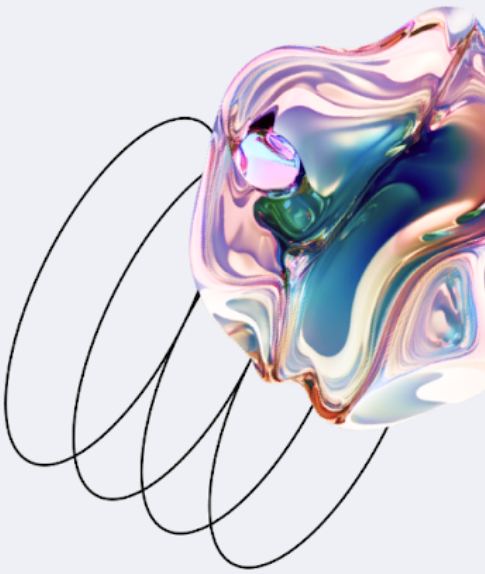
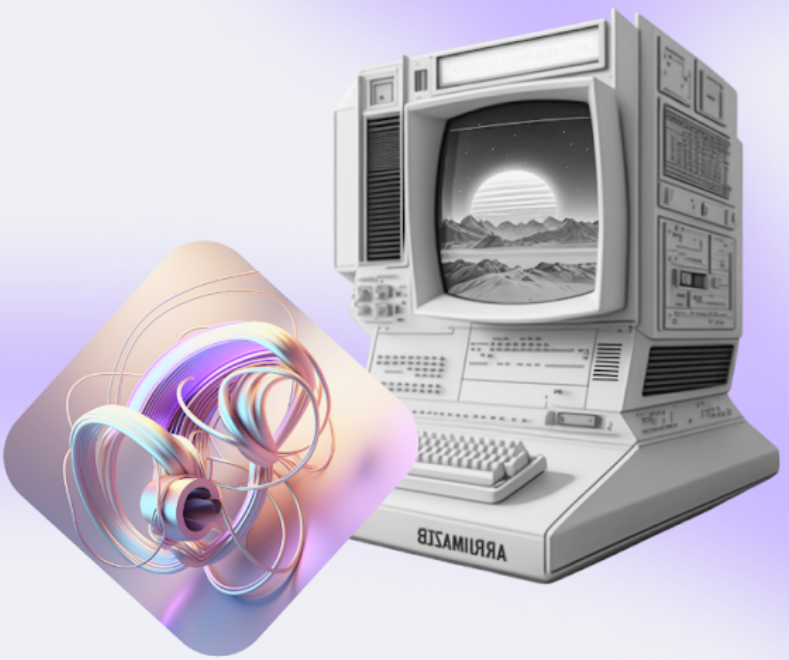
GeekBrains



Selenium в Python

Сбор и разметка данных



Оглавление

Введение

Selenium — это набор инструментов с открытым исходным кодом для автоматизации работы браузера. Selenium разработали в 2004 году, и с тех пор он стал одним из самых используемых инструментов автоматизации браузеров.

Selenium — универсальный инструмент. Его можно использовать:

- **Для автоматизации повторяющихся задач на сайтах**. Например, для заполнения форм, отправки данных и навигации по страницам. Автоматизация сэкономит время и снизит риск человеческой ошибки.
- **Для тестирования веб-приложений**, чтобы убедиться, что они работают так, как задумано. Например, для тестирования функциональности, производительности и совместимости с разными браузерами и операционными системами.
- **Для веб-скрейпинга**, чтобы извлечь данные с сайтов, которые трудно скрейпировать с помощью других инструментов. Например, с сайтов, использующих динамическое содержимое, JavaScript и так далее. Веб-скрейпинг — одно из основных применений Selenium.

Обзор экосистемы Selenium

У Selenium есть несколько компонентов для разных целей:

- **Selenium WebDriver** — основной компонент пакета Selenium. Используется для автоматизации веб-браузеров и тестирования веб-приложений.
- **Selenium Grid** — позволяет запускать тесты на нескольких машинах параллельно, чтобы ускорить их выполнение.
- **Selenium IDE** — предоставляет графический интерфейс для создания и выполнения тестов Selenium. Это простой и быстрый способ начать работу с Selenium, но не такой гибкий и мощный, как Selenium WebDriver.
- **Selenium Remote Control** (RC) — старый компонент, который сейчас во многом заменил Selenium WebDriver. Позволяет запускать тесты Selenium на удаленных машинах. Устарел и больше не поддерживается.

В этой лекции мы сосредоточимся на Selenium WebDriver — самом часто используемом компоненте для веб-скрейпинга.

