Введение в Selenium.

Selenium - это бесплатная и открытая библиотека для автоматизированного тестирования веб-приложений и автоматизации действий в браузере.

Selenium поддерживает все основные браузеры, но в этом курсе мы будем говорить только про браузер **Chrome**, т.к. он является самым популярным браузером на планете. К тому же использование других браузеров отличается только способом установки **webdriver**..

Webdriver - Это основной продукт, разрабатываемый в рамках проекта Selenium.

- **Webdriver** - самая важная сущность, ответственная за управление браузером. Основной ход скрипта строится именно вокруг экземпляра этой сущности.
- **Webelement** - вторая важная сущность, представляющая собой абстракцию над веб-элементом (кнопки, ссылки, поля ввода и др.). **Webelement** - это DOM-объект, находящийся на веб странице.
- **By** - класс, содержащий статические методы для идентификации элементов, о которым подробнее мы будем говорить в следующих степах.

Преимущества Selenium

Selenium + бесплатный продукт с открытым исходным кодом.

Selenium + очень гибкий инструмент, мы можем писать поистине большие скрипты, которые будут собирать информацию по заданным нами алгоритмам с множества сайтов.

Selenium + разрабатывается с 2004 года. За это время он стал самым популярным инструментом для тестирования веб-приложений.

Selenium + имеет огромное комьюнити, как в России, так и за её границами.

Selenium + благодаря своей популярности, хорошо и многократно описан в интернете с кучей примеров. Но таких задачек, как на этом курсе, вы не найдете больше нигде 😎

Недостатки Selenium

Selenium - не просто устанавливать. Если вы, как разработчик, все делаете на автомате, то вот ваш клиент с фриланс-биржи может и не разобраться с тем, как обновить **webdriver** при очередном автоматическом обновлении **Chrome**.

Selenium - подходит для работы только с веб-сайтами.

Selenium - может подвисать на слабых машинах, потому что во время работы используется Chrome, который умеет поглощать для своих нужд огромное количество оперативной памяти. Важно учитывать это и давать браузеру время, чтобы прогрузить все содержимое страницы.

Selenium - на собственном опыте замечено, что **Selenium** может не запускать один и тот же скрипт с первого раза, хотя с повторных запусков все работает без проблем. Имейте в виду эту особенность.

Установка Selenium. Синтаксис.

```
!pip install selenium

import selenium
from selenium import webdriver
from selenium.webdriver.common.by import By
```

```
options = webdriver.ChromeOptions()
options.add_argument('--headless')
options.add_argument("--disable-gpu")

driver = webdriver.Chrome(options=options)

driver.get(url)
```

Запускает браузер в режиме без графического интерфейса. Это означает, что браузер будет работать в фоновом режиме и не будет отображать открываемые веб-страницы. Это удобно для автоматизации задач и веб-скрапинга, особенно на серверах без графической оболочки.

Отключает использование графического процессора (GPU) при отрисовке.
Эти два параметра часто используют вместе из соображений совместимости и стабильности. Например, на некоторых системах, запуск в режиме --headless без отключения GPU может привести к нежелательным сайд-эффектам или ошибкам.

Преимущества запуска браузера в фоновом режиме.

- **Отсутствует отрисовка содержимого:** Поскольку браузер не отображает веб-страницы, потребление ресурсов (CPU и RAM) снижается.
- **Быстродействие:** Нет необходимости в отрисовке элементов на странице, что ускоряет работу. Это особенно заметно на слабых машинах.
- **Экономия экранного пространства:** Запущенный в фоновом режиме браузер не занимает место на экране и не мешает вашей текущей работе.
- **Повышенная стабильность:** В фоновом режиме меньше вероятность сбоев из-за отсутствия взаимодействия с пользовательским интерфейсом.
- **Удобство автоматизации:** Процесс становится более управляемым для автоматических скриптов и тестов, т.к. не требует открывания и закрывания окон.
- **Скрытность:** В фоновом режиме более трудно для окружающих заметить, что именно вы делаете, что может быть полезно для сбора данных или тестирования.
- **Экономия времени:** Вам не нужно ждать, пока страницы загрузятся и элементы отобразятся, что делает процесс более эффективным по времени.
- **Легкость интеграции:** В фоновом режиме удобнее интегрировать веб-парсинг или автоматизацию браузера в большие системы или конвейеры обработки данных.
- **Уменьшение вероятности капчи:** Некоторые сайты менее строго реагируют на запросы из браузера, запущенного в фоновом режиме, по сравнению с браузером, активно используемым пользователем.

