

Selenium WebDriver



Введение

WebDriver — это фреймворк для веб-автоматизации, который позволяет выполнять тесты для разных браузеров.

WebDriver также позволяет использовать язык программирования при создании тестовых сценариев (это невозможно в Selenium IDE).

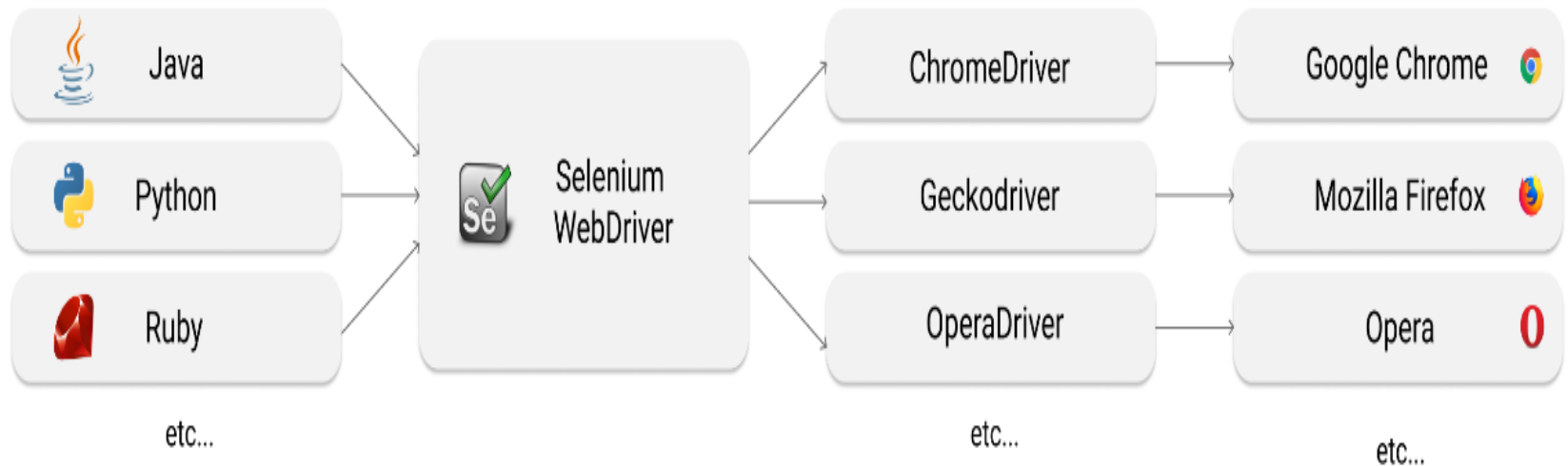
Введение

Selenium WebDriver — универсальный интерфейс, который позволяет манипулировать разными браузерами напрямую из кода на языке программирования.

Selenium WebDriver поддерживает довольно много языков программирования, хотя наибольшей популярностью в индустрии пользуются **Java** и **Python**.

Введение

клиентская библиотека
на конкретном языке
программирования



Что такое Selenium WebDriver

Selenium WebDriver, или просто **WebDriver** – это драйвер браузера, то есть не имеющая пользовательского интерфейса программная библиотека, которая позволяет различным другим программам взаимодействовать с браузером, управлять его поведением, получать от браузера какие-то данные и заставлять браузер выполнять какие-то команды.

WebDriver не имеет прямого отношения к тестированию. Он всего лишь предоставляет автотестам доступ к браузеру. На этом его функции заканчиваются.

Selenium WebDriver

Библиотеки **WebDriver** доступны на языках

- ✓ **Java,**
- ✓ **C#,**
- ✓ **Python,**
- ✓ **Ruby,**
- ✓ **JavaScript.**

Чаще всего **Selenium WebDriver** используется для тестирования функционала веб-сайтов/веб-ориентированных приложений. Автоматизированное тестирование удобно, потому что позволяет многократно запускать повторяющиеся тесты.

Регрессионное тестирование, то есть, проверка, что старый код не перестал работать правильно после внесения новых изменений, является типичным примером, когда необходима автоматизация.

Установка WebDriver + Python

Привязка **Selenium** к **Python** предоставляет собой простой **API** для написания тестов. С помощью **Selenium Python API** можно получить доступ ко всему функционалу Selenium WebDriver.

1. Сначала нужно установить **Python** на свой компьютер.

Ссылка: Python для Windows

**выбрать стабильную версию для Windows, и скачать файл для своей системы (64-разрядная или 32-разрядная)

Установка WebDriver + Python

2. Затем установить **Selenium WebDriver** с помощью **Pip**, официального менеджера пакетов **Python**.

Введите следующую команду для установки Selenium:

```
pip install selenium==3.14.0
```

Проверить, что библиотека действительно установлена:

```
pip list
```

```
(selenium_env) D:\environments>pip list
Package      Version
-----
pip          21.0.1
selenium     3.14.0
setuptools   49.2.1
urllib3      1.26.4
```


Установка WebDriver + Python

3. Загрузить драйвер, который интегрируется с выбранным браузером. Этот драйвер позволит **Selenium** управлять браузером и автоматизировать команды, которые будут использованы в скриптах.