Настройка среды разработки в Visual Studio Code

Создадим новый проект. Откроем VS Code, затем перейдем в меню File и выберем Open Folder. Создадим новую папку для проекта и назовем ее selenium_lesson. Откроем папку и начнем создавать файлы.

На панели Explorer щелкнем правой кнопкой мыши на папке проекта и выберем New File. Назовем файл main.py.

Прежде чем писать код, установим библиотеку Selenium.

Откроем терминал в Visual Studio Code: Terminal → New Terminal или сочетанием клавиш Ctrl + Shift + ‘ (для Windows) / Cmd + Shift + ‘ (для Mac).

Активируем виртуальную среду:

```
conda activate GeekBrain
```

Выполним команду для установки Selenium:

```
pip install selenium
```

Навигация по веб-страницам с помощью Selenium

Selenium WebDriver — инструмент для автоматизации тестирования веб-приложений. Позволяет разработчикам моделировать взаимодействие пользователя с веб-приложением для проверки его функциональности, производительности и удобства.

Представьте, что вы создаете интернет-магазин одежды. Вы хотите убедиться, что сайт работает так, как задумано, что на нем нет ошибок, которые могли бы повлиять на работу пользователей. Здесь на помощь приходит Selenium WebDriver.

С помощью Selenium WebDriver вы можете написать сценарии, которые имитируют просмотр сайта пользователем, добавление товаров в корзину и покупку. Эти сценарии можно запускать несколько раз, чтобы проверить последовательность поведения сайта и убедиться, что процесс оформления проходит гладко и без ошибок.

Давайте рассмотрим простой сценарий, который открывает веб-сайт, нажимает на товар, добавляет его в корзину и проверяет, правильный ли товар добавлен в корзину:

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

driver = webdriver.Chrome()
driver.get("https://www.example.com")

product = driver.find_element(By.XPATH
, "//a[@href='/products/shirt']")
product.click()

add_to_cart = driver.find_element(By.XPATH,
"//button[text()='Добавить в корзину']")
add_to_cart.click()

cart_items = driver.find_elements(By.XPATH,
"//td[@class='cart-item-name']")
assert len(cart_items) == 1, "В корзине должен быть только 1
товар"
assert cart_items[0].text == "Рубашка", "В корзину добавлен
неправильный товар"

driver.quit()
```

Модуль selenium.webdriver предоставляет все реализации WebDriver. Реализации, которые поддерживаются сейчас: Firefox, Chrome, IE и Remote. Класс Keys предоставляет клавиши клавиатуры (ENTER, F1, ALT и другие). Класс By используется для определения местоположения элементов в документе.

Затем создается экземпляр Chrome WebDriver.

Метод driver.get выполняет переход на страницу, заданную URL. WebDriver будет ждать, пока страница полностью загрузится, прежде чем вернуть управление вашему сценарию.

Существуют различные стратегии поиска элементов на странице — используйте ту, которая лучше подходит для вашего случая.

Метод Selenium для поиска элементов на странице:

```
find_element
```

Для поиска нескольких элементов (возвращается список):

```
find_elements
```

Атрибуты, доступные для класса By, используются для определения местоположения элементов на странице:

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

Класс By нужен, чтобы указать, какой атрибут используется для расположения элементов на странице. Способы использования атрибутов для расположения элементов на странице:

```
find_element(By.ID, "id")
find_element(By.NAME, "name")
find_element(By.XPATH, "xpath")
find_element(By.LINK_TEXT, "link text")
find_element(By.PARTIAL_LINK_TEXT, "partial link text")
find_element(By.TAG_NAME, "tag name")
find_element(By.CLASS_NAME, "class name")
find_element(By.CSS_SELECTOR, "css selector")
```

Если вы хотите найти несколько элементов с одинаковым атрибутом, замените find_element на find_elements.

Метод click() используется для перехода по ссылке.

Selenium предоставляет несколько методов для навигации и взаимодействия с веб-страницами. Рассмотрим некоторые из них.

1. **Метод click** — для перехода к определенному URL. Пример кода для перехода на домашнюю страницу Google:

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://www.google.com")
```

2. **Методы back и forward** — для навигации назад и вперед в истории браузера. Например, если вы перешли на несколько страниц, вы можете вернуться на предыдущую страницу с помощью кода:

```
driver.back()
```

3. **Метод refresh** — для обновления текущей страницы. Например:

```
driver.refresh()
```

4. **Атрибут title** — возвращает заголовок текущей страницы. Например:

```
print(driver.title)
```

5. **Атрибут current_url** — возвращает URL текущей страницы. Например:

```
print(driver.current_url)
```

Пример навигации в интернете с помощью Selenium — пользователь хочет автоматизировать процесс поиска товара в онлайн-магазине. Попробуем реализовать простой сценарий.