ChromeOptions().

`ChromeOptions` — это класс в библиотеке Selenium, предназначенный для настройки опций Chrome. Когда вы создаете объект этого класса, вы получаете возможность конфигурировать различные параметры и свойства браузера, прежде чем он будет запущен.

```
# Создание объекта ChromeOptions
options = webdriver.ChromeOptions()
```

Это включает в себя такие вещи как аргументы командной строки, использование прокси-сервера, установка расширений и множество других.

Метод `add_argument()` этого объекта служит для добавления аргументов командной строки к запуску браузера. Аргументы командной строки — это флаги или параметры, которые можно передать при запуске Chrome из командной строки, чтобы модифицировать его поведение. В контексте Selenium, `add_argument()` делает это за вас, передавая эти параметры при инициализации `webdriver.Chrome()`.

```
# Добавление аргументов командной строки
options.add_argument('--headless')  # Запуск браузера в фоновом режиме (без GUI)
options.add_argument('--disable-gpu')  # Отключение GPU (полезно для старых версий Chrome)
options.add_argument('--no-sandbox')  # Отключение режима "песочницы" (sandbox)
```

Объект `ChromeOptions` затем передается в конструктор `webdriver.Chrome()`, чтобы эти настройки были применены к новому экземпляру браузера.

```
# Запуск экземпляра браузера Chrome с заданными опциями
driver = webdriver.Chrome(options=options)
```

Большое количество команд для хрома можно найти по ссылке:
https://peter.sh/experiments/chromium-command-line-switches/

ChromeOptions().

Команда	Описание
`--disable-gpu`	Отключает аппаратное ускорение GPU. Иногда это делается для избежания проблем с графикой.
`--no-sandbox`	Запускает браузер без дополнительных мер безопасности.
`--incognito`	Запускает браузер в режиме инкогнито. В этом режиме не сохраняются куки и история просмотров.
`--window-size=width,height`	Устанавливает начальный размер окна браузера.
`--start-maximized`	Запускает браузер на весь экран.
`--disable-extensions`	Отключает все установленные расширения.
`--user-data-dir=path`	Устанавливает директорию для хранения профиля пользователя.
`--disable-infobars`	Убирает информационные строки в верхней части окна.
`--ignore-certificate-errors`	Игнорирует ошибки SSL-сертификатов. Полезно, если нужно обращаться к сайтам с недействительными сертификатами.
`--lang=ru`	Устанавливает язык интерфейса браузера на русский.

```
chrome_options.add_argument('--proxy-server=%s' % proxy)
```

работа с прокси в Selenium

ChromeOptions().

`--disable-popup-blocking`	Отключает блокировку всплывающих окон. Может быть полезным при автоматизации некоторых сценариев.
`--allow-running-insecure-content`	Позволяет загружать небезопасный контент на страницы, загруженные по HTTPS. Опасная опция, используйте с осторожностью.
`--disable-notifications`	Отключает уведомления браузера. Особенно полезно при автоматизированном тестировании.
`--disable-web-security`	Отключает меры безопасности веба. Не рекомендуется для обычного просмотра, но может быть полезно для тестирования.
`--disable-client-side-phishing-detection`	Отключает обнаружение фишинга на клиентской стороне.
`--enable-logging`	Включает журналирование в файл.
`--log-level=0`	Устанавливает уровень журналирования (0-3).
`--disable-cache`	Отключает кэш браузера. Полезно для тестирования изменений на веб-страницах в реальном времени.
`--enable-automation`	Подсказывает браузеру, что он управляется программой. Это может изменить поведение некоторых веб-сайтов.
`--disable-setuid-sandbox`	Отключает песочницу безопасности для браузера. Также не рекомендуется для обыденного использования.
`--disable-sync`	Отключает синхронизацию с аккаунтом Google.

Поиск элементов Selenium.

Локатор - это способ идентификации элементов на странице. Это аргумент, передаваемый методам поиска элементов.