В настоящее время **Selenium** поддерживает **Google Chrome, Firefox, Microsoft Edge** и **Safari**.

Официальный веб-драйвер для **Chrome** – это **ChromeDriver**, а **Geckodriver** – это официальный веб-драйвер для **Firefox**.

Скачать с сайта драйвер для поддерживаемой версии браузера. Разархивировать скачанный файл.

<https://sites.google.com/a/chromium.org/chromedriver/downloads>

Первое использование

В примере, который будем рассматривать, есть ряд основных шагов:

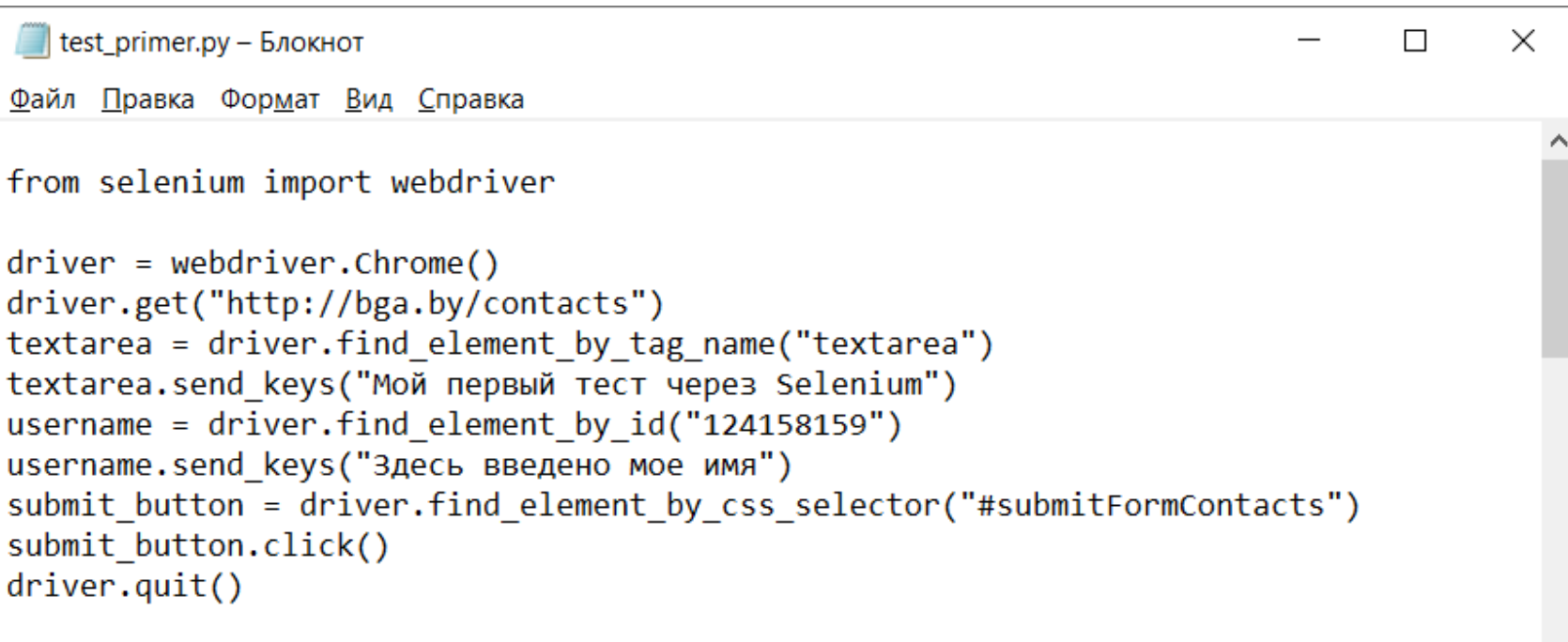
Шаг 1: Открываем браузер Chrome.

Шаг 2: Загружаем страницу

Шаг 3: В поле «Сообщение» вводим текст сообщения.

Шаг 4: В поле «Имя» вводим имя.

Шаг 5: Закрываем браузер.



```
test_primer.py – Блокнот
Файл  П_равка  Формат  В_ид  С_правка

from selenium import webdriver

driver = webdriver.Chrome()
driver.get("http://bga.by/contacts")
textarea = driver.find_element_by_tag_name("textarea")
textarea.send_keys("Мой первый тест через Selenium")
username = driver.find_element_by_id("124158159")
username.send_keys("Здесь введено мое имя")
submit_button = driver.find_element_by_css_selector("#submitFormContacts")
submit_button.click()
driver.quit()
```

Первое использование

```
from selenium import webdriver
```

импортируем пакет **webdriver** из **selenium**
webdriver это и есть набор команд для управления браузером

****В Python** ключевое слово **import** применяется для того, чтобы сделать код в одном **модуле** доступным для работы в другом.

