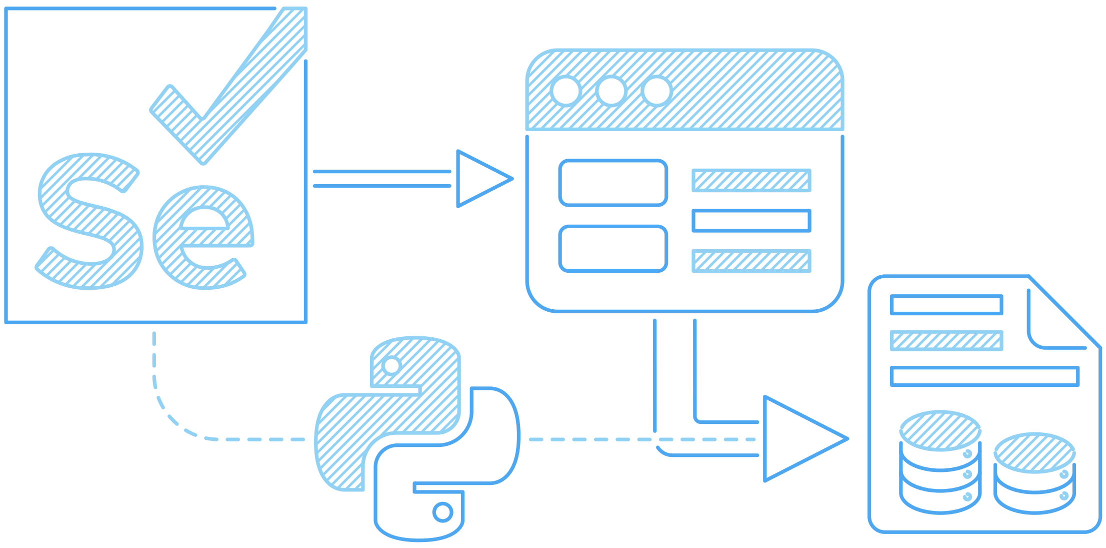
Web Scraping with Selenium and Python Tutorial + Example Project

Python Selenium Headless Browsers

Jan 10, 2022 (Updated A Month Ago)



The modern web is becoming increasingly complex and reliant on JavaScript, which makes traditional web scraping difficult. 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!

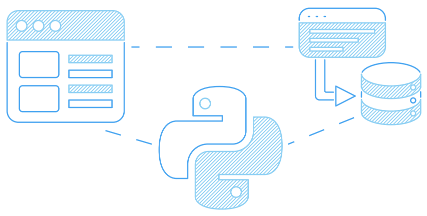
Browser automation is frequently used in web scraping to utilize browser rendering power to access dynamic content. Additionally, it's often used to avoid web scraper blocking as real browsers tend to blend in with the crowd easier than raw HTTP requests.

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.

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. 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

For a general introduction to web scraping in Python see our extensive introduction tutorial, which is focused on using HTTP clients rather than web browsers.



SCRAPFLY ACADEMY

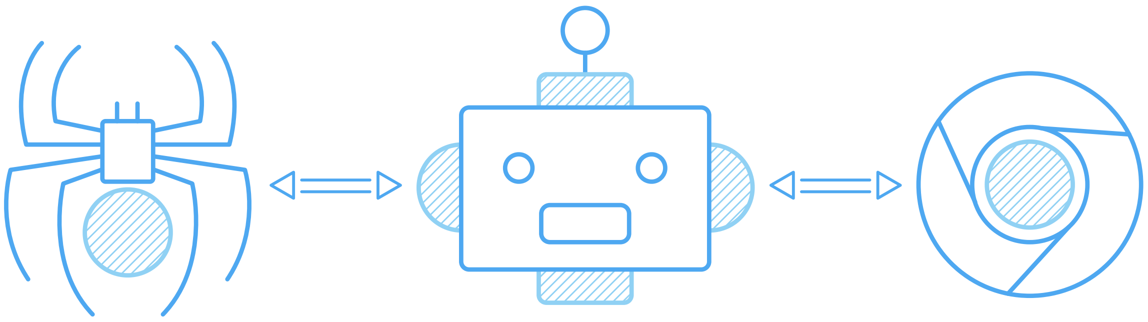
Learn more web scraping ☝️

What is Selenium?

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.

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.



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. 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.

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.
- Waiting for elements to load
- Button clicking and page scrolling.