Локаторы играют очень важную роль при работе с **Selenium**. Они обеспечивают путь к веб-элементам, которые необходимы для автоматизации определенных действий, таких как клик, ввод, установка флага и др.

Для начала, импортируем класс `By` из модуля `selenium.webdriver.common.by` :

```
from selenium.webdriver.common.by import By
```

Выглядит класс `By` следующим образом.

```
class By:
    """Set of supported locator strategies."""

    ID = "id"
    XPATH = "xpath"
    LINK_TEXT = "link text"
    PARTIAL_LINK_TEXT = "partial link text"
    NAME = "name"
    TAG_NAME = "tag name"
    CLASS_NAME = "class name"
    CSS_SELECTOR = "css selector"
```

Класс `By` — это, по сути, перечисление, которое содержит различные стратегии поиска элементов на веб-странице. Использование `By` позволяет сделать код более читаемым и поддерживаемым, так как это стандартизирует способы поиска элементов.

Поиск элементов Selenium.

- `By.ID` – Поиск элемента по уникальному идентификатору (`id`). Этот метод очень быстрый и надежный, но требует, чтобы у элемента был атрибут `id` .

```
element = driver.find_element(By.ID, "some_id")
```

- `By.CSS_SELECTOR` – Поиск элемента или элементов, используя селекторы CSS. Это гибкий и мощный метод, который может выразить сложные критерии поиска.

```
elements = driver.find_elements(By.CSS_SELECTOR, ".some_class")
```

- `By.XPATH` – Поиск элемента с помощью языка XPath. XPath позволяет создать более сложные запросы, но он менее читаемый и, возможно, будет работать медленнее, чем другие методы.

```
element = driver.find_element(By.XPATH, "//div[@attribute='value']")
```

- `By.NAME` – Поиск элемента по атрибуту `name` . Этот метод хорошо подходит для форм.

```
element = driver.find_element(By.NAME, "username")
```

Поиск элементов Selenium.

- `By.TAG_NAME` – Поиск элемента по названию HTML-тега. Этот метод полезен, если нужно выбрать, например, все изображения на странице.

```
images = driver.find_elements(By.TAG_NAME, "img")
```

- `By.CLASS_NAME` – Поиск элемента или элементов по классу. Этот метод полезен, если у элементов есть общий класс.

```
buttons = driver.find_elements(By.CLASS_NAME, "btn")
```

- `By.LINK_TEXT` – Поиск элемента по точному тексту ссылки. Очень удобно, если текст уникален.

```
element = driver.find_element(By.LINK_TEXT, "Continue")
```

- `By.PARTIAL_LINK_TEXT` – Поиск элемента по частичному тексту ссылки. Удобно, когда точный текст ссылки неизвестен или динамичен.

```
element = driver.find_element(By.PARTIAL_LINK_TEXT, "Cont")
```

Поиск элементов Selenium.

Локаторы используются с помощью двух универсальных методов - `find_element()`, который возвращает ровно один элемент, найденный первым, и `find_elements()`, который возвращает список найденных элементов.

`.find_element()`

Метод `find_element()` используется, когда вам нужно найти один конкретный элемент на странице. Он возвращает первый элемент, который соответствует заданным критериям поиска. Если элемент не найден, Selenium сгенерирует исключение `NoSuchElementException`.

```
# Ищем элемент с тегом img
elements = driver.find_element(By.TAG_NAME, 'img')
```

`.find_elements()`

Метод `find_elements()` полезен, когда вы хотите получить список всех элементов, которые соответствуют заданным критериям. В отличие от `find_element()`, этот метод вернёт пустой список, если ничего не найдено, вместо того чтобы генерировать исключение.

```
# Ищем все элементы с классом some_class
elements = driver.find_elements(By.CLASS_NAME, 'some_class')
```

Работаем с браузером.

Когда наш парсер отработает, мы бы хотели, чтобы он закрылся сам и тем самым корректно завершил свою работу. Но это может не произойти по множеству причин. Поэтому мы должны указать браузеру, что он должен закрыть окно после завершения работы, командой `driver.quit()`. Важно закрывать окно, потому что при создании `webdriver.Chrome()` создается процесс в **ОС**, который продолжит висеть. Команда `quit()` проще, чем закрывать окно браузера вручную, к тому же вы не будете засорять оперативную память.