Первое использование

```
driver = webdriver.Chrome()
```

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

Первое использование

```
driver.get("http://bga.by/contacts")
```

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

Первое использование

```
textarea =  
driver.find_element_by_tag_name("textarea")
```

Ищем поле для ввода текста

Метод **find_element_by_tag_name** позволяет найти нужный элемент на сайте по названию тега, указав путь к нему.

Первое использование

```
textarea.send_keys("Мой первый тест через Selenium")
```

Напишем текст в поле "Сообщение"

Первое использование

```
username =  
driver.find_element_by_id("l24l58l59")
```

Ищем поле для ввода текста

Метод **find_element_by_id** позволяет найти нужный элемент на сайте по его **ID**, указав путь к нему.

Первое использование

```
username.send_keys("Здесь введено мое  
имя")
```

Напишем текст в поле "*Имя*"

Первое использование

```
submit_button =  
driver.find_element_by_css_selector  
("#submitFormContacts")
```

Найдем кнопку, которая отправляет введенное решение

Первое использование

```
submit_button.click()
```

Скажем драйверу, что нужно нажать на кнопку.

Первое использование

```
driver.quit()
```

После выполнения всех действий не должны забыть закрыть окно браузера

Первое использование

КОНТАКТЫ

220012 г.Минск, ул.Сурганова, д.2, пом.33

Тел.: (017) 378-16-50

E-mail: info@bga.by

Пишите нам

Здесь введено мое имя

email*

Заполните поле!

Телефон

Мой первый тест через Selenium

*- Поля обязательные для заполнения

Отправить

Поиск элементов на странице

Любая веб-страница представляет собой html-файл, в котором с помощью языка разметки HTML описана её структура.

Умея описывать путь к элементу на странице, можно найти такой элемент и выполнить с ним необходимые действия, например, отправить текст в текстовое поле или нажать на необходимую кнопку.

Поиск элементов на странице

При работе с **Selenium** практически первое, с чем придется столкнуться - это локаторы.

Локатор - это строка, уникально идентифицирующая UI-элемент. Когда вы делаете клик мышкой, ввод текста и прочие действия, вы эти действия выполняете над вполне конкретным объектом. **Selenium** поступает так же. Но поскольку он не умеет читать ваши мысли, то ему надо четко указать объект, для которого надо применить то или иное действие.

Способы поиска элементов

- ✓ с помощью CSS-селекторов;
- ✓ с помощью языка запросов Xpath.

Поиск с помощью CSS-селекторов 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.

Поиск элементов с помощью CSS-селекторов

Элементы HTML-страницы, по которым можно найти элемент:

- ✓ id
- ✓ tag
- ✓ значение атрибута
- ✓ name
- ✓ class

Поиск элементов с помощью CSS-селекторов

id=<element_id> - соответствует элементу, у которого атрибут id равен значению element_id.

Например, у нас есть элемент, который в HTML записывается так:
`<input type=text id='some_input_id' name='some_input_name' value='' />`

В этом случае локатор будет иметь вид: `id=some_input_id`. Также следует отметить, что данный вид локаторов является одним из самых быстрых в нахождении и одним из самых уникальных. Это связано с тем, что в DOM-структуре ссылки на элементы, у которых задан ID, хранятся в отдельной таблице и через JScript (собственно именно через него осуществляется доступ к элементам на конечном уровне) обращение к элементам по ID идет достаточно короткой инструкцией, наподобие `some_input_id`.

Поиск элементов с помощью CSS-селекторов

name=<element_name> - соответствует элементу, у которого атрибут **name** равен значению **element_name**.

Эффективно применяется при работе с полями ввода формы (кнопки, текстовые поля, выпадающие списки). Как правило, значения элементов формы используются в запросах, которые идут на сервер и как раз атрибут **name** в этих запросах ставит в соответствие поле и его значение. Если брать предыдущий пример:

```
<input type=text id='some_input_id' name='some_input_name' value='' />
```

то данный элемент может быть также идентифицирован локатором вида **name=some_input_name**.

Данный тип локаторов тоже является достаточно быстрым в нахождении, но менее уникальным, так как на странице может быть несколько форм, у которых могут быть элементы с одинаковым именем.

Поиск элементов с помощью CSS-селекторов

`link=<link_text>` - специально для ссылок используется отдельно зарезервированный тип локаторов, который находит нужную ссылку по ее тексту. Это сделано отчасти потому, что ссылки как правило не имеют таких атрибутов как ID или name.

Соответственно, ссылка, которая в HTML записывается так:

```
<a href='http://some_url'>Link Text 2345</a>
```

в Селениуме идентифицируется локатором
`link=Link Text 2345`.

Поиск элементов с помощью XPath

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

xpath=<xpath_locator> - наиболее универсальный тип локаторов. Как XPath формируется: HTML, как и его более обобщенная форма - XML, представляет собой различное сочетание тегов, которые могут содержать вложенные теги, а те в свою очередь тоже могут содержать теги и т.д. То есть, выстраивается определенная иерархия, наподобие структуры каталогов в файловой системе. И задача XPath - отразить подобный путь к нужному элементу, с учетом иерархии.

