Click to prove you're human

Selenium web scraping example

Waiting for elements to load Button clicking and page scrolling. Not only that, but it automatically configures the backend browser for the best browser configurations and determines when the content has fully loaded for the given scrape target! For more on ScrapFly's browser rendering and more, see our official JavaScript rendering documentation. We also reviewed some common performance idioms, such as headless browsing and disabling of image loading. With this complete knowledge, we're ready to scrape complex javascript-powered websites such as twitch.tv! That being said, Selenium is not without its faults, and the biggest issue when it comes to developing web-scrapers using the selenium package is scaling. Check out my list of the top dataset providers! The modern web is becoming increasingly complex and reliant on JavaScript, which makes traditional web scraping difficult. Using web scraping tools — Use dedicated web scraping tools that will help you save time and money. Interested in skipping scraping? Traditional web scrapers in python cannot execute JavaScript, meaning they struggle with dynamic web pages, and this is where Selenium - a browser automation toolkit - comes in handy! Browser automation is frequently used in web scraping to utilize browser rendering power to access dynamic content. Webdriver is the first browser automation protocol designed by the W3C organization, and it's essentially a middleware protocol service that sits between the client and the browser, translating client commands to web browser actions. Websites often load content after a delay or based on user interactions like scrolling or clicking a button. Browser are resource heavy and slow, to add, Selenium doesn't support asynchronous programming which might speed things up like Playwright and Puppeteer does (as we've covered in [Scraping Dynamic Websites Using Browser Automation]) so we at ScrapFly offer a scalable Selenium like javascript rendering service - let's take a quick look! Alternative for Selenium Web Scraping Selenium is a powerful web scraping tool though it has its limitations and this is where Scrapfly can help! Let's take a quick look at how we can replicate our twitch.tv scraper in [ScrapFly's SDK]: from parsel import Selector from scrapfly import ScrapeConfig, ScrapflyClient, ScrapeApiResponse scrapfly = ScrapflyClient(key="YOUR KEY") result = scrapfly.scrape(ScrapeConfig(url=" , render_js=True, # ^ indicate to use browser rendering for this request. By sticking to best practices and really getting the most out of Selenium, I can create strong, reliable scrapers that fit exactly what I need. For this, let's take a look at how we can leverage our web scraping with Selenium project by using the Twitch.tv search bar. In this example, we've used parsel to extract content using XPATH and CSS selectors. asp=True, # ^ bypass common anti web scraping protection services. FAQ To wrap up this guide on web scraping with Python and Selenium, let's take a look at some frequently asked questions: Error: Geckodriver executable needs to be in PATH This error usually means that the geckodriver - Firefox's rendering engine - is not installed on the machine. Handling Dynamic Content One of Selenium's biggest advantages is handling dynamic content. They are still there, but they're not being downloaded and embedded into our viewport - saving us loads of resources and time! Finally, we can retrieve a fully rendered page and start parsing for data. In this section, we've covered the main advanced web scraping with Selenium functions: keyboard inputs, button clicking and javascript execution. For this selenium-wire python package can be used which extends Selenium with request/response capturing capabilities: driver.get(' ') for request in driver.requests: if request.response: print(request.url, request.response.status_code, request.response.headers['Content-Type']) Can Selenium be used with Scrapy? # configure webdriver options = Options() options.headless = True # hide GUI options.add_argument("--window-size=1920,1080") # set window size to native GUI size options.add_argument("start-maximized") # ensure window is full-screen ...),) parsed = [] for item in result.selector.xpath("//div[contains(@class,'tw-tower')]/div[@data-target]"): # ^ ScrapFly offers a shortcut to parsel's Selector object parsed.append({ 'title': item.css('h3::text').get(), 'url': item.css('.tw-link::attr(href)').get(), 'username': item.css('.tw-link::text').get(), 'tags': item.css('.tw-tag ::text').getall(), 'viewers': ''.join(item.css('.tw-media-card-stat::text').re(r'(\d+)')), }) print(json.dumps(parsed, indent=2)) ScrapFly simplifies the whole process to few parameter configurations. Parsing Dynamic Data Our first web scraping with selenium attempts were successful. Here’s how to deal with such scenarios: Example: Scraping Data After Scrolling Some websites load additional content when you scroll down the page. How to select a drop-down value in Selenium? It mimics the actions of a real user interacting with a website, making it an excellent choice for scraping dynamic pages that rely heavily on JavaScript. We reviewed most of the common functions used in scraping, such as navigation, button clicking, text input, waiting for content and custom javascript execution. In this case, our condition is a presence of an element that we select through a CSS selector. Step 1: Import the Required Libraries from selenium import webdriver from selenium.webdriver.common.by import By Step 2: Set Up the WebDriver driver = webdriver.Chrome(executable_path='/path/to/chromedriver') Step 3: Open the Web Page driver.get(" ") Step 4: Interact with the Web Page # Let’s assume we want to scrape all article