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

driver = webdriver.Chrome()
driver.get("https://www.amazon.com")

search_box = driver.find_element(By.ID, 'twotabsearchtextbox')
search_box.send_keys("laptops")
search_box.submit()

assert "laptops" in driver.title
```

В примере мы сначала переходим на домашнюю страницу Amazon с помощью метода get. Затем с помощью метода find_element находим на странице поле поиска и вводим в него поисковый запрос — laptops.

Далее мы запускаем поиск с помощью метода submit для элемента окна поиска. Наконец, проверяем, что страница с результатами поиска загрузилась, проверяя появление слова «laptops» в заголовке страницы.

Скрейпинг данных с помощью Selenium

В Selenium есть несколько методов для извлечения данных из веб-элементов: text, get_attribute и другие.

Метод text поможет извлечь текстовое содержимое. Например, если у вас есть элемент div с текстом «Hello, World!», его можно извлечь.

Метод get_attribute поможет получить значение определенного атрибута элемента.

Для примера рассмотрим HTML-код:

```
<html>
 <body>
   <div id="my-div">Hello World!!!v</div>
   <input type="text" name="username" value="johndoe">
 </body>
</html>
```

Можно извлечь текст из элемента div и значение поля ввода «имя пользователя» с помощью кода:

```
driver = webdriver.Chrome()
driver.get("https://www.example.com")

div_element = driver.find_element(By.ID, "my-div")
print(div_element.text)

username_element = driver.find_element(By.NAME, "username")
print(username_element.get_attribute("value"))
```

В примере мы сначала создаем экземпляр драйвера Chrome и переходим на веб-страницу. Затем используем методы find_element, чтобы найти элемент my-div и поле ввода с именем username соответственно. Затем используем атрибут text и метод get_attribute, чтобы извлечь текст и значения поля ввода соответственно.

Также можно извлекать данные из таблиц и списков на веб-странице, находя элементы таблицы или списка, а затем итеративно просматривая отдельные ячейки или элементы списка для извлечения данных.

Пример того, как с помощью Selenium можно выполнить скрейпинг 100 лучших фильмов IMDb:

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

driver = webdriver.Chrome()

driver.get("https://www.imdb.com/chart/top")

movie_title_elements = driver.find_elements(By.CSS_SELECTOR,
"td.titleColumn a")
rating_elements = driver.find_elements(By.CSS_SELECTOR,
"td.ratingColumn strong")

titles = [element.text for element in movie_title_elements]
ratings = [element.text for element in rating_elements]

for i in range(10):
   print("Rank {}: {} ({} stars)".format(i + 1, titles[i],
ratings[i]))

driver.quit()
```

В этом примере драйвер Chrome используется для открытия нового окна браузера и перехода на страницу IMDb Top Rated Movies. Метод find_elements выполняет поиск элементов, содержащих названия и рейтинги фильмов. А метод text извлекает информацию из элементов и сохраняет ее в списках названий и оценок.

Строка titles = [element.text for element in movie_title_elements] — это генератор списка в Python, который создает новый список titles, содержащий текст каждого элемента в списке movie_title_elements.

Генератор списка — это лаконичный способ написать цикл for, который создает новый список. В этом случае генератор списка выполняет итерации по каждому элементу в movie_title_elements и извлекает текст из каждого элемента с помощью метода text. Полученный текст затем добавляется в список титров.

Эквивалентный код с использованием цикла for:

```
titles = []
for element in movie_title_elements:
   title = element.text
   titles.append(title)
```

Наконец, первые 10 фильмов и их рейтинги выводятся в консоль.

```
Rank 1: Побег из Шоушенка (9,2 stars)
Rank 2: Крестный отец (9,2 stars)
Rank 3: Темный рыцарь (9,0 stars)
Rank 4: Крестный отец 2 (9,0 stars)
Rank 5: 12 разгневанных мужчин (9,0 stars)
Rank 6: Список Шиндлера (8,9 stars)
Rank 7: Властелин колец: Возвращение короля (8,9 stars)
Rank 8: Криминальное чтиво (8,8 stars)
Rank 9: Властелин колец: Братство кольца (8,8 stars)
Rank 10: Хороший, плохой, злой (8,8 stars)
```

Переход на следующую веб-страницу с помощью Selenium

Пример того, как можно перейти на следующую страницу с помощью Selenium:

