



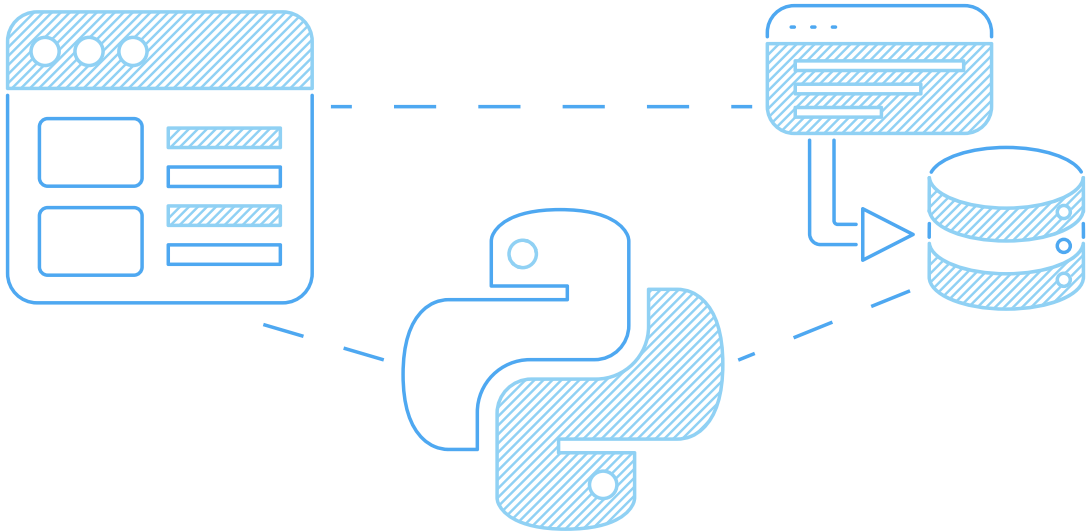


Everything to Know to Start Web Scraping in Python Today

by Bernardas Ališauskas Dec 02, 2024 #Python    



Web scraping with Python is a massive subject and this guide will introduce you to all main contemporary concepts and techniques. From how to web scrape basic HTML to scraping dynamic pages with headless browsers and AI — we'll cover it all!

We'll start from the *very basics* and cover **every relevant major technique and feature** with some real life examples. Finally, we'll wrap up with a full scraper breakdown and some tips on scraper scaling, scraper blocking and project deployment.

What is Web Scraping?

Web Scraping Overview

Basic Web Scraping Example

Step 1: Scraping The Data

Scraping HTML Pages

Scraping JavaScript Rendered HTML

Scrapfly Cloud Browsers

Capturing Browser Data

Capturing Screenshots

Scrapfly Screenshot API

Step 2: Parsing Scraped Data

Parsing Scraped HTML

Scraping Hidden Dataset

Parsing Scraped JSON

Scraping with AI and LLMs

Extracting Data with Extraction API

Step 3: Scaling Up

Scaling and Solving Challenges

Scraper Blocking

Power Up with Scrapfly

Existing Scrapers

Deploying Python Scrapers

Scraping Frameworks

Summary

[JOIN THE NEWSLETTER](#)

Get monthly web scraping insights 👉

[Learn at ScrapFly Academy](#)

What is Web Scraping?

To start, let's clarify what is web scraping and the parts that make it.

Web scraping is an automated web data retrieval and extraction process. It's an invaluable tool for collecting data from websites at scale to power applications, analytics, and research. Web scraping can be used to extract product prices, news articles, social media posts, and much more.

The web is full of public data and why not take advantage of it? Using Python we can scrape any page given the right techniques and tools so to start let's take a look at the main parts of web scraping.

Web Scraping Overview

Generally modern web scraping in Python development process looks something like this:

1. Retrieve web page data:
 1. Use one or multiple HTTP requests to reach the target page
 2. Or use a headless browser to render the page and capture the data
2. Parse the data:
 1. Extract the desired data from the HTML
 2. Or use AI and LLMs to interpret scraped data
3. Scale up and crawl:
 1. Fortify scrapers against blocking
 2. Distribute tasks concurrently or in parallel

There are a lot of small challenges and important details here that start to emerge as you dive deeper into web scraping though before we dig into those, let's start with a basic example that'll help us glimpse the entire process.

Basic Web Scraping Example

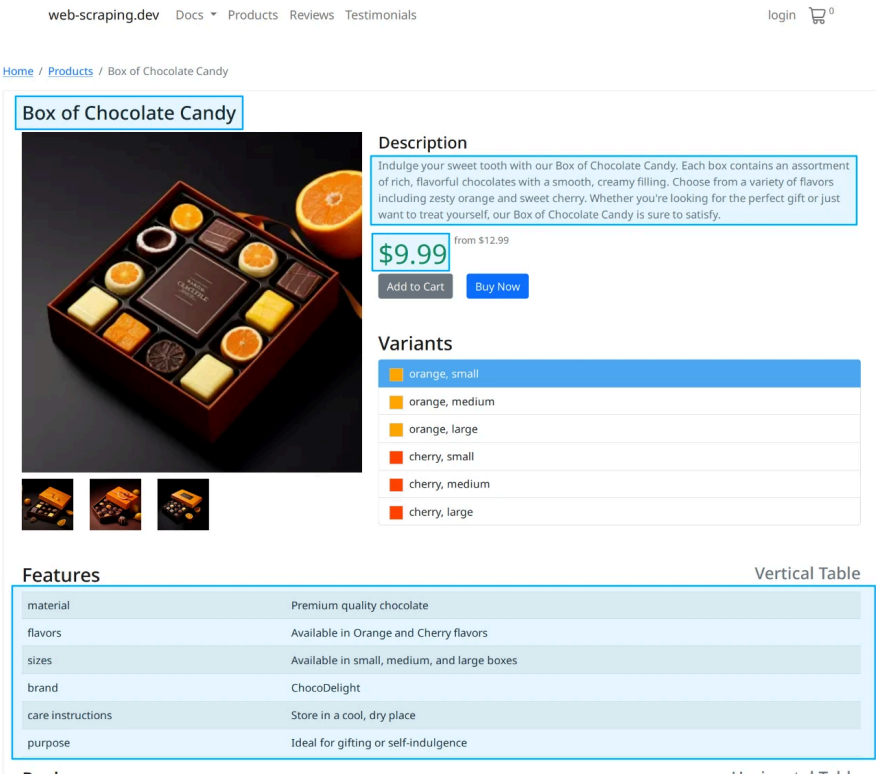
The most basic web scraping example needs:

- HTTP client library - [httpx](#) is a modern and fast HTTP client for Python.
- Data parsing library - [Beautifulsoup](#) is the most popular HTML parsing library.

To start, install those packages using `pip` terminal command:

```
$ pip install httpx beautifulsoup4
```

For our Python scraper example let's use <https://web-scraping.dev/product/1> page which is designed for testing and developing web scrapers. For this let's scrape the page and extract product name, description, price and features:



Which we can scrape with the following Python code:

```
import httpx
import bs4 # BeautifulSoup

# scrape page HTML
url = "https://web-scraping.dev/product/1"
response = httpx.get(url)
assert response.status_code == 200, f"Failed to fetch {url}, got {response.status_code}"

# parse HTML
soup = bs4.BeautifulSoup(response.text, "html.parser")
product = {}
product['name'] = soup.select_one("h3.product-title").text
product['price'] = soup.select_one("span.product-price").text
product['description'] = soup.select_one("p.product-description").text
product['features'] = {}
feature_tables = soup.select(".product-features table")
for row in feature_tables[0].select("tbody tr"):
    key, value = row.select("td")
    product['features'][key.text] = value.text

# show results
from pprint import pprint
print("scraped product:")
pprint(product)
{'description': 'Indulge your sweet tooth with our Box of Chocolate Candy. '
                'Each box contains an assortment of rich, flavorful chocolates '
                'with a smooth, creamy filling. Choose from a variety of '
                'flavors including zesty orange and sweet cherry. Whether '
                '"you're looking for the perfect gift or just want to treat "
                'yourself, our Box of Chocolate Candy is sure to satisfy.',
 'features': {'brand': 'ChocoDelight',
              'care instructions': 'Store in a cool, dry place',
              'flavors': 'Available in Orange and Cherry flavors',
              'material': 'Premium quality chocolate',
              'purpose': 'Ideal for gifting or self-indulgence',
              'sizes': 'Available in small, medium, and large boxes'},
 'name': 'Box of Chocolate Candy',
 'price': '$9.99 '}
```

This basic scraper example uses our [httpx](#) HTTP client to retrieve the page HTML and BeautifulSoup and CSS selectors to extract data. Returning a python dictionary with product details.

