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?

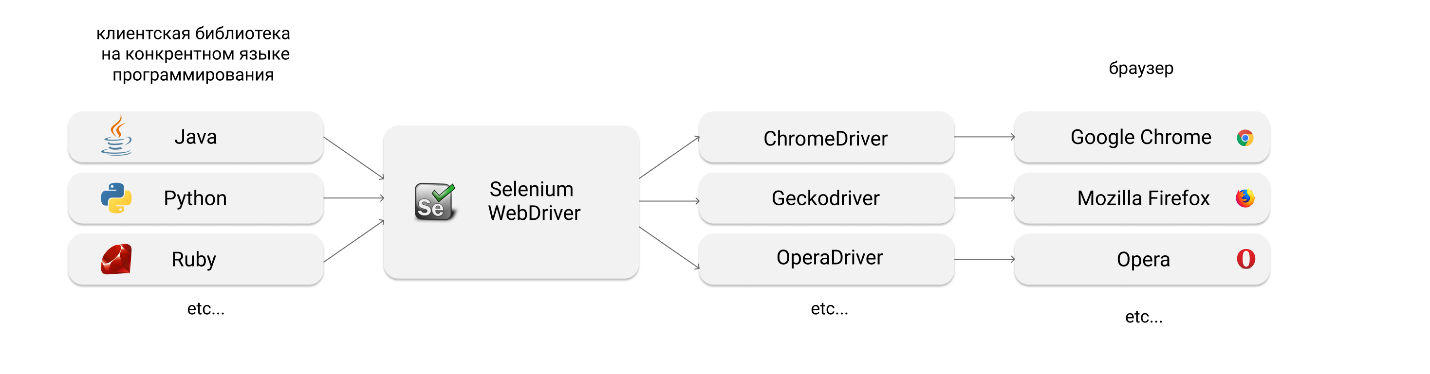
1. Очевидно, код на языке программирования. Там будет содержаться какая-то логика действий (например: залогинить пользователя, добавить товар в корзину) и проверки.
2. Браузер, чтобы имитировать действия пользователя. Желательно, какой-нибудь популярный. А еще лучше — несколько.

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.

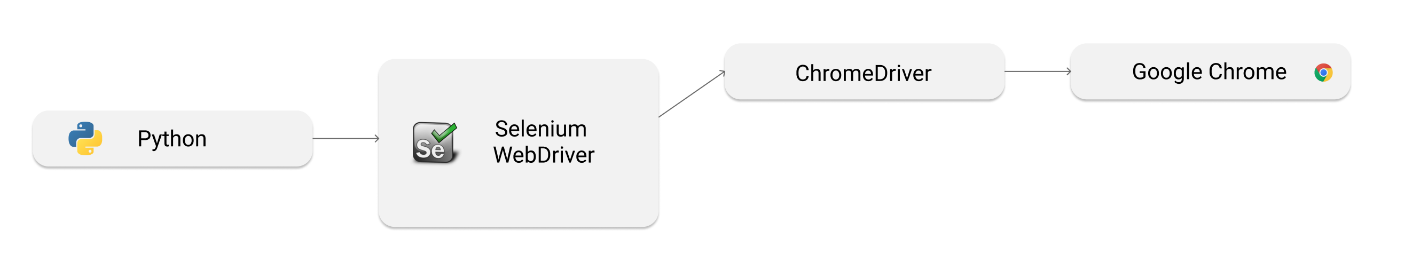
Итого получаем вот такую картинку:

[](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**

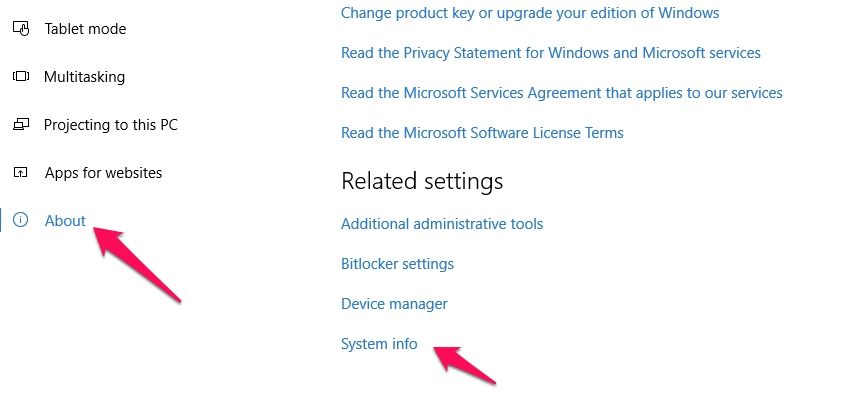
Для установки откройте сайт <https://sites.google.com/chromium.org/driver/> (старая версия сайта <https://sites.google.com/a/chromium.org/chromedriver/downloads>) и скачайте ту версию ChromeDriver, которая соответствует версии вашего браузера. Чтобы узнать версию браузера, откройте новое окно в Chrome, 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>.

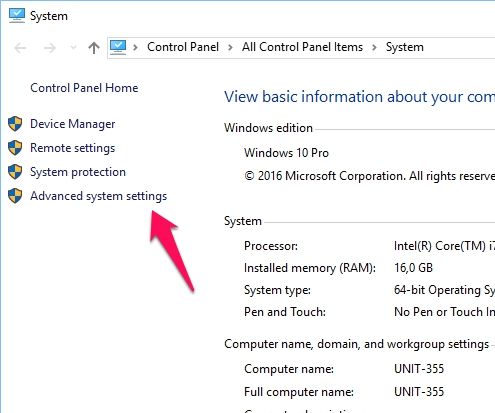
**Пример: как добавить путь в системную переменную PATH на Windows10**

1. Откройте настройки системы.

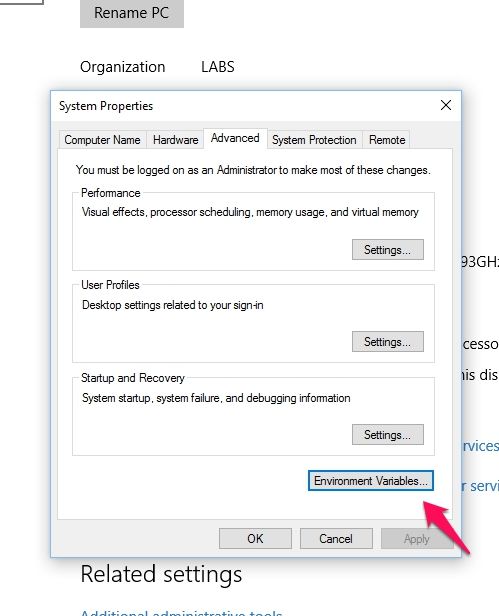
2. В настройках откройте вкладку About, затем System info:



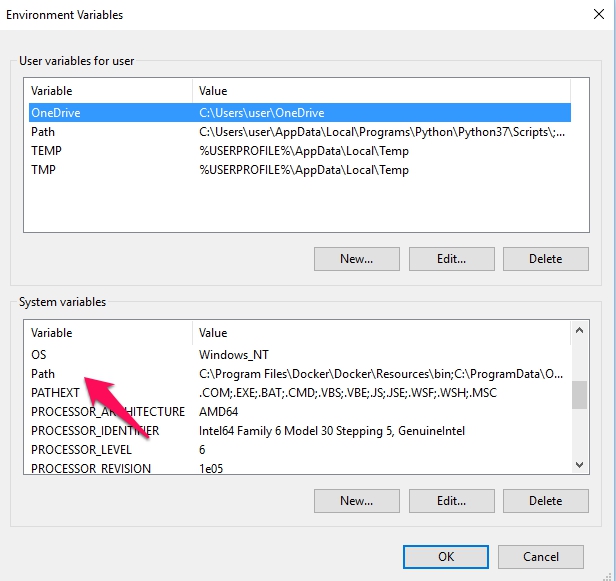
3. Выберите Advanced system settings:



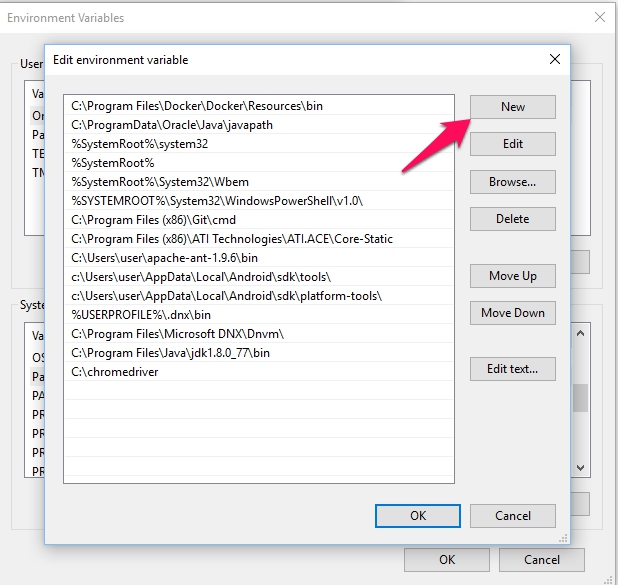
4. Выберите Environment Variables:



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



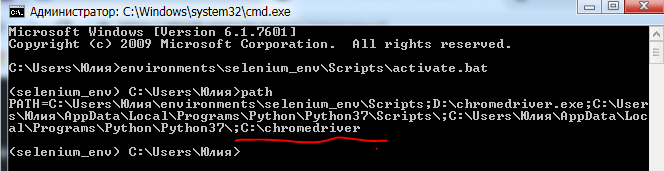
6. Нажмите кнопку New. Введите в новую строку путь к ChromeDriver — C:\chromedriver. Нажмите Enter. У вас должна появится строка с указанным путем:



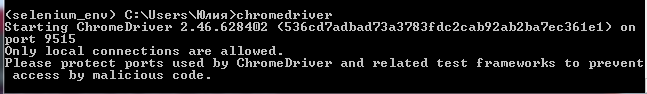
7. Если у вас была открыта командная строка Windows, не забудьте ее закрыть. Затем откройте новую командную строку, чтобы изменения переменной окружения стали доступны. Активируйте снова виртуальное окружение selenium\_env, которое мы создали в предыдущих шагах.

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">

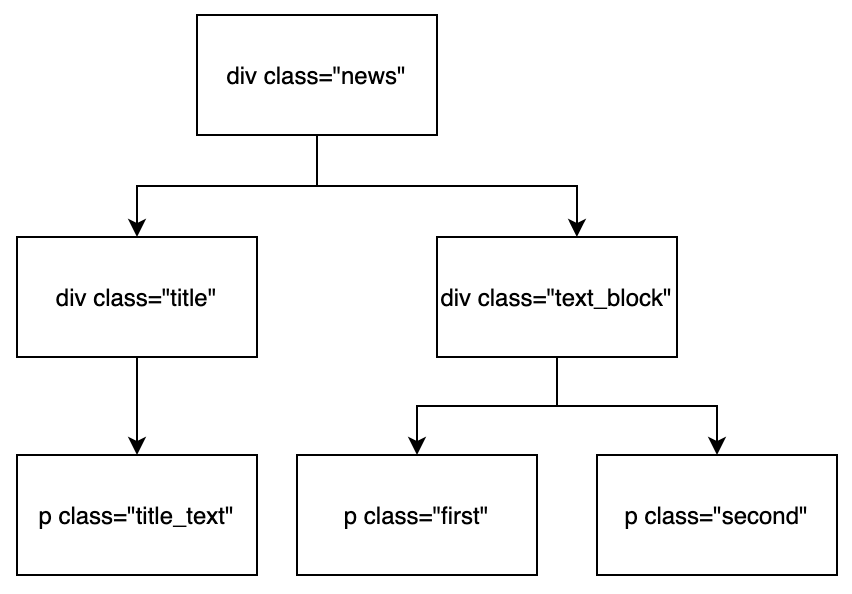
<p class="first">After 3 years from it’s a major release, now Selenium has put out its first alpha version of Selenium 4 on Apr 24, 2019.</p>

<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>

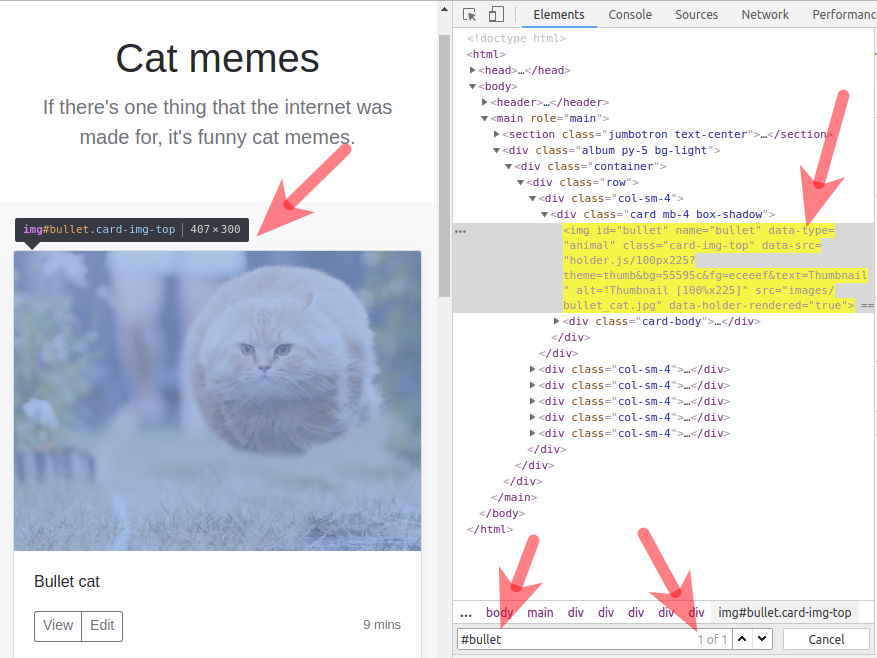
Для начала мы попробуем искать элементы вручную с помощью консоли браузера, а в следующем уроке научимся писать код, который выполняет ту же задачу поиска.