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

driver = webdriver.Chrome()

driver.get("https://www.example.com/movies")

next_button_locator = (By.XPATH, '//a[@class="next"]')

current_page = 1

while True:
   print(f"Scraping page {current_page}...")

   try:
       next_button = driver.find_element(*next_button_locator)
       next_button.click()
       current_page += 1
   except NoSuchElementException:
       break

driver.quit()
```

В примере сценарий переходит на сайт, содержащий несколько страниц, а затем использует цикл while для нажатия кнопки Next и скрейпинга данных на каждой странице.

next_button_locator определяется с помощью кортежа, который указывает метод By (в данном случае By.XPATH) и значение локатора (в данном случае выражение XPath //a[@class="next"]).

Блок try-except пытается найти кнопку Next и нажать на нее. Если кнопка Next не найдена, значит, мы достигли последней страницы — сценарий выйдет из цикла.

Рассмотрим использование Selenium для извлечения данных из простого сайта с несколькими страницами quotes.toscrape.com. Напишем сценарий, который использует Selenium WebDriver для навигации по страницам, поиска нужных данных и их извлечения.

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

# Запуск нового экземпляра браузера Chrome
browser = webdriver.Chrome()

# Переход на первую страницу веб-сайта
browser.get("http://quotes.toscrape.com/page/1/")

# Инициализация пустого списка для хранения цитат
quotes = []

while True:
    # Поиск всех цитат на странице с помощью xpath
    quote_elements = browser.find_elements(By.XPATH,
'//div[@class="quote"]')
    # Извлечение текста каждой цитаты
    for quote_element in quote_elements:
        quote = quote_element.find_element(By.XPATH,
'.//span[@class="text"]').text
        author = quote_element.find_element(By.XPATH,
'.//span/small[@class="author"]').text
        quotes.append({"quote": quote, "author": author})

    # Проверка наличия следующей кнопки
    next_button = browser.find_elements(By.XPATH,
'//li[@class="next"]/a')
    if not next_button:
        break

    # Нажатие следующей кнопки
```

```
    next_button[0].click()

    # Ожидание загрузки страницы
    time.sleep(1)

# Закрытие браузера
browser.close()

# Вывод цитат
for quote in quotes:
    print(quote["quote"], "by", quote["author"])
```

Этот код использует Selenium WebDriver для перехода к каждой странице цитат, извлечения текста и автора каждой цитаты и сохранения цитат в списке. Цикл while продолжается до тех пор, пока не останется больше страниц для сканирования, что определяется проверкой того, можно ли найти на странице кнопку Next. Код использует метод .find_elements() для поиска элементов цитаты и свойство .text для извлечения текста цитаты и автора.

Сохранение данных в локальном файле

Сохранение данных в локальном файле — важнейший шаг в веб-скрейпинге. После извлечения информации с веб-страницы нужно сохранить ее для использования и анализа. Selenium предоставляет несколько методов сохранения данных в локальный файл.

Запись в файл CSV. Это распространенный формат для хранения данных в табличной форме. Для записи данных в файл CSV можно использовать библиотеку csv в Python. Например:

```
import csv

with open('movies.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Title", "Year", "Genre"])
    for movie in movies:
        writer.writerow([movie.title, movie.year, movie.genre])
```

Запись в файл JSON. Для записи данных в файл JSON можно использовать библиотеку json в Python. Например:

```
import json

with open('movies.json', 'w') as file:
    json.dump(movies, file)
```

Запись в файл TXT. Это простой формат для хранения данных в виде обычного текста. Например:

```
with open('movies.txt', 'w') as file:
    for movie in movies:
        file.write(f"{movie.title} ({movie.year})
[{movie.genre}]\n")
```

Перепишем код, приведенный выше, для записи данных в файл CSV:

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

# Запуск нового экземпляра браузера Chrome
browser = webdriver.Chrome()

# Переход на первую страницу веб-сайта
browser.get("http://quotes.toscrape.com/page/1/")

# Инициализация пустого списка для хранения цитат
quotes = []

while True:
    # Поиск всех цитат на странице с помощью xpath
    quote_elements = browser.find_elements(By.XPATH,
'//div[@class="quote"]')
    # Извлечение текста каждой цитаты
    for quote_element in quote_elements:
        quote = quote_element.find_element(By.XPATH,
'.//span[@class="text"]').text
        author = quote_element.find_element(By.XPATH,
'.//span/small[@class="author"]').text
        quotes.append({"quote": quote, "author": author})

