Why

06 May 2025 20:08

API → Get data from API's

Requirement → Don't have all the necessary data.

↓

Data is costly → 5% missing

↓ Money / lost

Surveys, Manual Labor

Online, forms → messy

↓

structure → 10 fields → datatypes

3 → required

Data → Accurate, structured, cheap. → Automate

↓ effort ↓

↓ manual ∝

costly

Limits

API's → data acquisition → Automate

↓

Structured → JSON → Value

↓

Pandas df

Accurate → Google weather API → database.

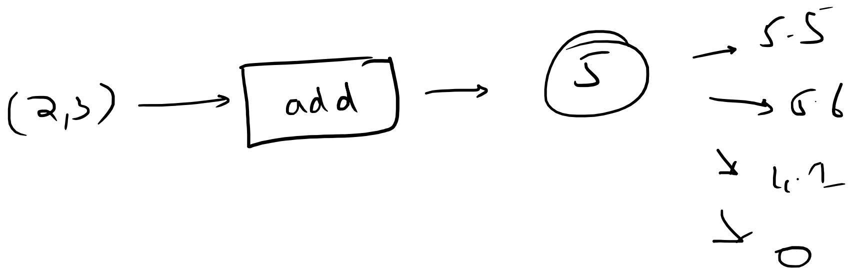
→ deterministic

↓

info → Mumbai → accurate

info → Mumbai → accurate

Web → inaccuracies → 2024



	Accuracy	Structured output	Automated.	Cost
1) Surveys	✓	medium	X	High
2) API	✓	✓	✓	medium
3) Web scraping	↓	effort	✓	lowest

Web scraping → Automated copy / paste.

What is Web Scraping

🕸️ What is Web Scraping?

Web scraping is the process of **automatically extracting data** from websites. It involves fetching the HTML of a webpage and then parsing it to extract the desired information (like product prices, article titles, stock prices, etc.).
At its core, web scraping mimics how a human would browse the web and copy useful data, but does so programmatically.

🔍 Why is Web Scraping Required?

Here are some common reasons for using web scraping:

1. **Data Collection**
   Many websites display data but do not offer a public API to access it. Web scraping helps collect such data for:
   - Market research ✓ → web scraping used a lot
   - Competitor analysis ✓
   - Price monitoring ✓
   - News aggregation ✓
2. **Automation**
   Automates repetitive tasks like:
   - Logging into websites ✓
   - Downloading reports ✓
   - Monitoring stock/crypto prices ✓
3. **Machine Learning & NLP** → text Based
   Real-world training data (e.g., tweets, reviews, blogs) is often scraped from the web to build and evaluate ML models.
4. **No API Available** ✓✓
   Some services either don't provide an API or limit access to it, and scraping becomes the only viable method to obtain data.

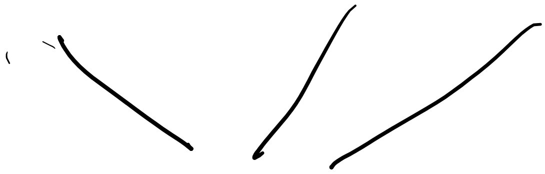
⚠ Ethical and Legal Considerations

- **Always** check the site's robots.txt file (e.g., https://example.com/robots.txt) to see what scraping is allowed.
- Avoid scraping personal data or content behind logins without permission.
- Many sites prohibit scraping in their **Terms of Service**.

Market research → up to date information

✓

Web scraping ✓ API ✓ Database ✓



Data is your bread / butter

limitations

Types

🔧 Types of Web Scraping Methods

1) 2)

There are several methods for web scraping, depending on the complexity of the website and the data you need. Here's a breakdown:

1. HTML Parsing (Static Scraping) → Text

text ✓

- **Use case**: Pages with static HTML content.
- **How it works**: Fetch HTML using requests and parse using libraries like BeautifulSoup or lxml.

✅ Simple and fast.
❌ Won't work on JavaScript-loaded content.

2. Browser Automation (Dynamic Scraping)

- **Use case**: Websites that load content dynamically using JavaScript (like infinite scroll).
- **Tools**: Selenium, Playwright, or Puppeteer (Node.js).

✅ Can handle JavaScript, forms, clicks.
❌ Slower, heavier on resources.