Другой способ – менеджер контекста with/as. С этим способом нам вообще не нужно думать о том, когда закрывать браузер, менеджер контекста делает это за нас в тот момент, когда это нужно.

```
with webdriver.Chrome() as
driver:
    …
```

`driver.close()` - закрывает текущее окно браузера, если во время работы вы открыли новое окно или вкладку.

`driver.quit()` - закрывает все окна, вкладки, процессы веб-драйвера, которые были запущены во время сессии.

Объект WebElement.

Когда вы работаете с Selenium, одной из самых основных задач является поиск и взаимодействие с элементами на веб-странице. Когда вы ищете элемент через методы вроде `find_element()` или `find_elements()`, возвращаемым типом данных является объект `WebElement`.

Сам же объект `WebElement` в **Selenium** представляет собой DOM-элемент на веб-странице. Этот объект предоставляет методы и атрибуты для взаимодействия с элементом, такие как клик, ввод текста или извлечение атрибутов и др.

```
from selenium import webdriver
from selenium.webdriver.common.by import By

url = 'http://parsinger.ru/selenium/3/3.html'
with webdriver.Chrome() as browser:
    browser.get(url)
    elem = browser.find_element(By.CLASS_NAME, 'text')
    print(elem)
```

Вывод:

```
<selenium.webdriver.remote.webelement.WebElement (session="a86f9c5223a7fa5ac8d6c1911f5bfc16",
element="9F9569F9515A0E022F0E665284FFB19D_element_2")>
```

Объект WebElement.

Этот объект `WebElement` — ваш ключ к манипуляциям с элементом на странице. Вы можете применять к нему различные методы, такие как:

1. `.click()` для симуляции клика мышью.

```
browser.find_element(By.ID, "some_button_id").click()
```

2. `.send_keys()` для ввода текста (полезно для текстовых полей).

```
browser.find_element(By.NAME, "some_textbox_name").send_keys("Hello, World!")
```

3. `.get_attribute('some_attribute')` для получения атрибутов, например, `href` у ссылок.

```
browser.find_element(By.TAG_NAME, "a").get_attribute("href")
```

4. `.text` для получения видимого текста элемента.

```
browser.find_element(By.CLASS_NAME, "some_class_name").text
```

5. И многое другое.

Комбинирование .find_element() и .find_elements()

Сценарий 1: Каскадный поиск

Иногда, чтобы добраться до конкретного элемента, нужно сначала найти его "родительский" элемент, и уже внутри него искать дочерний.

```
# Ищем родительский элемент
parent_element = driver.find_element(By.ID, 'parent_id')

# Ищем дочерний элемент внутри родительского
child_element = parent_element.find_element(By.CLASS_NAME, 'child_class')
```

```
# Или тот же самый поиск в одну строку
element = driver.find_element(By.ID, 'parent_id').find_element(By.CLASS_NAME, 'child_class')
```

Сценарий 2: Поиск внутри списка элементов

Представьте, что у вас на странице несколько однотипных блоков, и в каждом из них есть кнопка или какой-то другой элемент, с которым нужно взаимодействовать.

```
# Ищем все блоки
blocks = driver.find_elements(By.CLASS_NAME, 'block')

# Проходим по каждому блоку и кликаем на кнопку внутри
for block in blocks:
    button = block.find_element(By.CLASS_NAME, 'button')
    button.click()
```

Сценарий 3: Проверка существования элементов

Вы можете сначала проверить, есть ли на странице интересующие вас элементы, и только затем с ними взаимодействовать.

```
# Ищем все элементы с классом 'some_class'
elements = driver.find_elements(By.CLASS_NAME, 'some_class')

# Если элементы найдены, кликаем на первый
if elements:
    elements[0].click()
```

Основные методы Selenium.

Навигация по истории браузера

Далее, `browser` — это ссылка на конкретный экземпляр `webdriver.Chrome()`. Например:

```
with webdriver.Chrome(options=options_chrome) as browser:
    ...
```

- `browser.back()` - С помощью этого метода вы можете вернуться на предыдущую страницу, как если бы нажали стрелочку "назад" в браузере.
  - 
- `browser.forward()` - Аналогично предыдущему, но перемещает вперёд по истории браузера.
  - 
- `browser.refresh()` - Этот метод обновляет текущую страницу, как если бы вы нажали кнопку обновления в браузере.
  - 

Основные методы Selenium.