Например, XPath вида:

A/B/C/D

указывает на некоторый элемент с тегом D, который находится внутри тега C, а тот в свою очередь - внутри тега B, который находится внутри тега A, который находится на самом верхнем уровне иерархии.

Поиск элементов с помощью XPath

Если брать использование XPath в Селениуме, то там зачастую полный путь указывать не нужно, более того, вредно, особенно, если вложенность тега нужного элемента достаточно высока. Как правило, удобно указывать путь, начиная с некоторого промежуточного элемента, пропуская теги более высокого порядка.

Например, такой XPath: `//table/tbody/tr/td/a` ссылается на первую ссылку в первой строке тела первой таблицы. Обратите внимание на начало данной записи.

Строка `'//'` означает, что поиск элемента начинается с некоторого произвольного места.

Поиск элементов с помощью XPath

Проверять XPath-запросы можно точно так же как и CSS-селекторы — в консоли разработчика.

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 начинают запрос с символа /, то должны указать элемент, являющийся корнем документа. Корнем **всегда** будет элемент с тегом <html>. Пример: `/html/body/header`

Если начинают запрос с символа //, то ищут всех потомков корневого элемента без указания корневого элемента. В этом случае, для поиска того же хедера, выполняют запрос `//header`, так как других заголовков нет.

Но такой поиск не рекомендуется!!!!

Например, запрос `//div` вернет все элементы с тегом <div>.

Поиск элементов с помощью Xpath.

Примеры

Абсолютный путь:

```
WebElement userName = driver.findElement(By.xpath("html/body/div/div/form/input"));
```

Относительный путь:

```
WebElement userName = driver.findElement(By.xpath("//input"));
```



Поиск элементов с помощью Xpath.

Примеры

Поиск дочернего элемента любого уровня:

```
WebElement userName = driver.findElement(By.xpath("//div//a"));
```



Поиск элемента по тексту:

```
WebElement userName = driver.findElement(By.xpath("//*[text()='Первая ссылка']/.."));
```



Поиск элементов с помощью Xpath.

Примеры

Поиск по значениям атрибутов:

```
WebElement userName = driver.findElement(By.xpath("//input[@id='username']"));
```

Поиск по названию атрибутов:

```
List<WebElement> imagesWithAlt = driver.findElements(By.xpath ("img[@alt]"));
```

Поиск родительского элемента:

```
WebElement userName = driver.findElement(By.xpath("//input[@id='username']/.."));
```

Поиск элементов с помощью локаторов CSS

css=<css_path> - данный тип локаторов основан на описаниях таблиц стилей (CSS), соответственно и синтаксис такой же. В отличие от локаторов по ID, по имени или по тексту ссылки, данный тип локаторов может учитывать иерархию объектов, а также значения атрибутов, что делает его ближайшим аналогом XPath.

Многие браузеры реализуют CSS движок, чтобы разработчики смогли применять CSS таблицы в своих проектах. Что позволяет разделить между собой контент страницы с её оформлением. В CSS есть паттерны, согласно которым стили, создаваемые разработчиком, применяются к элементам страницы (DOM). Эти паттерны называются локаторы (selectors).

А в силу того, что объект находится по данному локатору быстрее, чем XPath, рекомендуется прибегать к помощи CSS вместо XPath.

Абсолютный путь:

```
WebElement userName = driver.findElement(By.cssSelector("html body div div form input"));
```

Относительный путь:

```
WebElement userName = driver.findElement(By.cssSelector("input"));
```

Поиск непосредственного дочернего элемента:

```
WebElement userName = driver.findElement(By.cssSelector("div>a"));
```

Поиск дочернего элемента любого уровня:

```
WebElement userName = driver.findElement(By.cssSelector("div a"));
```

Поиск по ID элемента:

```
1 WebElement userName = driver.findElement(By.cssSelector("input#username"));
2
3 //или
4
5 WebElement userName = driver.findElement(By.cssSelector("#username"));
```

Поиск по классу:

```
1 WebElement userName = driver.findElement(By.cssSelector("input.classname"));
2
3 //или
4
5 WebElement userName = driver.findElement(By.cssSelector(".classname"));
```

Поиск по значениям атрибутов html тегов:

```
WebElement previousButton = driver.findElement(By.cssSelector("img[alt='Previous']"));
```

Поиск по названию атрибутов:

```
List<WebElement> imagesWithAlt = driver.findElements(By.cssSelector("img[alt]"));
```

Поиск по началу строки:

```
WebElement previousButton = driver.findElement(By.cssSelector("header[id^='page-']"));
```

Поиск по окончанию строки:

```
WebElement previousButton = driver.findElement(By.cssSelector("header[id$='page-']"));
```

Поиск по частичному совпадению строки:

```
WebElement previousButton = driver.findElement(By.cssSelector("header[id*='page-']"));
```

Поиск элементов с помощью Selenium

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

- ✓ `find_element_by_id`
- ✓ `find_element_by_name`
- ✓ `find_element_by_xpath`
- ✓ `find_element_by_link_text`
- ✓ `find_element_by_partial_link_text`
- ✓ `find_element_by_tag_name`
- ✓ `find_element_by_class_name`
- ✓ `find_element_by_css_selector`

Поиск элементов с помощью Selenium.