This is a very basic single page scraper example and to fully understand web scraping in Python, let's break each individual step down and dive deeper into python web scraping next.

Step 1: Scraping The Data

Generally there are two ways to retrieve web page data in web scraping:

1. Use a **fast HTTP client** to retrieve the HTML
2. Use a **slow real web browser** to render the page, capture the browser data and page HTML

[HTTP clients](#) are very fast and resource-efficient. However, many modern pages are not only compromised of HTML but javascript and complex loading operations where HTTP client is often not enough. In those cases using a real web browser is required to load the page fully.

[Web browsers](#) can render the page and capture the data as a human would see it and the web browser can be automated through Python. This is called a "headless browser" automation (headless meaning no GUI). Headless browsers can be used to scrape data from pages that require javascript rendering or complex page interactions like clicking buttons, inputs or scrolling.

Both of these methods are used in web scraping and the choice depends on the target page and the data you need. Here are some examples:

- [web-scraping.dev/product/1](#) page is a static HTML page and we can easily scrape it using an HTTP client.
- [web-scraping.dev/reviews](#) page is a dynamic page that loads reviews using javascript. We need a headless browser to scrape this page.

If you're unfamiliar with web basic like what is URL, HTML, request etc. then take a brief look at this short Scrapfly Academy lesson before continuing:

► [Related Lesson: Basic Website Reverse Engineering](#)

Let's take a look at each of these 2 web data retrieval methods in Python.

Scraping HTML Pages

Most web pages are pretty simple and contain static HTML that can be easily scraped using an HTTP client.

To confirm whether a page is static or dynamic you can *disable Javascript in your browser and see whether the results remain in the page display or page source*. If the data is in the page source then it's static and can be scraped with an HTTP client.

To scrape HTML data use a HTTP client like [httpx](#) (recommended) which can send HTTP requests to the web server which in turn return responses that either contain page HTML or a failure indicator.

To start, there are several types of requests identified by **HTTP method**:

Method	Description
GET	Retrieve data. The most common method encountered in web scraping and the most simple one.
POST	Submit data. Used for submitting forms, search queries, login requests etc. in web scraping.
HEAD	Retrieve page metadata (headers) only. Useful for checking if the page has changed since the last visit.
PUT, PATCH, DELETE	Update, modify, and delete data. Rarely used in web scraping.





Here are some examples how to perform each type of web scraping request in popular Python `httpx` client:

GET POST HEAD

```
import httpx

response = httpx.get('https://web-scraping.dev/product/1')
print(response.text) # prints the page HTML
# <!doctype html>
# <html lang="en">
#   <head>
#     <meta charset="utf-8">
#     <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
# ...
print(response.status_code) # prints the response status code, 200 means success!
200
```

As result each response has some key indicators whether our scrape was successful. To start the status code is a numeric code where each number has a specific meaning:

Code Range	Meaning
2xx	 success!
3xx	 redirect to different location. Page has moved or client failed.
4xx	 client error. The request is faulty or being blocked.
5xx	 server error. The server is inaccessible or client is being blocked.

For more on status codes encountered in web scraping see our full breakdown in each of our HTTP status code articles:

Status Code	Brief
403	Forbidden / Blocked
405	Request method not allowed
406	Not Acceptable (<code>Accept-</code> headers)
409	Request Conflict

Status Code	Brief
413	Payload Too Large
412	Precondition Failed (<code>If-</code> headers)
415	Unsupported Media Type (for POST requests)
422	Unprocessable Entity (mostly bad POST request data)
429	Too many Request (Throttling)
502	Service Unavailable

For more on static HTML page scraping see our related lesson on Scrapfly Academy:

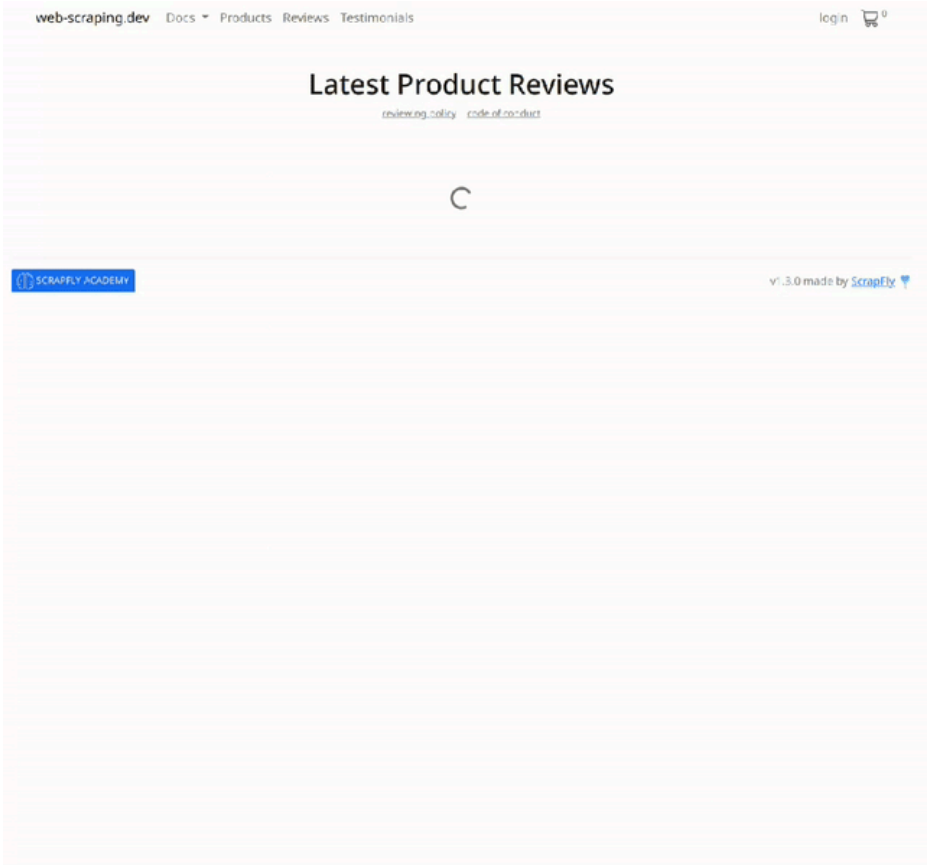
► **Related Lesson: Static HTML Scraping**

Next just scraping page HTML is often not enough for many websites so let's take a look at how a real web browsers can be used in web scraping to scrape dynamic pages.

Scraping JavaScript Rendered HTML

Modern pages are complex and use javascript to generate some page sections on demand. You've probably encountered this when scrolling through page and new content loads or when you click on a button and a new section appears.

An example of such page would be web-scraping.dev/reviews page which loads reviews using javascript:



the loading circle is a common indicator of Javascript loading

To scrape JS rendered pages, we can automate real web browsers like Chrome or Firefox to load the page and return the page contents for us. This is called **headless browser scraping** and headless simply means there is no visible GUI.

Each web browser exposes an automation API called [CDP \(Chrome Devtools Protocol\)](#) which allows applications to connect to a browser and control it. There are several popular libraries that implement this API in Python and click on each of these to see our full introduction guide:

- [Selenium](#) is the most mature and well known library for Python. It has a giant community but can be a bit harder to use due to it's age.
- [Playwright](#) is the newest and most feature rich option in Python developed by Microsoft.