Работа со скриншотами

- `browser.get_screenshot_as_file("../file_name.jpg")` - Сохраняет скриншот страницы в файл по указанному пути. Возвращает `True` если всё прошло успешно, и `False` при ошибках ввода-вывода.
- `browser.save_screenshot("file_name.jpg")` - Сохраняет скриншот в папке с проектом.
- `browser.get_screenshot_as_png()` - Возвращает скриншот в виде двоичных данных (binary data), которые можно передать или сохранить в файл в конструкторе `with/as` ;
- `browser.get_screenshot_as_base64()` - Возвращает скриншот в виде строки в кодировке **Base64**. Удобно для встроенных изображений в HTML.

Открытие и закрытие страниц и браузера

- `browser.get("http://example_url.ru")` - Открывает указанный URL в браузере.
- `browser.quit()` - Закрывает все вкладки и окна, завершает процесс драйвера, освобождает ресурсы.
- `browser.close()` - Закрывает только текущую вкладку.

Исполнение JavaScript

- `browser.execute_script("script_code")` - Выполняет JavaScript код на текущей странице.
- `browser.execute_async_script("script_code" , *args )` - Асинхронно выполняет JavaScript код. Удобно для работы с AJAX и промисами.

Основные методы Selenium.

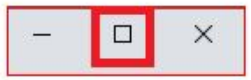
Время ожидания

- `browser.set_page_load_timeout()` - Устанавливает таймаут на загрузку страницы. Выбрасывает исключение, если время вышло.

Поиск элементов

- `browser.find_element(By.ID, 'example_id')` - Возвращает первый найденный элемент по заданному локатору.
- `browser.find_elements(By.ID, 'example_id')` - Возвращает список всех элементов, соответствующих локатору.

Работа с окном браузера

- `browser.get_window_position()` - Возвращает словарь с текущей позицией окна браузера (`{'x': 10, 'y': 50}`).
- `browser.maximize_window()` - Разворачивает окно на весь экран.
  - 
- `browser.minimize_window()` - Сворачивает окно.
  - 
- `browser.fullscreen_window()` - Переводит окно в полноэкранный режим, как при нажатии **F11**.
- `browser.get_window_size()` - Возвращает размер окна в виде словаря (`{'width': 945, 'height': 1020}`).
- `browser.set_window_size(800,600)` - Устанавливает новый размер окна.

Основные методы Selenium.

Работа с элементами

1. `element.click()` - Симулирует клик по элементу.
2. `element.send_keys("text")` - Вводит текст в текстовое поле. Очень полезно для автоматизации ввода данных.
3. `element.clear()` - Очищает текстовое поле.
4. `element.is_displayed()` - Проверяет, отображается ли элемент на странице.
5. `element.is_enabled()` - Проверяет, доступен ли элемент для взаимодействия (например, не заблокирован).
6. `element.is_selected()` - Проверяет, выбран ли элемент (актуально для радиокнопок и чекбоксов).
7. `element.get_attribute("attribute")` - Возвращает значение указанного атрибута элемента.
8. `element.text` - Возвращает текст элемента.
9. `element.submit()` - Отправляет форму, в которой находится элемент.

Основные методы Selenium.

Работа с cookies

- `browser.get_cookies()` - Возвращает список всех **cookies**.
- `browser.get_cookie(name_cookie)` - Возвращает конкретную cookie по имени.
- `browser.add_cookie(cookie_dict)` - Добавляет новую **cookie** к вашему текущему сеансу;
- `browser.delete_cookie(name_cookie)` - Удаляет **cookie** по имени.
- `browser.delete_all_cookies()` - удаляет все файлы **cookie** в рамках текущего сеанса;

Ожидание элементов

- `browser.implicitly_wait(10)` - Устанавливает неявное ожидание на поиск элементов или выполнение команд.
- `browser.WebDriverWait(driver, timeout).until(condition)`

Фреймы

1. `browser.switch_to.frame("frame_name")` - Переключает фокус на указанный фрейм.
2. `browser.switch_to.default_content()` - Возвращает фокус на основное содержимое страницы, выходя из фрейма.

