Selenium Webdriver](https://stepik.org/lesson/138920/step/4)
* [Задание: поиск элемента по тексту в ссылке](https://stepik.org/lesson/138920/step/5)
* [Поиск всех необходимых элементов с помощью find\_elements](https://stepik.org/lesson/138920/step/6)
* [Задание: использование метода find\_elements](https://stepik.org/lesson/138920/step/7)
* [Задание: поиск элемента по XPath](https://stepik.org/lesson/138920/step/8)
* [Уникальность селекторов: часть 1](https://stepik.org/lesson/138920/step/9)
* [Уникальность селекторов: часть 2](https://stepik.org/lesson/138920/step/10)
* [Задание: уникальность селекторов](https://stepik.org/lesson/138920/step/11)
* [Уникальность селекторов: часть 3](https://stepik.org/lesson/138920/step/12)
* [Итоги модуля](https://stepik.org/lesson/138920/step/13)

## 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). 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.

Если хотите узнать больше про исключения, как их кидать, ловить и как с ними жить, то советуем к прохождению вот этот урок: [Ошибки и исключения](https://stepik.org/lesson/24463/step/1?unit=6771).

## 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.

**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:**

1

* 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.

**Часть 2. Полезные методы Selenium**

2.1 Основные методы Selenium

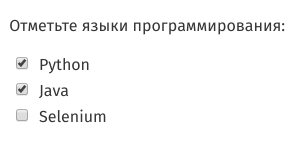
## Содержание урока:

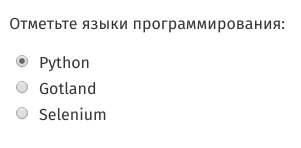
* [Как работать с элементами типа checkbox и radiobutton?](https://stepik.org/lesson/165493/step/2)
* [Задание на терминологию 1](https://stepik.org/lesson/165493/step/3)
* [Задание на терминологию 2](https://stepik.org/lesson/165493/step/4)
* [Задание: кликаем по checkboxes и radiobuttons (капча для роботов)](https://stepik.org/lesson/165493/step/5)
* [Метод get\_attribute](https://stepik.org/lesson/165493/step/6)
* [Задание: поиск сокровища с помощью get\_attribute](https://stepik.org/lesson/165493/step/7)

## Как работать с элементами типа checkbox и radiobutton?

Checkbox (чекбокс или флажок) и radiobutton (радиобаттон или переключатель) — часто используемые в формах элементы. Основная разница между ними состоит в том, что флажки позволяют выбирать/отключать любой из представленных вариантов, а переключатели позволяют выбрать только один из вариантов. Далее мы будем называть эти элементы на англоязычный манер: checkbox и radiobutton.

Так выглядят checkboxes:



А это radiobuttons:  
  
Оба этих элемента создаются при помощи тега input со значением атрибута type равным checkbox или radio соответственно. В html-коде страницы вы увидите:

<input type="checkbox">

<input type="radio">

Если checkbox или radiobutton выбран, то у элемента появится новый атрибут checked без значения. Часто атрибут checked уже установлен для одного из элементов по умолчанию.

<input type="checkbox" checked>

<input type="radio" checked>

Radiobuttons объединяются в группу, где все элементы имеют одинаковые значения атрибута name, но разные значения атрибута value:

<input type="radio" name="language" value="python" checked>

<input type="radio" name="language" value="selenium">

Checkboxes могут иметь как одинаковые, так и разные значения атрибута name. Поэтому и те, и другие лучше искать с помощью значения id или значения атрибута value. Если вы видите на странице чекбокс с уникальным значением name, то можете искать по name.