Here are Python selenium web scraping example and Python playwright web scraping example for easier comparison:

Python Selenium	Python Playwright	Scrapfly SDK
<pre># pip install selenium # also install selenium chrome driver: # install the chrome webdriver from https://sites.google.com/a/chromium.org/chromedriver/downloads 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 # start a configured Chrome browser 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 driver = webdriver.Chrome(options=options) # scrape the reviews page driver.get("https://web-scraping.dev/reviews") # wait up to 5 seconds for element to appear element = WebDriverWait(driver=driver, timeout=5).until(EC.presence_of_element_located((By.CSS_SELECTOR, '.review'))) # retrieve the resulting HTML and parse it for dataao html = driver.page_source print(html) driver.close()</pre>		

This type of scraping is generally called **dynamic page scraping** and while Selenium and Playwright are great solutions there's much more to this topic that we cover in our Scrapfly Academy lesson 📌

► [Related Lesson: Dynamic Page Scraping](#)

As headless browsers run real web browsers scaling browser based web scrapers can be very difficult. For this, many scaling extensions are available like [Selenium Grid](#) for Selenium. Alternatively, you can defer this expensive task to Scrapfly's cloud browsers.

Scrapfly Cloud Browsers

Scrapfly's [Web Scraping API](#) can manage a fleet of cloud headless browsers for you and simplify the entire browser automation process.

Here's an example how to automate a web browser with Scrapfly in Python:

```
# pip install scrapfly-sdk[all]
from scrapfly import ScrapflyClient, ScrapeConfig

client = ScrapflyClient(key="YOUR SCRAPFLY KEY")

# We can use a browser to render the page, screenshot it and return final HTML
api_result = client.scrape(ScrapeConfig(
    "https://web-scraping.dev/reviews",
    # enable browser rendering
    render_js=True,
    # we can wait for specific part to load just like with Playwright:
    wait_for_selector=".reviews",
    # for targets that block scrapers we can enable block bypass:
    asp=True,
    # or fully control the browser through browser actions:
    js_scenario=[
        # input keyboard
        {"fill": {"value": "watercolor", "selector": 'input[autocomplete="twitch-
nav-search"]' }},
        # click on buttons
        {"click": {"selector": 'button[aria-label="Search Button"]' }},
        # wait explicit amount of time
        {"wait_for_navigation": {"timeout": 2000}}
    ]
))
print(api_result.result)
```

[See the Web Scraping API](#)

Capturing Browser Data

There's more to scraping with web browsers than just rendered HTML capture. The browsers are capable of non-html features:

- Local Storage, IndexedDB database storage
- Executing background requests for retrieving more data from the server (like comments, reviews, etc)
- Storing data in Javascript, CSS and other non-html resources

When web scraping with a headless browser we can also capture all of this data which can be crucial for many more complex web scraping processes.

So, if **you see your page missing some data** in the HTML maybe it wasn't there in the first place! Let's see how to capture non-html data in Python and Playwright or Selenium libraries.

Capturing Network Requests

XHR (XMLHttpRequests) are background requests that are made by the page to retrieve more data from the server. These requests are often used to load comments, reviews, graph data and other dynamic page elements.

To start, we can view what XHR requests are being made by the browser using browser developer tools. See this quick video how that is being done using Chrome:

How to find background requests a website is making using Chrome Devtools



To capture XHR in Python we must use a headless browser which can be setup with a background request listener. Here's an example how to capture XHR requests in Python and Playwright or Selenium libraries:

Python Playwright Python Selenium Scrapfly SDK

```
from playwright.sync_api import sync_playwright

def intercept_response(response):
    print(response)
    return response

with sync_playwright() as pw:
    browser = pw.chromium.launch(headless=False)
    context = browser.new_context(viewport={"width": 1920, "height": 1080})
    page = context.new_page()

    # Intercept requestes and responses
    page.on("response", intercept_response)

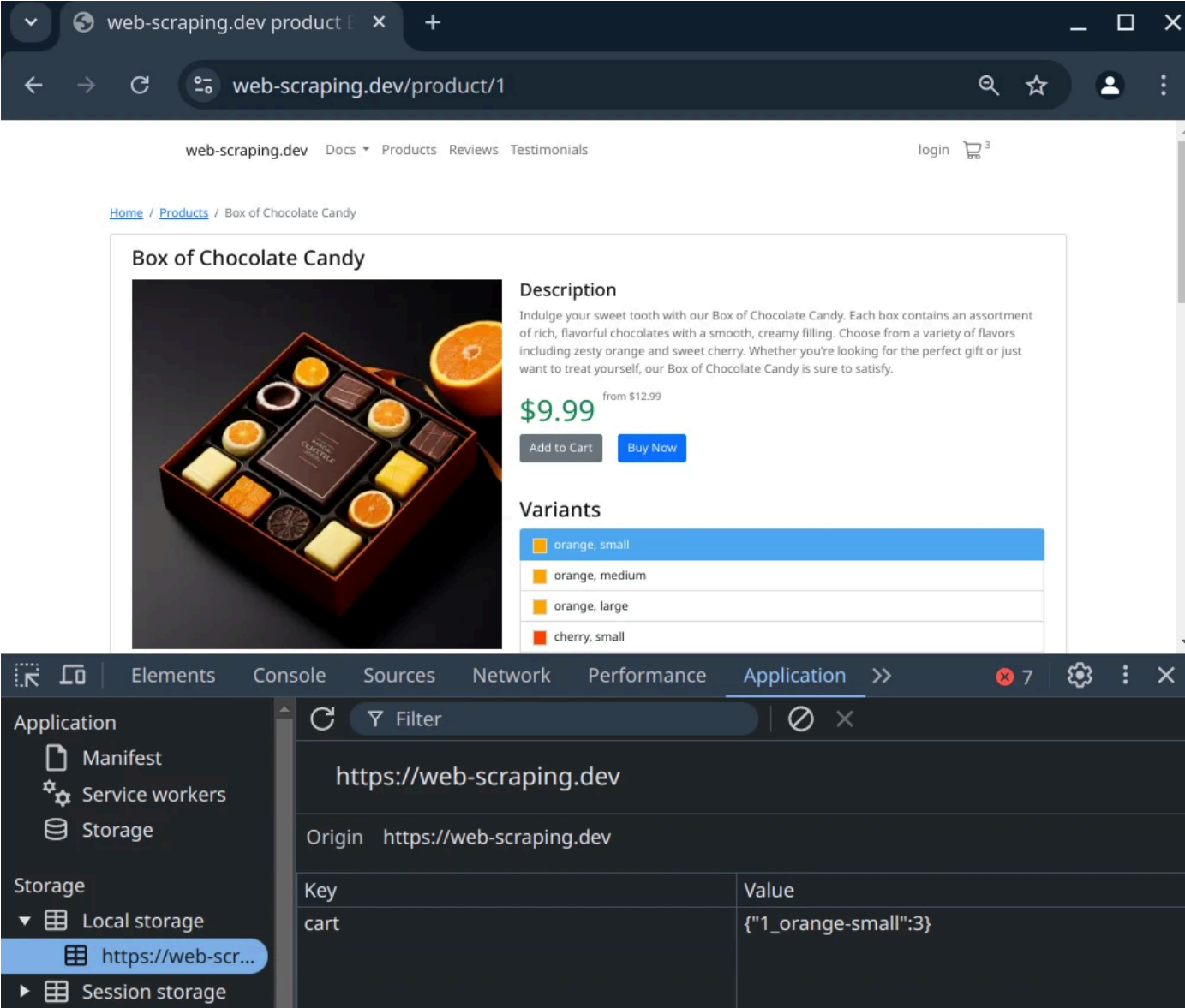
    page.goto("https://web-scraping.dev/reviews")
```

For more see our [complete intro to background request scraping](#).

Capturing Local Storage

Local Storage is a browser feature that allows web pages to store key-value data in the browser itself. This data is not visible in the page HTML and can be used to store user preferences, shopping cart items, and other client-side data.