**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

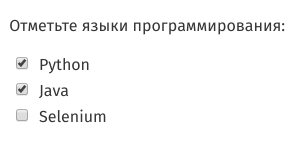
## Содержание урока:

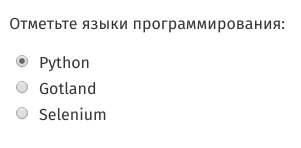
* [Как работать с элементами типа checkbox и radiobutton?](https://stepik.org/lesson/165493/step/2)
* [Задание на терминологию 1](https://stepik.org/lesson/165493/step/3)
* [Задание на терминологию 2](https://stepik.org/lesson/165493/step/4)
* [Задание: кликаем по checkboxes и radiobuttons (капча для 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

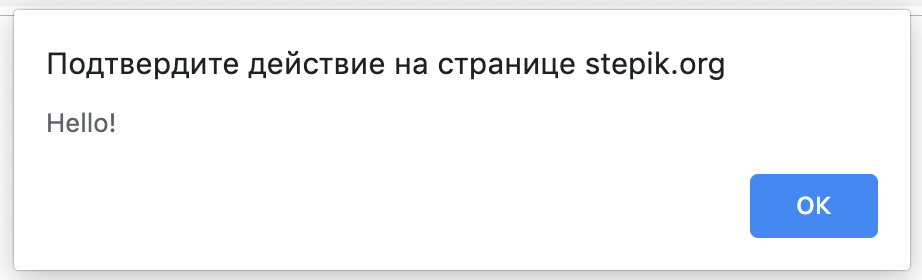
## Содержание урока

* [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:

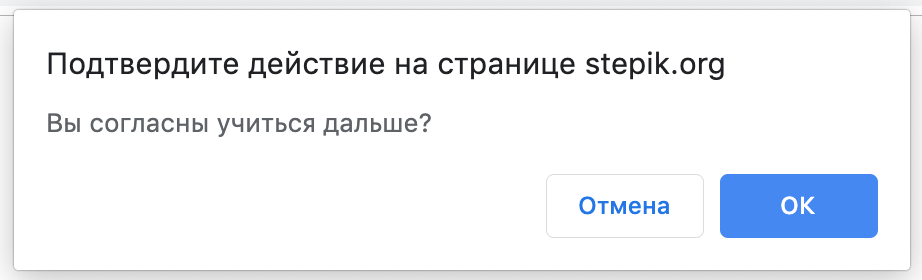
alert = browser.switch\_to.alert

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()

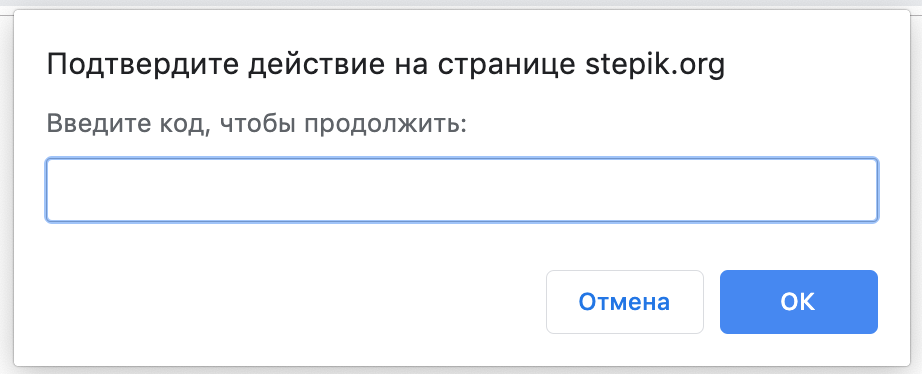
То же самое, что и при нажатии пользователем кнопки "Отмена".

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)

## Немного про Git и почему это важно

(Если вы свободно работаете с Git, можете пропустить теорию, решить задачки и получить свои баллы. Всем остальным читать внимательно!)

Представьте, что вы долго работали над проектом с вашим кодом, но в день перед дедлайном ваш жесткий диск решил сгореть.

Или вы сделали необратимое изменение, и весь ваш код перестал работать, хотя вот вчера же все было хорошо. Что делать?

Или вам нужно работать совместно над проектом с несколькими людьми. Как синхронизировать действия?

Для решения всех этих проблем люди придумали системы контроля версий. Git — одна из наиболее популярных, и де-факто стандарт в современном мире IT практически для всех, кто пишет код.

Git позволяет сохранять изменения в файлах с кодом локально и хранить полную копию кода на удаленном сервере — например, github.com. Репозиторий с вашим проектом полезно иметь для себя: чтобы иметь доступ к коду с разных компьютеров, показывать его другим людям и работать над кодом совместно, а еще это отличное дополнение к резюме — оно покажет ваши навыки на практике.

По завершении этого курса у вас будет репозиторий с решениями задач по основным темам, которые мы охватим, и небольшой финальный проект со структурой из нескольких файлов, приближенный к тому, что бывает в проектах по автоматизации тестирования в реальной жизни.

Стоит сказать, что полноценное изучение всех-всех команд выходит за рамки данного курса, однако умение работать с системами контроля версий — важный навык для тестировщика, а для автоматизатора — жизненно важный! В данном уроке мы рассмотрим только основные команды, которые будут нужны для работы с этим курсом.

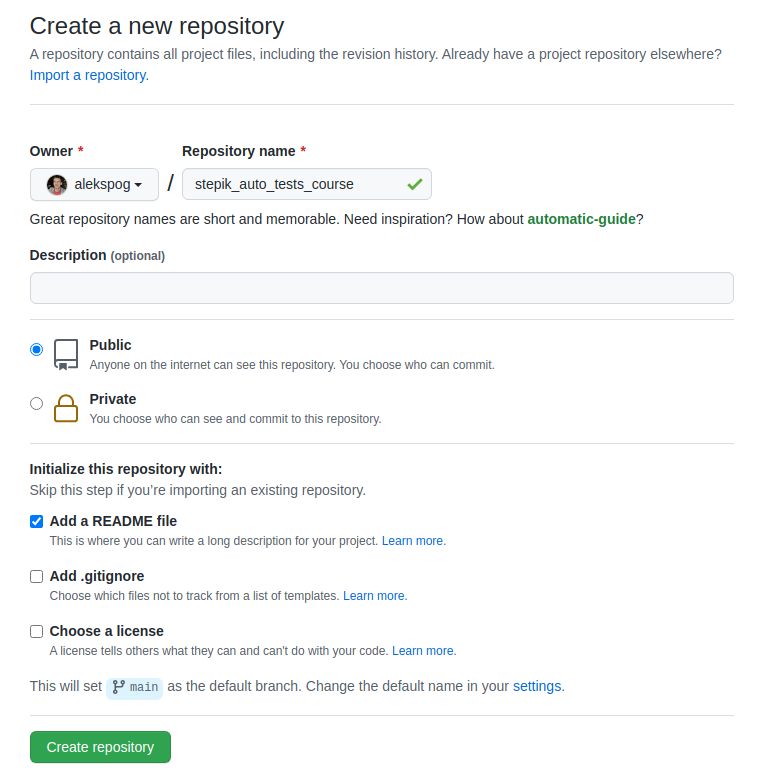
Подробнее предлагаем вам почитать самостоятельно — вам это обязательно пригодится.

## Первые шаги: создаем аккаунт и свой первый репозиторий

Для начала нужно создать аккаунт на [https://github.com](https://github.com/). Выберите имя пользователя, введите почту и пароль. Лучше сразу подтвердить адрес электронной почты.

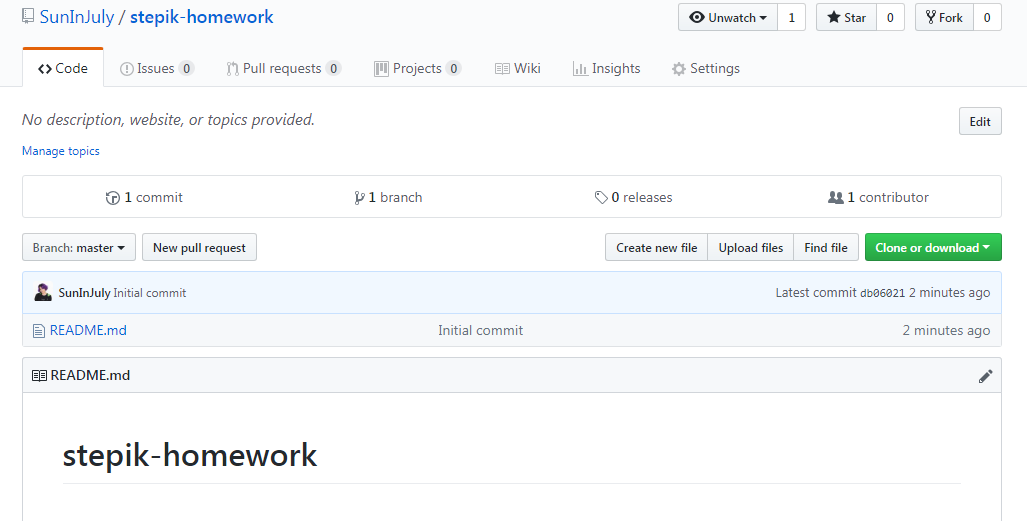
Время создать свой первый репозиторий — это место, где будет храниться ваш проект.

1. В правом верхнем углу рядом с аватаркой нажмите **+**и выберите "New repository".
2. Выберите имя для вашего репозитория. Лучше всего выбирать что-то осмысленное — чтобы через год легко было догадаться, что это и о чем. Например, "stepik\_auto\_tests\_course".
3. Сформулируйте короткое описание для проекта. Здесь можно написать о том, что это домашние задания к курсу.
4. Выбираем Public (публичный) репозиторий: это означает, что любой сможет его найти и открыть.
5. Ставим галочку Initialize this repository with README.
6. Нажимаем кнопку Create repository.



Ура! Теперь у вас есть собственный репозиторий! Пока здесь ничего нет, но мы это скоро исправим.

У вас должно получиться примерно следующее:



## Клонируем репозиторий

Дальше мы будем работать с файлами локально — прямо на своем компьютере. Для этого нам понадобится клиент Git — это программа, которая отслеживает и записывает изменения в файлах.

**Ubuntu:**

**apt-get install git**

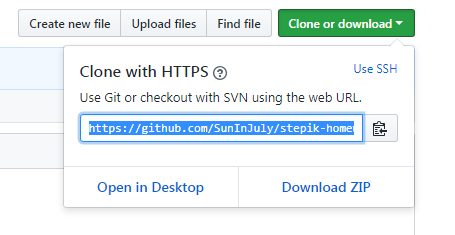
**Windows:**

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

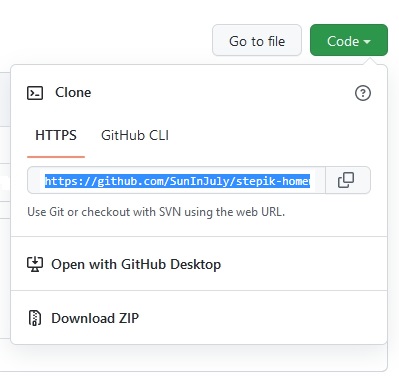
В Ubuntu можно выполнять команды в родном терминале, в Windows нужно запустить установленный **git bash.**

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

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



Некоторое время назад Git немного изменил дизайн. Теперь нужно нажать на зеленую кнопку Code и скопировать адрес из строки:

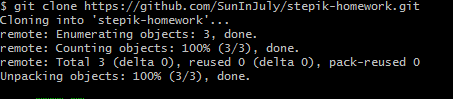


Чтобы склонировать к себе введите в консоли:

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

Чтобы перейти в каталог репозитория, введите:

**cd \*имя репозитория \***



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

## Мануал и полезные команды

В этом шаге мы рассмотрим несколько полезных команд, которые вам могут пригодиться при работе с Git.

Для начала посмотрим, что Git умеет. Введите в консоль ключевое слово **git**, чтобы увидеть популярные команды:



Если вы хотите посмотреть подсказку по какой-то конкретной команде, то можете ввести **git help <команда>** или **git <команда> --help**. Вы увидите подробную документацию по этой команде из руководства пользователя. Мануал, как и поисковик, — ваш лучший друг, не пренебрегайте его помощью.

Обратите внимание на две команды: **status** и **log**.

Попробуйте запустить команду **git status** и посмотрите на вывод. Данная команда позволяет отслеживать, какие изменения в репозитории вы сделали, что добавили в отслеживаемые файлы, что попало в коммит, что нет и так далее. Словом, это полезная команда, которая позволяет понимать, что происходит.

## Первый коммит

Для первого изменения будем использовать уже существующий файл: README.md.

Откройте его в текстовом редакторе (например, блокноте). Напишите пару слов о своем проекте: что вы собираетесь в нем хранить. Например, можете оставить там ссылку на этот курс 😉.

Сохраните и закройте файл.

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

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

**git add README.md**

Таким образом можно добавить любой файл, просто укажите его имя. Обратите внимание, что имена файлов зависят от регистра (case sensitive)!

Теперь гит будет отслеживать все изменения, которые вы будете производить в этом файле. Попробуйте выполнить **git status**и посмотрите, что изменилось в выводе.

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

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

Чтобы сделать коммит, нужно ввести команду:

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

(обратите внимание на кавычки!)

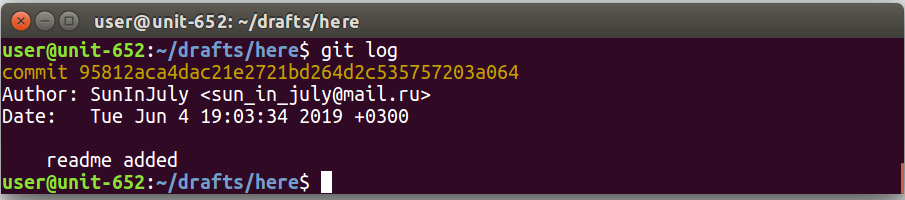
Если все прошло как надо, в ответе вы увидите, сколько строк и файлов изменилось:



Ура, вы сделали первый коммит!

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

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



## Добавление изменений на сервер (push)

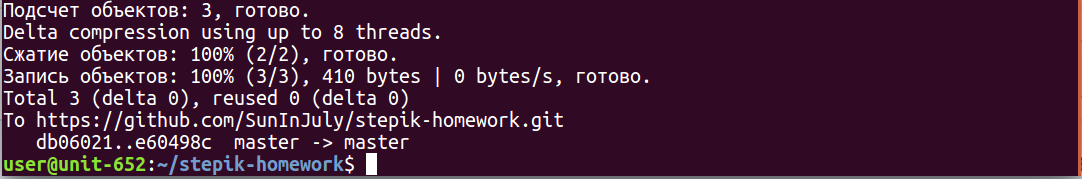
Сейчас у вашего репозитория есть две разные копии — одна локальная, которая уже содержит изменения в файле, и удаленная — на гитхабе. Необходимо наши локальные коммиты положить в удаленный репозиторий. Для этого есть специальная команда **git push <репозиторий><название ветки>**.