- ✓ **find_element_by_id** – поиск по уникальному атрибуту id элемента. Если разработчики проставляют всем элементам в приложении уникальный id, то лучше всего будет использовать этот метод, так как он наиболее стабильный;
- ✓ **find_element_by_css_selector** – поиск элемента с помощью правил на основе CSS. Это универсальный метод поиска, так как большинство веб-приложений использует CSS для вёрстки и задания оформления страниц. Если find_element_by_id вам не подходит из-за отсутствия id у элементов, то скорее всего лучше использовать именно этот метод в тестах;
- ✓ **find_element_by_xpath** – поиск с помощью языка запросов XPath, позволяет выполнять очень гибкий поиск элементов;
- ✓ **find_element_by_name** – поиск по атрибуту name элемента;
- ✓ **find_element_by_tag_name** – поиск элемента по названию тега элемента;
- ✓ **find_element_by_class_name** – поиск по значению атрибута class;
- ✓ **find_element_by_link_text** – поиск ссылки на странице по полному совпадению;
- ✓ **find_element_by_partial_link_text** – поиск ссылки на странице, если текст селектора совпадает с любой частью текста ссылки.

Поиск элементов с помощью Selenium. Пример

Рассмотрим следующий исходный код страницы:

```
<html>
  <body>
    <form id="loginForm">
      <input name="username" type="text" />
      <input name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
    </form>
  </body>
</html>
```

Элемент *form* может быть определен следующим образом:

```
login_form = driver.find_element_by_id('loginForm')
```

Поиск элементов с помощью Selenium. Пример

Рассмотрим следующий исходный код страницы:

```
<html>
<body>
  <form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <input name="continue" type="submit" value="Login" />
    <input name="continue" type="button" value="Clear" />
  </form>
</body>
</html>
```

Элементы с именами *username* и *password* могут быть определены следующим образом:

```
username = driver.find_element_by_name('username')
password = driver.find_element_by_name('password')
```

Поиск элементов с помощью Selenium. Пример поиска по XPath

Рассмотрим следующий исходный код страницы:

```
<html>
<body>
  <form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <input name="continue" type="submit" value="Login" />
    <input name="continue" type="button" value="Clear" />
  </form>
</body>
</html>
```

Элемент *form* может быть определен следующим образом:

```
login_form = driver.find_element_by_xpath("/html/body/form[1]")
login_form = driver.find_element_by_xpath("//form[1]")
login_form = driver.find_element_by_xpath("//form[@id='loginForm']")
```

Поиск элементов с помощью Selenium. Пример поиска по XPath

```
<html>
  <body>
    <form id="loginForm">
      <input name="username" type="text" />
      <input name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
      <input name="continue" type="button" value="Clear" />
    </form>
  </body>
</html>
```

Элемент *username* может быть определен следующим образом:

```
username = driver.find_element_by_xpath("//form[input/@name='username']")
username = driver.find_element_by_xpath("//form[@id='loginForm']/input[1]")
username = driver.find_element_by_xpath("//input[@name='username']")
```

Поиск элементов с помощью Selenium. Пример поиска по XPath

```
<html>
  <body>
    <form id="loginForm">
      <input name="username" type="text" />
      <input name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
      <input name="continue" type="button" value="Clear" />
    </form>
  </body>
</html>
```

Кнопка *Clear* может быть найдена следующими способами:

```
clear_button =
driver.find_element_by_xpath("//input[@name='continue'][@type='button']")

clear_button = driver.find_element_by_xpath("//form[@id='loginForm']/input[4]")
```

Поиск элементов с помощью Selenium.

Есть второй способ для поиска элементов с помощью универсального метода **find_element()** и полей класса By из библиотеки selenium.

Пример:



test_primer_1.py – Блокнот

Файл Правка Формат Вид Справка

```
from selenium import webdriver

from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("http://bga.by/contacts")
textarea = driver.find_element(By.TAG_NAME, "textarea")
textarea.send_keys("Мой первый тест через Selenium")
username = driver.find_element(By.ID, "124158159")
username.send_keys("Здесь введено мое имя")
submit_button = driver.find_element(By.CSS_SELECTOR, "#submitFormContacts")
submit_button.click()
driver.quit()
```

Поиск элементов с помощью Selenium.

Второй способ более удобен для оформления архитектуры тестовых сценариев с помощью подхода Page Object Model.