Чтобы снять/поставить галочку в элементе типа checkbox или выбрать опцию из группы radiobuttons, надо указать WebDriver метод поиска элемента и выполнить для найденного элемента метод click():

option1 = browser.find\_element(By.CSS\_SELECTOR, "[value='python']")

option1.click()

Также вы можете увидеть тег label рядом с input. Этот тег используется, чтобы сделать кликабельным текст, который отображается рядом с флажком. Этот текст заключен внутри тега label. Элемент label связывается с элементом input с помощью атрибута for, в котором указывается значение атрибута id для элемента input:

<div>

<input type="radio" id="python" name="language" checked>

<label for="python">Python</label>

</div>

<div>

<input type="radio" id="java" name="language">

<label for="java">Java</label>

</div>

В этом случае можно также отметить нужный пункт с помощью WebDriver, выполнив метод click() на элементе label.

option1 = browser.find\_element(By.CSS\_SELECTOR, "[for='java']")

option1.click()

## Задание: кликаем по checkboxes и radiobuttons (капча для роботов)

Продолжим использовать силу роботов 🤖 для решения повседневных задач. На данной [странице](https://suninjuly.github.io/math.html) мы добавили капчу для роботов, то есть тест, являющийся простым для компьютера, но сложным для человека.

Ваша программа должна выполнить следующие шаги:

1. Открыть страницу <https://suninjuly.github.io/math.html>.
2. Считать значение для переменной x.
3. Посчитать математическую функцию от x (код для этого приведён ниже).
4. Ввести ответ в текстовое поле.
5. Отметить checkbox "I'm the robot".
6. Выбрать radiobutton "Robots rule!".
7. Нажать на кнопку Submit.

Для этой задачи вам понадобится использовать атрибут .text для найденного элемента. Обратите внимание, что скобки здесь не нужны:

x\_element = browser.find\_element(By.CSS\_SELECTOR, selector\_value)

x = x\_element.text

y = calc(x)

Атрибут text возвращает текст, который находится между открывающим и закрывающим тегами элемента. Например, text для данного элемента <div class="message">У вас новое сообщение.</div> вернёт строку: "У вас новое сообщение".

Используйте функцию calc(), которая рассчитает и вернет вам значение функции, которое нужно ввести в текстовое поле. Не забудьте добавить этот код в начало вашего скрипта:

import math

def calc(x):

  return str(math.log(abs(12\*math.sin(int(x)))))

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Скопируйте его в поле ниже.

## Метод get\_attribute

Мы уже знаем, как найти нужный элемент на странице и как получить видимый пользователю текст. Для более детальных проверок в тесте нам может понадобиться узнать значение атрибута элемента. Атрибуты могут быть стандартными свойствами, которые понимает и использует браузер для отображения и вёрстки элементов или для хранения служебной информации, например, name, width, height, color и многие [другие](https://www.w3schools.com/tags/ref_attributes.asp). Также атрибуты могут быть созданы разработчиками проекта для задания собственных стилей или правил.

Значение атрибута представляет собой строку. Если значение атрибута отсутствует, то это равносильно значению атрибута равному "false". Давайте еще раз взглянем на страницу <http://suninjuly.github.io/math.html>. На ней есть radiobuttons, для которых выбрано значение по умолчанию. 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>

## Задание: поиск сокровища с помощью get\_attribute

В данной задаче вам нужно с помощью роботов решить ту же математическую задачу, как и в прошлом задании. Но теперь значение переменной **х** спрятано в "сундуке", точнее, значение хранится в атрибуте **valuex** у картинки с изображением сундука.

Ваша программа должна:

1. Открыть страницу <http://suninjuly.github.io/get_attribute.html>.
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.
4. Посчитать математическую функцию от x (сама функция остаётся неизменной).
5. Ввести ответ в текстовое поле.
6. Отметить checkbox "I'm the robot".
7. Выбрать radiobutton "Robots rule!".
8. Нажать на кнопку "Submit".

Для вычисления значения выражения в п.4 используйте функцию calc(x) из предыдущей задачи.

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Скопируйте его в поле ниже и нажмите кнопку "Submit", чтобы получить баллы за задание.