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


Выполним команду:

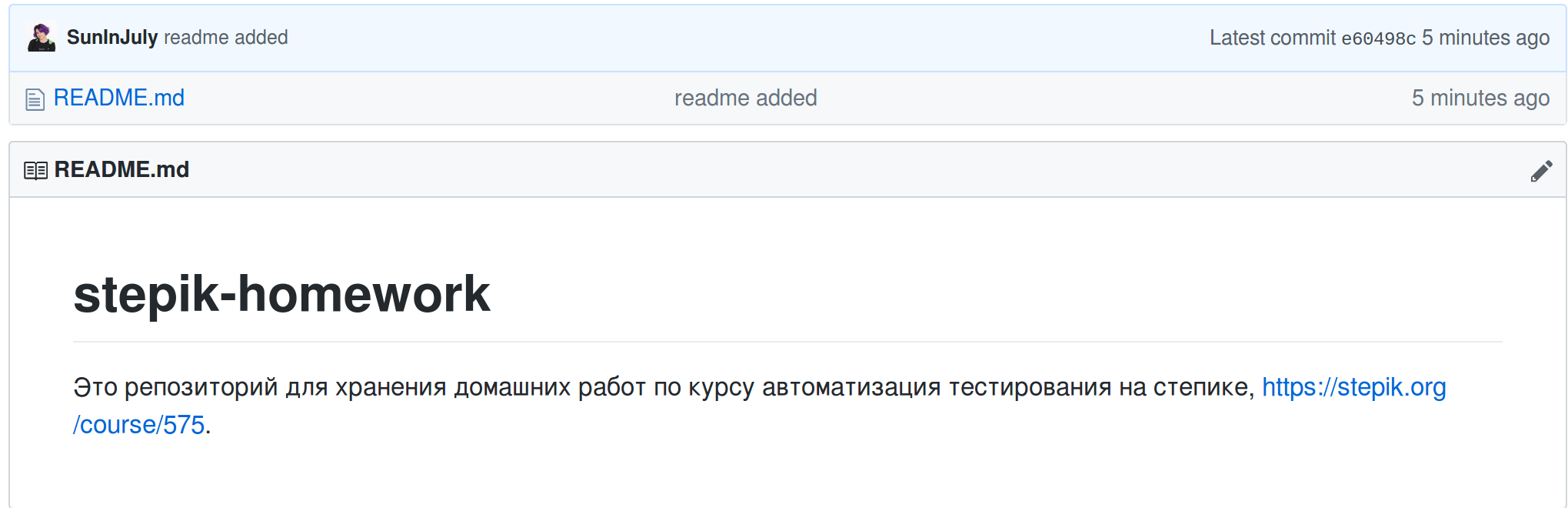
**git push origin master**

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



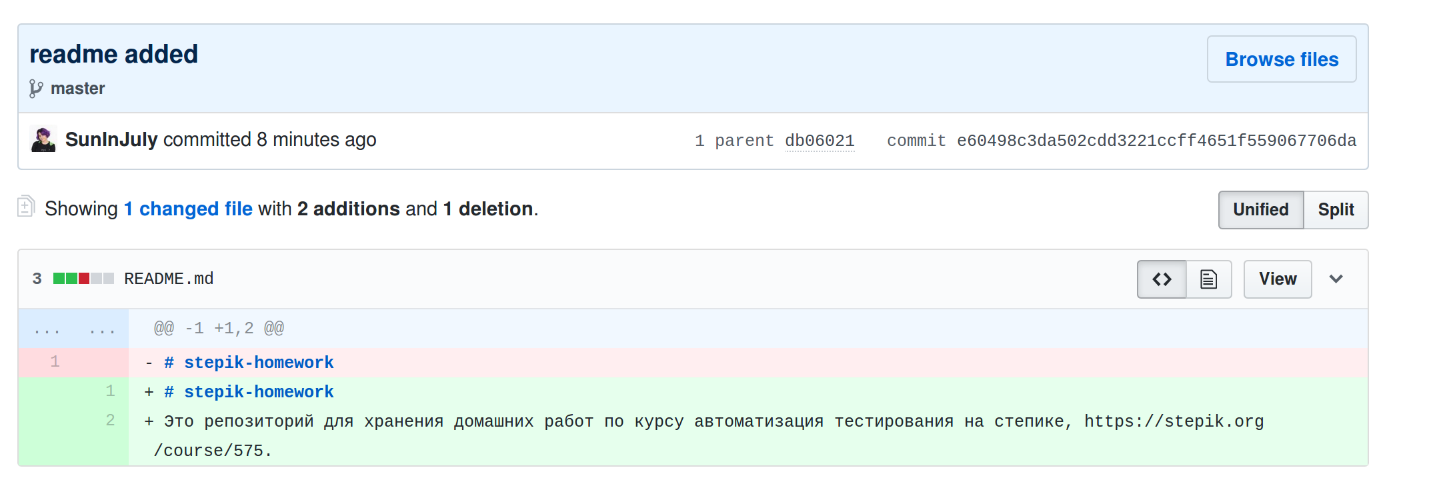
Это сообщение о том, сколько данных откуда и куда отправились.

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



Обратите внимание, что рядом с файлом появилась дополнительная информация: когда файл был последний раз модифицирован и какое было сообщение у этого коммита.

Можно, например, кликнуть на сообщение коммита и посмотреть, какие были изменения:



Зеленые строки — это те, которые добавились, а красные — те, которые были удалены.

Вот и все, вы молодец!

3.2 Тестирование web-приложений и тестовые фреймворки

## Тестирование web-приложений и тестовые фреймворки

Далее мы рассмотрим, как использовать Selenium Webdriver для написания автоматических тестов. Почему мы еще не можем назвать тестами скрипты, которые мы писали в предыдущих модулях?

Для этого нам придётся познакомиться с тестовыми фреймворками unittest и PyTest, которые позволяют создавать легко читаемые проверки ожидаемых результатов в тестах, удобно настраивать запуск большого количества тестов в нужных окружениях, организовывать хранение тестов и генерацию отчётов для последующего анализа.

В качестве основы для данного урока мы адаптировали руководство про написание юнит-тестов в Python:

<https://realpython.com/python-testing/>

## Содержание урока

* [Автоматизированное и ручное тестирование](https://stepik.org/lesson/36285/step/2)
* [Юнит-тесты и интеграционные тесты](https://stepik.org/lesson/36285/step/3)
* [Задание: пирамида тестирования](https://stepik.org/lesson/36285/step/4)
* [Структура теста](https://stepik.org/lesson/36285/step/5)
* [Проверка ожидаемого результата](https://stepik.org/lesson/36285/step/6)
* [Составные сообщения об ошибках](https://stepik.org/lesson/36285/step/7)
* [Задание: составные сообщения об ошибках](https://stepik.org/lesson/36285/step/8)
* [Задание: составные сообщения об ошибках и поиск подстроки](https://stepik.org/lesson/36285/step/9)
* [Тестовый сценарий](https://stepik.org/lesson/36285/step/10)
* [Выбор test runner](https://stepik.org/lesson/36285/step/11)
* [unittest](https://stepik.org/lesson/36285/step/12)
* [Задание: оформляем тесты в стиле unittest](https://stepik.org/lesson/36285/step/13)

**Автоматизированное и ручное тестирование**

Чтобы начать писать хорошие автотесты, нужно ﻿разобраться, в чем плюсы и минусы ручного и автоматизированного тестирования.

| **Ручные тесты** | **Автотесты** |
| --- | --- |
| Шаги могут быть описаны достаточно абстрактно, поэтому тестировщик может увидеть новые проблемы, следуя сценарию теста.  В ручном тесте может содержаться шаг "Зарегистрировать нового пользователя" без пояснений, как это делать, в автотесте должны быть строго описаны данные для теста и сценарий регистрации. | Строгий порядок шагов и их чёткая детализация. Проверяют только то, что было заложено изначально в сценарий теста, поэтому могут быть пропущены новые ошибки в продукте. |
| - Из-за абстрактности описания шагов ручные тесты иногда могут приводить к ложноположительным результатам, когда ожидаемый результат достигнут, но в реальности тестировщик выполнил сценарий неправильно, пропустив баг. | + Запуск автотестов обычно приводит к одним и тем же результатам, т.к. сценарий описан точно. |
| - Требуется много времени для проверки регрессии, что приводит к усталости тестировщика и ошибкам в проверках из-за человеческого фактора (человеку сложно долго выполнять монотонную работу). | + Можно запускать на каждый коммит в тестируемом приложении, что позволяет раньше обнаруживать ошибки в продукте. |

Отдельно стоит описать отрицательные стороны автоматизированного тестирования:

* возможная нестабильность теста, которая не связана с качеством самого теста, а вызывается внешними проблемами: нестабильное сетевое соединение, проблемы с серверами, обновление кода продукта в момент запуска тестов;
* требуется достаточно много времени на разработку и поддержку набора автотестов.

Несмотря на наличие минусов автотестов, в большинстве случаев их использование на проекте помогает быстрее находить ошибки в коде приложения и поддерживать качество продукта на достаточном уровне. Автотесты помогают отделу тестирования оптимизировать свою работу, чтобы сделать счастливее пользователей, которые быстрее получают новые фичи и меньше страдают от ошибок в продукте, так как разработчики раньше узнают о багах и могут заранее принять меры для их устранения, не откладывая релиз продукта. Таким образом, благодаря автотестам налаживается бизнес, который может увеличить скорость внедрения фич, приносящих прибыль.

Ручные тесты в идеальном случае остаются только на этапе проверки новых фич и в виде исследовательского тестирования, которое позволяет найти проблемы в сложных пользовательских сценариях.

## Юнит-тесты и интеграционные тесты

Если вы работаете в тестировании, то уже знаете разницу между юнит-тестами и интеграционными тестами. Юнит-тесты проверяют очень маленький кусок кода, обычно конкретную функцию, и чаще всего их пишут разработчики, которые хорошо понимают возможные крайние случаи для своего стека технологий. Интеграционные тесты проверяют взаимодействие сразу нескольких систем. Они могут создаваться и поддерживаться как разработчиками и тестировщиками, так и аналитиками (если для них разработан удобный фреймворк для написания тестов).

Юнит-тесты всегда автоматизированы, так как проверяют непосредственно работу кода. Интеграционные тесты могут быть ручными и автоматизированными. Иногда выделяют отдельную категорию end-to-end (е2е) тестов, которые проверяют полный стек технологий приложения и пользовательский сценарий взаимодействия с приложением как с черным ящиком. Если говорить про UI-тесты, которые разрабатываются с помощью Selenium, то их стоит отнести к разряду end-to-end тестов, так как они проверяют совместную работу всех систем web-продукта: работу frontend и backend, работу базы данных, дополнительные сервисы, такие как аналитика, платежные системы и так далее.

Подробно про разные типы автотестов мы говорить не будем, но советуем вам изучить теорию самостоятельно. Вот, например, отличная и подробная статья: [Пирамида тестов на практике](https://habr.com/ru/post/358950/).

**Структура теста**

Для написания UI-тестов можно использовать те же возможности Python, что и для написания юнит-тестов, которые создаются разработчиками.

Любой тест должен содержать:

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

Давайте обсудим, как именно можно производить проверки.

## Проверка ожидаемого результата

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

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

source selenium\_env/bin/activate

python

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

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

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

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

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

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AssertionError

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

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

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AssertionError: Should be absolute value of a number

## Составные сообщения об ошибках

Отдельно хочется поговорить про качество сообщений об ошибках, которые показываются при падении теста. Почему это важно? Хорошо написанный текст помогает быстро локализовать найденный баг и разобраться в том, что произошло и из-за чего тест упал. Хороший assert сэкономит вам часы вашей работы, особенно когда количество тестов переходит за сотню.

В целом, тут как с любым фидбеком: важно давать его точно и актуально. Если вы проверяете наличие элемента, то обязательно пишите, что это за элемент по смыслу на странице:

assert self.is\_element\_present('create\_class\_button', timeout=30), "No create class button"

Примечание: Функция is\_element\_present() вспомогательная. Как её реализовать и использовать, мы разберемся чуть позжe.

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

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

Если вы работаете с каким-то текстом (например, проверяете информационное сообщение, текущий url, ссылку, placeholder в input-элементе или любой другой текст), в сообщении об ошибке всегда лучше выводить оба значения: то, которое ожидалось, и то, которое получили по факту. Всё как в хорошем багрепорте: ожидаемый и фактический результат.

### Форматирование строк с помощью конкатенации

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

actual\_result = "abrakadabra"

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

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

### Форматирование строк с помощью str.format

Гораздо лучше воспользоваться возможностью python для форматирования строк. Дополнительно можно почитать здесь: <https://realpython.com/python-string-formatting/#2-new-style-string-formatting-strformat>

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

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

Попробуйте запустить её в интерпретаторе:

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

Такая строка при исполнении кода превратится в:

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

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

### Форматирование строк с помощью f-strings

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

Подробнее про f-strings можно почитать здесь: <https://realpython.com/python-string-formatting/#3-string-interpolation-f-strings-python-36>. Так как мы предполагаем, что вы используете последнюю версию Python, то предлагаем вам применять именно этот подход в данном курсе.

Пример 1:

str1 = "one"

str2 = "two"

str3 = "three"

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

Итог выполнения выражений в интерпретаторе:

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

Пример 2:

actual\_result = "abrakadabra"

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

Итог выполнения выражений в интерпретаторе:

Wrong text, got abrakadabra, something wrong

Пример 3:

>>> f"{2+3}"

'5'

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

**неправильно:**

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

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

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

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

**правильно:**

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

assert catalog\_text == "Каталог", \

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

## Задание: составные сообщения об ошибках и поиск подстроки

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

s = 'My Name is Julia'

if 'Name' in s:

print('Substring found')

index = s.find('Name')

if index != -1:

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

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

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

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

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

Реализуйте подобную проверку самостоятельно.

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

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

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

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

## Тестовые сценарии

Созданные тесты нужно сохранить в файле, чтобы его было удобно запускать и хранить в системе контроля версий. Давайте создадим файл**test\_abs\_project.py** и напишем в нём следующий код:

def test\_abs1():

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

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

test\_abs1()

print("All tests passed!")

Мы поместили тестовый сценарий в функцию для разделения тест-кейсов и возможности их независимого запуска.

Не вдаваясь в подробности, скажем только, что конструкция **if \_\_name\_\_ == "\_\_main\_\_"** служит для подтверждения того, что данный скрипт был запущен напрямую, а не вызван внутри другого файла в качестве модуля. Весь код написанный в теле этого условия будет выполнен только если пользователь запустил файл самостоятельно. Подробнее можно ознакомиться в видео [Олега Молчанова](https://www.youtube.com/watch?v=cW_-zGG4ef4).

В этой конструкции мы вызвали функцию**test\_abs1()**, которая выполняет тестовый сценарий.

С помощью **print("All tests passed!")** мы вывели сообщение, если все тесты прошли успешно.

Чтобы запустить тест, выполните в консоли команду:

python test\_abs\_project.py

Вы должны увидеть в консоли сообщение**"All tests passed!"**.

Если нам нужно добавить еще один тест, мы можем написать его как функцию в этом же файле. В приведенном примере мы уже не увидим сообщение "Everything passed", так как падение любого теста вызывает выход из программы:

def test\_abs1():

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

def test\_abs2():

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

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

test\_abs1()

test\_abs2()

print("Everything passed")

Запустите файл снова. Вы должны увидеть сообщение об упавшем втором тесте:

$ python test\_abs\_project.py

Traceback (most recent call last):

  File "test\_abs\_project.py", line 9, in <module>

    test\_abs2()

  File "test\_abs\_project.py", line 5, in test\_abs2

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

AssertionError: Should be absolute value of a number

**Выбор test runner**

В предыдущих шагах мы научились писать простые тесты и запускать их с помощью Python. Приведём здесь код тестов и результаты запуска из предыдущего шага еще раз.

**test\_abs\_project.py:**

def test\_abs1():

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

def test\_abs2():

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

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

    test\_abs1()

test\_abs2()

    print("Everything passed")

Консоль:

$ python test\_abs\_project.py

Traceback (most recent call last):

  File "test\_project.py", line 9, in <module>

    test\_abs2()

  File "test\_project.py", line 5, in test\_abs2

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

AssertionError: Should be absolute value of a number

Рассмотрим минусы такого подхода к запуску автотестов:

* Когда тестов становится много, сложно становится запускать только тесты из нужных тест-сьютов.
* Для каждого теста нужно создавать тестовые данные и окружение отдельно. Например, если мы захотим для каждого теста запускать браузер, а после завершения теста браузер закрывать, то логику работы с браузером придется дублировать в коде каждого теста.
* Если один из тестов завершится с ошибкой, например, тест упадёт с ошибкой AssertionError, то последующие тесты не запустятся. Мы не узнаем, были ли проблемы в этих тестах, пока не починим упавший тест или пока не запустим эти тесты по отдельности.

Для решения этих проблем и упрощения написания и запуска тестов существуют специальные фреймворки, которые называются test runners (тест-раннеры). Можно выделить три основных тестовых фреймворка для Python: **unittest**, **PyTest** и **nose**. Модуль **unittest** является встроенным инструментом Python — и это его большой плюс. **PyTest** и **nose** устанавливаются дополнительно, они позволяют получить расширенные возможности по сравнению с **unittest**. Мы кратко рассмотрим, как используется **unittest**, а затем изучим возможности **PyTest**, который позволяет писать более простой код тестов по сравнению с **unittest** и гибко настраивать запуск тестов. Еще один плюс использования **PyTest** в том, что для него существует большое количество плагинов, которые позволяют решить практически любую проблему, связанную с запуском автотестов.