Поля класса `By`, которые можно использовать для поиска:

- **`By.ID`** – поиск по уникальному атрибуту `id` элемента;
- **`By.CSS_SELECTOR`** – поиск элементов с помощью правил на основе CSS;
- **`By.XPATH`** – поиск элементов с помощью языка запросов XPath;
- **`By.NAME`** – поиск по атрибуту `name` элемента;
- **`By.TAG_NAME`** – поиск по названию тега;
- **`By.CLASS_NAME`** – поиск по атрибуту `class` элемента;
- **`By.LINK_TEXT`** – поиск ссылки с указанным текстом. Текст ссылки должен быть точным совпадением;
- **`By.PARTIAL_LINK_TEXT`** – поиск ссылки по частичному совпадению текста.

Поиск элементов с помощью Selenium.

!Важно. Вы можете столкнуться с ситуацией, когда на странице будет несколько элементов, подходящих под заданные вами параметры поиска.

В этом случае WebDriver вернет вам только первый элемент, который встретит во время поиска по HTML. Если вам нужен не первый, а второй или следующие элементы, вам нужно либо задать более точный селектор для поиска, либо использовать методы **find_elements_by**.

find_elements в Selenium возвращает вам список веб-элементов, соответствующих значению локатора, в отличие от **find_element**, который возвращает только один веб-элемент. Если на веб-странице нет соответствующих элементов, **findElements** возвращает пустой список.

Основные различия методов поиска

find_element_by	find_elements_by
Returns the first matching web element within the web page even if multiple web elements match the locator value.	Returns a list of multiple web elements matching the locator value.
Throws NoSuchElementException in case there are no matching elements.	Returns an empty list in case there are no matching elements.
Returns a single web element	Returns a collection of web elements.
No indexing is required since only one element is returned.	Each web element is indexed starting from 0.

Фреймворк **unittest**

unittest – это *framework* для тестирования, входящий в стандартную библиотеку языка *Python*.

Его архитектура выполнена в стиле *xUnit*. *xUnit* представляет собой семейство *framework*'ов для тестирования в разных языках программирования, в *Java* – это *JUnit*, *C#* – *NUnit* и т.д.

Пример

Модуль *calc.py*:

```
def add(a, b):  
    return a + b
```

```
def sub(a, b):  
    return a - b
```

```
def mul(a, b):  
    return a * b
```

```
def div(a, b):  
    return a / b
```

Для того, чтобы протестировать эту библиотеку, мы можем создать отдельный файл с названием *test_calc.py* и поместить туда функции, которые проверяют корректность работы функций из *calc.py*.

Пример

```
import calc

def test_add():
    if calc.add(1, 2) == 3:
        print("Test add(a, b) is OK")
    else:
        print("Test add(a, b) is Fail")

def test_sub():
    if calc.sub(4, 2) == 2:
        print("Test sub(a, b) is OK")
    else:
        print("Test sub(a, b) is Fail")

def test_mul():
    if calc.mul(2, 5) == 10:
        print("Test mul(a, b) is OK")
    else:
        print("Test mul(a, b) is Fail")

def test_div():
    if calc.div(8, 4) == 2:
        print("Test div(a, b) is OK")
    else:
        print("Test div(a, b) is Fail")

test_add()
test_sub()
test_mul()
test_div()
```

Пример

В результате, в окне консоли, будет напечатано следующее:

```
Test add(a, b) is OK  
Test sub(a, b) is OK  
Test mul(a, b) is OK  
Test div(a, b) is OK
```

Пример

Это были четыре теста, которые проверяют работоспособность функций в простейшем случае. При написании тестов, как обычных программ, возникает ряд неудобств, в первую очередь связанных с унификацией выходной информации о пройденных и не пройденных тестах, сами тесты получаются довольно громоздкими, также необходимо продумывать архитектуру тестирующего приложения и т.д.

В дополнение к этому можно отметить отсутствие гибких инструментов для запуска требуемых только на данном этапе тестов, пропуска тестов по условию (например для разрабатываемой библиотеки, начиная с определённой версии, не выполнять конкретные тесты) и т.п. Все это приводит к мысли о том, что нужен какой-то *framework*, который возьмет на себя обязанности по поддержанию инфраструктуры проекта с тестами.

Пример тестирования приложения с использованием *unittest*

```
import unittest
import calc

class CalcTest(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

if __name__ == '__main__':
    unittest.main()
```


Пример тестирования приложения с использованием *unittest*

В результате, в окне консоли, будет напечатано следующее:

```
....
```

```
-----  
Ran 4 tests in 0.000s
```

```
OK
```

Пример тестирования приложения с использованием *unittest*

Запуск можно сделать с запросом расширенной информации по пройденным тестам, для этого необходимо добавить ключ `-v`:

```
python -v test_calc.py
```

```
test_add (test_calc_v2.CalcTest) ... ok
test_div (test_calc_v2.CalcTest) ... ok
test_mul (test_calc_v2.CalcTest) ... ok
test_sub (test_calc_v2.CalcTest) ... ok
-----
Ran 4 tests in 0.002s
OK
```

Основные структурные элементы *unittest*

Test suite – это коллекция тестов, которая может в себя включать как отдельные *test case*'ы так и целые коллекции (т.е. можно создавать коллекции коллекций).