2.2 Работа с файлами, списками и js-скриптами

## Содержание урока:

* [Работа со списками](https://stepik.org/lesson/228249/step/2)
* [Задание: работа с выпадающим списком](https://stepik.org/lesson/228249/step/3)
* [Метод execute\_script](https://stepik.org/lesson/228249/step/4)
* [﻿Пример задания для execute\_script](https://stepik.org/lesson/228249/step/5)
* [Задание на execute\_script](https://stepik.org/lesson/228249/step/6)
* [Загрузка файлов](https://stepik.org/lesson/228249/step/7)
* [Задание: загрузка файла](https://stepik.org/lesson/228249/step/8)

## Работа со списками

На веб-страницах мы также встречаем раскрывающиеся (выпадающие) списки. У таких списков есть несколько важных особенностей:

1. У каждого элемента списка обычно есть уникальное значение атрибута value
2. В списках может быть разрешено выбирать как только один, так и несколько вариантов, в зависимости от типа списка
3. Визуально списки могут различаться тем, что в одном случае все варианты скрыты в выпадающем меню (<http://suninjuly.github.io/selects1.html>), а в другом все варианты или их часть видны (<http://suninjuly.github.io/selects2.html>)

Но для взаимодействия с любым вариантом списка мы будем использовать одни и те же методы Selenium.

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

from selenium.webdriver.common.by import By

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 "--".

## Метод execute\_script

Рассмотрим еще один очень полезный и мощный метод, но он требует хотя бы минимальных знаний JavaScript. С помощью метода execute\_script можно выполнить программу, написанную на языке JavaScript, как часть сценария автотеста в запущенном браузере. Зачем это может понадобиться, если в автотестах мы стараемся взаимодействовать с интерфейсом сайта как обычный пользователь, нажимая кнопки, выбирая пункты меню и вводя текст в текстовые поля?

Дело в том, что стандартные методы, доступные в Selenium, не могут покрыть всех возможных ситуаций работы с веб-приложением. Сайты в интернете могут решать самые разные задачи, начиная от простого блога до сложных финансовых или графических приложений. Разработчики имеют доступ к огромному количеству различных библиотек для решения бизнес-сценариев, что приводит к появлению на веб-странице нестандартных редакторов текстов, уникальных меню, оригинальных видео-плееров и т.д. Порой это приводит к тому, что для нажатия вроде бы обычной кнопки тестировщику понадобится писать настоящий JavaScript-сценарий. Если вы столкнулись с такой ситуацией, то в первую очередь обратитесь за помощью к вашим фронтенд-разработчикам, чтобы они подсказали  пример нужного скрипта. Прежде чем использовать данный скрипт в тестах, вы можете проверить, как он работает прямо в браузере, выполнив код в консоли браузера. Затем можете добавить его в ваш автотест с помощью execute\_script(javascript\_code).

Давайте попробуем 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 внутри скрипта вам также понадобится использовать кавычки, а для выделения скрипта вы уже используете двойные кавычки, то в скрипте следует поставить одинарные:

browser.execute\_script("document.title='Script executing';")

Такой формат записи тоже будет работать:

browser.execute\_script('document.title="Script executing";')

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

from selenium.webdriver.common.by import By

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.

## 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. 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.

Если совсем непонятно что происходит, пример:

Допустим, мы написали код скрипта и сохранили код в lesson2\_step7.py в своей 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)**.