## unittest

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

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

Для unittest существуют собственные дополнительные правила:

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

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

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

import unittest

class TestAbs(unittest.TestCase):

def test\_abs1(self):

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

def test\_abs2(self):

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

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

unittest.main()

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

python test\_abs\_project.py

.F

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

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

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

Traceback (most recent call last):

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

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

AssertionError: Should be absolute value of a number

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

Ran 2 tests in 0.000s

FAILED (failures=1)

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

В следующем уроке мы рассмотрим преимущества и особенности использования тестового фреймворка **PyTest**. Если вы хотите использовать **unittest** ﻿в своих проектах, вы можете изучить [документацию](https://docs.python.org/3/library/unittest.html) самостоятельно.

3.3 Тестирование с помощью PyTest

## Содержание урока

* [PyTest — преимущества и недостатки](https://stepik.org/lesson/193188/step/2)
* [Задание: вывод PyTest](https://stepik.org/lesson/193188/step/3)
* [Фиксируем пакеты в requirements.txt](https://stepik.org/lesson/193188/step/4)
* [PyTest: правила запуска тестов](https://stepik.org/lesson/193188/step/5)
* [Задание: запуск тестов](https://stepik.org/lesson/193188/step/6)
* [PyTest — отчёты](https://stepik.org/lesson/193188/step/7)
* [PyTest — как пишут тесты](https://stepik.org/lesson/193188/step/8)
* [PyTest — проверка ожидаемого результата (assert)](https://stepik.org/lesson/193188/step/9)

## PyTest — преимущества и недостатки

Для написания данного урока мы вдохновлялись [статьёй](https://habr.com/post/269759/) на Хабре, адаптировав ее к специфике тестирования веб-приложений с помощью Selenium WebDriver.

Рассмотрим преимущества использования PyTest:

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

**Для Windows:**

> selenium\_env\Scripts\activate.bat

(selenium\_env) С:\Users\user\environments> pip install pytest==7.1.2

**Для Linux и macOS:**

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

(selenium\_env) $ pip install pytest==7.1.2

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

pytest test\_abs\_project.py

2) Подробный отчёт с поддержкой цветовых схем из коробки.

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

4) Для проверок используется стандартный assert из Python.

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

6) Дополнительные возможности по настройке фикстур.

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

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

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

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

Рассмотрим минусы PyTest:

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

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

## Фиксируем пакеты в requirements.txt

Количество пакетов в нашем проекте растет, а мы тем временем все дальше уходим от учебных кусочков скриптов в сторону настоящего тестового проекта, поэтому в этом шаге давайте зафиксируем все пакеты, которые мы используем. Это стандартная практика, которая позволяет быстро переключаться в свежее виртуальное окружение, а также работать нескольким людям над одним проектом, получая одинаковые результаты.

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

После чего выполните в терминале команду:

pip freeze > requirements.txt

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

Как их оттуда достать? Попробуйте создать новое виртуальное окружение (если нужно, вернитесь в [модуль 1](https://stepik.org/lesson/25969/step/3?unit=196192) за инструкциями) и активировать. После чего выполните команду:

pip install -r requirements.txt

В свежем окружении все пакеты установлены одной командой!

## ****PyTest: правила запуска тестов****

В этом шаге мы коротко обсудим важные особенности запуска тестов с помощью PyTest. Когда мы выполняем команду **pytest**, тест-раннер собирает все тесты для запуска по определенным правилам:

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

pytest scripts/selenium\_scripts

# найти все тесты в директории scripts/selenium\_scripts

pytest test\_user\_interface.py

# найти и выполнить все тесты в файле

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

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

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

Подробности: [Conventions for Python test discovery](https://docs.pytest.org/en/stable/goodpractices.html#conventions-for-python-test-discovery)

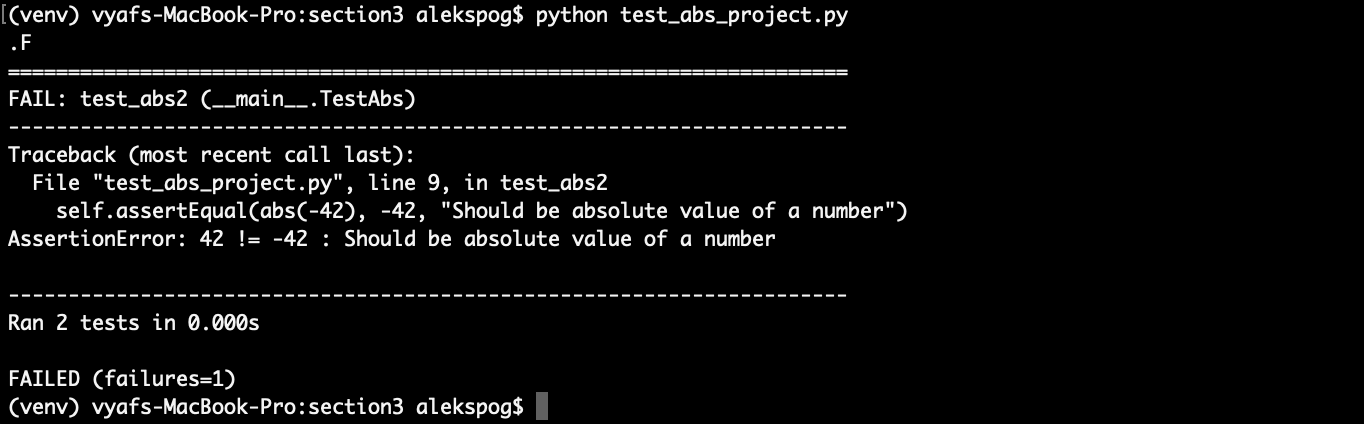
## PyTest — отчёты

Вы могли заметить, что PyTest позволяет генерировать подробный отчёт с поддержкой цветовых схем и форматированием прямо из коробки.

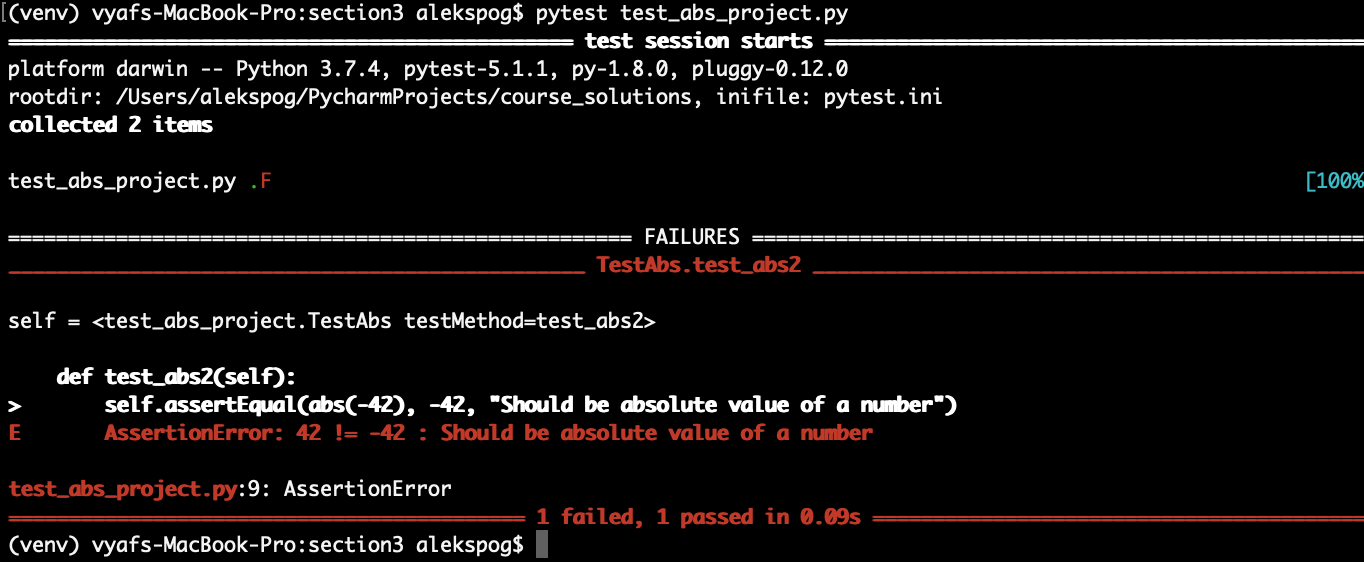
Давайте еще раз запустим наши тесты с помощью unittest и PyTest, чтобы сравнить выводимый результат.

Мы видим, что в PyTest-отчёте упавший тест выделен красным шрифтом, что делает разбор логов более приятным занятием.

**unittest:**



**PyTest:**



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



Другие полезные команды для манипуляции выводом тестов PyTest можно найти по ссылке: [Useful py.test commands.](https://gist.github.com/amatellanes/12136508b816469678c2" \t "_blank)

Useful py.test commands.

py.test test\_sample.py --collect-only # collects information test suite

py.test test\_sample.py -v # outputs verbose messages

py.test -q test\_sample.py # omit filename output

python -m pytest -q test\_sample.py # calling pytest through python

py.test --markers # show available markers

# In order to create a reusable marker.

/\*

# content of pytest.ini

[pytest]

markers =

webtest: mark a test as a webtest.

\*/

py.test -k "TestClass and not test\_one" # only run tests with names that match the "string expression"

py.test test\_server.py::TestClass::test\_method # cnly run tests that match the node ID

py.test -x # stop after first failure

py.test --maxfail=2 # stop after two failures

py.test --showlocals # show local variables in tracebacks

py.test -l # (shortcut)

py.test --tb=long # the default informative traceback formatting

py.test --tb=native # the Python standard library formatting

py.test --tb=short # a shorter traceback format

py.test --tb=line # only one line per failure

py.test --tb=no # no tracebak output

py.test -x --pdb # drop to PDB on first failure, then end test session

py.test --durations=10 # list of the slowest 10 test durations.

py.test --maxfail=2 -rf # exit after 2 failures, report fail info.

py.test -n 4 # send tests to multiple CPUs

py.test -m slowest # run tests with decorator @pytest.mark.slowest or slowest = pytest.mark.slowest; @slowest

py.test --traceconfig # find out which py.test plugins are active in your environment.

py.test --instafail # if pytest-instafail is installed, show errors and failures instantly instead of waiting until the end of test suite.

# Test using parametrize

/\*

import pytest

@pytest.mark.parametrize(

('n', 'expected'), [

(1, 2),

(2, 3),

(3, 4),

(4, 5),

pytest.mark.xfail((1, 0)),

pytest.mark.xfail(reason="some bug")((1, 0)),

pytest.mark.skipif('sys.version\_info >= (3,0)')((10, 11)),

]

)

def test\_increment(n, expected):

assert n + 1 == expected

\*/

## PyTest — как пишут тесты

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

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

def test\_abs1():

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

def test\_abs2():

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

Запустим тесты в этом файле:

pytest test\_abs.py

Код тестов стал короче и читабельнее.

3.4 Использование фикстур в PyTest

## Содержание урока

* [Классические фикстуры (fixtures)](https://stepik.org/lesson/237257/step/2)
* [Фикстуры, возвращающие значение](https://stepik.org/lesson/237257/step/3)
* [Финализаторы — закрываем браузер](https://stepik.org/lesson/237257/step/4)
* [Область видимости scope](https://stepik.org/lesson/237257/step/5)
* [Автоиспользование фикстур](https://stepik.org/lesson/237257/step/6)
* [Задание: область видимости фикстур](https://stepik.org/lesson/237257/step/7)

## Классические фикстуры (fixtures)

Важной составляющей в использовании PyTest является концепция фикстур. Фикстуры в контексте PyTest — это вспомогательные функции для наших тестов, которые не являются частью тестового сценария.

Назначение фикстур может быть самым разным. Одно из распространенных применений фикстур — это подготовка тестового окружения и очистка тестового окружения и данных после завершения теста. Но, вообще говоря, фикстуры можно использовать для самых разных целей: для подключения к базе данных, с которой работают тесты, создания тестовых файлов или подготовки данных в текущем окружении с помощью API-методов. Более подробно про фикстуры в широком смысле вы можете прочитать в [Википедии](https://en.wikipedia.org/wiki/Test_fixture#Software).

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

Можно создавать фикстуры для модулей, классов и отдельных функций. Давайте попробуем написать фикстуру для инициализации браузера, который мы затем сможем использовать в наших тестах. После окончания тестов мы будем автоматически закрывать браузер с помощью команды**browser.quit()**, чтобы в нашей системе не оказалось множество открытых окон браузера. Вынесем инициализацию и закрытие браузера в фикстуры, чтобы не писать этот код для каждого теста.

Будем сразу объединять наши тесты в тест-сьюты, роль тест-сьюта будут играть классы, в которых мы будем хранить наши тесты.

Рассмотрим два примера: создание экземпляра браузера и его закрытие только один раз для всех тестов первого тест-сьюта и создание браузера для каждого теста во втором тест-сьюте. Сохраните следующий код в файл**test\_fixture1.py** и запустите его с помощью PyTest. Не забудьте указать параметр **-s**, чтобы увидеть текст, который выводится командой print().

pytest -s test\_fixture1.py

**test\_fixture1.py:**

from selenium import webdriver

from selenium.webdriver.common.by import By

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

class TestMainPage1():

@classmethod

def setup\_class(self):

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

self.browser = webdriver.Chrome()

@classmethod

def teardown\_class(self):

print("quit browser for test suite..")

self.browser.quit()

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

self.browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self):

self.browser.get(link)

self.browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

class TestMainPage2():

def setup\_method(self):

print("start browser for test..")

self.browser = webdriver.Chrome()

def teardown\_method(self):

print("quit browser for test..")

self.browser.quit()

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

self.browser.get(link)

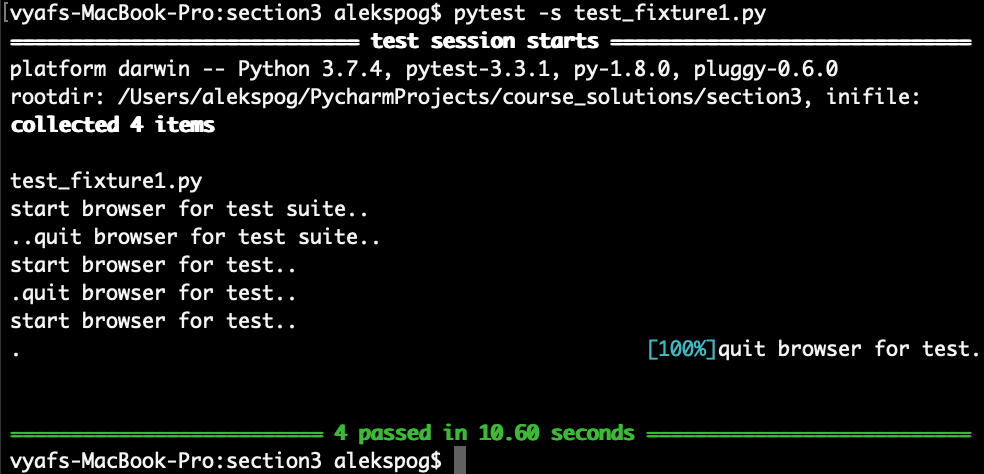
self.browser.find\_element(By.CSS\_SELECTOR, "#login\_link")

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self):

self.browser.get(link)

self.browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

В консоли видим:



Мы видим, что в первом тест-сьюте браузер запустился один раз, а во втором — два раза.

Данные и кэш, оставшиеся от запуска предыдущего теста, могут влиять на результаты выполнения следующего теста, поэтому лучше всего запускать отдельный браузер для каждого теста, чтобы тесты были стабильнее. К тому же если вдруг браузер зависнет в одном тесте, то другие тесты не пострадают, если они запускаются каждый в собственном браузере.