The easiest way to explore these basic functions is to experiment with Selenium in an interactive REPL like `ipython`. See this quick demo:

0:00 / 0:44 1×

Selenium driver in ipython demonstration

To further learn about Selenium for web scraping let's start with our example project.
We'll be scraping current streams from https://www.twitch.tv/ art section where users stream their art creation process. We'll be collecting dynamic data like stream name, viewer count and author.

Our current task is to:

1. Start a Chrome web browser
2. Go to https://www.twitch.tv/directory/game/Art
3. Wait for the page to finish loading

4. Pick up HTML content of the current browser instance
5. Parse data from the HTML content

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("https://www.twitch.tv/directory/game/Art")
```

If we run this script, we'll see a browser window open up and take us our twitch URL. However, often when web scraping we don't want to have our screen be taken up with all the GUI elements. For this, we can use the **headless mode**, which strips the browser of all GUI elements and lets it run silently in the background. In Selenium, we can enable it through the `options` keyword argument:

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

# configure webdriver
options = Options()
options.headless = True  # hide GUI
options.add_argument("--window-size=1920,1080")  # set window size to native GUI si
options.add_argument("start-maximized")  # ensure window is full-screen

...
driver = webdriver.Chrome(options=options)
#                         ^^^^^^^^^^^^^^^
driver.get("https://www.twitch.tv/directory/game/Art")
```

Additionally, when scraping we don't need to render images, which is a slow and intensive process. 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 si
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(
    # this will disable image loading
    "prefs", {"profile.managed_default_content_settings.images": 2}
)
...

driver = webdriver.Chrome(options=options, chrome_options=chrome_options)
#                                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
driver.get("https://www.twitch.tv/directory/game/Art")
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. 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. 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. 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 si
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("https://www.twitch.tv/directory/game/Art")
# wait for page to load
element = WebDriverWait(driver=driver, timeout=5).until(
    EC.presence_of_element_located((By.CSS_SELECTOR, 'div[data-target=directory-fir
)
print(driver.page_source)
```

Here, we are using a special `WebDriverWait` object which blocks our program until a specific condition is met. In this case, our condition is a presence of an element that we select through a CSS selector.

Parsing Dynamic Data

Our first web scraping with selenium attempts were successful. We've started a browser, told it to go to twitch.tv and wait for the page to load and retrieve the page contents.

With this content at hand, we can level-up our project and parse related dynamic data from the HTML:

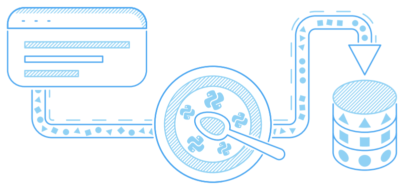
```
from parsel import Selector

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. 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. In this example, we've used parsel to extract content using XPATH and CSS selectors.

Web Scraping with Python and BeautifulSoup

For parsing with BeautifulSoup, see our in-depth article which covers introduction, tips and tricks and best practices



In this section, we covered the first basic Selenium web scraper. 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. 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.

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.

For starters, sometimes we might need to **click buttons** and **input text** into forms to access content we want to web scrape. 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.

We'll find HTML elements for the search box and search button and send our inputs there:

```python
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome()
driver.get("https://www.twitch.tv/")
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. 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.

Finally, the last important feature used in web-scraping is **JavaScript execution**. 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.
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:

```python
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://www.twitch.tv/directory/game/Art")
# 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.
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.

In this section, we've covered the main advanced web scraping with Selenium functions: keyboard inputs, button clicking and javascript execution. 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. 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

ScrapFly.io implements core web browser automation functions: page rendering, session/proxy management, custom javascript evaluation and page loading rules - all of which help create a highly scalable and easy-to-manage web scraper.

> One important feature of the ScrapFly API is the seamless mixing of browser rendering and traditional HTTP requests - allowing developers to optimize scrapers to their full scraping potential.

Let's quickly take a 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="https://www.twitch.tv/directory/game/Art",
        render_js=True,
        # ^ indicate to use browser rendering for this request.
        country="US"
        # ^ applies proxy to request to ensure we are getting results in english la
        asp=True,
        # ^ bypass common anti web scraping protection services.
    ),
)

parsed = []
for item in result.selector.xpath("//div[contains(@class,'tw-tower')]/div[@data-tar
#                  ^ 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))
```

▶ in ScrapFly player

ScrapFly simplifies the whole process to few parameter configurations. 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.

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. 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?

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?

To take screenshots we can use webdriver commands: `webdriver.save_screenshot()` and `webdriver.get_screenshot_as_file()`. Screenshots are very useful for debugging headless browser workflows.

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. For example `Keys.ENTER` will send the enter key.

How to select a drop-down value in Selenium?

To select drop-down values we can take advantage of Selenium's UI utils. `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')
```

How to scroll Selenium browser to a specific object?