    # Проверка наличия следующей кнопки
    next_button = browser.find_elements(By.XPATH,
'//li[@class="next"]/a')
```

```
    if not next_button:
        break

    # Нажатие следующей кнопки
    next_button[0].click()

    # Ожидание загрузки страницы
    time.sleep(1)

# Запись данных в файл CSV
with open("quotes.csv", "w", newline="") as file:
    writer = csv.DictWriter(file, fieldnames=["quote", "author"])
    writer.writeheader()
    writer.writerows(quotes)

# Закрытие браузера
browser.close()
```

Продвинутые техники скрейпинга в Selenium

Мы рассмотрели основы использования Selenium для навигации по веб-страницам и извлечения данных. Но при работе с большими и более сложными сайтами базовых инструментов может не хватить.

В этом разделе рассмотрим продвинутые методы: обработку динамического контента, работу с JavaScript и решение проблем CAPTCHA.

Эти методы веб-скрейпинга важны, потому что позволяют эффективно работать с динамическими и сложными современными сайтами. Технологии вроде JavaScript и CAPTCHA усложняют процесс извлечения данных с сайтов традиционными методами скрейпинга, но Selenium позволяет их обойти.

Работа с динамическими веб-страницами

Динамические веб-страницы — это страницы, которые изменяются в ответ на взаимодействие с пользователем и другие переменные (например, обновление данных).

В отличие от статических веб-страниц с фиксированным содержанием и структурой, динамические могут обновлять свое содержимое, изображения и другие элементы в режиме реального времени. Из-за этого традиционные методы сбора данных могут не сработать.

Пример — сайт с прогнозом погоды. На нем может быть динамическая страница, которая обновляет прогноз на основе местоположения пользователя или выбранного города. Информация о погоде может меняться в зависимости от времени суток, дня недели или других факторов.

Динамические веб-страницы создаются с помощью языков программирования — JavaScript, PHP и других. Они взаимодействуют с базами данных и API для получения и обновления информации, что делает содержимое веб-страницы динамичным.

Selenium предлагает несколько методов, которые позволяют взаимодействовать с динамической веб-страницей, как если бы мы были пользователями.

- **Ожидание загрузки элементов страницы.** Мы можем использовать метод WebDriverWait для ожидания загрузки определенных элементов страницы перед сбором данных. Это помогает убедиться, что страница полностью загрузилась, прежде чем мы начнем скрейпинг.

- **Взаимодействие с элементами страницы.** Мы можем использовать Selenium для взаимодействия с элементами страницы — кнопками, ссылками и формами. Это позволяет нам инициировать действия на странице, такие как нажатие на ссылки или заполнение форм.

- **Работа со всплывающими окнами.** Мы можем использовать Selenium для работы со всплывающими и диалоговыми окнами, чтобы взаимодействовать с ними по мере необходимости.

- **Скроллинг.** Мы можем использовать Selenium для прокрутки веб-страницы и загрузки нового содержимого. Это полезно для веб-страниц, которые загружают дополнительное содержимое по мере прокрутки пользователем.

Selenium дает ряд мощных инструментов для скрейпинга динамических веб-страниц. Один из них — возможность выполнять JavaScript в окне браузера. Это полезно для взаимодействия со страницей, изменения содержимого или запуска действий, которые требуют ручного взаимодействия.

Например, если сайт загружает содержимое динамически с помощью JavaScript, традиционные методы скрейпинга не смогут извлечь данные. Но с помощью Selenium мы можем выполнить код JavaScript, который загружает содержимое, что позволит нам извлечь нужные данные.

Пример того, как выполнить JavaScript в Selenium:

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://www.example.com")

# Выполнение JavaScript для взаимодействия со страницей
result = driver.execute_script("return document.title")

print(result)
driver.quit()
```

В примере мы создаем экземпляр webdriver, переходим на сайт, а затем выполняем код JavaScript "return document.title", который возвращает значение заголовка страницы. Результат выполнения сохраняется в переменной result, которую мы можем использовать по мере необходимости.

Другой мощный инструмент — ожидание загрузки элементов на странице. Он полезен для извлечения данных со страниц, которые требуют времени для загрузки.

Некоторые элементы могут загружаться дольше, чем другие, и нам нужно дождаться их полной загрузки, прежде чем пытаться извлечь из них данные. Для ожидания загрузки элементов мы можем использовать класс WebDriverWait в Selenium, который позволяет дождаться наступления определенного условия, прежде чем приступить к работе.