An example of local storage can be found on [web-scraping.dev/product/1](#) product pages where the page stores car information in local storage:



local storage data as seen in Chrome Devtools when 'add to cart' button is pressed

To capture Local Storage data in Python we must execute Javascript in the browser to retrieve the data. Here's an example how to capture Local Storage data in Python and Playwright or Selenium libraries:

Python Playwright Python Selenium Scrapfly SDK

```
from playwright.sync_api import sync_playwright

def capture_local_storage(page):
    # Retrieve all local storage data
    local_storage_data = page.evaluate "() => Object.entries(localStorage)")
    return {key: value for key, value in local_storage_data}

with sync_playwright() as pw:
    browser = pw.chromium.launch(headless=True) # Headless mode
    context = browser.new_context()
    page = context.new_page()

    # Navigate to the target website
    page.goto("https://example.com")

    # Capture local storage data
    local_storage = capture_local_storage(page)
    print("Local Storage Data:", local_storage)

    # Cleanup
    context.close()
    browser.close()
```

Capturing Screenshots

Screenshots are a great way to capture the entire page state at a specific moment in time. They can be used to verify the page state, debug the scraping process, or even as a part of the scraping process itself.

To capture screenshots in Python and Playwright or Selenium libraries we can use the `screenshot()` method which implement the entire screenshot process:

Python PlaywrightPython SeleniumScrapfly SDK

```
from playwright.sync_api import sync_playwright

with sync_playwright() as pw:
    browser = pw.chromium.launch(headless=True)  # Run in headless mode
    context = browser.new_context()
    page = context.new_page()

    # Navigate to the target URL
    page.goto("https://web-scraping.dev/reviews")

    # Capture a screenshot and save it to a file
    page.screenshot(path="screenshot.png")

    print("Screenshot saved as screenshot.png")

    # Cleanup
    context.close()
    browser.close()
```

Screenshot capture can be surprisingly challenging with page pop-ups, javascript exeuction and blocks. This is where Scrapfly can assist you once again!

Scrapfly Screenshot API

Scrapfly's [Screenshot API](#) uses headless browsers optimized for screenshot capture to make it as simple and seamless as possible:

- Automatically block ads and popups
- Bypass CAPTCHAs and other bot detection mechanisms
- Target specific areas and different resolutions
- Scroll the page and control the entire capture flow

Here's an example of how easy it is to capture screenshot of any page using the Screenshot API:

```
# pip install scrapfly-sdk[all]
from scrapfly import ScrapflyClient, ScreenshotConfig

client = ScrapflyClient(key="your scrapfly key")

api_result = client.screenshot(ScreenshotConfig(
    url="https://web-scraping.dev/reviews",
))
print(api_result.image)  # binary image
print(api_result.metadata)  # json metadata
```

See the Screenshot API

No matter which method you use to scrape websites you'll end up with the entire page dataset which on its own not very useful. Next, let's take a look at how to parse this data and extract the information we need.

Step 2: Parsing Scraped Data

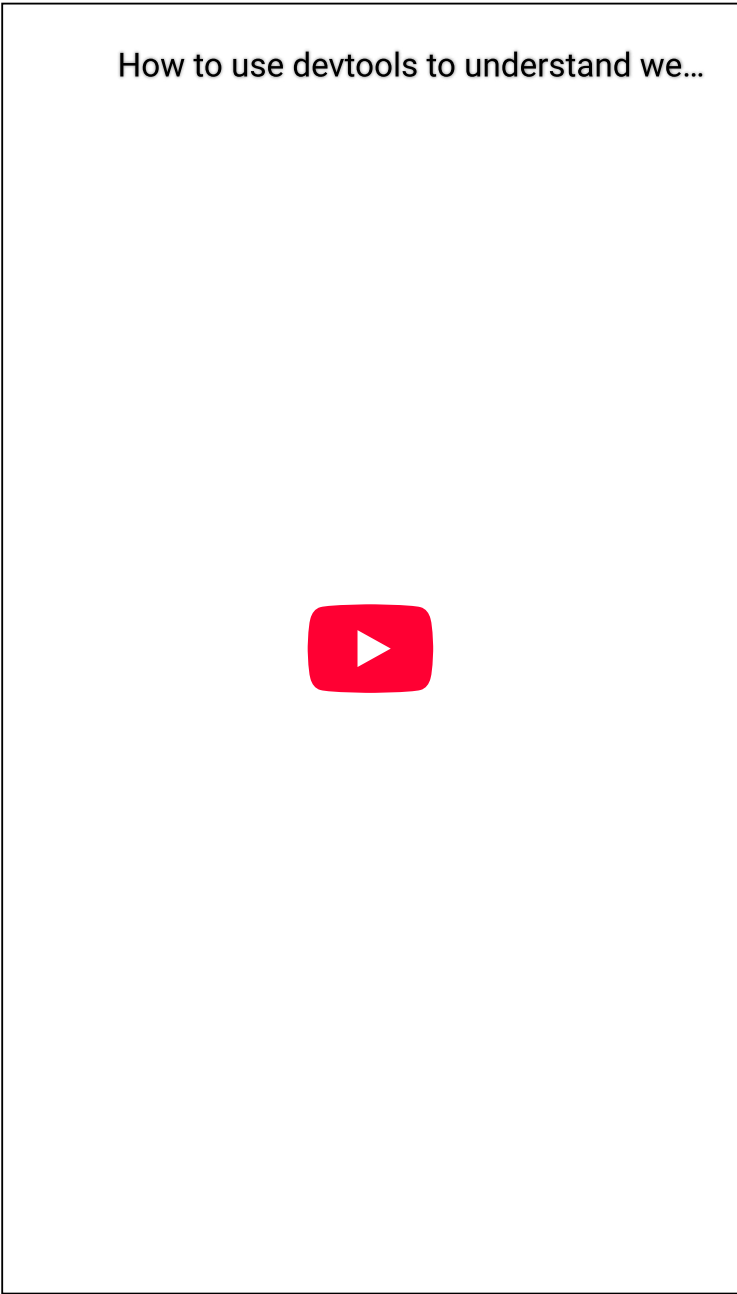
Websites generally return data in one of two formats: HTML for page content and JSON for API responses (like background requests). Each of these formats requires a different approach to extract the data.

Parsing Scraped HTML

To parse HTML there are 3 standard options (click on each for our dedicated tutorial):

- [CSS Selectors](#) - commonly encountered in web development like CSS style application rules and DOM parsing, CSS selectors allow to select HTML elements by attributes like element tag, class, id or location in the document.
- [XPath](#) - a more powerful and flexible option, XPath allows to select HTML elements by their attributes, exact location in the document and even by their text content.
- [Beautifulsoup](#) - offers a Pythonic way to parse HTML using `.find()`, `.find_all()`, `.select()` methods.

To create your parsing rules it's first important to understand the structure of the HTML document. Take a look at this short video on how to explore HTML using Chrome developer tools:



With that, here's an example how to parse product HTML from [web-scraping.dev/product/1](#) with each of these methods:

[BeautifulSoup](#) [CSS Selectors](#) [XPath](#) [Scrapfly SDK](#)

```
import httpx
from bs4 import BeautifulSoup

resp = httpx.get("https://web-scraping.dev/product/1")
soup = BeautifulSoup(resp.text, "html.parser")

product = {}
# use css selectors with beautifulsoup
product["name"] = soup.select_one(".product-title").text
# or use find method
product["price"] = soup.find("span", class_="product-price").text
product["description"] = soup.find("p", class_="product-description").text
print(product)
{
  "name": "Box of Chocolate Candy",
  "price": "$9.99 ",
  "description": "Indulge your sweet tooth with our Box of Chocolate Candy. Each box
contains an assortment of rich, flavorful chocolates with a smooth, creamy filling.
Choose from a variety of flavors including zesty orange and sweet cherry. Whether
you're looking for the perfect gift or just want to treat yourself, our Box of
Chocolate Candy is sure to satisfy.",
}
```

Overall, **CSS selectors as sufficiently powerful** for most web scraping operations though for more complex pages the more advanced features of XPath or BeautifulSoup can be very useful.