titles from a blog page titles = driver.find_elements(By.CLASS_NAME, 'article-title') for title in titles: print(title.text) Step 5: Close the Browser driver.quit() This simple script demonstrates how to open a web page, locate elements by their class name, and extract text from them. For example, to scroll to the last product item we'd use the scrollIntoView() javascript function: driver.execute_script(""" let items=document.querySelectorAll('.product-box .product'); items[items.length-1].scrollIntoView(); """) How to capture HTTP requests in Selenium? We've launched an optimized instance of a browser, told it to go to our web page, wait for content to load and return us a rendered document! These basic functions will get you pretty far in web scraping already. How to type specific keyboard keys in Selenium? To send non-character keyboard keys we can use defined constants in the from selenium.webdriver.common.keys import Keys constant. We'll be scraping current streams from art section where users stream their art creation process. WebDrivers are specific to each browser (e.g., ChromeDriver for Google Chrome, GeckoDriver for Firefox). country="US" # ^ applies proxy to request to ensure we are getting results in english language. Advanced Functions For Selenium Web Scraping Selenium is a pretty powerful automation library that is capable of much more than what we've discovered through our twitch.tv example. However, some edge cases might require more advanced automation functionality such as element button clicking, the input of text and custom javascript execution - let's take a look at these. Then, to submit our search, we have an option to either send a literal ENTER key or find search button and click it to submit our search form. In this web scraping with Selenium tutorial, we'll take a look at what Selenium is; its common functions used in web scraping dynamic pages and web applications. Web Scraping with Python and BeautifulSoup In this section, we covered the first basic Selenium web scraper. Web scraping with AI — What’s better than utilizing the power of AI to improve your web scraping operations? Selenium lets me interact with web pages just like a regular user would. It’s especially useful when I need to scrape data from complex websites that other tools can’t handle. For example Keys.ENTER will send the enter key. Selenium was initially a tool created to test a website's behavior, but quickly, the demand of web scraping with Selenium has increased This tool is quite widespread and is capable of automating different browsers like Chrome, Firefox, Opera and even Internet Explorer through middleware controlled called Selenium webdriver. This can overload the server and get your IP banned. Selenium can handle these as well: alert = driver.switch_to.alert alert.accept() Headless Browsing for Faster Scraping Running a browser in headless mode can speed up the scraping process, especially when running scripts on a server. Rotate IPs and User-Agents: For large-scale scraping, consider rotating IP addresses and user-agent strings to reduce the risk of being blocked. Selenium can simulate scrolling, enabling you to scrape all the data, not just what’s initially visible. Alternatively, you can specify the WebDriver’s path directly in your script. To reduce bandwidth usage when scraping using Selenium we can disable loading of images through a preference option: chrome_options = webdriver.ChromeOptions() chrome_options.add_experimental_option(# this will disable image loading "prefs", {"profile.managed_default_content_settings.images": 2}) How to take a screenshot in Selenium? In-Depth Guide. Handling JavaScript: Many modern websites load content dynamically using JavaScript. Screenshots are very useful for debugging headless browser workflows. from selenium.webdriver.support.ui import Select object allows us to select values and execute various actions: from selenium.webdriver.support.ui import Select select = Select(driver.find_element_by_id('dropdown-box')) # select by visible text select.select_by_visible_text('first option') # or by value select.select_by_value('1') The best way to reliably scroll through dynamic pages is to use javascript code execution. I can click buttons, fill out forms, and even handle content that loads after the page has initially loaded. We'll find HTML elements for the search box and search button and send our inputs there: from selenium import webdriver from selenium.webdriver.common.keys import Keys driver = webdriver.Chrome() driver.get(") search_box = driver.find_element_by_css_selector('input[aria-label="Search Input"]') search_box.send_keys('fast painting') # either press the enter key search_box.send_keys(Keys.ENTER) # or click search button search_button = driver.find_element_by_css_selector('button[icon="NavSearch"]') search_button.click() In the code above, we used a CSS selector to find our search box and input some keys. Finally, the last important feature used in web-scraping is JavaScript execution. Use Random Delays: To avoid detection as a bot, use random delays between actions: import random time.sleep(random.uniform(2, 5)) Avoid Overloading the Server: Don’t make too many requests in a short time. The easiest way to explore these basic functions is to experiment with Selenium in an interactive REPL like ipython. Setting Up the WebDriver: After downloading, ensure that the WebDriver is accessible through your system’s PATH. Further, we advise taking a look at avoiding bot detection: How Javascript is Used to Block Web Scrapers? Handle Exceptions Gracefully: Always handle exceptions like timeouts and element not found errors to ensure your script doesn’t crash. To select drop-down values we can take advantage of Selenium's UI utils. We've already