Пример Python-кода, который позволяет дождаться загрузки элемента:

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

driver = webdriver.Chrome()
driver.get("https://quotes.toscrape.com/page/1/")
```

```
# ожидание загрузки элемента
wait = WebDriverWait(driver, 10)
element =
wait.until(EC.presence_of_element_located((By.CSS_SELECTOR,
".quote")))

# извлечение данных из элемента
quote = element.text

driver.quit()
```

В примере мы сначала инициализируем веб-драйвер Chrome и переходим на нужный URL. Затем создаем экземпляр класса WebDriverWait и указываем максимальное время ожидания — 10 секунд. Метод until используется для ожидания, пока нужный элемент не появится на странице. В этом случае мы ищем элемент с CSS-классом quote. Когда элемент присутствует, мы извлекаем текст из элемента и сохраняем его в переменной. Наконец, мы закрываем драйвер.

Взаимодействие с динамическими элементами (выпадающим меню или модальными окнами) с помощью Selenium может быть непростым. Эти элементы часто изменяют структуру страницы и требуют дополнительного взаимодействия с пользователем, чтобы стать видимыми.

Пример Python-кода для взаимодействия с выпадающим меню с помощью Selenium:

```
from selenium import webdriver
from selenium.webdriver.support.ui import Select

driver = webdriver.Chrome()
driver.get("https://www.example.com/dropdown-menu")

# Найдите выпадающее меню и выберите нужную опцию
dropdown = driver.find_element(By.ID, "dropdown-menu")
select = Select(dropdown)
select.select_by_visible_text("Option 2")

driver.quit()
```

В коде мы сначала переходим на целевой сайт, содержащий выпадающее меню. Затем используем метод find_element, чтобы найти выпадающее меню на странице. Далее создаем элемент Select.

Объект Select в коде — это объект Selenium, который позволяет взаимодействовать с выпадающими меню на веб-странице. Объект Select предоставляет методы для выбора опций из выпадающего меню и для проверки выбранной опции. Объект Select инициализирует WebElement, представляющий выпадающее меню, и предоставляет такие методы, как select_by_index, select_by_value и select_by_visible_text для выбора опций из выпадающего меню.

Это позволяет нам взаимодействовать с выпадающим меню, как если бы оно было обычным элементом HTML <select>. Наконец, мы используем метод select_by_visible_text для выбора нужной опции.

Для модальных окон процесс аналогичен. Сначала нужно найти элемент запуска модального окна (например, кнопку, которая открывает модальное окно) и щелкнуть его, чтобы открыть модальное окно. Затем можно взаимодействовать с модальным окном так же, как и с любым другим элементом веб-страницы.

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

driver = webdriver.Chrome()
driver.get("https://www.example.com/modal-window")

# Нахождение триггера модального окна и клик на нем, чтобы
открыть модальное окно
modal_trigger = driver.find_element(By.ID, "modal-trigger")
modal_trigger.click()

# Ожидание отображения модального окна
modal = WebDriverWait(driver,
10).until(EC.presence_of_element_located((By.ID, "modal")))

# Взаимодействовать с модальным окном (например, заполнение
формы)
modal_input = modal.find_element(By.ID, "modal-input")
modal_input.send_keys("Hello, World!")

# Закрытие модального окна
modal_close = modal.find_element(By.ID, "modal-close")
modal_close.click()

# Закрытие браузера
driver.quit()
```

В этом примере мы сначала переходим на сайт, содержащий модальное окно. Затем с помощью метода find_element находим элемент модального окна на странице и щелкаем его, чтобы открыть модальное окно. Затем используем метод WebDriverWait для ожидания отображения модального окна с таймаутом в 10 секунд. После отображения модального окна находим элемент модального ввода с помощью метода find_element и вводим в него текст "Hello, World!". Далее находим кнопку закрытия модального окна и нажимает ее, чтобы закрыть модальное окно. Наконец, закрываем браузер с помощью метода quit.

Перейдем к практике. Возьмем для работы сайт duckduckgo.com. Будем использовать выпадающее меню и модальное окно при скрейпинге.