APIs, Scraping ✗

3. API Scraping

- **Use case**: Some sites use internal APIs to load data in the frontend.
- **How it works**: Intercept network traffic (using browser DevTools), find API endpoints, and make direct requests.

✅ Clean and structured data.
❌ May require ~~reverse engineering~~ and authentication.

4. Headless Browsers

- **Use case**: Similar to browser automation but without a GUI. → Rendering
- **Tools**: Playwright, Selenium (headless mode)
- ✅ Great for automated scraping at scale.

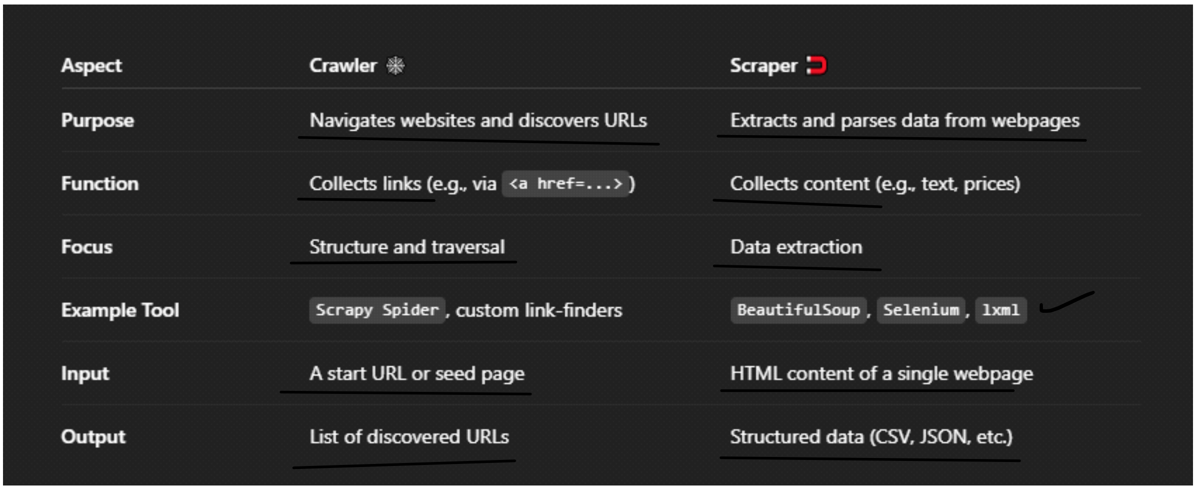
❌ Still heavier than requests-based methods. → BS, API

5. Scraping with Crawl Libraries

- **Use case**: Large-scale web crawling and scraping across multiple pages/sites.
- **Tool**: Scrapy → Advanced

✅ Handles throttling, pagination, pipelines.
❌ Steeper learning curve.

Crawler vs Scraper



Aspect	Crawler 🕸️	Scraper 🧲
Purpose	Navigates websites and discovers URLs	Extracts and parses data from webpages
Function	Collects links (e.g., via `<a href=...>`)	Collects content (e.g., text, prices)
Focus	Structure and traversal	Data extraction
Example Tool	`Scrapy Spider`, custom link-finders	`BeautifulSoup`, `Selenium`, `lxml` ✓
Input	A start URL or seed page	HTML content of a single webpage
Output	List of discovered URLs	Structured data (CSV, JSON, etc.)

🔁 How They Work Together

In most scraping workflows:

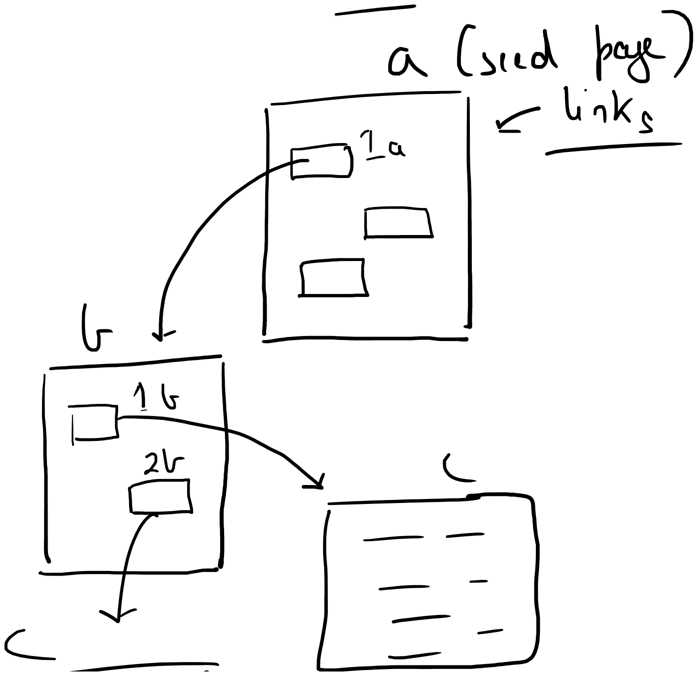
1. **Crawler** finds all relevant pages (e.g., product pages across categories).
2. **Scraper** extracts the useful information (e.g., price, name) from each page.