briefly covered 3 available tools for running headless browsers Playwright, Puppeteer and Selenium in our overview article How to Scrape Dynamic Websites Using Headless Web Browsers, and in this one, we'll dig a bit deeper into understanding Selenium - the most popular browser automation toolkit out there. We'll cover some general tips and tricks and common challenges and wrap it all up with an example project by scraping twitch.tv Hands on Python Web Scraping Tutorial and Example Project What is Selenium? However, the most reliable one is to check whether an element is present in the page via CSS selectors: from parsel import Selector from selenium import webdriver from selenium.webdriver.common.by import By from selenium.webdriver.chrome.options import Options from selenium.webdriver.support.ui import WebDriverWait from selenium.webdriver.support import expected_conditions as EC import time # configure webdriver options = Options() options.headless = True # hide GUI options.add_argument("--window-size=1920,1080") # set window size to native GUI size options.add_argument("start-maximized") # ensure window is full-screen # configure chrome browser to not load images and javascript chrome_options = webdriver.ChromeOptions() chrome_options.add_experimental_option("prefs", {"profile.managed_default_content_settings.images": 2}) driver = webdriver.Chrome(options=options, chrome_options=chrome_options) driver.get(") # wait for page to load element = WebDriverWait(driver=driver, timeout=5).until(EC.presence_of_element_located((By.CSS_SELECTOR, 'div[data-target=directory-first-item]'))) print(driver.page_source) Here, we are using a special WebDriverWait object which blocks our program until a specific condition is met. driver = webdriver.Chrome(options=options) # ^^^^^^^^^^^^^^^^ driver.get(") Additionally, when scraping we don't need to render images, which is a slow and intensive process. With Selenium, I can mimic real user actions, which makes it a game-changer for anyone diving into data science or web development. In Selenium, we can instruct the Chrome browser to skip image rendering through the chrome_options keyword argument: from selenium import webdriver from selenium.webdriver.chrome.options import Options from selenium.webdriver.support.ui import WebDriverWait # configure webdriver options = Options() options.headless = True # hide GUI options.add_argument("--window-size=1920,1080") # set window size to native GUI size options.add_argument("start-maximized") # ensure window is full-screen ... In Selenium, we can enable it through the options keyword argument: from selenium import webdriver from selenium.webdriver.chrome.options import Options ... driver = webdriver.Chrome(options=options, chrome_options=chrome_options) # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ driver.get(") driver.quit() If we were to set out options.headless setting back to False we'd see that all the pages load without any media images. Additionally, it's often used to avoid web scraper blocking as real browsers tend to blend in with the crowd easier than raw HTTP requests. See this quick demo: Selenium driver in ipython demonstrationTo further learn about Selenium for web scraping let's start with our example project. Selenium webdriver translates our python client's commands to something a web browser can understand Currently, it's one of two available protocols for web browser automation (the other being Chrome Devtools Protocol) and while it's an older protocol it's still capable and perfectly viable for web scraping - let's take a look at what can it do! How to Install Selenium Selenium webdriver for python can be installed using the pip command: $ pip install selenium However, we also need webdriver-enabled browsers. Conclusion Web scraping with Selenium gives me the power to pull data from complex and dynamic websites. Setting Up Selenium Before diving into examples, you need to set up Selenium in your Python environment. Selenium is an open-source automation tool primarily used for testing web applications. For this, we can use the headless mode, which strips the browser of all GUI elements and lets it run silently in the background. There are many ways to scroll content in Selenium controlled web browser, but using the scrollIntoView() method is one of the most reliable ways to navigate the browser's viewport. It's much more efficient to pick up the HTML source of the rendered page and use parsel or beautifulsoup packages to parse this content in a more efficient and pythonic fashion. With this content at hand, we can level-up our project and parse related dynamic data from the HTML: from parsel import Selector sel = Selector(text=driver.page_source) parsed = [] for item in sel.xpath("//div[contains(@class,'tw-tower')]/div[@data-target]"): parsed.append({ 'title': item.css('h3::text').get(), 'url': item.css('.tw-link::attr(href)').get(), 'username': item.css('.tw-link::text').get(), 'tags': item.css('.tw-tag ::text').getall(), 'viewers': ''.join(item.css('.tw-media-card-stat::text').re(r'(\d+)')), }) While selenium offer parsing capabilities of its own, they are sub-par to what's available in python's ecosystem. Unlike static HTML pages, where data can be easily retrieved using traditional scraping methods like BeautifulSoup or Scrapy, dynamic pages require a more robust solution to render and interact with the content — this is Selenium’s strength. Read more here. User Interaction Simulation: Selenium can simulate user interactions like clicking buttons, filling forms, and scrolling pages. Our driver is able to deliver us the content of the current browser window (called page source) through driver.page_source attribute but if we call it too early we'll get an almost empty page as nothing has loaded