Код, который можно использовать в качестве отправной точки:

```
# Импорт необходимых библиотек
from selenium import webdriver
from selenium.webdriver.common.by import By
import csv

# Установка веб-драйвера
driver = webdriver.Chrome()

# Переход на веб-сайт DuckDuckGo
driver.get("https://duckduckgo.com/")

# Поиск строки поиска и ввод поискового запроса
search_bar = driver.find_element(By.NAME, "q")
search_bar.send_keys("Selenium books")

# Поиск кнопки поиска и нажатие на нее
search_button = driver.find_element(By.XPATH,
"//input[@type='submit']")
search_button.click()

# Поиск выпадающего меню "Время" и щелчок по нему
time_dropdown = driver.find_element(By.XPATH,
"//*[@id='links_wrapper']/div[1]/div[1]/div/div[3]/a")
time_dropdown.click()

# Поиск опции "За последний месяц" в выпадающем меню времени и
щелчок по ней
time_last_month = driver.find_element(By.XPATH,
"*//a[@data-value='m']")
time_last_month.click()
```

```
more_btn = driver.find_element(By.XPATH,
"//a[@class='result--more__btn btn btn--full']")
more_btn.click()


# Поиск всех результатов на странице
results = driver.find_elements(By.XPATH,
"//div[@class='nrn-react-div']")
result_data = []

# Извлечение заголовка и URL каждого результата
for result in results:
   result_title = result.find_element(By.XPATH,
".//a[@class='eVNpHGjtxRBq_gLOfGDr LQNqh2U1kzYxREs65IJu']").text
   result_url = result.find_element(By.XPATH,
".//a[@class='eVNpHGjtxRBq_gLOfGDr
LQNqh2U1kzYxREs65IJu']").get_attribute("href")

   result_data.append([result_title, result_url])

driver.quit()

# Запись данных в файл CSV
with open("duckduckgo_results.csv", "w", newline="") as file:
   writer = csv.writer(file)
   writer.writerow(["Result Title", "URL"])
   writer.writerows(result_data)
```

Сначала мы импортируем нужные библиотеки, включая webdriver из библиотеки selenium, csv для записи данных в CSV-файл и By из selenium.webdriver.common.by для указания локаторов, используемых для поиска элементов на странице.

С помощью драйвера Chrome создается объект веб-драйвера, который будет использоваться для взаимодействия с сайтом DuckDuckGo. Метод get вызывается на объекте драйвера для перехода на сайт DuckDuckGo. Метод find_element используется, чтобы найти строку поиска на странице с помощью локатора NAME и имени "q". Затем метод send_keys используется для ввода поискового запроса "Selenium books" в строку поиска.

Далее метод find_element используется, чтобы найти кнопку поиска на странице с помощью локатора XPATH и выражения XPath "//input[@type='submit']". Затем используется метод click, чтобы нажать на кнопку поиска.

Затем с помощью метода find_element ищется выпадающее меню времени на странице с помощью локатора XPATH и выражения XPath. Затем используется метод click, чтобы открыть выпадающее меню. Метод find_element снова используется для поиска опции «За последний месяц» в выпадающем меню с помощью локатора XPATH и выражения XPath "*//a[@data-value='m']". Затем метод click используется для выбора опции «За последний месяц».

После нахождения всех результатов на странице с помощью локатора XPATH используется цикл для извлечения заголовка и URL каждого результата. С помощью атрибута text извлекаем текст заголовка, а с помощью метода get_attribute — извлекаем URL с помощью атрибута href. Извлеченные данные хранятся в списке со списками, result_data.

Метод quit вызывается на объекте драйвера, чтобы закрыть веб-драйвер.

Оператор with используется для открытия нового файла CSV с именем duckduckgo_results.csv в режиме записи ("w"). Затем объект csv.writer используется для записи строки заголовка, ["Result Title", "URL"], и полученных данных.

Обход CAPTCHA и обнаружение ботов

CAPTCHA (Completely Automated Public Turing tests to tell Computers and Humans Apart) и механизмы обнаружения ботов — две важные темы, которые нужно учитывать при скрейпинге сайтов. Это меры для предотвращения автоматизированного скрейпинга. Их цель — обеспечить доступ к сайту только для пользователей-людей.

CAPTCHA обычно представлены в виде изображений или головоломок. Они требуют от пользователя выполнить задание, например определить последовательность цифр или символов, чтобы доказать, что он человек. CAPTCHA могут быть сложными или невозможными для ПО, что усложняет скрейпинг.

Механизмы обнаружения ботов — это методы для обнаружения и блокировки автоматизированного скрейпинга. Они могут включать блокировку IP-адресов, блокировку пользовательских агентов и JavaScript-задачи, которые требуют от пользователя выполнить задание, чтобы доказать, что он не бот.

При скрейпинге сайтов важно учитывать CAPTCHA и механизмы обнаружения ботов и применять стратегии, чтобы избежать или обойти их. Рассмотрим несколько распространенных методов обхода.