2.3 Работа с окнами

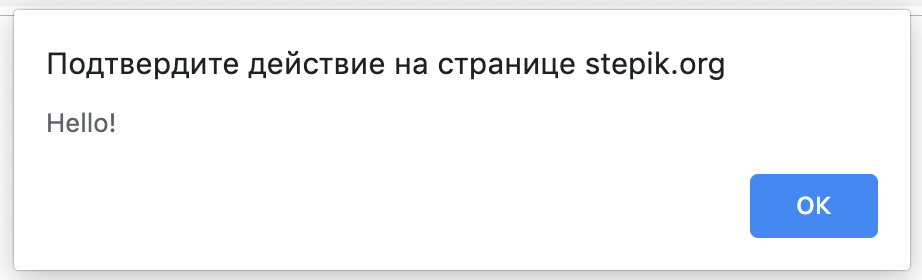
## Содержание урока

* [Alerts и как с ними жить](https://stepik.org/lesson/184253/step/2)
* [Задание: типы модальных окон](https://stepik.org/lesson/184253/step/3)
* [Задание: принимаем alert](https://stepik.org/lesson/184253/step/4)
* [Переход на новую вкладку браузера](https://stepik.org/lesson/184253/step/5)
* [Задание: переход на новую вкладку](https://stepik.org/lesson/184253/step/6)

## 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:

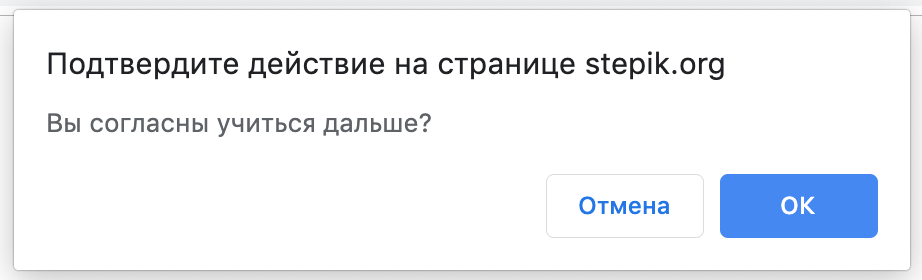
alert = browser.switch\_to.alert

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()

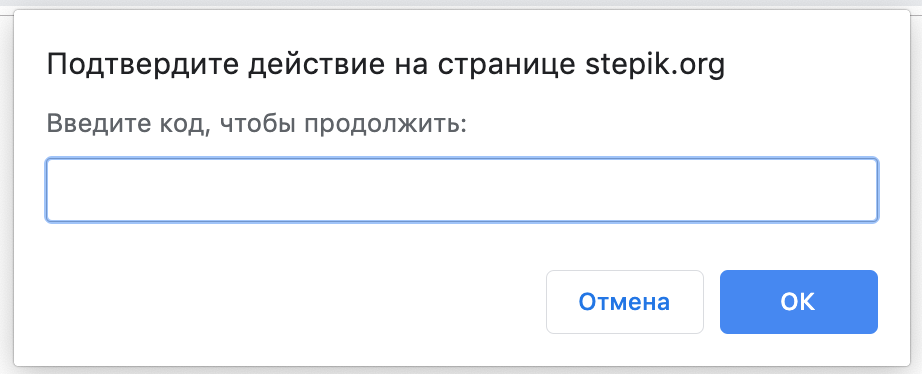
То же самое, что и при нажатии пользователем кнопки "Отмена".

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.

2.4 Настройка ожиданий

## Содержание урока

* [Немного про современный веб](https://stepik.org/lesson/181384/step/2)
* [Как работают методы get и find\_element](https://stepik.org/lesson/181384/step/3)
* [Давайте быстрее это починим: time.sleep()](https://stepik.org/lesson/181384/step/4)
* [Есть способы получше: Selenium Waits (Implicit waits)](https://stepik.org/lesson/181384/step/5)
* [Задание: Про Exceptions](https://stepik.org/lesson/181384/step/6)
* [Explicit Waits (WebDriverWait и expected\_conditions)](https://stepik.org/lesson/181384/step/7)
* [Задание: ждем нужный текст на странице](https://stepik.org/lesson/181384/step/8)
* [Итоги урока](https://stepik.org/lesson/181384/step/9)

## Немного про современный веб

Разработчики хорошо потрудились, чтобы в 2022 году веб-страницы выглядели красиво и быстро открывались, а переходы между страницами были практически незаметны. Страницы сайтов интерактивны и мгновенно реагируют на действия пользователя. Для реализации такого комфортного пользовательского опыта чаще всего используют подход [Single-Page Application](https://en.wikipedia.org/wiki/Single-page_application) (или [одностраничных приложений](https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%BD%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)), что в общем случае означает наличие одной страницы на сайте. Содержимое страницы при этом динамически обновляется с помощью JavaScript, который незаметно обменивается с сервером информацией, например, посредством REST API.