Минусы запуска браузера на каждый тест: каждый запуск и закрытие браузера занимают время, поэтому тесты будут идти дольше. Возможно, вы захотите оптимизировать время прогона тестов, но лучше это делать с помощью других инструментов, которые мы разберём в дальнейшем.

Обычно такие фикстуры переезжают вместе с тестами, написанными с помощью unittest, и приходится их поддерживать, но сейчас все пишут более гибкие фикстуры **@pytest.fixture**, которые мы рассмотрим в следующем шаге.

## Фикстуры, возвращающие значение

Мы рассмотрели базовый подход к созданию фикстур, когда тестовые данные задаются и очищаются в setup и teardown методах. PyTest предлагает продвинутый подход к фикстурам, когда фикстуры можно задавать глобально, передавать их в тестовые методы как параметры, а также имеет набор встроенных фикстур. Это более гибкий и удобный способ работы со вспомогательными функциями, и сейчас вы сами увидите почему.

**Возвращаемое значение**

Фикстуры могут возвращать значение, которое затем можно использовать в тестах. Давайте перепишем наш предыдущий пример с использованием PyTest фикстур. Мы создадим фикстуру **browser**, которая будет создавать объект WebDriver. Этот объект мы сможем использовать в тестах для взаимодействия с браузером. Для этого мы напишем метод browser и укажем, что он является фикстурой с помощью декоратора **@pytest.fixture**. После этого мы можем вызывать фикстуру в тестах, передав ее как параметр. По умолчанию фикстура будет создаваться для каждого тестового метода, то есть для каждого теста запустится свой экземпляр браузера.

pytest -s -v test\_fixture2.py

**test\_fixture2.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture

def browser():

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

browser = webdriver.Chrome()

return browser

class TestMainPage1():

# вызываем фикстуру в тесте, передав ее как параметр

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

browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

## Финализаторы — закрываем браузер

Вероятно, вы заметили, что мы не использовали в этом примере команду **browser.quit()**. Это привело к тому, что несколько окон браузера оставались открыты после окончания тестов, а закрылись только после завершения всех тестов. Закрытие браузеров произошло благодаря встроенной фикстуре — сборщику мусора. Но если бы количество тестов насчитывало больше нескольких десятков, то открытые окна браузеров могли привести к тому, что оперативная память закончилась бы очень быстро. Поэтому надо явно закрывать браузеры после каждого теста. Для этого мы можем воспользоваться **финализаторами**. Один из вариантов финализатора — использование ключевого слова Python: **yield**. После завершения теста, который вызывал фикстуру, выполнение фикстуры продолжится со строки, следующей за строкой со словом **yield**:

test\_fixture3.py

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture

def browser():

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

browser = webdriver.Chrome()

yield browser

# этот код выполнится после завершения теста

print("\nquit browser..")

browser.quit()

class TestMainPage1():

# вызываем фикстуру в тесте, передав ее как параметр

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

browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

Есть альтернативный способ вызова teardown кода с помощью встроенной фикстуры **request** и ее метода **addfinalizer**. Можете изучить его сами по документации [PyTest](https://docs.pytest.org/en/latest/how-to/fixtures.html" \l "adding-finalizers-directly" \t "_blank).

Рекомендуем также выносить очистку данных и памяти в фикстуру, вместо того чтобы писать это в шагах теста: финализатор выполнится даже в ситуации, когда тест упал с ошибкой.

## Область видимости scope

Для фикстур можно задавать область покрытия фикстур. Допустимые значения: “**function**”, “**class**”, “**module**”, “**session**”. Соответственно, фикстура будет вызываться один раз для тестового метода, один раз для класса, один раз для модуля или один раз для всех тестов, запущенных в данной сессии.

Запустим все наши тесты из класса **TestMainPage1** в одном браузере для экономии времени, задав scope="class" в фикстуре browser:

test\_fixture5.py

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture(scope="class")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1():

# вызываем фикстуру в тесте, передав ее как параметр

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

print("start test1")

browser.get(link)

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

print("finish test1")

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

print("start test2")

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

print("finish test2")

Мы видим, что в данном примере браузер открылся один раз и тесты последовательно выполнились в этом браузере. Здесь мы проделали это в качестве примера, но мы крайне рекомендуем всё же запускать отдельный экземпляр браузера для каждого теста, чтобы повысить стабильность тестов. Фикстуры, которые занимают много времени для запуска и ресурсов (обычно это работа с базами данных), можно вызывать и один раз за сессию запуска тестов.

## Автоиспользование фикстур

При описании фикстуры можно указать дополнительный параметр **autouse=True,** который укажет, что фикстуру нужно запустить для каждого теста даже без явного вызова:

test\_fixture\_autouse.py

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

@pytest.fixture(autouse=True)

def prepare\_data():

print()

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

class TestMainPage1():

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

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

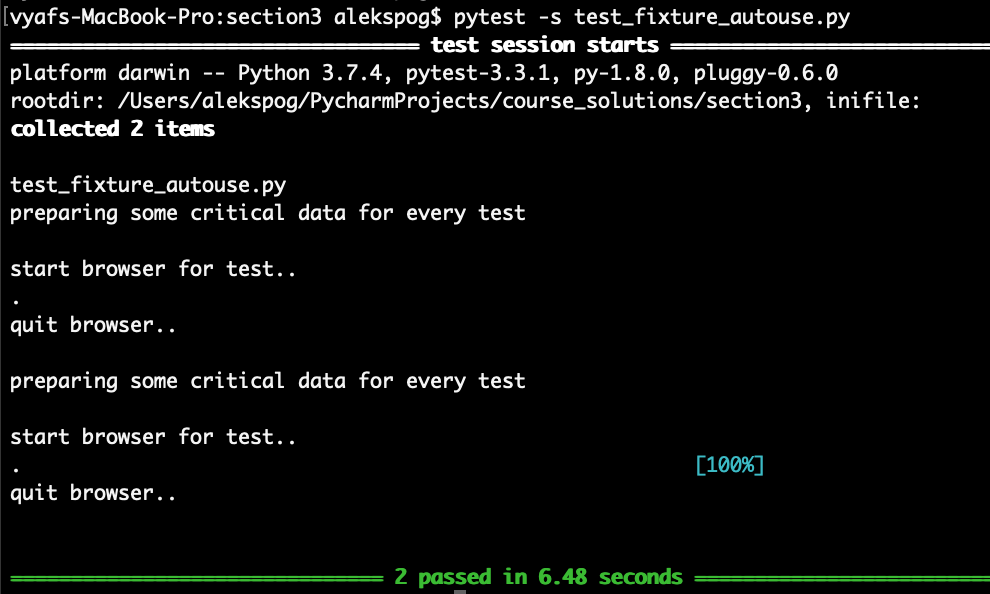
browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")



Попробуйте запустить этот код и увидите, что для каждого теста фикстура подготовки данных выполнилась без явного вызова. Нужно быть аккуратнее с этим параметром, потому что фикстура выполняется для всех тестов. Без явной необходимости автоиспользованием фикстур лучше не пользоваться.

**Итог**

Вспомогательные функции — это очень мощная штука, которая решает много проблем при работе с автотестами. Основной плюс в том, что их удобно использовать в любых тестах без дублирования лишнего кода.

Дополнительные материалы про фикстуры, которые мы настоятельно советуем почитать, приведены ниже:

<https://habr.com/ru/company/yandex/blog/242795/>

<https://docs.pytest.org/en/stable/fixture.html>

3.5 PyTest — маркировка

## Содержание урока

* [Маркировка тестов часть 1](https://stepik.org/lesson/236918/step/2)
* [Маркировка тестов часть 2](https://stepik.org/lesson/236918/step/3)
* [Пропуск тестов](https://stepik.org/lesson/236918/step/4)
* [XFail: помечать тест как ожидаемо падающий](https://stepik.org/lesson/236918/step/5)
* [Задание: пропуск тестов](https://stepik.org/lesson/236918/step/6)
* [Задание: запуск тестов](https://stepik.org/lesson/236918/step/7)

## Маркировка тестов часть 1

Когда тестов становится много, хорошо иметь способ разделять тесты не только по названиям, но также по каким-нибудь заданным нами категориям. Например, мы можем выбрать небольшое количество критичных тестов (smoke), которые нужно запускать на каждый коммит разработчиков, а остальные тесты обозначить как регрессионные (regression) и запускать их только перед релизом. Или у нас могут быть тесты, специфичные для конкретного браузера (internet explorer 11), и мы хотим запускать эти тесты только под данный браузер. Для выборочного запуска таких тестов в PyTest используется маркировка тестов или **метки (marks)**. Для маркировки теста нужно написать декоратор вида **@pytest.mark.mark\_name**, где mark\_name — произвольная строка.

Давайте разделим тесты в одном из предыдущих примеров на smoke и regression.

**test\_fixture8.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1():

@pytest.mark.smoke

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

browser.get(link)

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

@pytest.mark.regression

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

Чтобы запустить тест с нужной маркировкой, нужно передать в командной строке параметр **-m** и нужную метку:

pytest -s -v -m smoke test\_fixture8.py

Если всё сделано правильно, то должен запуститься только тест с маркировкой smoke.

При этом вы увидите warning, то есть предупреждение:

PytestUnknownMarkWarning: Unknown pytest.mark.smoke - is this a typo? You can register custom marks to avoid this warning - for details, see https://docs.pytest.org/en/latest/mark.html

PytestUnknownMarkWarning,

Это предупреждение появилось потому, что в последних версиях PyTest настоятельно рекомендуется регистрировать метки явно перед использованием. Это, например, позволяет избегать опечаток, когда вы можете ошибочно пометить ваш тест несуществующей меткой, и он будет пропускаться при прогоне тестов.

### Как же регистрировать метки?

Создайте файл pytest.ini в корневой директории вашего тестового проекта и добавьте в файл следующие строки:

[pytest]

markers =

smoke: marker for smoke tests

regression: marker for regression tests

Текст после знака ":" является поясняющим — его можно не писать.

Снова запустите тесты:

pytest -s -v -m smoke test\_fixture8.py

Теперь предупреждений быть не должно.

Так же можно маркировать целый тестовый класс. В этом случае маркировка будет применена ко всем тестовым методам, входящим в класс.

## Маркировка тестов часть 2

### **Инверсия**

Чтобы запустить все тесты, не имеющие заданную маркировку, можно использовать инверсию. Для запуска всех тестов, не отмеченных как smoke, нужно выполнить команду:

pytest -s -v -m "not smoke" test\_fixture8.py

### **Объединение тестов с разными маркировками**

Для запуска тестов с разными метками можно использовать логическое ИЛИ. Запустим smoke и regression-тесты:

pytest -s -v -m "smoke or regression" test\_fixture8.py

### **Выбор тестов, имеющих несколько маркировок**

Предположим, у нас есть smoke-тесты, которые нужно запускать только для определенной операционной системы, например, для Windows 10. Зарегистрируем метку win10 в файле pytest.ini, а также добавим к одному из тестов эту метку.

**pytest.ini:**

[pytest]

markers =

smoke: marker for smoke tests

regression: marker for regression tests

win10

**test\_fixture81.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1:

@pytest.mark.smoke

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

browser.get(link)

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

@pytest.mark.smoke

@pytest.mark.win10

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

Чтобы запустить только smoke-тесты для Windows 10, нужно использовать логическое И:

pytest -s -v -m "smoke and win10" test\_fixture81.py

Должен выполниться тест test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page.

## ****Пропуск тестов****

В PyTest есть стандартные метки, которые позволяют пропустить тест при сборе тестов для запуска (то есть не запускать тест) или запустить, но отметить особенным статусом тот тест, который ожидаемо упадёт из-за наличия бага, чтобы он не влиял на результаты прогона всех тестов. Эти метки не требуют дополнительного объявления в pytest.ini.

**Пропустить тест**

Итак, чтобы пропустить тест, его отмечают в коде как **@pytest.mark.skip**:

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1():

@pytest.mark.skip

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

browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

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

## XFail: помечать тест как ожидаемо падающий

**Отметить тест как падающий**

Теперь добавим в наш тестовый класс тест, который проверяет наличие кнопки "Избранное":

def test\_guest\_should\_see\_search\_button\_on\_the\_main\_page(self, browser):

browser.get(link)

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

Предположим, что такая кнопка должна быть, но из-за изменений в коде она пропала. Пока разработчики исправляют баг, мы хотим, чтобы результат прогона ﻿всех ﻿наших тестов был успешен, но падающий тест помечался соответствующим образом, чтобы про него не забыть. Добавим маркировку **@pytest.mark.xfail**для падающего теста.

**test\_fixture10.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1():

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

browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

@pytest.mark.xfail

def test\_guest\_should\_see\_search\_button\_on\_the\_main\_page(self, browser):

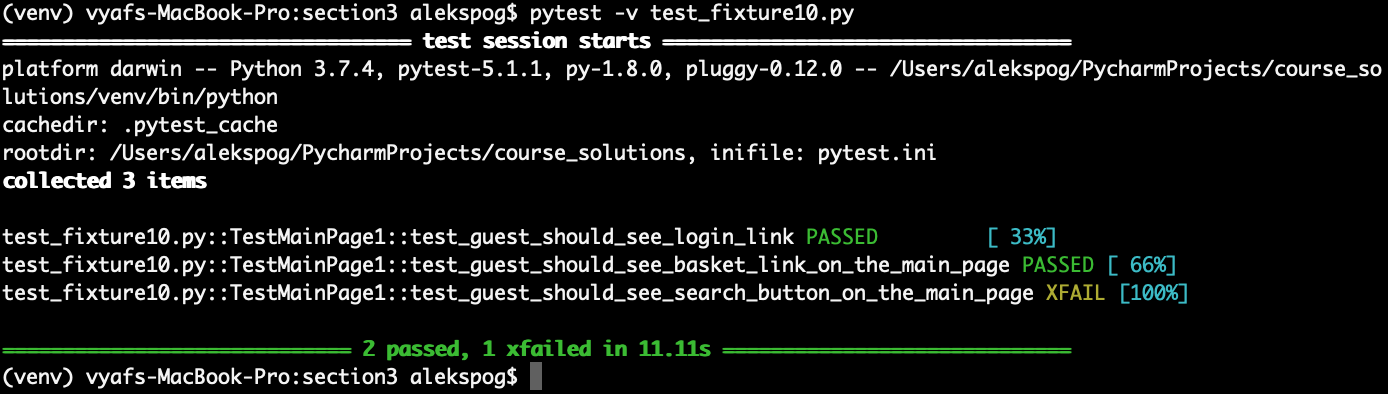
browser.get(link)

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

Запустим наши тесты:

pytest -v test\_fixture10.py

Наш упавший тест теперь отмечен как **xfail**, но результат прогона тестов помечен как успешный:



Когда баг починят, мы это узнаем, ﻿﻿так как теперь тест будет отмечен как **XPASS**(“unexpectedly passing” — неожиданно проходит). После этого маркировку **xfail**для теста можно удалить. Кстати, к маркировке **xfail** можно добавлять параметр **reason**. Чтобы увидеть это сообщение в консоли, при запуске нужно добавлять параметр pytest **-rx**.