You can think of a **crawler as the explorer** and a **scraper as the miner**.

Crawler → Spider

index web pages

Data → Text ∝

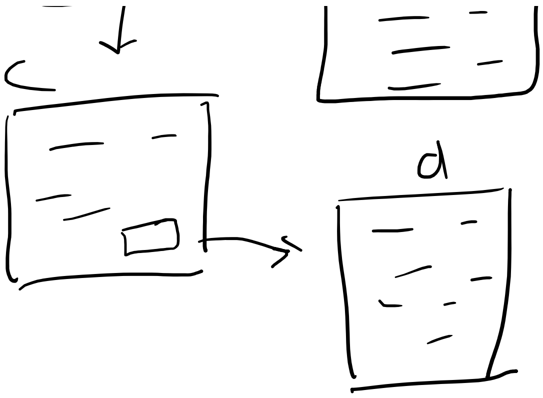


[1a → 1b → 1c]

[1a → 2b → 2c → d]

Crawlers → Indexing

Scraper → Data scrape



Scraper → Data scrape

Crawlers

Ethical Considerations → Most imp.

⚖️ Ethical Considerations Around Web Scraping

Web scraping can be incredibly useful, but it comes with ethical responsibilities to respect others' content, resources, and privacy. Here are the key considerations:

Robots

1. Respect robots.txt

- Websites often include a robots.txt file (e.g., https://example.com/robots.txt) specifying which parts of the site can be accessed by bots.
- While it's not legally binding, it's a widely respected **standard for ethical scraping**.

```
User-agent: *
Disallow: /private/
```

☑ Always check and respect this file before scraping.

2. Avoid Overloading Servers

- Excessive requests can strain a website's infrastructure (especially small or non-commercial sites).
- Add **delays or rate-limiting** between requests.

☑ Be a "polite scraper" — simulate human-like browsing speed.

DDoS

3. Respect Terms of Service (ToS)

- Most websites include ToS that explicitly **disallow scraping** or limit its usage.
- Violating ToS can have legal consequences, even if the data is publicly visible.

☑ Always review and follow the site's ToS.

4. Do Not Scrape Personal or Sensitive Data

- Avoid collecting data that is personally identifiable (PII) or confidential unless the user has explicitly consented.
- Examples: emails, addresses, login-protected content.

✗ Never scrape data behind login forms or paywalls without permission.

5. Attribute the Source

- If you're using scraped data for research, blog posts, or reports, **credit the original source**.

☑ Acknowledge the website if you're republishing or sharing scraped content.

6. Use Official APIs When Available

- Many sites offer public APIs specifically to access their data in a safe, legal way.

☑ Prefer APIs over scraping whenever possible.

Ethical scraping = Legal + Respectful + Responsible.

API → Accurate
API → structured output
API → Automate

Github → Code → Open AI

20$ → Code

Robots.txt

📄 What is the robots.txt File?

The robots.txt file is a **standard protocol** used by websites to **communicate with web crawlers and bots** about which parts of the site they are allowed or disallowed to access.

📍 Location

It's always located at the **root** of a website:
https://example.com/robots.txt

⚙️ Purpose

- To **guide bots** (like Googlebot, Bingbot, or your own scraper) on what they can or cannot crawl.
- To **prevent overloading** the site or blocking access to sensitive/unnecessary areas.

🛠️ Example Structure

```
User-agent: *          # Applies to all bots
Disallow: /private/    # Don’t crawl this folder
Allow: /public/        # Do crawl this folder
```

More Examples:

```
User-agent: Googlebot
Disallow: /no-google/
User-agent: *
Disallow: /tmp/
```

✅ Important Points

- robots.txt **doesn't enforce** access control — it's a **voluntary convention**, not a security measure.
- Scrapers can technically **ignore it**, but doing so is **unethical** and can get your bot blocked or blacklisted.