Here's a quick overview table comparing XPath and CSS selectors and BeautifulSoup:

Feature	CSS Selectors	XPath	BeautifulSoup
select by class	✓	✓	✓
select by id	✓	✓	✓
select by text	✗	✓	✓
select descendant elements (child etc)	✓	✓	✓
select ancestor element (father etc)	✗	✓	✓
select sibling elements	limited	✓	✓

While both XPath and BeautifulSoup are more powerful than CSS selectors, CSS selectors are very popular in web development and really easy to use. Because of this, most developers tend to use CSS selectors where possible and switch to XPath or BeautifulSoup when CSS selectors are not enough.

Whichever you choose see our handy cheatsheets:

- [CSS Selectors Cheatsheet](#)
- [Xpath Cheatsheet](#)

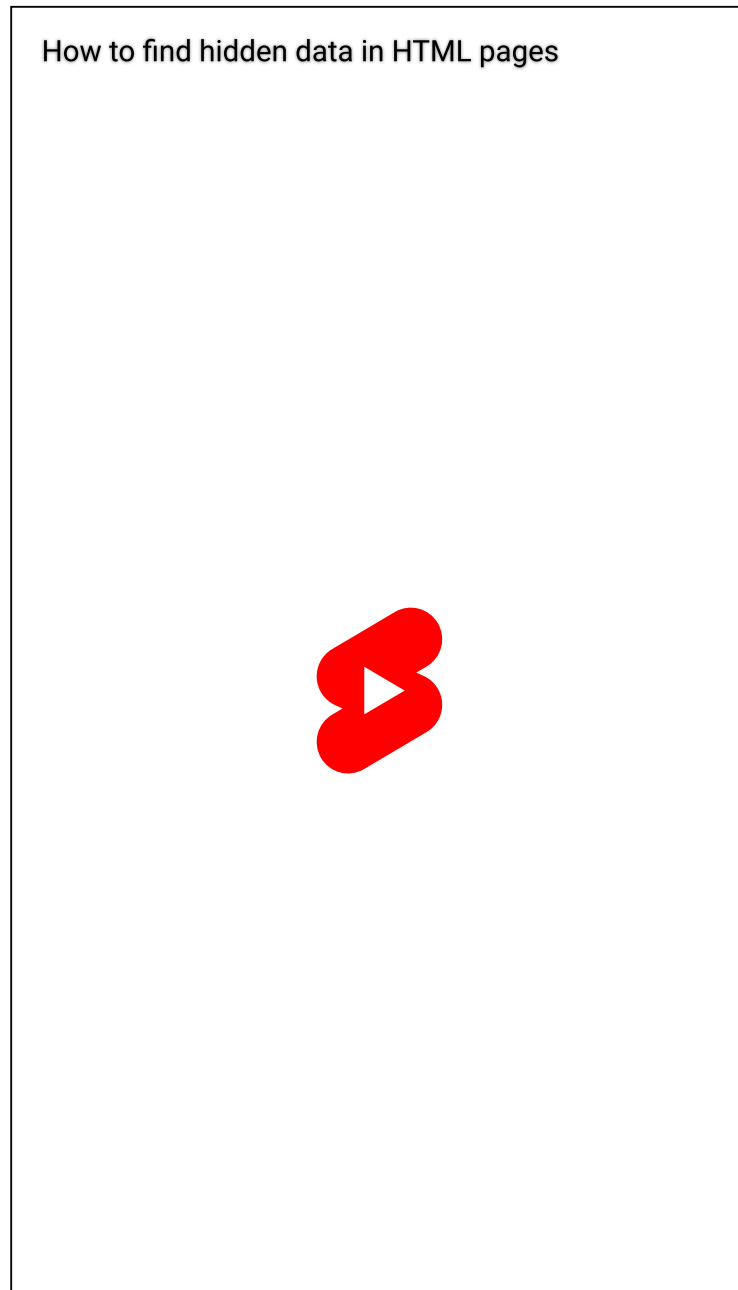
Next, modern websites not always store the data in the HTML element themselves. Often data is stored in variables or scripts and then unpacked to the page HTML on page load by javascript. We can scrape this as well, let's take a look at how next.

Scraping Hidden Dataset

Often we might see HTML parsing techniques return empty results even though we are sure the data is there. This is often caused by hidden data.

[Hidden Web Data](#) is when pages store page data in JavaScript variables or hidden HTML elements and then unpack this data into page HTML using javascript on page load. This is a common web development technique that allows server side and client side development logic separation but can be annoying to deal with in scraping.

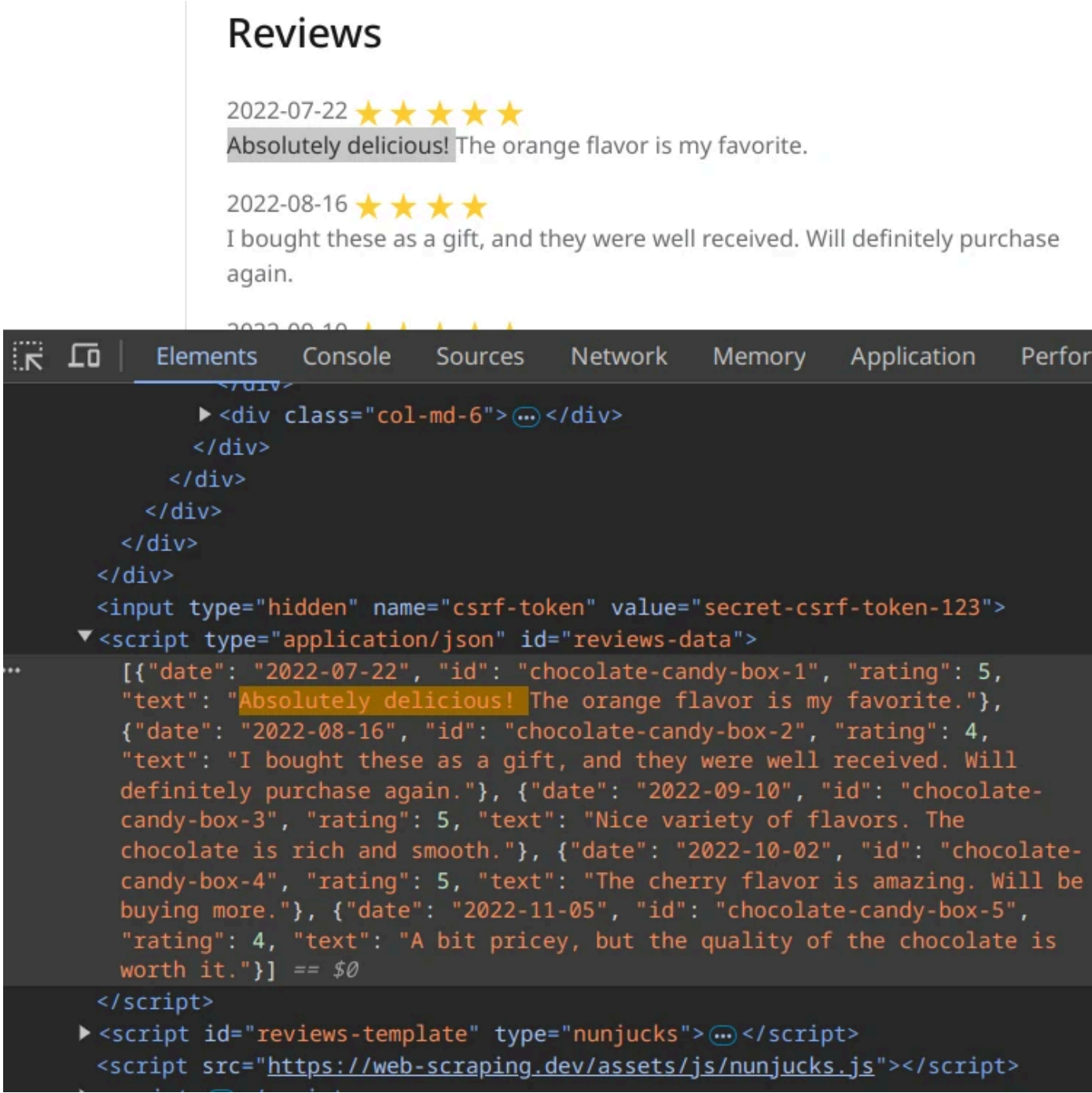
Take a look at this short video for a quick overview of hidden web data:



To verify whether that's the case it's best to find unique identifier and browser around the page source code to see if the data is there. Often it's hidden in:

- `<script>` tags as JSON objects
- `data-` attributes of HTML elements
- `style="display: none"` element that are hidden by CSS

For example, the web-scraping.dev/product/1 stores the first set of product reviews in page source and then on page load turns them into HTML elements we see on the screen:



Here we can verify this by searching the page source for unique review text.