В целом все довольны. Разве что создателям автотестов на интерфейсы приходится туго. Неожиданно появляющиеся или пропадающие элементы на странице, непредсказуемое время полной отрисовки страницы, изменяющийся текст в кнопках или в сообщениях веб-сайта — эти особенности работы SPA-приложений приходится учитывать в автотестах, и, стоит признать, это является одним из самых сложных и головоломных аспектов разработки автотестов на Selenium (да и в других фреймворках для написания end-to-end тестов тоже).

В этом уроке мы рассмотрим подробнее самые распространенные проблемы и познакомимся со способами их решения.

## Как работают методы get и find\_element

Разберем еще один простой тест на WebDriver, проверяющий работу кнопки.

Тестовый сценарий выглядит так:

1. Открыть страницу <http://suninjuly.github.io/wait1.html>
2. Нажать на кнопку "Verify"
3. Проверить, что появилась надпись "Verification was successful!"

Для открытия страницы мы используем метод get, затем находим нужную кнопку с помощью одного из методов find\_element\_by\_ и нажимаем на нее с помощью метода click. Далее находим новый элемент с текстом и проверяем соответствие текста на странице ожидаемому тексту.

Вот как выглядит код автотеста:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

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

Попробуйте сначала выполнить тест вручную, а затем запустить автотест. В первом случае, вы завершите тест успешно, во втором случае автотест упадет с сообщением NoSuchElementException для элемента c **id="verify"**. Почему так происходит?

Команды в Python выполняются синхронно, то есть, строго последовательно. Пока не завершится команда get, не начнется поиск кнопки. Пока кнопка не найдена, не будет сделан клик по кнопке и так далее.

Но тест будет работать абсолютно стабильно, только если в данной веб-странице не используется JavaScript (что маловероятно для современного веба). Метод get дожидается информации от браузера о том, что страница загружена, и только после этого наш тест переходит к поиску кнопки. Если страница интерактивная, то браузер будет считать, что страница загружена, при этом продолжат выполняться загруженные браузером скрипты. Скрипт может управлять появлением кнопки на странице и показывать ее, например, с задержкой, чтобы кнопка красиво и медленно возникала на странице. В этом случае наш тест упадет с уже известной нам ошибкой NoSuchElementException, так как в момент выполнения команды button = browser.find\_element(By.ID, "verify") элемент с **id="verify"** еще не отображается на странице. На данной странице пауза перед появлением кнопки установлена на 1 секунду, метод **find\_element()** сделает только одну попытку найти элемент и в случае неудачи уронит наш тест.

## Давайте быстрее это починим: time.sleep()

Теперь, когда мы уже знаем, что кнопка появляется с задержкой, мы можем добавить паузу до начала поиска элемента. Мы уже использовали библиотеку **time** ранее. Давайте применим ее и сейчас:

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/wait1.html")

time.sleep(1)

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

button.click()

message = browser.find\_element(By.ID, "verify\_message")

assert "successful" in message.text

Теперь тест проходит. Но что если элемент с сообщением тоже будет появляться с задержкой? Добавить еще один **time.sleep()** перед поиском сообщения? А если изменится время задержки при появлении кнопки? Увеличим длительность паузы? А еще на разных машинах с разной скоростью интернета кнопка может появляться через разные промежутки времени. Можно перед каждым действием добавить задержку, но тогда значительную часть времени прогона тестов будут занимать бесполезные ожидания, при этом с увеличением количества тестов эта проблема будет только расти.