Advantages

☑ Advantages of Web Scraping

Web scraping offers several benefits, especially in data-driven fields like data science, business intelligence, and automation.

1. Access to Large-Scale Data

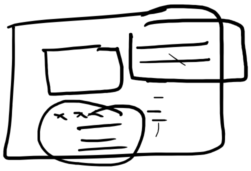
- Easily collect data from multiple pages, sites, or even entire domains. → expensive
- Enables **big data analysis** and modeling when official APIs or datasets are unavailable.

2. Real-Time Information

- Scraping can be automated to collect **up-to-date data** (e.g., ~~prices,~~ news, weather, stock info).
- Ideal for building **dashboards, alerts, or monitoring systems**.

3. Cost-Effective

- Avoids paying for commercial data feeds or services.
- Uses **open web content** to build your own custom datasets.



4. Customizable

- Tailor scraping scripts to extract **only the data you need** in the exact format you want.
- Enables **targeted data collection** (e.g., specific tags, products, reviews).

5. Useful for Competitive Intelligence

- Track competitors':
  - Product listings
  - Pricing changes
  - Content strategies
  - Customer reviews

6. Enables Machine Learning/NLP Projects

- Use scraped data to:
  - Train recommendation systems
  - Perform sentiment analysis
  - Build language models or chatbots

7. Automation of Manual Tasks

- Automate repetitive browser tasks like:
  - Filling forms
  - Downloading reports
  - Navigating web pages

Limitations

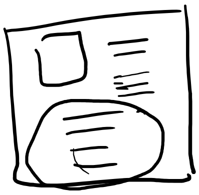
⚠️ **Disadvantages / Limitations of Web Scraping**
Despite its usefulness, web scraping comes with several challenges and drawbacks:

1. Legal and Ethical Risks

- Scraping may **violate terms of service**.
- Some sites may **legally pursue** scrapers, especially if personal or copyrighted data is involved.

  🛑 Always ensure you're compliant with robots.txt and ToS.

hardiode



2. Website Structure Changes ∝

- Scrapers are **fragile**: even minor changes in HTML structure can break your script.

  🔁 Requires constant maintenance.

3. JavaScript-Rendered Content ✓

- Many modern websites use JavaScript (React, Angular, etc.) to load content dynamically.
- Requires **heavier tools** like Selenium or Playwright, which are slower.

4. IP Blocking / Rate Limiting

- Sites may detect scraping activity and:
  - Block your IP ✓
  - Require CAPTCHA ✓
  - Throttle requests ✓

  🧩 Solutions: use proxies, rotate user agents, add delays.

{1: {name: – ,
age: – ,
subjects: [ ]}}

2:

3:

5. Data Quality Issues

- Extracted data may have:
  - Duplicates
  - Missing values
  - Inconsistent formats (e.g., dates, currency symbols) ✓

  🧼 Requires post-processing and cleaning.

∝

{10A: {1: { }, 2: { }}}

6. Resource Intensive

- Dynamic scraping tools use browsers → higher memory and CPU usage.
- May not scale well without proper infrastructure.

7. No Guarantees of Availability

- The site you're scraping from can:
  - Change structure
  - Add anti-scraping protections
  - Shut down or remove content

In short, web scraping is powerful, but **not always stable or sustainable** without care and respect for source websites.

Client Server Model

customer -> client → request → server → Kitchen

← Response ←

🌐 Client-Server Model on the Web

The **client-server model** is the foundational concept behind how data is exchanged on the web. It defines how the **client** (your browser or any app) and the **server** (the web host) interact with each other.

🔄 Basic Flow of the Client-Server Model

1. **Client (Request)**:
   - The client is typically a **web browser** (e.g., Chrome, Firefox) or any app that makes requests to a server.
   - The client initiates a **request** for resources (such as a webpage, image, or data) from the server.
2. **Server (Response)**:
   - The server hosts the requested resource (a webpage, file, API endpoint).
   - The server receives the client's request, processes it, and returns the requested resource back to the client.

🖥️ The Components of the Model

1. Client

- **What is it?**
  Any device or software (browser, mobile app, etc.) that interacts with the web.
- **Responsibilities:**
  - **Sending Requests**: Sends HTTP requests to access resources.
  - **Rendering Responses**: Displays the response (HTML, images, etc.) from the server to the user.
- **Examples:**
  - Web browsers like Chrome, Safari, etc.
  - Mobile apps that interact with web APIs.