JavaScript Alerts

1. `browser.switch_to.alert` - Переключает фокус на всплывающее окно JavaScript.

Скроллинг страниц.

Полоса прокрутки представляет собой тонкую, длинную часть на краю дисплея компьютера. Используя полосу прокрутки, мы можем просматривать весь контент или всю страницу, прокручивая её вверх и вниз или влево и вправо с помощью мыши.

Самый простой способ прокрутки страницы по пикселям — это использование метода `.execute_script()`, который выполняет код JavaScript на странице. К примеру, `window.scrollBy(0, 5000)` прокрутит страницу вниз на 5000 пикселей.

```
window.scrollBy(X, Y);
```

- `X` - смещение в пикселях по горизонтали;
- `Y` - смещение в пикселях по вертикали.

Напишем простой код, который прокрутит страницу в низ за 10 итераций.

```
import time
from selenium import webdriver

with webdriver.Chrome() as browser:
    browser.get('http://parsinger.ru/scroll/1/')
    for i in range(10):
        browser.execute_script("window.scrollBy(0,5000)")
        time.sleep(2)
```

Мы сделали 10 итераций и не оказались в самом низу страницы, потому что не знаем, какова высота нашего сайта в пикселях. Мы можем использовать большие значения, например, `window.scrollBy(0, 500000)`, чтобы одним скроллингом прокрутить всю страницу. Это подходит, когда у вас обычный сайт без динамической подгрузки данных.

Скроллинг страниц.

Если вам необходимо прокрутить страницу до самого низа, до последнего пикселя, одним из самых простых способов будет использование скрипта `window.scrollTo(0, document.body.scrollHeight)`.

```
import time
from selenium import webdriver

with webdriver.Chrome() as browser:
    browser.get('http://parsinger.ru/scroll/1/')
    browser.execute_script("window.scrollTo(0, document.body.scrollHeight);")
    time.sleep(2)
```

Где значение `0` означает горизонтальное смещение в пикселях от начальной точки прокрутки. В данном случае, `0` говорит о том, что прокрутка будет совершена без горизонтального смещения, то есть по вертикали.

При написании парсеров часто необходимо сначала совершить необходимое количество скроллинга, чтобы загрузилась вся нужная вам информация. После того, как вся необходимая информация появилась на странице, мы собираем всё при помощи метода `.find_elements()`.

.execute_script()

Синтаксис: `webdriver.execute_script(script, *args)`

- `.execute_script("return arguments[0].scrollIntoView(true);", element)` — прокручивает родительский контейнер элемента таким образом, чтобы `element`, для которого вызывается `scrollIntoView`, был виден пользователю.
- `.execute_script("window.open('http://parsinger.ru', 'tab2');")` — создаст новую вкладку в браузере с именем "tab2".
- `.execute_script("return document.body.scrollHeight")` — вернёт значение высоты элемента `<body>`.
- `.execute_script("return window.innerHeight")` — вернёт значение высоты окна браузера.
- `.execute_script("return window.innerWidth")` — вернёт значение ширины окна браузера.
- `.execute_script("window.scrollBy(X, Y)")` — прокрутит документ на заданное число пикселей по осям X и Y.
  - `X` — смещение в пикселях по горизонтали.
  - `Y` — смещение в пикселях по вертикали.
- `.execute_script("alert('Ура Selenium')")` — вызывает модальное окно Alert.
- `.execute_script("return document.title;")` — возвращает `title` открытого документа.
- `.execute_script("return document.documentURI;")` — возвращает URL документа.
- `.execute_script("return document.readyState;")` — возвращает состояние загрузки страницы; вернёт "complete", если страница загрузилась.