yet! Fortunately, Selenium has many ways of checking whether the page has loaded. Traditional scraping tools often fail here because they only retrieve the initial HTML. Check out these open source attempts scrapy-selenium and scrapy-headless. Scrapy is a popular web scraping framework in Python however because of differing architectures making scrapy and selenium work together is tough. We'll be collecting dynamic data like stream name, viewer count and author. Selenium essentially provides us with a full, running Javascript Interpreter which allows us to fully control the page document and a big chunk of the browser itself! To illustrate this, let's take a look at scrolling. You can see the official release page for download instructions Alternatively, we can use any other Firefox instance by changing executable_path argument in the webdriver initiation, e.g: webdriver.Firefox(executable_path=r'your\path\geckodriver.exe') How to disable loading of images? We've started a browser, told it to go to twitch.tv and wait for the page to load and retrieve the page contents. In this short Python with Selenium tutorial, we took a look at how we can use this web browser automation package for web scraping. Basic Web Scraping Example Now, let’s dive into a basic example where we’ll scrape some data from a website using Selenium. Here’s a quick guide: Install Selenium: pip install selenium Download a WebDriver: Selenium requires a WebDriver to interact with browsers. However, often when web scraping we don't want to have our screen be taken up with all the GUI elements. This is especially useful for running automated scraping scripts in production environments. Here’s how to set it up: from selenium.webdriver.chrome.options import Options options = Options() options.headless = True driver = webdriver.Chrome(executable_path='/path/to/chromedriver', options=options) Best Practices for Web Scraping with Selenium While Selenium is a powerful tool, it’s important to follow best practices to avoid issues: Respect Website’s Robots.txt: Before scraping, check the website’s robots.txt file to ensure you’re not violating their policies. Web scraping with Node.js — One of the easiest ways to scrape websites, read more here. Here’s an example where we simulate a form submission: username = driver.find_element(By.NAME, 'username') password = driver.find_element(By.NAME, 'password') submit_button = driver.find_element(By.ID, 'submit') username.send_keys("myUsername") password.send_keys("myPassword") submit_button.click() time.sleep(3) result = driver.find_element(By.ID, 'result') print(result.text) Dealing with Pop-ups and Alerts Web pages often contain pop-ups or alerts that can interfere with your scraping. Our current task is to: Before we begin let's install Selenium itself: To start with our we scraping with Selenium code, we'll create a selenium webdriver object and launch a Chrome browser: from selenium import webdriver driver = webdriver.Chrome() driver.get(") If we run this script, we'll see a browser window open up and take us our twitch URL. This knowledge should help you get started with Selenium web scraping. To take screenshots we can use webdriver commands: webdriver.save_screenshot() and webdriver.get_screenshot_as_file(). Navigating, Waiting and Retrieving In Selenium When it comes to web scraping, we essentially need a few basic functionalities of Selenium API: Web pages navigation. Headless Browsing: Selenium supports headless browsing, which means you can run the browser without a graphical user interface (GUI). Selenium, however, can execute JavaScript, allowing you to scrape data that appears only after the page has fully loaded. This is crucial for scraping data that requires such interactions, like loading additional content through infinite scroll. Since Twitch is using so-called "endless pagination" to get results from the 2nd page, we must instruct our browser to scroll to the bottom to trigger the loading of the next page: from selenium import webdriver driver = webdriver.Chrome() driver.get(") # find last item and scroll to it driver.execute_script(""" let items=document.querySelectorAll('.tw-tower>div'); items[items.length-1].scrollIntoView(); """) In this example, we used the JavaScript execution to find all web elements in the page that represent videos and then scroll the view to the last element, which tells the page to generate the second page of results. When the web browser connects to a web page it performs many HTTP requests from the document itself to image and data requests. For starters, sometimes we might need to click buttons and input text into forms to access content we want to web scrape. Best Alternatives to Selenium Web scraping with APIs — Using APIs for web scraping can save a lot of time and resources, read more here. It’s a bit tricky to learn than some other tools, but the payoff is huge. We recommend either Firefox and Chrome browsers: ChromeDriver for Chrome driver Geckodriver for Firefox driver Foor more details on how to install Selenium with Python, refer to the official selenium installation instrucitons. from selenium.webdriver.common.keys import Keys driver.find_element(By.TAG_NAME, 'body').send_keys(Keys.END) import time time.sleep(2) new_content = driver.find_elements(By.CLASS_NAME, 'new-content-class') for item in new_content: print(item.text) Handling Form Submissions and Button Clicks Selenium allows you to interact with various elements on the page, such as forms and buttons. Why Use Selenium for Web Scraping? # configure chrome browser to not load images and javascript chrome_options = webdriver.ChromeOptions() chrome_options.add_experimental_option(# this will disable image loading "prefs", {"profile.managed_default_content_settings.images": 2}) ...