Test runner – это компонент, который координирует взаимодействие запуск тестов и предоставляет пользователю результат их выполнения.

Test fixture – обеспечивает подготовку окружения для выполнения тестов, а также организацию мероприятий по их корректному завершению (например очистка ресурсов).

Основные структурные элементы *unittest*

Основным строительным элементом при написании тестов с использованием *unittest* является *TestCase*.

Он представляет собой класс, который должен являться базовым для всех остальных классов, методы которых будут тестировать те или иные автономные единицы исходной программы.

TestCase

```
import unittest
import calc

class CalcTests(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

if __name__ == '__main__':
    unittest.main()
```

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.

__name__ -- это специальная внутренняя переменная, которая инициализируется, как `'__main__'` если файл с исходным кодом был непосредственно запущен, а не импортирован.

При импорте, переменная будет содержать имя модуля, из которого произошел импорт.

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_"

Для unittest существуют собственные дополнительные правила:

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

Unittest. Алгоритм создания

- Импортировать unittest в файл:
 - **import unittest**
- Создать класс, который должен наследоваться от класса ***TestCase***:
 - **class CalcTest(unittest.TestCase):**
- Превратить тестовые функции в методы, добавив ссылку на экземпляр класса ***self*** в качестве первого аргумента функции:
 - **def test_add(self):**
- Изменить ***assert*** на ***self.assertEqual()***
- Заменить строку запуска программы на ***unittest.main()***

Методы класса *TestCase*

Все методы класса *TestCase* можно разделить на три группы:

- методы, используемые при запуске тестов;
- методы, используемые при непосредственном написании тестов (проверка условий, сообщение об ошибках);
- методы, позволяющие собирать информацию о самом тесте.

Методы, используемые при запуске тестов.

- *setUp()*

Метод вызывается перед запуском теста. Как правило, используется для подготовки окружения для теста.

Методы, используемые при запуске тестов.

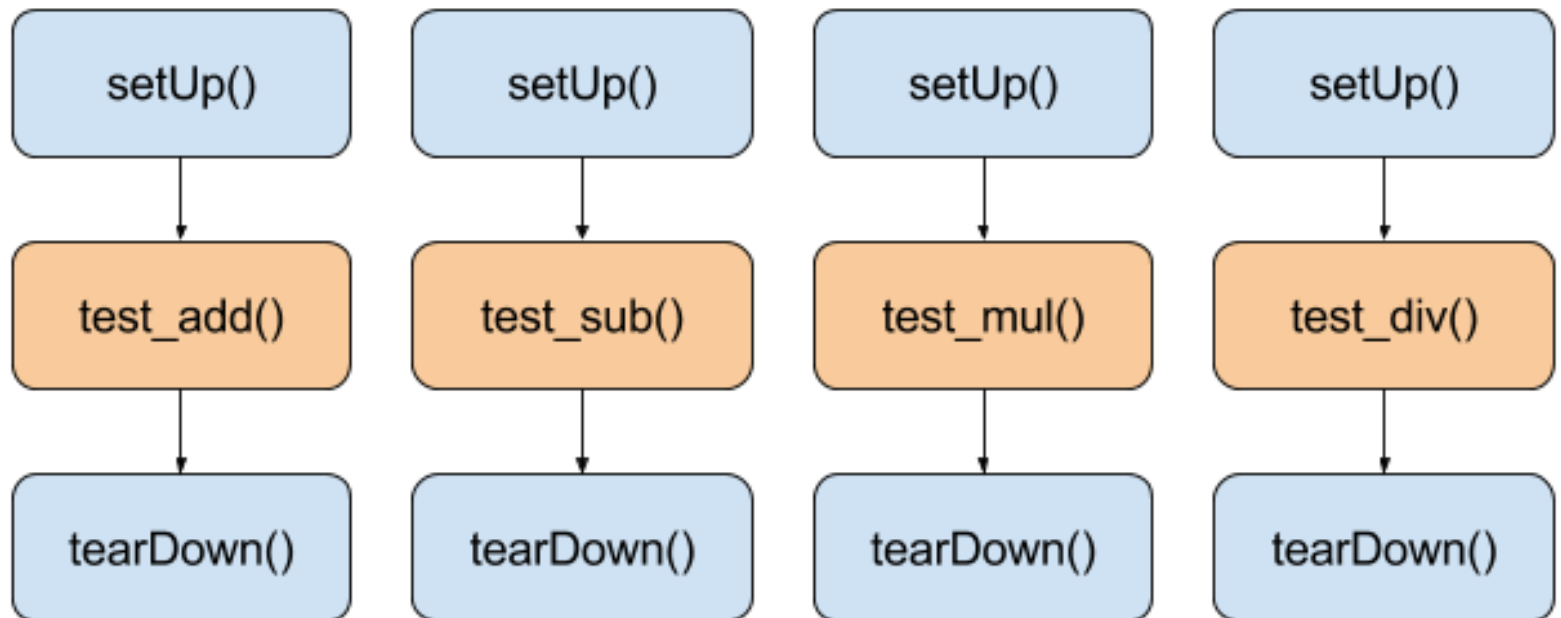
- *tearDown()*

Метод вызывается после завершения работы теста. Используется для “приборки” за тестом.