**test\_fixture10a.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1():

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

browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

@pytest.mark.xfail(reason="fixing this bug right now")

def test\_guest\_should\_see\_search\_button\_on\_the\_main\_page(self, browser):

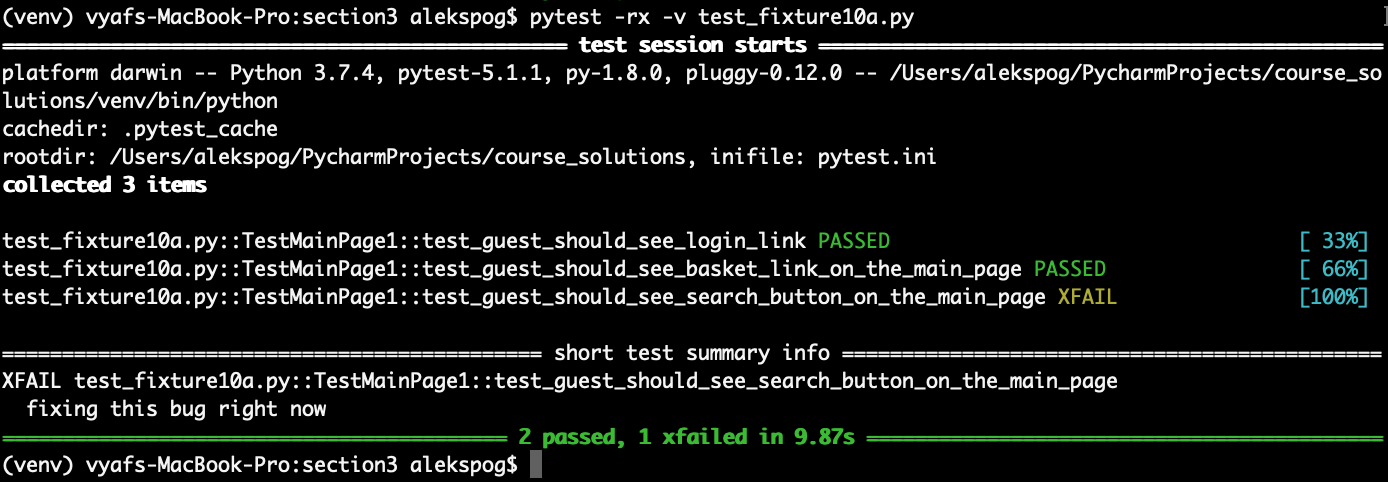
browser.get(link)

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

Запустим наши тесты:

pytest -rx -v test\_fixture10a.py

Сравните вывод в первом и во втором случае.



**XPASS-тесты**

Поменяем селектор в последнем тесте, чтобы тест начал проходить.

**test\_fixture10b.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

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

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1():

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

browser.get(link)

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

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, ".basket-mini .btn-group > a")

@pytest.mark.xfail(reason="fixing this bug right now")

def test\_guest\_should\_see\_search\_button\_on\_the\_main\_page(self, browser):

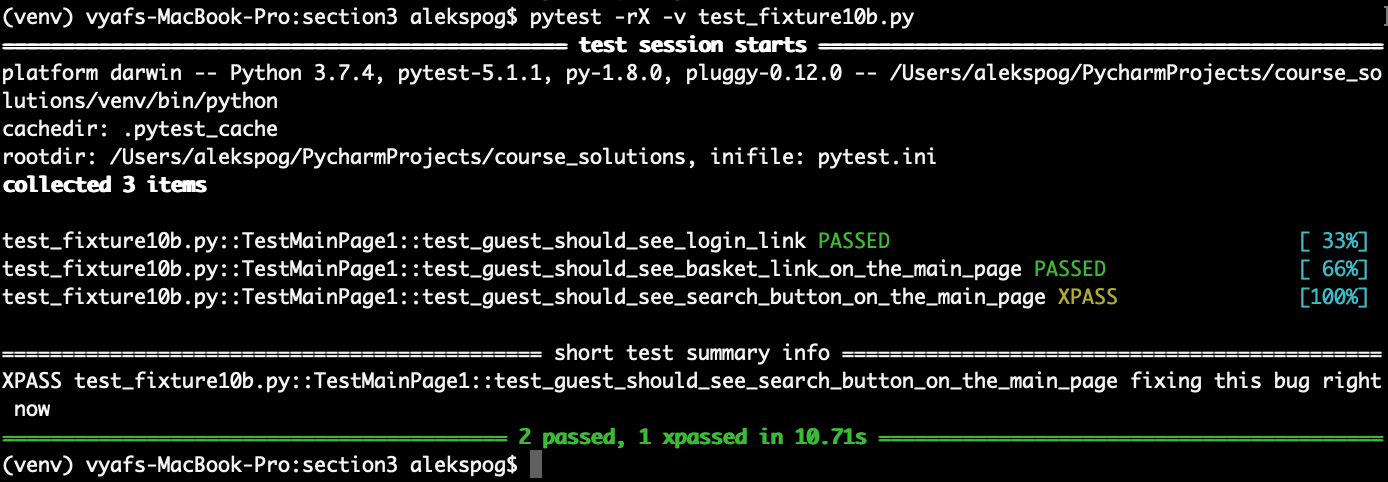
browser.get(link)

browser.find\_element(By.CSS\_SELECTOR, "input.btn.btn-default")

Запустите тесты. Здесь мы добавили символ X в параметр -r, чтобы получить подробную информацию по XPASS-тестам:

pytest -rX -v test\_fixture10b.py

И изучите отчёт:



Дополнительно об использовании этих меток можно почитать в документации: [Skip and xfail: dealing with tests that cannot succeed](https://pytest.org/en/stable/skipping.html" \t "_blank).  Там есть много разных интересных особенностей, например, как пропускать тест только при выполнении условия, как сделать так, чтобы внезапно прошедший xfailed тест в отчете стал красным, и так далее.

## Задание: пропуск тестов

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

**test\_xfail.py:**

import pytest

def test\_succeed():

assert True

@pytest.mark.xfail

def test\_not\_succeed():

assert False

@pytest.mark.skip

def test\_skipped():

assert False

Запустите полученные тесты. Обратите внимание на статус прогона тестов. Найдите последнюю строчку с итогами запуска, скопируйте текст между символами === и отправьте его в качестве ответа на это задание.

3.6 PyTest — параметризация, конфигурирование, плагины

## Содержание урока

* [Conftest.py — конфигурация тестов](https://stepik.org/lesson/237240/step/2)
* [Параметризация тестов](https://stepik.org/lesson/237240/step/3)
* [Задание: авторизация на сайте](https://stepik.org/lesson/237240/step/4?unit=209628)
* [Задание: параметризация тестов](https://stepik.org/lesson/237240/step/4)
* [Установка Firefox и Selenium-драйвера geckodriver](https://stepik.org/lesson/237240/step/5)
* [Conftest.py и передача параметров в командной строке](https://stepik.org/lesson/237240/step/6)
* [Плагины и перезапуск тестов](https://stepik.org/lesson/237240/step/7)
* [Запуск автотестов для разных языков интерфейса](https://stepik.org/lesson/237240/step/8)
* [Задание: запуск автотестов с указанием языка интерфейса](https://stepik.org/lesson/237240/step/9)

## Conftest.py — конфигурация тестов

Ранее мы добавили фикстуру browser, которая создает нам экземпляр браузера для тестов в данном файле. Когда файлов с тестами становится больше одного, приходится в каждом файле с тестами описывать данную фикстуру. Это очень неудобно. Для хранения часто употребимых фикстур и хранения глобальных настроек нужно использовать файл**conftest.py,** который должен лежать в директории верхнего уровня в вашем проекте с тестами. Можно создавать дополнительные файлы conftest.py в других директориях, но тогда настройки в этих файлах будут применяться только к тестам в под-директориях.

Создадим файл **conftest.py** в корневом каталоге нашего тестового проекта и перенесем туда фикстуру **browser**. Заметьте, насколько лаконичнее стал выглядеть файл с тестами.

**conftest.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

Теперь, сколько бы файлов с тестами мы ни создали, у тестов будет доступ к фикстуре browser. Фикстура передается в тестовый метод в качестве аргумента. Таким образом можно удобно переиспользовать одни и те же вспомогательные функции в разных частях проекта.

**test\_conftest.py:**

from selenium.webdriver.common.by import By

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

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

browser.get(link)

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

### **ОЧЕНЬ ВАЖНО!**

Есть одна важная особенность поведения конфигурационных файлов, о которой вы обязательно должны знать. PyTest автоматически находит и подгружает файлы conftest.py, которые находятся в директории с тестами. Если вы храните все свои скрипты для курса в одной директории, будьте аккуратны и следите, чтобы не возникало ситуации, когда вы запускаете тесты из папки tests:

tests/

├── conftest.py

├── subfolder

│ └── conftest.py

│ └── test\_abs.py

следует избегать!

В таком случае применяются ОБА файла conftest.py, что может вести к непредсказуемым ошибкам и конфликтам.

Таким образом можно переопределять разные фикстуры, но мы в рамках курса рекомендуем придерживаться одного файла на проект/задачу и держать их горизонтально, как-нибудь так:

selenium\_course\_solutions/

├── section3

│ └── conftest.py

│ └── test\_languages.py

├── section4

│ └── conftest.py

│ └── test\_main\_page.py

правильно!

Будьте внимательны и следите, чтобы не было разных conftest во вложенных друг в друга директориях, особенно, когда будете скачивать и проверять задания сокурсников.

[Override a fixture on a folder (conftest) level](https://docs.pytest.org/en/7.1.x/how-to/fixtures.html?highlight=fixture%20folder#override-a-fixture-on-a-folder-conftest-level)

## Параметризация тестов

PyTest позволяет запустить один и тот же тест с разными входными параметрами. Для этого используется декоратор **@pytest.mark.parametrize()**. Наш сайт доступен для разных языков. Напишем тест, который проверит, что для сайта с русским и английским языком будет отображаться ссылка на форму логина. Передадим в наш тест ссылки на русскую и английскую версию главной страницы сайта.

В **@pytest.mark.parametrize()** нужно передать параметр, который должен изменяться, и список значений параметра. В самом тесте наш параметр тоже нужно передавать в качестве аргумента. Обратите внимание, что внутри декоратора имя параметра оборачивается в кавычки, а в списке аргументов теста кавычки не нужны.

**test\_fixture7.py:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

@pytest.fixture(scope="function")

def browser():

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

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

@pytest.mark.parametrize('language', ["ru", "en-gb"])

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

link = f"http://selenium1py.pythonanywhere.com/{language}/"

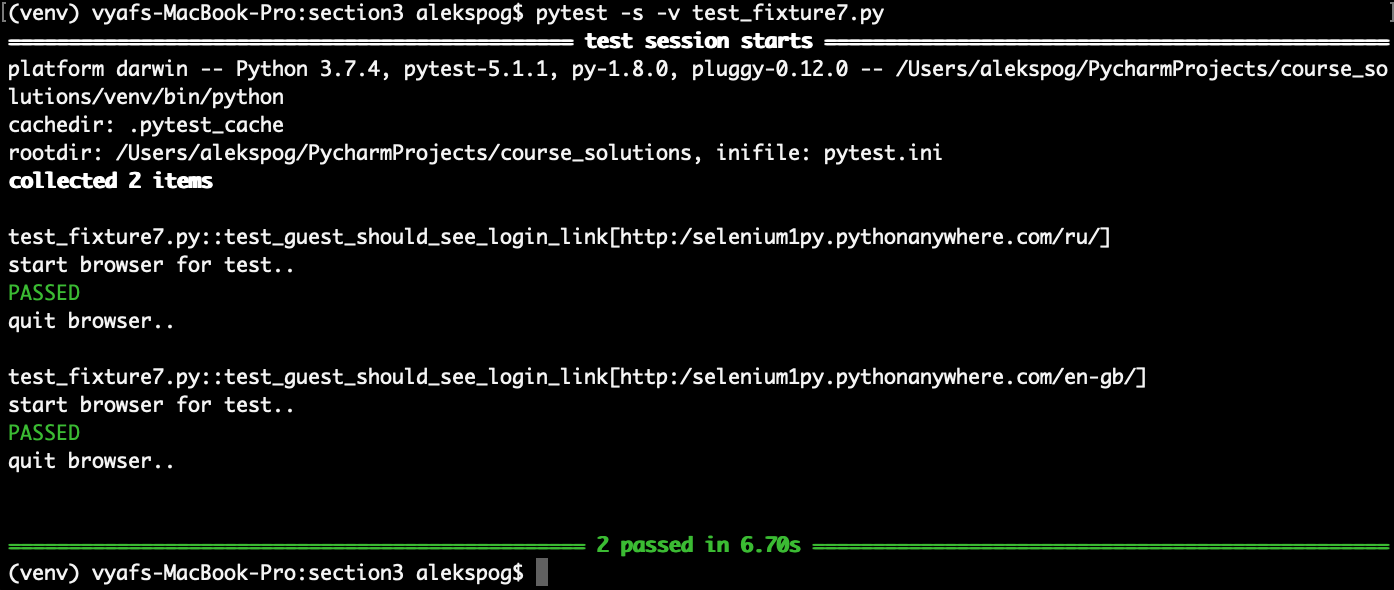
browser.get(link)

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

Запустите тест:

pytest -s -v test\_fixture7.py

 Вы увидите, что запустятся два теста.  В названии каждого теста в квадратных скобках будет написан параметр, с которым он был запущен. Таким образом мы можем быстро и без дублирования кода увеличить количество проверок для похожих сценариев.



Можно задавать параметризацию также для всего тестового класса, чтобы все тесты в классе запустились с заданными параметрами. В таком случае отметка о параметризации должна быть перед объявлением класса:

@pytest.mark.parametrize('language', ["ru", "en-gb"])

class TestLogin:

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

link = f"http://selenium1py.pythonanywhere.com/{language}/"

browser.get(link)

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

# этот тест запустится 2 раза

def test\_guest\_should\_see\_navbar\_element(self, browser, language):

# этот тест тоже запустится дважды

Дополнительно, полезный туториал из документации: [Parametrizing fixtures and test functions](https://docs.pytest.org/en/latest/how-to/parametrize.html?highlight=parametrize" \t "_blank)

## Установка Firefox и Selenium-драйвера geckodriver

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

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

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

Для установки Firefox скачайте его с официального сайта и установите в вашей ОС: <https://www.mozilla.org/firefox/new/>.

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

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

from selenium import webdriver

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

driver = webdriver.Firefox()

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

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

Если при попытке выполнения кода вы увидели подобное сообщение:

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

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

## Conftest.py и передача параметров в командной строке

Встроенная фикстура **request** может получать данные о текущем запущенном тесте, что позволяет, например, сохранять дополнительные данные в отчёт, а также делать многие другие интересные вещи. В этом шаге мы хотим показать, как можно настраивать тестовые окружения с помощью передачи параметров через командную строку.

Это делается с помощью встроенной функции pytest\_addoption и фикстуры request. Сначала добавляем в файле conftest обработчик опции в функции pytest\_addoption, затем напишем фикстуру, которая будет обрабатывать переданные в опции данные. Подробнее можно ознакомиться здесь: <https://docs.pytest.org/en/latest/example/simple.html?highlight=addoption>

Добавим логику обработки командной строки в conftest.py. Для запроса значения параметра мы можем вызвать команду:

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

**conftest.py:**

import pytest

from selenium import webdriver

def pytest\_addoption(parser):

parser.addoption('--browser\_name', action='store', default=None,

help="Choose browser: chrome or firefox")

@pytest.fixture(scope="function")

def browser(request):

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

browser = None

if browser\_name == "chrome":

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

browser = webdriver.Chrome()

elif browser\_name == "firefox":

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

browser = webdriver.Firefox()

else:

raise pytest.UsageError("--browser\_name should be chrome or firefox")

yield browser

print("\nquit browser..")

browser.quit()

**test\_parser.py:**

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

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

browser.get(link)

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

Если вы теперь запустите тесты без параметра, то получите ошибку:

pytest -s -v test\_parser.py

\_pytest.config.UsageError: --browser\_name should be chrome or firefox

Можно задать значение параметра по умолчанию, чтобы в командной строке не обязательно было указывать параметр --browser\_name, например, так:

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

help="Choose browser: chrome or firefox")

Давайте укажем параметр:

pytest -s -v --browser\_name=chrome test\_parser.py

А теперь запустим тесты на Firefox:

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

Вы должны увидеть, как сначала тесты запустятся в браузере Chrome, а затем — в Firefox.