2. Server

- **What is it?**
  A computer or system hosting the web application or service.
- **Responsibilities:**
  - **Processing Requests**: Receives and processes incoming requests.
  - **Sending Responses**: Returns the requested resource (HTML page, JSON data, etc.).
- **Examples:**
  - Web servers like Apache, Nginx, or cloud servers (AWS, Google Cloud).

🔑 Key Technologies Involved

1. **HTTP (HyperText Transfer Protocol)**
   - Protocol that allows the client and server to communicate.
   - Clients send **GET, POST, PUT, DELETE** requests, and servers respond with **status codes** (200 OK, 404 Not Found, etc.).
2. **HTML / CSS / JavaScript**
   - The server sends back **HTML** content, which is rendered by the client's browser.
   - **CSS** handles the styling, and **JavaScript** manages the interactivity.
3. **APIs (Application Programming Interfaces)**
   - A server might return data in the form of an API response (often JSON or XML) instead of an HTML page. ✓
   - This allows **decoupled communication** between the client and server (especially in modern web apps or mobile apps).

📝 A Typical Interaction Example

1. **Request**:

- Client: Browser requests https://example.com/index.html (GET request).

2. **Response**:
    - Server: The server processes the request and returns the HTML for the index.html page.
3. **Rendering**:
    - Client: The browser renders the HTML, loads CSS, and runs JavaScript to display the page.

Web Scraping

Request and Response Bodies

Request and Response Bodies in Web Communication

In the **client-server model** of the web, both the **request** and **response** involve **bodies** that contain the data being exchanged between the client (browser, app, etc.) and the server. Let's break down both:

Request Body

The **request body** is the part of an HTTP request that carries the data sent from the **client to the server**. Not all HTTP methods (e.g., **GET**) send a body, but methods like **POST**, **PUT**, and **PATCH** often include data in the body.

When is a Request Body Used?

- **POST**: To submit form data or send a payload to the server (e.g., creating a new resource).
- **PUT**: To update a resource with new data (e.g., updating a user's details).
- **PATCH**: To partially update a resource.
- **DELETE**: While typically no body, some implementations allow a request body with **DELETE** to specify deletion criteria.

1. **JSON**: ✓ json =
   ```
   {
    "name": "John Doe",
    "email": "john.doe@example.com"
   }
   ```
   requests . post (url, json=json)
2. **Form Data (x-www-form-urlencoded)**:
   - Common in HTML forms.
   - Example: name=John+Doe&email=john.doe%40example.com
3. **Multipart Form Data**:
   - Used for uploading files, along with other form data.
   - Often seen in file upload scenarios.
4. **XML**:
   ```
   <user>
    <name>John Doe</name>
    <email>john.doe@example.com</email>
   </user>
   ```

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 68
{
 "name": "John Doe",
 "email": "john.doe@example.com"
}
```

Response Body

The **response body** is the part of an HTTP response that carries the data sent from the **server to the client**. It contains the **actual content** requested, such as HTML, JSON, images, or other resources.

When is a Response Body Used?

- Every response to a **GET** request usually includes a response body containing the requested resource (e.g., a webpage or API data). HTML , JSON
- Responses to **POST**, **PUT**, and **DELETE** requests can also include a body, often confirming the success or failure of an action.

1. **HTML**: The content of a webpage that gets rendered in the browser.
2. **JSON**: Common for APIs to send back structured data.
   ```
   {
     "status": "success",
     "message": "User created"
   }
   ```
3. **XML**: Used for structured data exchange in some APIs.
   ```
   <response>
     <status>success</status>
     <message>User created</message>
   </response>
   ```
4. **Plain Text**: Sometimes used for simple status messages or logs.
   User created successfully.
5. **Images, Files**: If the server returns media or file content, the body may contain binary data (images, videos, PDFs, etc.).

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 314
<html>
  <head>
    <title>Welcome</title>
  </head>
  <body>
    <h1>Welcome to Our Website</h1>
  </body>
</html>
```

🔑 Key Differences Between Request and Response Bodies:

Aspect	**Request Body**	**Response Body**
Purpose	Sent from client to server to provide data	Sent from server to client as the result of the request
Common Methods	POST, PUT, PATCH (sometimes DELETE)	GET, POST, PUT, DELETE (for status, confirmation, etc.)
Content Types	JSON, Form Data, XML, Multipart/Form-Data	HTML, JSON, XML, Images, Files, Text
Usage	To create, update, or delete resources	To send back the requested data or confirmation

Recap

09 May 2025 17:37

Incomplete data → API → data

↘ Web Scraping → data.