So, here instead of parsing HTML we'd scrape the page source, extract the `<script>` element and load the hidden dataset. Here's an example of how to scrape hidden data in Python and it's popular HTML parsing libraries:

BeautifulSoup CSS Selectors Scrapfly SDK

```
import httpx
from bs4 import BeautifulSoup

resp = httpx.get("https://web-scraping.dev/product/1")
soup = BeautifulSoup(resp.text, "html.parser")

reviews = soup.find("script", id="reviews-data").text
print(reviews)
[
  {
    "date": "2022-07-22",
    "id": "chocolate-candy-box-1",
    "rating": 5,
    "text": "Absolutely delicious! The orange flavor is my favorite.",
  },
  {
    "date": "2022-08-16",
    "id": "chocolate-candy-box-2",
    "rating": 4,
    "text": "I bought these as a gift, and they were well received. Will definitely purchase again.",
  },
  {
    "date": "2022-09-10",
    "id": "chocolate-candy-box-3",
    "rating": 5,
    "text": "Nice variety of flavors. The chocolate is rich and smooth.",
  },
  {
    "date": "2022-10-02",
    "id": "chocolate-candy-box-4",
    "rating": 5,
    "text": "The cherry flavor is amazing. Will be buying more.",
  },
  {
    "date": "2022-11-05",
    "id": "chocolate-candy-box-5",
    "rating": 4,
    "text": "A bit pricey, but the quality of the chocolate is worth it.",
  },
]
```

Hidden web data can be placed in almost any part of the entire of page resources so for more on that see [our full intro to hidden web data scraping](#).

Parsing Scraped JSON

JSON is the second most popular data format encountered in web scraping and modern websites can return colossal datasets that can be difficult to parse for valuable information. These datasets are often complex, nested and even obfuscated to prevent scraping.

To parse JSON in Python there are a few standard ways:

1. Convert JSON to Python dictionaries and use tools like [nested-lookup](#) to traverse the data and find the information you need.
2. Use standard parsing path languages like [Jmespath](#) and [JsonPath](#) (similar to CSS selectors and Xpath)

Here's an example how to parse JSON data from [web-scraping.dev/reviews](#) with each of these methods:

JmesPath JsonPath

```
# review data from XHR on https://web-scraping.dev/reviews
reviews_dataset = {
  "data": {
    "reviews": {
      "edges": [
        {
          "node": {
            "rid": "teal-potion-4",
            "text": "Unique flavor and great energy boost. It's the
perfect gamer's drink!",
            "rating": 5,
            "date": "2023-05-18"
          },
          "cursor": "cmV2aWV3X2lkOnRlYWwtcG90aW9uLTQ="
        },
        {
          "node": {
            "rid": "red-potion-4",
            "text": "Good flavor and keeps me energized. The bottle design
is really fun.",
            "rating": 5,
            "date": "2023-05-17"
          },
          "cursor": "cmV2aWV3X2lkOnJlZC1wb3Rpb24tNA=="
        }
      ],
      "pageInfo": {
        "startCursor": "cmV2aWV3X2lkOnRlYWwtcG90aW9uLTQ=",
        "endCursor": "cmV2aWV3X2lkOmRhcmstcmVklXBvdGlvbi0z",
        "hasPreviousPage": False,
        "hasNextPage": True
      }
    }
  }
}

# Jmespath is the most complete package for JMESPath in Python
# pip install jmespath
import jmespath
review_texts = jmespath.search("data.reviews.edges[].node.text", reviews_dataset)
print(review_texts)

[
  "Unique flavor and great energy boost. It's the perfect gamer's drink!",
  'Good flavor and keeps me energized. The bottle design is really fun.'
]
```

Overall, JSON parsing is becoming a bigger part of web scraping as more websites use client-side rendered data rather than server side rendered HTMLs though as you can see — it's not very difficult to parse JSON data in Python given the right toolset!

For more see our full introductions to [JMesPath](#) and [JsonPath](#) JSON parsing languages.

Finally, modern Python has access to new tools like AI models and LLMs which can greatly assist with parsing or just perform the entire parsing task for you. Let's take a look at these next.

Scraping with AI and LLMs

AI and LLMs are becoming a bigger part of web scraping as they can greatly assist with parsing complex data or even perform the entire parsing task for you. Generally we can separate this process into two parts:

- Machine learning AI models. These are specially trained models to understand HTML for data extraction.

- Large Language models (LLMs). These are general purpose models that can be fine-tuned to understand HTML for data extraction.

While dedicated AI models are very difficult to build the LLMs are easily accessible in Python through services like OpenAPI, Gemini and other LLM marketplaces. Let's take a look at some examples.

Extracting Data with LLMs

To extract data using LLMs from scraped HTML we can add our HTML to the LLM prompt and ask questions about it like:

Extract price data from this product page

or more interestingly we can ask LLMs to make intelligent assessment of the data like:

Is this product in stock?
Summarize review sentiment of this product and grade it in a scale 1-10

Furthermore, we can use LLMs to assist us with traditional parsing technologies with prompts like:

For HTML: Generate CSS selector that select the price number in this product page
For JSON: Generate JmesPath selector that selects the review text in this product review JSON

These prompts can be very powerful and can save a lot of time and effort in web scraping. Here's an example how to use LLMs to extract data from HTML in Python with OpenAI API:

```
import httpx
import openai
import os

# Replace with your OpenAI API key
openai.api_key = os.environ['OPENAI_KEY']

response = httpx.get('https://web-scraping.dev/product/1')

llm_response = openai.chat.completions.create(
    model="gpt-4o",
    messages = [{
        "role": "user",
        "content": f"Extract price data from this product page:\n\n{response.text}"
    }]
)

# Extract the assistant's response
llm_result = llm_response.choices[0].message.content
print(llm_result)
'The price data extracted from the product page is as follows:\n\n- Sale Price: $9.99\n- Original Price: $12.99'
```

Important recent discovery is that LLMs perform better with data formats more simple than HTML. In some cases, it can be beneficial to convert HTML to Markdown or plain text before feeding it to the LLM parser. Here's an example how to convert HTML to Markdown using [html2text](#) in Python and then parse it with OpenAI API:

```
import httpx # pip install httpx
import openai # pip install openai
import html2text # pip install html2text
import os

# Replace with your OpenAI API key
openai.api_key = os.environ['OPENAI_KEY']

# Fetch the HTML content
response = httpx.get('https://web-scraping.dev/product/1')

# Convert HTML to Markdown
html_to_markdown_converter = html2text.HTML2Text()
html_to_markdown_converter.ignore_links = False # Include links in the markdown
output
markdown_text = html_to_markdown_converter.handle(response.text)

print("Converted Markdown:")
print(markdown_text)

# Send the Markdown content to OpenAI
llm_response = openai.chat.completions.create(
    model="gpt-4",
    messages=[
        {
            "role": "user",
            "content": f"Extract price data from this product page in markdown
format:\n\n{markdown_text}"
        }
    ]
)

# Extract the assistant's response
llm_result = llm_response.choices[0].message.content
print(llm_result)
'The price data for the product "Box of Chocolate Candy" is $9.99, discounted from the
original price of $12.99.'
```

Web scraping with LLMs is still a relatively new field and there are a lot of tools and tricks are worth experimenting with and for more on that see our [web scraping with LLM and RAG guide](#).

Extracting Data with Extraction API

Scrapfly's Extraction API contains LLM and AI models optimized for web scraping and can save a lot of time figuring this new technology out.