.execute_script()

- `.execute_script("return document.anchors;")` — возвращает список всех якорей на странице.
  - `[x.tag_name for x in browser.execute_script("return document.anchors;")]` — этот код позволяет получить список всех тегов с якорями. Очень полезная инструкция, особенно если при скроллинге элемент для "зацепления" не найден.
- `.execute_script("return document.cookie;")` — возвращает строку, содержащую все cookies документа, разделённые точкой с запятой.
- `.execute_script("return document.domain;")` — возвращает домен текущего документа.
- `.execute_script("return document.forms;")` — возвращает список всех форм на странице.
- `.execute_script("window.scrollTo(x-coord, y-coord);")` — прокручивает документ до указанных координат:
  - `x-coord` — позиция по горизонтальной оси, которая будет отображена вверху
  - `y-coord` — позиция по вертикальной оси, которая будет отображена вверху слева.
- `.execute_script("return document.getElementsByClassName('container');")` — возвращает список всех элементов с классом `'container'`.
- `.execute_script("return document.getElementsByTagName('container');")` — возвращает список всех элементов с тегом `'container'`.
- `.execute_script("return document.getElementById('some-id');")` — возвращает элемент с указанным **ID** `'some-id'`.

Неявное ожидание (Implicit waits)

Selenium имеет встроенный способ автоматического ожидания элементов, называемый неявным ожиданием(**Implicit waits**). Это глобальная настройка, которая применяется ко всем вызовам поиска элементов на протяжении всей сессии. По умолчанию значение равно 0, что означает, что если элемент не найден, он немедленно вернет ошибку. Если установлено неявное ожидание, драйвер будет ждать указанное время перед возвратом ошибки.

```
driver.implicitly_wait(2)
driver.get('https://www.selenium.dev/selenium/web/dynamic.html')
driver.find_element(By.ID, "adder").click()
added = driver.find_element(By.ID, "box0")
```

`Implicit Wait` — это время, которое Selenium будет ждать по умолчанию перед тем как выбросить исключение, если он не может найти элемент. Это как бы глобальный таймер, который применяется ко всем операциям поиска элементов.

Аналогия из реальной жизни

Представьте, что вы идете в кафе и заказываете кофе. Бариста говорит, что приготовление займет около 5 минут. Вы решаете подождать. Эти 5 минут — это ваше "ожидание"(`Implicit Wait`). Если через 5 минут или раньше кофе готов, вы его забираете и всё у вас хорошо. Если нет, вы начинаете задаваться вопросом, что происходит, возможно даже начинаете нервничать. В контексте Selenium, если элемент не появляется в течение заданного времени ожидания, возникает исключение.

Почему необходимо использовать Implicit Wait?

1. **Простота**: Это простой способ убедиться, что ваш код будет ждать достаточное количество времени перед тем как продолжить выполнение.
2. **Глобальность**: Один раз установив, он применяется ко всем последующим операциям поиска.

Неявное ожидание (Implicit waits)

Важные замечания

- Не рекомендуется смешивать `Implicit Waits` и `Explicit Waits` в одном тесте, так как это может привести к непредсказуемым результатам.
- `Implicit Wait` не всегда является лучшим решением, особенно если на веб-странице много динамически загружаемого контента. В таких случаях лучше использовать `Explicit Waits`.

Как это работает?

1. **Задание времени ожидания**: Когда вы устанавливаете `Implicit Wait`, вы фактически говорите Selenium: "Если я попытаюсь найти элемент и не найду его сразу, не выбрасывай ошибку. Вместо этого жди заданное мной время, пытаясь найти этот элемент".

2. **Поиск элемента**: Когда вы пытаетесь найти элемент на странице (например, используя `find_element_by_id`), Selenium сначала проверяет, виден ли этот элемент сразу.