## Есть способы получше: Selenium Waits (Implicit Waits)

Надеемся, вы поняли, что решение с**time.sleep()**плохое: оно не масштабируемое и трудно поддерживаемое.

Идеальное решение могло бы быть таким: нам всё равно надо избежать ложного падения тестов из-за асинхронной работы скриптов или задержек от сервера, поэтому мы будем ждать появление элемента на странице в течение заданного количества времени (например, 5 секунд). Проверять наличие элемента будем каждые 500 мс. Как только элемент будет найден, мы сразу перейдем к следующему шагу в тесте. Таким образом, мы сможем получить нужный элемент в идеальном случае сразу, в худшем случае за 5 секунд.

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.

Улучшим наш тест с помощью неявных ожиданий. Для этого нам нужно будет убрать time.sleep() и добавить одну строчку с методом **implicitly wait**:

from selenium import webdriver

from selenium.webdriver.common.by import By

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**.

## 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

from selenium.webdriver.common.by import By

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

**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

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"))

)

## 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

from selenium.webdriver.common.by import By

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

**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

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"))

)

## Итоги урока

В этом уроке мы изучили способы сделать наши тесты менее зависимыми от внешних условий, на которые мы не можем влиять: например, особенности работы JavaScript и непредсказуемая длительность сетевых запросов. Также узнали, почему надо избегать использования **time.sleep()** в автотестах, стали пользоваться неявными и явными ожиданиями WebDriver (**Implicit** и**Explicit Waits**) и научились разбираться в исключениях WebDriver.

2.5 Итоги второго модуля

## Итоги модуля

В этом уроке мы еще раз пробежимся по основным методам и концепциям, которые мы использовали в этом модуле. Это поможет вам закрепить знания и лучше запомнить всё то, что мы изучили.

**Содержание:**

* [Итоги модуля: темы](https://stepik.org/lesson/236205/step/2)
* [Задание: работа с alert](https://stepik.org/lesson/236205/step/3)
* [Задание: методы Selenium Webdriver](https://stepik.org/lesson/236205/step/4)
* [Задание: методы Selenium Webdriver](https://stepik.org/lesson/236205/step/5)
* [Значения get\_attribute()](https://stepik.org/lesson/236205/step/6)
* [Expected conditions](https://stepik.org/lesson/236205/step/7)

2.6 Полезные ссылки к первому и второму модулям

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**

* <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://www.selenium.dev/documentation/webdriver/waits/>﻿﻿
* <https://stackoverflow.com/questions/15122864/selenium-wait-until-document-is-ready>
* <https://blog.codeship.com/get-selenium-to-wait-for-page-load/>
* <http://barancev.github.io/slow-loading-pages/>
* <http://barancev.github.io/page-loading-complete/>

**3  Тестовые фреймворки**

3.1 Лирическое отступление про Git

## Содержание урока

* [Немного про Git и почему это важно](https://stepik.org/lesson/187065/step/2)
* [Соцопрос](https://stepik.org/lesson/187065/step/3)
* [Первые шаги: создаем аккаунт и свой первый репозиторий](https://stepik.org/lesson/187065/step/4)
* [Клонируем репозиторий](https://stepik.org/lesson/187065/step/5)
* [Мануал и полезные команды](https://stepik.org/lesson/187065/step/6)
* [Первый коммит](https://stepik.org/lesson/187065/step/7)
* [Добавление изменений на сервер (push)](https://stepik.org/lesson/187065/step/8)
* [Задание: команды Git](https://stepik.org/lesson/187065/step/9)
* [Добавьте в репозиторий задачи из предыдущего модуля](https://stepik.org/lesson/187065/step/10)
* [Задание: проверьте решения других учащихся](https://stepik.org/lesson/187065/step/11)
* [Полезные ссылки](https://stepik.org/lesson/187065/step/12)