- **Использование прокси IP-адресов.** Сайты часто блокируют IP-адреса, которые связаны со скрейпингом. Используя прокси IP-адреса, вы можете изменить свой IP-адрес и избежать блокировки.
- **Ротация агентов пользователя.** Сайты могут обнаружить скрейпинговую активность, находя закономерности в строке агента пользователя, которая отправляется с каждым запросом. Ротация позволяет изменить строку агента пользователя и избежать блокировки.
- **Решение CAPTCHA.** CAPTCHA могут быть сложными или невозможными для скрейпингового ПО. Но существуют специальные сервисы, которые помогут решить CAPTCHA: например, Anti-Captcha и DeathByCaptcha.
- **Обход вызовов JavaScript.** Некоторые сайты используют вызовы JavaScript для обнаружения ботов. Отключив JavaScript в веб-драйвере или используя headless-браузер, вы можете обойти эти вызовы.
- **Использование headless-браузеров.** Headless-браузеры не отображают графический интерфейс, поэтому сайтам сложнее обнаружить, что вы используете автоматическое ПО для скрейпинга.
- **Замедление запросов.** Веб-сайты могут блокировать скрейпинг, если запросы отправляются слишком быстро. Замедляя запросы, вы можете избежать срабатывания механизмов обнаружения ботов.

На некоторых сайтах применяется множество мер для предотвращения скрейпинга, так что обойти их сложнее. Можно попробовать использовать несколько методов или применить разные подходы, чтобы получить данные.

К сожалению, мы не можем показать реальный пример скрейпинга информации с сайта, использующего CAPTCHA, или защищенного от обнаружения ботов, потому что это неэтично и не вполне законно. Покажем код, отражающий общий принцип:

```
import time
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
import requests

# Использование headless браузера, чтобы избежать механизмов
обнаружения ботов
options = Options()
options.headless = True
```

```
# Создание экземпляра веб-драйвера
driver = webdriver.Chrome(options=options)

# Загрузка веб-сайта
driver.get("https://example.com/")

# Решение задачи CAPTCHA
captcha_element = driver.find_element(By.ID, "captcha")
captcha_image_src = captcha_element.get_attribute("src")
captcha_image_data = requests.get(captcha_image_src).content

# Использование OCR для извлечения текста из изображения CAPTCHA
captcha_text = "CAPTCHA_SOLUTION"

# Ввод текста CAPTCHA в форму
captcha_input = driver.find_element(By.ID, "captcha_input")
captcha_input.send_keys(captcha_text)

# Отправка формы
submit_button = driver.find_element(By.ID, "submit_button")
submit_button.click()

# Ожидание загрузки страницы перед извлечением данных
time.sleep(3)

# Извлечение данных со страницы
data = driver.find_elements(By.XPATH,
"//div[@class='data-element']")
data_list = []
for item in data:
   data_list.append(item.text)

# Закрытие веб-драйвера
driver.quit()

# Вывод извлеченных данных
print(data_list)
```

В этом коде мы сначала используем headless-браузер, чтобы избежать механизмов обнаружения ботов. Headless-браузер не отображает графический интерфейс, так что сайтам сложнее обнаружить, что вы используете автоматизированное ПО для скрейпинга.

Затем мы загружаем веб-сайт и решаем задачу CAPTCHA. Чтобы решить CAPTCHA, мы извлекаем источник изображения CAPTCHA с помощью метода get_attribute и загружаем изображение с помощью библиотеки requests. В примере мы используем OCR (оптическое распознавание символов) для извлечения текста из изображения CAPTCHA, но для решения CAPTCHA можно использовать сервис Anti-Captcha или DeathByCaptcha.

После решения CAPTCHA мы вводим текст в форму и отправляем ее. Затем мы ждем загрузки страницы, после чего извлекаем данные с помощью метода find_elements. Наконец, мы закрываем веб-драйвер и выводим извлеченные данные.

Заключение

Сегодня мы разобрали основы работы с Selenium — мощным инструментом для автоматизации веб-браузеров и скрейпинга сайтов. Мы обсудили преимущества Selenium, включая возможность взаимодействия с динамическими страницами. Посмотрели, как использовать Selenium для скрейпинга статической и динамической страницы. Обсудили проблемы, связанные с CAPTCHA и механизмами обнаружения ботов, а также существующие техники их обхода.

Лучший способ прокачать свои навыки скрейпинга — это практика. Ищите сайты для скрейпинга и работайте над проектами, чтобы расширить опыт. Используйте Selenium в сочетании с другими инструментами: Beautiful Soup и Scrapy. Все это поможет добиться результатов.

Что можно почитать еще?

1. Документация Selenium with Python
2. Selenium WebDriver API Reference
3. Modern Web Automation With Python and Selenium