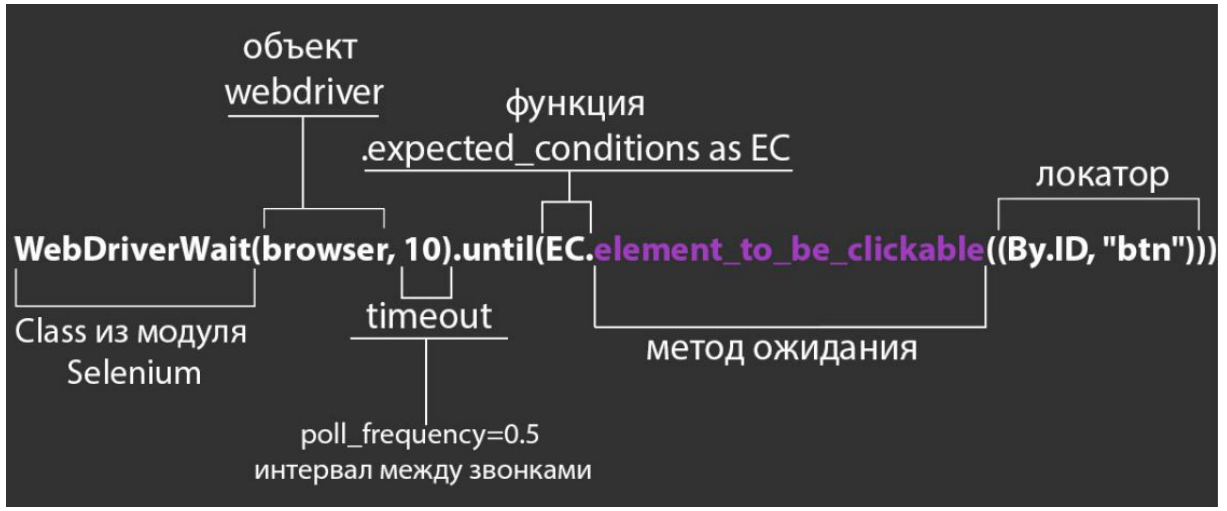
3. **Ожидание**: Если элемент не найден сразу:

   - Selenium будет периодически (*каждые 0.2 сек, по умолчанию*) проверять наличие элемента на странице.
   - Это повторяется до тех пор, пока элемент не будет найден или пока не истечет установленное время `Implicit Wait`.

4. **Результат**:

   - Если элемент появляется на странице в течение заданного времени ожидания, код продолжает выполнение.
   - Если время истекло, и элемент так и не был найден, Selenium выбросит исключение `NoSuchElementException`.

expected_conditions as EC




```
WebDriverWait(browser, poll_frequency=0.5, timeout=10).until(EC.element_to_be_clickable((By.ID, "btn")))
```

- Параметры `WebDriverWait` :
  - `browser` : Экземпляр WebDriver (например, Ie, Firefox, Chrome или Remote)
  - `poll_frequency=float` : (*необязательный*): Интервал ожидания между попытками. По умолчанию равен значению 0.5 секунды.
  - `timeout=float` : Время ожидания в секундах до таймаута
- `.until(method)` : Ожидает, пока предоставленный `method` вернет что-либо, кроме `False` . Если `method` продолжает возвращать `False` после истечения времени ожидания, будет вызвано исключение `TimeoutException` .
- `.until_not(method)` : Ожидает, пока предоставленный `method` не вернет `False` . Если метод не вернет `False` до истечения времени ожидания, будет вызвано исключение `TimeoutException` .

Явные ожидания (Explicit waits)

```python
import time

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.support.ui import WebDriverWait
```

- `EC.frame_to_be_available_and_switch_to_it(locator)`
- `EC.invisibility_of_element_located(locator)`
- `EC.invisibility_of_element(element)`
- `EC.element_to_be_clickable(mark)`
- `EC.staleness_of(element)`
- `EC.element_to_be_selected(element)`
- `EC.element_located_to_be_selected(locator)`
- `EC.element_selection_state_to_be(element, is_selected)`
- `EC.element_located_selection_state_to_be(locator, is_selected)`
- `EC.number_of_windows_to_be(num_windows)`
- `EC.new_window_is_opened(current_handles)`
- `EC.alert_is_present()`

- `EC.text_to_be_present_in_element(locator, text_)`
- `EC.text_to_be_present_in_element_value(locator, text_)`
- `EC.text_to_be_present_in_element_attribute(locator, attribute_, text_)`
- `EC.element_attribute_to_include(locator, attribute_)`

- `EC.title_is(title: str)`
- `EC.title_contains(title: str)`
- `EC.url_contains(url: str)`
- `EC.url_matches(pattern: str)`
- `EC.url_to_be(url: str)`
- `EC.url_changes(url: str)`

- `EC.presence_of_element_located(locator)`
- `EC.visibility_of_element_located(locator)`
- `EC.visibility_of(element)`
- `EC.presence_of_all_elements_located(locator)`
- `EC.visibility_of_any_elements_located(locator)`
- `EC.visibility_of_all_elements_located(locator)`