For example here's how easy it is to LLM query your documents:

```
from scrapfly import ScrapflyClient, ScrapeConfig, ExtractionConfig

client = ScrapflyClient(key="SCRAPFLY KEY")

# First retrieve your html or scrape it using web scraping API
html = client.scrape(ScrapeConfig(url="https://web-scraping.dev/product/1")).content

# Then, extract data using extraction_prompt parameter:
api_result = client.extract(ExtractionConfig(
    body=html,
    content_type="text/html",
    extraction_prompt="extract main product price only",
))
print(api_result.result)
{
  "content_type": "text/html",
  "data": "9.99",
}
```

Or how to extract popular data object like product, reviews, articles etc. without any user input at all:

```
from scrapfly import ScrapflyClient, ScrapeConfig, ExtractionConfig

client = ScrapflyClient(key="SCRAPFLY KEY")

# First retrieve your html or scrape it using web scraping API
html = client.scrape(ScrapeConfig(url="https://web-scraping.dev/product/1")).content
# Then, extract data using extraction_model parameter:
api_result = client.extract(ExtractionConfig(
    body=html,
    content_type="text/html",
    extraction_model="product",
))
print(api_result.result)
```

► **Example Output**

[See the Extraction API](#)

Step 3: Scaling Up

Finally, scaling web scraping is a major topic and a challenge of its own and we can break down scaling challenges into two parts:

- **Scaling technical challenges** like concurrency, scheduling, deploying and resource use.
- **Scraper blocking challenges** like proxies, user agents, rate limiting and fingerprint resistance.

Scaling and Solving Challenges

There are several important problems when it comes to scaling web scraping in Python and let's take a look at the most common challenges and solutions.

Speed of Scraping

The biggest bottle neck of python web scrapers is the IO wait. Every time a request is sent it has to travel around the world and back which is physically slow process and there's nothing we can do about it. That's an issue for a single request but for many requests we can actually do a lot!

Once we have multiple scrape requests planned we can scale up our scraping significantly by sending multiple requests concurrently. This is called **concurrent scraping** and can be achieved in Python with libraries like `asyncio` or `httpx` which support concurrent requests. Here's an `httpx` example:

```
import asyncio
import httpx
import time

async def run():
    urls = [
        "https://web-scraping.dev/product/1",
        "https://web-scraping.dev/product/2",
        "https://web-scraping.dev/product/3",
        "https://web-scraping.dev/product/4",
        "https://web-scraping.dev/product/5",
        "https://web-scraping.dev/product/6",
        "https://web-scraping.dev/product/7",
        "https://web-scraping.dev/product/8",
        "https://web-scraping.dev/product/9",
        "https://web-scraping.dev/product/10",
    ]
    async with httpx.AsyncClient() as client:
        # run them concurrently
        _start = time.time()
        tasks = [client.get(url) for url in urls]
        responses = await asyncio.gather(*tasks)
        print(f"Scraped 10 pages concurrently in {time.time() - _start:.2f} seconds")
        # will print
        "Scraped 10 pages concurrently in 1.41 seconds"

    with httpx.Client() as client:
        # run them sequentially
        _start = time.time()
        for url in urls:
            response = client.get(url)
        print(f"Scraped 10 pages sequentially in {time.time() - _start:.2f} seconds")
        "Scraped 10 pages sequentially in 10.41 seconds"

asyncio.run(run())
```

Just by running 10 requests concurrently we've reduced the scraping time from 10 seconds to 1.4 seconds which is an incredible improvement! This can scale up even higher based on scraper limits in a single process.

For a detailed dive in web scraping speed see our full guide below:

Web Scraping Speed: Processes, Threads and Async

Guide to solving IO blocks and CPU block scaling issues encountered in web scraping in Python.



Retrying and Failure Handling

Another important part of scaling web scraping is handling failures and retries. Web scraping is a very fragile process and many things can go wrong like:

- Network errors
- Server errors
- Page changes

- Rate limiting
- Captchas and scraper blocking

All of these issues are difficult to handle but they can also be temporary and transient. For this, it's a good practice to equip every Python scraper with **retrying and failure handling** logic as a bare minimum to ensure they are robust and reliable.

Here's an example how to retry `httpx` requests in Python using `tenacity` general purpose retrying library:

```
import httpx
from tenacity import retry, stop_after_attempt, wait_fixed, retry_if_exception,
retry_if_result

# Retry policy when no exception is raised but response object indicates failure
def is_response_unsuccessful(response: httpx.Response) -> bool:
    return response.status_code != 200

# Retry policy when an exception is raised
def should_retry(exception: Exception) -> bool:
    return isinstance(exception, (httpx.RequestError, httpx.TimeoutException))

# apply tenacity retry decorator to scrape function
@retry(
    stop=stop_after_attempt(5), # Retry up to 5 times
    wait=wait_fixed(2), # Wait 2 seconds between retries
    retry=retry_if_exception(should_retry) |
    retry_if_result(is_response_unsuccessful),
)
def fetch_url(url: str, **kwargs) -> httpx.Response:
    with httpx.Client() as client:
        response = client.get(url, **kwargs)
        # Explicitly raise an exception for non-200 responses
        response.raise_for_status()
        return response

# Example usage
try:
    url = "https://web-scraping.dev/product/1"
    response = fetch_url(url)
    print(f"Success! Status code: {response.status_code}")
    print(response.text[:200]) # Print first 200 characters of the response
except Exception as e:
    print(f"Failed to fetch the URL. Error: {e}")
```

Tenacity provides a lot of options for handling any web scraping target though not all issues can be solved with retries. Scraper blocking is by far the biggest challenge facing the industry so let's take a look at that next.

Scraper Blocking

While web scraping is perfectly legal around the world it's not always welcome by the websites being scraped. Many pages desire to protect their public data from competitors and employ bot detection mechanisms to block scrapers.

There are many ways to identify scrapers but they all can be summarized by the fact that **scrapers appear different from regular users**. Here's a list of key attributes:

- Request IP address is being tracked or analyzed
- Requests are performed through HTTP version 1.1 rather than HTTP2+ which is used by modern browsers. To address this use `http2=True` in `httpx` client.
- Request headers are different of those of a regular browser. To address this use `headers` parameter in `httpx` client to set what you see in your web browser.
- Scraper's TLS fingerprint is different from regular users.

- Scraper doesn't execute Javascript.
- Scraper Javascript fingerprint gives away it's a scraper.
- Scraper doesn't store cookies or local storage data.
- Scraper doesn't execute background requests (XHR).

Basically, any lack of normal user technology or behavior can be used to identify a scraper and we've covered all of these in our full guide below:

5 Tools to Scrape Without Blocking and How it All Works

Guide to all of the ways scrapers are blocked and what to do to preven it



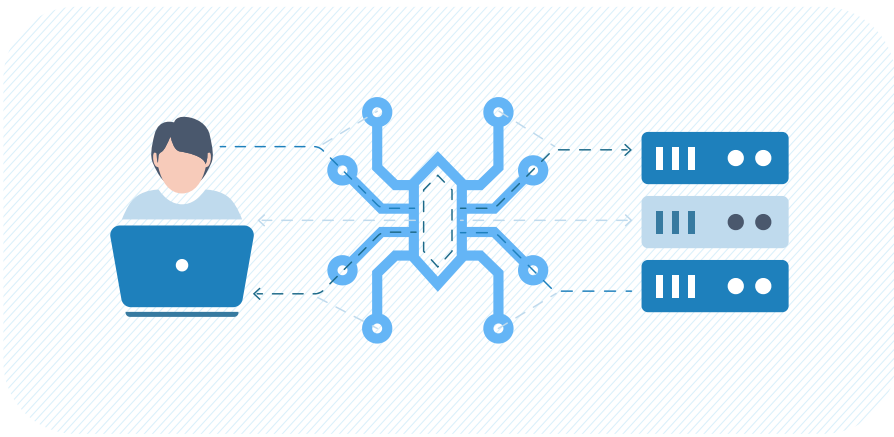
With that being said there are a few open source options available that can give you a head start against scraper blocking:

- [Curl Impersonate](#) - enhances `libcurl` which powers many HTTP clients to appear more like a web browser.
- [Undetected Chromedriver](#) - enhances Selenium to appear more like a user web browser.
- [Rotate IP proxies](#) - distribute traffic through multiple IP addresses to avoid IP blocking.