Заметим, что методы *setUp()* и *tearDown()* вызываются для всех тестов в рамках класса, в котором они переопределены.

По умолчанию, эти методы ничего не делают. Если их добавить в *utest_calc.py*, то перед [после] тестов *test_add()*, *test_sub()*, *test_mul()*, *test_div()* будут выполнены *setUp()* [*tearDown()*].

Методы, используемые при запуске тестов.



Методы, используемые при запуске тестов.

- *setUpClass()*

Метод действует на уровне класса, т.е. выполняется перед запуском тестов класса. При этом синтаксис требует наличие декоратора *@classmethod*.

```
@classmethod  
def setUpClass(cls):
```

```
    ...
```

Методы, используемые при запуске тестов.

- ***tearDownClass()***

Запускается после выполнения всех методов класса, требует наличия декоратора *@classmethod*.

```
@classmethod
```

```
def tearDownClass(cls):
```

```
...
```

Методы, используемые при запуске тестов.

- ***skipTest(reason)***

Данный метод может быть использован для пропуска теста, если это необходимо.

Assertion-методы для проверки

TestCase класс предоставляет набор *assertion*-методов для проверки и генерации ошибок:

assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	is True
assertFalse(x)	is False
<u>assertIs(a, b)</u>	a is b
<u>assertIsNot(a, b)</u>	a is not b
<u>assertIsNone(x)</u>	x is None
<u>assertIsNotNone(x)</u>	x is not None
<u>assertIn(a, b)</u>	a in b
<u>assertNotIn(a, b)</u>	a not in b

Assert-методы для проверки

Assert'ы для проверки различных ситуаций:

<u><code>assertGreater(a, b)</code></u>	<code>a > b</code>
<u><code>assertGreaterEqual(a, b)</code></u>	<code>a >= b</code>
<u><code>assertLess(a, b)</code></u>	<code>a < b</code>
<u><code>assertLessEqual(a, b)</code></u>	<code>a <= b</code>
<u><code>assertRegex(s, r)</code></u>	<code>r.search(s)</code>
<u><code>assertNotRegex(s, r)</code></u>	<code>not r.search(s)</code>
<u><code>assertCountEqual(a, b)</code></u>	а и b имеют одинаковые элементы (порядок неважен)

Методы, позволяющие собирать информацию о самом тесте.

- *countTestCases()*

Возвращает количество тестов в объекте класса-наследника от *TestCase*.

- *id()*

Возвращает строковый идентификатор теста. Как правило это полное имя метода, включающее имя модуля и имя класса.

- *shortDescription()*

Возвращает описание теста, которое представляет собой первую строку *docstring*'а метода, если его нет, то возвращает *None*.

ПРИМЕР

```
import unittest
import calc
```

```
class CalcTest(unittest.TestCase):
    """Calc tests"""

    @classmethod
    def setUpClass(cls):
        """Set up for class"""
        print("setUpClass")
        print("=====")

    @classmethod
    def tearDownClass(cls):
        """Tear down for class"""
        print("=====")
        print("tearDownClass")

    def setUp(self):
        """Set up for test"""
        print("Set up for [" + self.shortDescription() + "]")

    def tearDown(self):
        """Tear down for test"""
        print("Tear down for [" + self.shortDescription() + "]")
        print("")
```

```
    def test_add(self):
        """Add operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        """Sub operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        """Mul operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        """Div operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.div(8, 4), 2)
```

```
if __name__ == '__main__':
    unittest.main()
```

ПРИМЕР

```
setUpClass
=====
test_add (simple_ex.CalcTest)
Add operation test ... Set up for [Add operation test]
id: simple_ex.CalcTest.test_add
Tear down for [Add operation test]
ok

test_div (simple_ex.CalcTest)
Div operation test ... Set up for [Div operation test]
id: simple_ex.CalcTest.test_div
Tear down for [Div operation test]
ok

test_mul (simple_ex.CalcTest)
Mul operation test ... Set up for [Mul operation test]
id: simple_ex.CalcTest.test_mul
Tear down for [Mul operation test]
ok

test_sub (simple_ex.CalcTest)
Sub operation test ... Set up for [Sub operation test]
id: simple_ex.CalcTest.test_sub
Tear down for [Sub operation test]
ok
=====
tearDownClass
-----

Ran 4 tests in 0.016s
OK
```

Контрольные вопросы:

1. Что представляет собой **Selenium WebDriver**?
2. Как инициализировать драйвер браузера?
3. Как осуществить поиск элементов на странице?
4. Перечислить методы **WebDriver** для поиска элементов.
5. Что представляет собой **unittest**?
6. Описать алгоритм создания теста на основе фреймворка **unittest**.