## Плагины и перезапуск тестов

Для PyTest написано большое количество [плагинов](https://docs.pytest.org/en/latest/explanation/flaky.html?highlight=plugins#plugins), то есть ﻿дополнительных модулей, которые расширяют возможности этого фреймворка. Полный список доступных плагинов доступен [здесь](https://docs.pytest.org/en/latest/reference/plugin_list.html).

Рассмотрим еще одну проблему, с которой вы обязательно столкнетесь, когда будете писать end-to-end тесты на Selenium. Flaky-тесты или "мигающие" авто-тесты, т.е. такие тесты, которые по независящим от нас внешним обстоятельствам или из-за трудновоспроизводимых багов, могут иногда падать, хотя всё остальное время они проходят успешно. Это может происходить в момент прохождения тестов из-за одновременного обновления сайта, из-за сетевых проблем или странных стечений обстоятельств. Конечно, надо стараться исключать такие проблемы и искать причины возникновения багов, но в реальном мире бывает, что это требует слишком много усилий. Поэтому мы будем перезапускать упавший тест, чтобы еще раз убедиться, что он действительно нашел баг, а не упал случайно.

Это сделать очень просто. Для этого мы будем использовать плагин **pytest-rerunfailures**.

Сначала установим плагин в нашем виртуальном окружении. После установки плагин будет автоматически найден PyTest, и можно будет пользоваться его функциональностью без дополнительных изменений кода:

pip install pytest-rerunfailures

Чтобы указать количество перезапусков для каждого из упавших тестов, нужно добавить в командную строку параметр:

**"--reruns n"**, где n — это количество перезапусков. Если при повторных запусках тесты пройдут успешно, то и прогон тестов будет считаться успешным. Количество перезапусков отображается в отчёте, благодаря чему можно позже анализировать проблемные тесты.﻿﻿  
Дополнительно мы указали параметр **"--tb=line"**, чтобы сократить лог с результатами теста. Можете почитать подробнее про настройку вывода в [документации PyTest](https://docs.pytest.org/en/stable/usage.html#modifying-python-traceback-printing):

pytest -v --tb=line --reruns 1 --browser\_name=chrome test\_rerun.py

Давайте напишем два теста: один из них будет проходить, а другой — нет. Посмотрим, как выглядит перезапуск.

**test\_rerun.py:**

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

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

browser.get(link)

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

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

browser.get(link)

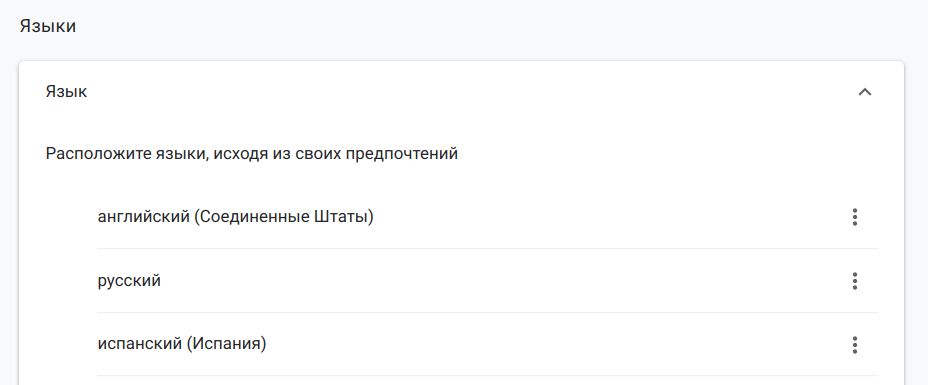
browser.find\_element(By.CSS\_SELECTOR, "#magic\_link")

Мы увидим сообщение: "1 failed, 1 passed, 1 rerun in 9.20s﻿", то есть упавший тест был перезапущен, но при втором запуске тоже упал. Если бы во второй раз мигающий тест все-таки прошёл успешно, то мы бы увидели сообщение: ﻿﻿"2 passed, 1 rerun in 9.20s"﻿, и итоговый результат запуска всех тестов считался бы успешным.

## Запуск автотестов для разных языков интерфейса

Цель: научиться запускать автотесты для разных локалей, т.е. для разных языков интерфейсов.

Мы уже запускали автотесты для разных языков в одном из предыдущих [шагов](https://stepik.org/lesson/237240/step/2), используя параметризацию с помощью разных ссылок, но такой подход сложно масштабировать на большое количество тестов. Давайте сделаем так, чтобы сервер сам решал, какой язык интерфейса нужно отобразить, основываясь на данных браузера. Браузер передает данные о языке пользователя через запросы к серверу, указывая в Headers (заголовке запроса) параметр **accept-language**. Если сервер получит запрос с заголовком {accept-language: ru, en}, то он отобразит пользователю русскоязычный интерфейс сайта. Если русский язык не поддерживается, то будет показан следующий язык из списка, в данном случае пользователь увидит англоязычный интерфейс. Это, кстати, примерно то же самое, что и выставить предпочтительный язык в настройках своего браузера:



Чтобы указать язык браузера с помощью WebDriver, используйте класс Options и метод **add\_experimental\_option**, как указано в примере ниже:

from selenium.webdriver.chrome.options import Options

options = Options()

options.add\_experimental\_option('prefs', {'intl.accept\_languages': user\_language})

browser = webdriver.Chrome(options=options)

Для Firefox объявление нужного языка будет выглядеть немного иначе:

fp = webdriver.FirefoxProfile()

fp.set\_preference("intl.accept\_languages", user\_language)

browser = webdriver.Firefox(firefox\_profile=fp)

В конструктор webdriver.Chrome или webdriver.Firefox вы можете добавлять разные аргументы, расширяя возможности тестирования ваших веб-приложений: можно указывать прокси-сервер для контроля сетевого трафика или запускать разные версии браузера, указывая локальный путь к файлу браузера. Предполагаем, что эти возможности вам понадобятся позже и вы сами сможете найти настройки для этих задач.

## Полезные ссылки

### Git

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

### Тестирование веб-приложений

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

### Тестирование с помощью PyTest

* [Статья про PyTest](https://habr.com/ru/post/269759/)
* [Введение в PyTest](https://coderlessons.com/tutorials/python-technologies/uznaite-pytest/pytest-kratkoe-rukovodstvo)
* [Документация PyTest](https://docs.pytest.org/en/latest/)
* [Conventions for Python test discovery](https://docs.pytest.org/en/stable/goodpractices.html)
* [Полезные флаги pytest](https://gist.github.com/amatellanes/12136508b816469678c2)

### Использование фикстур в PyTest

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

### Параметризация, конфигурирование, плагины

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

4.1 Что такое Page Object?

## Содержание урока:

* [Задание: ООП](https://stepik.org/lesson/199980/step/2)
* [Немного о Code Style](https://stepik.org/lesson/199980/step/3)
* [Code Style: базовые принципы](https://stepik.org/lesson/199980/step/4)
* [Code Style в автотестах](https://stepik.org/lesson/199980/step/5)
* [Подготовка окружения](https://stepik.org/lesson/199980/step/6)
* [Что такое Page Object?](https://stepik.org/lesson/199980/step/7)
* [Почему именно Page Object?](https://stepik.org/lesson/199980/step/8)
* [Задание: методы в Page Object](https://stepik.org/lesson/199980/step/9)

### Внимание!

В этом модуле мы будем использовать подходы объектно-ориентированного программирования, так что если вы плохо знакомы с принципами ООП, то просим вас разобраться самостоятельно:

<https://stepik.org/lesson/24461/step/1?unit=6767>

<https://tproger.ru/translations/oop-principles-cheatsheet/>

Page Objects Model:

[https://page-objects.readthedocs.io](https://page-objects.readthedocs.io/)

<https://selenium-python.readthedocs.io/page-objects.html>

## Немного о Code Style

Среди тех, кто более или менее регулярно пишет код, существует определенное соглашение о "стиле кода". Стиль кода — это всё то, что не относится к его функциональности: форматирование, имена переменных, функций, констант и так далее. Python прекрасен тем, что его очень легко читать, но даже такой простой для понимания язык в своём коде можно превратить в нечитаемую кашу. Нечитаемая каша опасна тем, что вы не разберетесь в своем коде уже через пару недель, а другой человек не разберется никогда. Хорошо написанный код экономит время при починке тестов, при внедрении нового человека в команду, да и при написании нового кода тоже. В общем, это очень важная тема, и следует всегда помнить о читабельности кода.

Мы совсем немного затронули эту тему в предыдущих модулях, а теперь, раз уж мы потихоньку идём в сторону большей абстракции, настало время поговорить об этом чуть более подробно.

### Отступы

Отступы являются частью синтаксиса в Python и означают вложенность блока, будь то тело функции условного выражения, цикла, и так далее. Самое важное для нас в будущих шагах, что все функции внутри класса так же должны быть отделены отступом:

@pytest.mark.regression

# тест вне класса: отступа нет

def test\_student\_can\_see\_lesson\_name\_in\_lesson\_in\_course\_after\_joining(self, driver):

# все строки внутри теста с отступом

page = CoursePromoPage(url=self.course.url, driver=driver)

page.open()

class TestLessonNameInCourseForTeacher():

@pytest.mark.regression

# тест внутри класса, нужен дополнительный отступ

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

# еще один отступ для каждой строки, и так с любым уровнем вложенности

page = LessonPlayerPage(url=self.lesson\_url, driver=driver)

page.open()

try:

# плюс один отступ на каждый уровень вложенности

dangerous\_function()

except:

close\_something()

В некоторых теоретических шагах мы будем давать отдельно функции вне контекста классов, и вам придется расставлять отступы самостоятельно. Будьте готовы и не пугайтесь.

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

## Code Style: базовые принципы

### Имена переменных и функций

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

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

Функции пишутся через\_нижнее\_подчеркивание:

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

Классы пишут с помощью CamelCase:

class TestLessonNameWithoutCourseForGuest():

Константы пишут в стиле UPPERCASE:

MAIN\_PAGE = "/catalog"

### Максимальная простота кода

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

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

## Code Style в автотестах

Здесь мы попытались собрать важные принципы написания автотестов:

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

def should\_be\_reply\_comment()

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

Если у вас нет большого опыта в написании кода, в статьях по ссылкам вы можете найти дополнительные рекомендации по оформлению кода.

Английский язык:

<https://docs.python-guide.org/writing/style/>

<https://www.python.org/dev/peps/pep-0008/>

Русский язык:

<https://habr.com/ru/post/266969/>

<https://habr.com/ru/post/206868/>

<https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html>

## Подготовка окружения

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

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

Итак:

1. Создайте отдельный **публичный** репозиторий с осмысленным названием на GitHub.
2. Склонируйте его к себе на локальную машину.
3. Добавьте туда файл conftest.py из предыдущего модуля. Убедитесь дополнительно, что там есть параметр для задания языка интерфейса, по умолчанию равный "**en**".
4. Убедитесь что ни во вложенных папках, ни во внешних папках нет других файлов conftest.py, почему это важно смотри здесь: [Conftest.py — конфигурация тестов](https://stepik.org/lesson/237240/step/2).
5. Добавьте в репозиторий файл requirements.txt из предыдущего модуля.
6. Создайте пустой файл \_\_init\_\_.py, чтобы работали относительные импорты.
7. Создайте файл test\_main\_page.py и добавьте в него тест из предыдущего модуля:
8. def test\_guest\_can\_go\_to\_login\_page(browser):
9. link = "http://selenium1py.pythonanywhere.com/"
10. browser.get(link)
11. login\_link = browser.find\_element(By.CSS\_SELECTOR, "#login\_link")

login\_link.click()

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

## Что такое Page Object Model?

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

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

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

Рассмотрим такой простой тест-кейс:

1. Открыть главную страницу
2. Перейти на страницу логина

Ожидаемый результат:

Открыта страница логина

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

test\_main\_page.py:

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

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

browser.get(link)

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

login\_link.click()

Что здесь происходит?

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

Что мы на самом деле имеем в виду?

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

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

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

login\_link.click()

и наш тест упрощается:

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

browser.get(link)

go\_to\_login\_page(browser)

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

Круто? А мы еще даже не написали Page Object. Дальше будет круче.

Дополнительно про Page Object вы можете почитать здесь:

<https://github.com/SeleniumHQ/selenium/wiki/PageObjects>

<https://martinfowler.com/bliki/PageObject.html>

<https://medium.com/tech-tajawal/page-object-model-pom-design-pattern-f9588630800b>

## Почему именно Page Object?

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

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

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

Рассмотрим страницу товара в интернет магазине <http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209/>.

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

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

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

Тесты будут выглядеть примерно так:

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

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

page.open() # открываем страницу в браузере

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

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

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

4.2 Пишем первые тесты с помощью Page Object

## Содержание урока

* [Базовая страница для проекта: BasePage](https://stepik.org/lesson/238819/step/2)
* [Page Object для главной страницы сайта](https://stepik.org/lesson/238819/step/3)
* [Первый тест на основе Page Object](https://stepik.org/lesson/238819/step/4)
* [Методы-проверки в Page Object](https://stepik.org/lesson/238819/step/5)
* [Проверка элемента на странице](https://stepik.org/lesson/238819/step/6)
* [Элементы страниц в паттерне Page Object](https://stepik.org/lesson/238819/step/7)
* [Реализация LoginPage](https://stepik.org/lesson/238819/step/8)
* [Переходы между страницами](https://stepik.org/lesson/238819/step/9)
* [Удобство поддержки тестов — инкапсуляция бизнес-логики в методах](https://stepik.org/lesson/238819/step/10)
* [Задание: Page Object](https://stepik.org/lesson/238819/step/11)
* [Задание: Page Object](https://stepik.org/lesson/238819/step/12)

## Базовая страница для проекта: BasePage

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

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

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

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

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

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

В Python такие вещи делаются с помощью следующей конструкции:

class BasePage():

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

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

self.browser = browser

self.url = url

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

Объявите ниже в том же классе:

def open(self):

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

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

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

class BasePage():

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

self.browser = browser

self.url = url

def open(self):

self.browser.get(self.url)

## Page Object для главной страницы сайта

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

1. Откройте файл main\_page.py

2. В нем нужно сделать импорт базового класса BasePage:

from .base\_page import BasePage

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

class MainPage(BasePage):

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

4. Перенесите метод из предыдущего урока в класс MainPage:

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

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

login\_link.click()

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

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

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

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

Итого, файл main\_page.py будет выглядеть так:

from .base\_page import BasePage

from selenium.webdriver.common.by import By

class MainPage(BasePage):

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

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

login\_link.click()

## Первый тест на основе Page Object

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

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

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

from .pages.main\_page import MainPage

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

from .pages.main\_page import MainPage

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

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

page = MainPage(browser, link) # инициализируем Page Object, передаем в конструктор экземпляр драйвера и url адрес

page.open() # открываем страницу

page.go\_to\_login\_page() # выполняем метод страницы — переходим на страницу логина

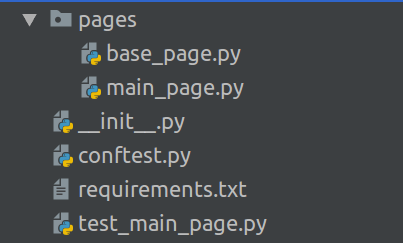
4. Убедитесь, что тест проходит, запустив его все той же командой:

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

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

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

Обратите внимание на  структуру файлов на данном этапе:



## Методы-проверки в Page Object

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

Допустим, нам нужно проверять такой сценарий:

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

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

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

Для первой пробы можно реализовать его самым примитивным образом:

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

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

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

Добавляем в файл с тест-кейсами новый тест:

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

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

page = MainPage(browser, link)

page.open()

page.should\_be\_login\_link()

Запустите получившийся тест:

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

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

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

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

## Проверка элемента на странице

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

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

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

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

self.browser = browser

self.url = url

self.browser.implicitly\_wait(timeout)

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

Чтобы перехватывать исключение, нужна конструкция try/except:

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

try:

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

except (имя исключения):

return False

return True

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

from selenium.common.exceptions import имя\_исключения

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

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

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

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

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

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

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

4. Сделайте коммит изменений (с осмысленным сообщением).