The best way to reliably scroll through dynamic pages is to use javascript code execution. 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?

When the web browser connects to a web page it performs many HTTP requests from the document itself to image and data requests. For this selenium-wire python package can be used which extends Selenium with request/response capturing capabilities:

```
driver.get('https://www.google.com')
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?

Scrapy is a popular web scraping framework in Python however because of differing architectures making scrapy and selenium work together is tough. Check out these open source attempts scrapy-selenium and scrapy-headless.

Web Scraping With Python and Selenium Summary

In this short Python with Selenium tutorial, we took a look at how we can use this web browser automation package for web scraping.
We reviewed most of the common functions used in scraping, such as navigation, button clicking, text input, waiting for content and custom javascript execution.
We also reviewed some common performance idioms, such as headless browsing and disabling of image loading.

This knowledge should help you get started with Selenium web scraping. Further, we advise taking a look at avoiding bot detection: How Javascript is Used to Block Web Scrapers? In-Depth Guide.

Check out ScrapFly Python SDK

Try ScrapFly for FREE!

Related Questions

- What Python libraries support HTTP2?
- How to scrape HTML table to Excel Spreadsheet (.xlsx)?
- How to use proxies with Python httpx?
- Python httpx vs requests vs aiohttp - key differences
- How to click on a pop up dialog in Playwright?
- How to scrape images from a website?

How to select dictionary key recursively in Python?
What are some ways to parse JSON datasets in Python?
Selenium: chromedriver executable needs to be in PATH?
How to fix python requests ConnectTimeout error?
How to fix Python requests MissingSchema error?
How to check if element exists in Playwright?
How to use cURL in Python?
Selenium: geckodriver executable needs to be in PATH?
How to fix Python requests ReadTimeout error?

More >

PYTHON SELENIUM HEADLESS BROWSERS

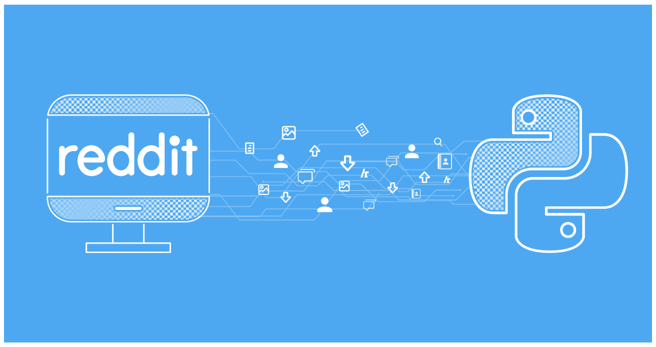
Related Posts

May 03, 2024

How to Build a Minimum Advertised Price (MAP) Monitoring Tool

Learn what minimum advertised price monitoring is and how to apply its concept using Python web scraping.

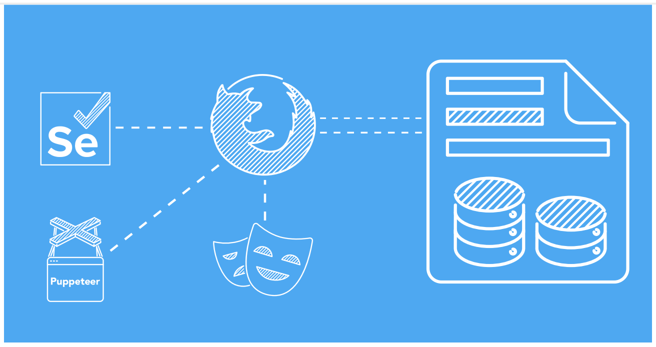
PROJECT PYTHON



Apr 22, 2024

How to Scrape Reddit Posts, Subreddits and Profiles

In this article, we'll explore how to scrape Reddit. We'll extract various social data types from subreddits, posts, and user pages. All of which through plain HTTP requests without



Apr 17, 2024

How to Scrape With Headless Firefox

Discover how to use headless Firefox with Selenium, Playwright, and Puppeteer for web scraping, including practical examples for each library.

HEADLESS BROWSERS PUPPETEER SELENIUM NODEJS PLAYWRIGHT PYTHON

Company