These open source solutions can get you quite far!

Scraper blocking is incredibly diverse subject so new tools are being developed and available every day which can be difficult to keep track off and that's where Scrapfly can help you out.

Power Up with Scrapfly



ScrapFly provides [web scraping](#), [screenshot](#), and [extraction](#) APIs for data collection at scale.

- [Anti-bot protection bypass](#) - scrape web pages without blocking!
- [Rotating residential proxies](#) - prevent IP address and geographic blocks.
- [JavaScript rendering](#) - scrape dynamic web pages through cloud browsers.
- [Full browser automation](#) - control browsers to scroll, input and click on objects.
- [Format conversion](#) - scrape as HTML, JSON, Text, or Markdown.
- [Python](#) and [Typescript](#) SDKs, as well as [Scrapy](#) and [no-code tool integrations](#).

Start Scraping

Existing Scrapers

There are a lot of great open source tools for popular web scraping targets and often there's no reason to reinvent the wheel.

Here's a collection web scraping guides and scrapers on Github for the most popular scraping targets covered by Scrapfly:

See Scrapfly Scrape Guides

45+ open source scrapers!

Deploying Python Scrapers

To deploy Python scrapers docker is the most popular option as it allows packing the entire scraper environment into a single container and run it anywhere.

For this, let's use Python's [poetry package manager](#) to create a simple web scraping project:

1. Install Poetry using the [official install instructions](#)
2. Create your python project using `poetry init`
3. Create a `scraper.py` file with your scraping logic
4. Package everything with `Dockerfile` and `docker build` command

Here's a simple Dockerfile for a modern Python web scraper Python 3.12 and Poetry as package manager:

```
# Use the official lightweight Python image
FROM python:3.12-slim as base

# Set environment variables for Poetry
ENV POETRY_VERSION=1.7.0 \
    POETRY_VIRTUALENVS_IN_PROJECT=true \
    POETRY_NO_INTERACTION=1

# Set working directory
WORKDIR /app

# Install Poetry
RUN apt-get update && apt-get install -y curl && \
    curl -sSL https://install.python-poetry.org | python3 - && \
    ln -s /root/.local/bin/poetry /usr/local/bin/poetry && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

# Copy only Poetry files to leverage caching
COPY pyproject.toml poetry.lock ./
# Copy your project files,
COPY scraper.py .
# Install the project (if it's a package)
RUN poetry install --no-dev

# Specify the command to run your Python script
CMD ["poetry", "run", "python", "scraper.py"]
```

This Dockerfile will create a lightweight Python image with your scraper and all dependencies installed. You can deploy this to your favorite platform capable of running Docker containers like AWS, GCP, Azure or Heroku and run your scraper at any point!

Scraping Frameworks

Lastly, there are a few popular web scraping frameworks that can help you with building and scaling your web scraping operations. Most popular by far one is [Scrapy](#) which is a feature rich and powerful web scraping framework for Python.

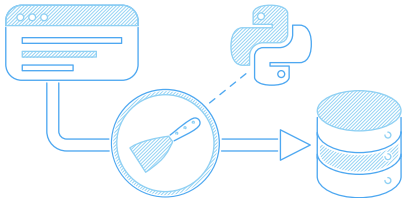
Scrapy bundles a lot of technologies we've covered in this guide into a single cohesive framework and here's a quick breakdown:

Pros	Cons
<div><div></div><div>Powerful HTTP client capable of concurrent requests.</div></div>	<div><div></div><div>High learning curve and complex architecture.</div></div>
<div><div></div><div>Crawling rules and crawling path generators for crawling websites.</div></div>	<div><div></div><div>Does NOT support headless browser scraping.</div></div>
<div><div></div><div>XPath and CSS selectors for parsing HTML.</div></div>	<div><div></div><div>No AI or LLM support.</div></div>
<div><div></div><div>Auto Retries and failure handling.</div></div>	

So, Scrapy offers quite a bit out of the box and can be a great starting point for people who are familiar with web scraping but need a more robust solution. For more on Scrapy see our full introduction guide below:

Web Scraping With Scrapy: The Complete Guide in 2024

Full introduction to scrapy and scrapy example project



Summary

In this extensive guide we did a deep overview of contemporary web scraping techniques and tools in Python.

We've covered a lot so let's summarize the key points:

- Scraping can be done using HTTP client or headless web browser.
- HTTP clients are faster but can't scrape dynamic pages.
- Headless browsers can scrape dynamic pages but are slower and harder to scale.
- Headless browsers can capture screenshots, browser data and network requests.
- HTML Parsing can be done using CSS selectors, XPath, BeautifulSoup or AI models.
- Datasets can be hidden in HTML `<script>` elements
- JSON Parsing can be done using JmesPath or JsonPath.
- LLMs can be used to extract data from HTML and JSON.
- Concurrency through Python asyncio can speed up scraping dozens to hundreds of times.

For more on each of these topics see [Scrapfly Academy](#) — free web scraping course by Scrapfly.

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?
- 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?
- Python httpx vs requests vs aiohttp - key differences
- How to handle popup dialogs in Playwright?
- How to scrape images from a website?
- 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    

Related Posts



Mar 10, 2025

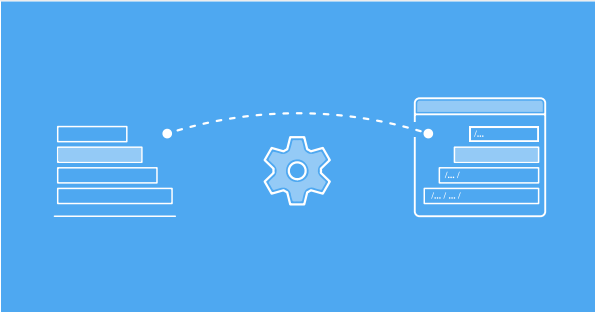
Guide to List Crawling: Everything You Need to Know

In-depth look at list crawling - how to extract valuable data from list-formatted content like tables, listicles and paginated pages.

WEB CRAWLING

BEAUTIFULSOUP

PYTHON




Jan 29, 2025

How to Find All URLs on a Domain

Learn how to efficiently find all URLs on a domain using Python and web crawling. Guide on how to crawl entire domain to collect all website data

WEB CRAWLING

PYTHON



Jan 22, 2025

How to Capture and Convert a Screenshot to PDF

Quick guide on how to effectively capture web screenshots as PDF documents

https://scrapfly.io/blog/everything-to-know-about-web-scraping-python/

26/28

SCREENSHOTS PYTHON
NODEJS

Company

Careers
Terms of service
Privacy Policy
Data Processing Agreement
KYC Compliance
Status

Integrations

Zapier
Make
N8n
LlamaIndex
LangChain

Social



Tools

Convert cURL commands to Python code
JA3/TLS Fingerprint
HTTP2 Fingerprint
Xpath/CSS Selector Tester

Resources

API Documentation
Web Scraping Academy
Is Web Scraping Legal?
Web Scraping Tools
FAQ

Learn Web Scraping

Web Scraping with Python
Web Scraping with PHP
Web Scraping with Ruby
Web Scraping with R
Web Scraping with NodeJS
Web Scraping with Python Scrapy
How to Scrape without getting blocked tutorial
Web Scraping with Python and BeautifulSoup
Web Scraping with Nodejs and Puppeteer
How To Scrape GraphQL
Best Proxies for Web Scraping
Top 5 Best Residential Proxies

Usage

What is Web Scraping used for?
Web Scraping for AI Training
Web Scraping for Compliance
Web Scraping for eCommerce
Web Scraping for Finance
Web Scraping for Fraud Detection
Web Scraping for Jobs
Web Scraping for Lead Generation

- Web Scraping for News & Media
- Web Scraping for Real Estate
- Web Scraping for SERP & SEO
- Web Scraping for Social Media
- Web Scraping for Travel

© 2025 Scrapfly - The Best Web Scraping API For Developers