## Элементы страниц в паттерне Page Object

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

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

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

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

Запустите тесты командой:

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

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

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

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

Давайте этим и займемся:

1. В папке pages создайте новый файл locators.py

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

from selenium.webdriver.common.by import By

class MainPageLocators():

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

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

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

from .locators import MainPageLocators

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

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

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

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

5. Запустите тесты с помощью той же самой команды:

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

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

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

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

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

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

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

## Реализация LoginPage

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

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

should\_be\_login\_url

should\_be\_login\_form

should\_be\_register\_form

Реализуйте их самостоятельно:

1. В файле locators.py создайте класс LoginPageLocators

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

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

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

5. Добавьте изменения в коммит с осмысленным сообщением

Теперь посмотрим, как можно осуществлять переход между страницами.

**Переходы между страницами**

Переход можно реализовать двумя разными способами.

Первый способ: возвращать нужный Page Object.

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

from .login\_page import LoginPage

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

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

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

link.click()

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

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

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

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

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

page = MainPage(browser, link)

page.open()

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

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

Плюсы такого подхода:

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

минусы:

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

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

1. Закомментируйте строку с возвращаемым значением

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

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

link.click()

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

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

from .pages.login\_page import LoginPage

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

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

page = MainPage(browser, link)

page.open()

page.go\_to\_login\_page()

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

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

Плюсы:

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

Минусы:

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

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

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

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

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

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

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

link.click()

alert = self.browser.switch\_to.alert

alert.accept()

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

4.3 Улучшаем дизайн тестов

## Содержание урока

Вы уже знаете практически всю теорию, связанную с паттерном Page Object Model. Как вы могли заметить, особых сложностей там нет. В этом уроке мы сосредоточимся на практических задачах, которые вы будете выполнять самостоятельно, а попутно мы еще расскажем несколько трюков и хороших практик для написания автотестов.

Обратите внимание: некоторые из задач будут иметь проверки на Stepik, некоторые нужно будет выполнить полностью самостоятельно. Не пренебрегайте ими, в конце курса у вас должен получится полноценный тестовый проект, за который можно получить больше всего баллов в курсе.

Этот проект будет оцениваться другими учащимися. А еще этот проект можно приложить к своему первому резюме, если у вас не было опыта работы в роли автоматизатора тестов.

* [Задание: добавление в корзину со страницы товара](https://stepik.org/lesson/201964/step/2?unit=176022)
* [Задание: независимость от данных](https://stepik.org/lesson/201964/step/3)
* [Задание: независимость от контента, ищем баг](https://stepik.org/lesson/201964/step/4)
* [Отрицательные проверки: как проверить отсутствие элемента](https://stepik.org/lesson/201964/step/5)
* [Задание: отрицательные проверки](https://stepik.org/lesson/201964/step/6)
* [Плюсы наследования: магия ООП](https://stepik.org/lesson/201964/step/7)
* [Плюсы наследования: пример](https://stepik.org/lesson/201964/step/8)
* [Задание: наследование](https://stepik.org/lesson/201964/step/9)
* [Задание: наследование и отрицательные проверки](https://stepik.org/lesson/201964/step/10)
* [Группировка тестов: setup](https://stepik.org/lesson/201964/step/11)
* [Задание: setup](https://stepik.org/lesson/201964/step/12)
* [Задание: группировка тестов и setup](https://stepik.org/lesson/201964/step/13)
* [Финишная прямая: готовим код к ревью](https://stepik.org/lesson/201964/step/14)
* [Финальное задание: ревью](https://stepik.org/lesson/201964/step/15)

## Задание: добавление в корзину со страницы товара

Давайте, прежде чем двигаться дальше, закрепим знания на практике.

Представьте, что вы работаете тестировщиком-автоматизатором в IT-отделе интернет-магазина. QA Lead поручил вам задание автоматизировать следующий тестовый сценарий:

1. Открываем страницу товара (<http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209/?promo=newYear>). Обратите внимание, что в ссылке есть параметр "?promo=newYear". Не теряйте его в авто-тесте, чтобы получить проверочный код.
2. Нажимаем на кнопку "Добавить в корзину".
3. \*Посчитать результат математического выражения и ввести ответ. Используйте для этого метод **solve\_quiz\_and\_get\_code()**, который приведен ниже. Например, можете добавить его в класс **BasePage**, чтобы использовать его на любой странице. Этот метод нужен только для проверки того, что вы написали тест на Selenium. После этого вы получите код, который нужно ввести в качестве ответа на данное задание. Код будет выведен в консоли интерпретатора, в котором вы запускаете тест. Не забудьте в конце теста добавить проверки на ожидаемый результат.

Ожидаемый результат:

1. Сообщение о том, что товар добавлен в корзину. Название товара в сообщении должно совпадать с тем товаром, который вы действительно добавили.
2. Сообщение со стоимостью корзины. Стоимость корзины совпадает с ценой товара.

Тест нужно написать, используя паттерн Page Object. Для этого вам нужно:

1. Добавить новый файл для тест-кейсов, связанных со страницей товара. Назовите файл с тестами test\_product\_page.py.
2. Создать класс Page Object для страницы товара. Опишите его в файле product\_page.py в папке pages.
3. Описать в нем метод для добавления в корзину.
4. Дописать методы-проверки.
5. Описать необходимые локаторы к элементам страницы.
6. Написать сам тест-кейс, используя все вышеописанное. Назовите тест **test\_guest\_can\_add\_product\_to\_basket.**

Можете начинать работу с любого пункта, но хорошей практикой считается написать сначала шаги и структуру теста, а потом описывать конкретную реализацию.

\*Используйте этот метод в тесте для получения проверочного кода:

from selenium.common.exceptions import NoAlertPresentException # в начале файла

def solve\_quiz\_and\_get\_code(self):

alert = self.browser.switch\_to.alert

x = alert.text.split(" ")[2]

answer = str(math.log(abs((12 \* math.sin(float(x))))))

alert.send\_keys(answer)

alert.accept()

try:

alert = self.browser.switch\_to.alert

alert\_text = alert.text

print(f"Your code: {alert\_text}")

alert.accept()

except NoAlertPresentException:

print("No second alert presented")

Чтобы увидеть проверочный код в консоли, запускайте PyTest с параметром **-s**:

pytest -s test\_foo.py

## Задание: независимость от данных

Хорошие автотесты должны быть максимально независимы от данных. Худшее, что можно сделать в тесте это "захардкодить" проверки для объектов, которые существуют только на вашем конкретном инстансе. Почему? Потому что данные будут постоянно меняться, и при каждом таком изменении придется чинить автотесты. Еще это ухудшает переиспользование метода: допустим, мы хотим прогнать тест для множества товаров, тогда придется писать большое количество проверок: по одной для каждого товара. В конечном итоге, это сказывается на качестве продукта, так как такие тесты работают на узкой выборке страниц.

Общая рекомендация: ваши тесты не должны зависеть от того, что вы не можете контролировать. Это может быть информация, уже хранящаяся в базе данных, или сторонние сервисы, которые использует ваше приложение. Вы можете проверять конкретные данные только в случае, когда используете специально подготовленную тестовую базу, инициируемую перед каждым запуском тестов, или добавляете нужные данные в базу данных напрямую или через API приложения.

Попробуйте запустить автотест, который мы написали на предыдущем шаге, на странице <http://selenium1py.pythonanywhere.com/catalogue/coders-at-work_207/?promo=newYear2019>.

Если в предыдущем тесте после добавления товара в корзину вы проверяли в сообщении сайта фиксированную строку "The shellcoder's handbook", то тест упадет, так как теперь мы добавили другой товар. Если тест прошел, то вы молодец и можете просто вставить новый проверочный код в этом задании.

Чтобы тест был независимым от контента:

* Измените методы проверки таким образом, чтобы они принимали как аргумент название товара и цену товара.
* Сделайте метод, который вытаскивает из элемента текст-название товара и возвращает его.
* Сделайте такой же метод для цены.
* Теперь проверяйте, что название товара в сообщении совпадает с заголовком товара.

## Задание: независимость контента, ищем баг

Эта задача для настоящих ниндзя автотестинга. Не потому что она сложная, а потому что сейчас мы будем ловить с вами настоящий баг с помощью наших автотестов. Для нашего интернет-магазина было запущено несколько новых промо-акций, одна из которых привела к появлению бага. Промо-акция включается путем добавления параметра ?promo=offerN к ссылке на товар.

К счастью, нам не придется менять наш тест, чтобы проверить изменения в коде. Мы просто запустим всё тот же тест на странице <http://selenium1py.pythonanywhere.com/catalogue/coders-at-work_207/> с параметризацией. Вам нужно определить, при каком значении параметра promo автотест упадет. Для этого проверьте результат работы PyTest и найдите url, на котором произошла ошибка. Значение параметра может изменяться от offer0 до offer9.

Пример ссылки: <http://selenium1py.pythonanywhere.com/catalogue/coders-at-work_207/?promo=offer0>. Если баг будет найден на этой странице, то введите в качестве ответа <http://selenium1py.pythonanywhere.com/catalogue/coders-at-work_207/?promo=offer0>.

Запустить сразу несколько тестов вы можете, используя **@pytest.mark.parametrize**. Мы уже сделали для вас шаблон теста:

@pytest.mark.parametrize('link', ["http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer0",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer1",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer2",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer3",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer4",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer5",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer6",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer7",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer8",

"http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer9"])

def test\_guest\_can\_add\_product\_to\_basket(browser, link):

# ваша реализация теста

Подсказка: баг должен быть найден методом проверки.

После того как вы обнаружили баг, учитывая что чинить его не собираются, лучше всего пометить падающий тест как **xfail** или **skip.**Помните, как мы такое проворачивали в третьем модуле? Освежить память: [XFail: помечать тест как ожидаемо падающий](https://stepik.org/lesson/236918/step/5?unit=209305).

С параметризацией делается это примерно так:

@pytest.mark.parametrize('link', ["okay\_link",

pytest.param("bugged\_link", marks=pytest.mark.xfail),

"okay\_link"])

Подробнее: [Skip/xfail with parametrize](https://pytest.org/en/stable/skipping.html#skip-xfail-with-parametrize)

После всех манипуляций не забудьте зафиксировать изменения коммитом.

## Отрицательные проверки: как проверить отсутствие элемента

Иногда в ходе написания авто-тестов возникает ситуация, когда нам нужно проверить не только присутствие элемента на странице, но и то, что элемента на странице нет. Здесь стоит разделять две принципиально разные ситуации, в зависимости от того как ведет себя веб-приложение:

**1. Элемент потенциально может появиться на странице** (но вообще-то не должен). Например, мы открываем страницу товара, и ожидаем, что там нет сообщения об успешном добавлении в корзину. Мы проверяем, что элемента нет, но при позитивном сценарии, когда мы добавляем товар в корзину, сообщение тоже появляется не сразу. Если при негативной проверке мы не добавим ожидание, а сразу выдадим результат: "True, элемента действительно нет, все хорошо", мы рискуем нарваться на ложно-зеленый тест. То есть, можем пропустить баг.

**2. Элемент присутствует на странице и должен исчезнуть** со временем или в результате действий пользователя. Это может быть, например, удаление товара из корзины, или исчезновение лоадера с загрузкой.

### Почему нужно писать такие проверки с осторожностью?

Во-первых, нам приходится всегда гарантированно ждать. В первом примере нам всегда нужно ждать несколько секунд, чтобы убедиться, что элемент не появился. Если мы используем нашу написанную функцию is\_element\_present, то тест с такой проверкой будет ждать полные и честные 10 секунд:

def should\_not\_be\_success\_message(self):

assert not self.is\_element\_present(\*ProductPageLocators.SUCCESS\_MESSAGE),\

"Success message is presented"

Что очень много для зелёного теста. То есть implicit\_wait уже в такой ситуации не подходит, придется использовать явное ожидание и аккуратно подбирать условия. Время ожидания тоже придется подбирать эмпирически, путем проб, ошибок, ложноположительных и ложноотрицательных результатов.

Во-вторых, еще одна загвоздка с отрицательными проверками в том, что они могут давать ложноположительные срабатывания, если селектор устарел. Проверяем, что элемента с таким селектором нет, — проверка проходит, так как у элемента уже другой селектор. Элемент есть на экране — это баг, а тест зеленый. Это плохо!

Поэтому на каждый негативный тест обязательно должен приходиться положительный тест. В одном тесте проверяем, что элемента нет, в соседнем тесте, что элемент есть. Тогда мы сможем отслеживать актуальность селектора и не пропустим такой баг.

Дополнительная ссылка на пост Виталия Котова про распространенные грабли и отрицательные проверки, в том числе:

[UI-автотесты: как делать не стоит](https://habr.com/ru/company/badoo/blog/419419/)

### Как же тогда реализовывать такие проверки?

Нужно ориентироваться на конкретную ситуацию, но общий совет — использовать явные ожидания и [Expected Conditions](https://selenium-python.readthedocs.io/waits.html" \t "_blank), о которых мы говорили в предыдущих модулях.

Можно добавить в BasePage абстрактный метод, который проверяет, что элемент не появляется на странице в течение заданного времени:

def is\_not\_element\_present(self, how, what, timeout=4):

try:

WebDriverWait(self.browser, timeout).until(EC.presence\_of\_element\_located((how, what)))

except TimeoutException:

return True

return False

Тогда его использование Page Object для страницы товара будет выглядеть так:

def should\_not\_be\_success\_message(self):

assert self.is\_not\_element\_present(\*ProductPageLocators.SUCCESS\_MESSAGE), \

"Success message is presented, but should not be"

Если же мы хотим проверить, что какой-то элемент исчезает, то следует воспользоваться явным ожиданием вместе с функцией until\_not, в зависимости от того, какой результат мы ожидаем:

def is\_disappeared(self, how, what, timeout=4):

try:

WebDriverWait(self.browser, timeout, 1, TimeoutException).\

until\_not(EC.presence\_of\_element\_located((how, what)))

except TimeoutException:

return False

return True

Метод-проверка в классе про страницу товара будет выглядеть аналогично **should\_not\_be\_success\_message**, напишите его самостоятельно.

Обратите внимание на разницу между методами **is\_not\_element\_present** и **is\_disappeared**.

**is\_not\_element\_present**: упадет, как только увидит искомый элемент. Не появился: успех, тест зеленый.

**is\_disappeared**: будет ждать до тех пор, пока элемент не исчезнет.

Резюмируя, можно сказать, что разрабатывать такие проверки нужно очень аккуратно, использовать явные ожидания для сокращения времени прогона теста и всегда добавлять позитивную проверку на элемент в другом тесте. Без явной необходимости таких проверок лучше избегать.