API, Web scraping → data fetch from HTML structure.

↗ Structured output

↓ difficult.

HTML

🌍 What is HTML?

HTML (HyperText Markup Language) is the standard language used to create and structure content on the **web**. It consists of a series of **elements** or **tags** that tell the browser how to display text, images, videos, links, forms, and other content on a webpage.
HTML defines the **structure** of a webpage by using elements wrapped in **tags**. These elements describe content (e.g., headings, paragraphs, images, links), and the browser renders it accordingly.

🏷️ Commonly Used HTML Tags and Their Functionality

Tag	Description	Example
<html>	Root element of an HTML document.	<html>...</html>
<head>	Contains meta-information about the document (e.g., title, styles, scripts).	<head><title>Page Title</title></head>
<title>	Sets the title of the webpage (appears in browser tab).	<title>My Page</title>
<body>	Contains the content of the webpage (text, images, etc.).	<body><h1>Welcome</h1></body>
<h1> to <h6>	Header tags that define headings from the most important (<h1>) to the least (<h6>).	<h1>This is a main heading</h1>
<p>	Defines a paragraph of text.	<p>This is a paragraph.</p>
<a>	Defines a hyperlink. Specifies the URL destination with the href attribute.	<a href="https://example.com">Click here</a>
<img>	Embeds an image. Requires the src attribute for the image path.	<img src="image.jpg" alt="A description of the image">
<ul>	Creates an unordered (bulleted) list.	<ul><li>Item 1</li><li>Item 2</li></ul>
<ol>	Creates an ordered (numbered) list.	<ol><li>Step 1</li><li>Step 2</li></ol>
<li>	Defines a list item (used inside <ul> or <ol>).	<li>List item 1</li>
<div>	Defines a division or section in a document (block-level container).	<div><p>Content goes here</p></div>
<span>	Defines a small section of text (inline container).	<span class="highlight">Important text</span>
<br>	Inserts a line break.	Line 1<br>Line 2
<strong>	Defines important text (usually bold).	<strong>This is important</strong>
<em>	Defines emphasized text (usually italicized).	<em>This is emphasized text</em>
<form>	Defines an HTML form for user input.	<form action="/submit" method="post">...</form>
<input>	Defines an input field (text, radio button, checkbox, etc.).	<input type="text" name="username">
<button>	Defines a clickable button.	<button type="submit"> Submit</button>
<table>	Defines a table.	<table><tr><td>Row 1, Cell 1</td></tr> </table>
<tr>	Defines a row in a table.	<tr><td>Row 1, Cell 1</td><td>Row 1, Cell 2</td></tr>
<td>	Defines a cell in a table (used inside <tr>).	<td>Cell Content</td>
<th>	Defines a table header cell (bold and centered by default).	<th>Header 1</th>

→ anchor

<ul>
<li> first </li>

- first
- second

container

<iframe>	Embeds another document (e.g., another webpage, video) within the current page.	<iframe src="https://www.youtube.com/embed/example"></iframe>
<link>	Defines the relationship between the current document and an external resource (usually used to link stylesheets).	<link rel="stylesheet" href="styles.css">
<meta>	Defines metadata about the HTML document (e.g., character set, viewport settings).	<meta charset="UTF-8">
<footer>	Defines a footer section for a webpage.	<footer>Contact us at: contact@example.com</footer>
<header>	Defines a header section for a webpage.	<header><h1>Welcome to my website</h1></header>

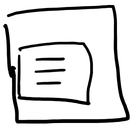
📝 **Key Points:**

- **Block-level tags** (e.g., <div>, <h1>) typically start on a new line and occupy the full width available.
- **Inline tags** (e.g., <span>, <a>) do not start on a new line and only take up as much space as their content.
- HTML tags typically come in pairs: **opening** (<tag>) and **closing** (</tag>) tags, though some tags (like <img> or <br>) are **self-closing**.

💾 Example: Basic HTML Structure

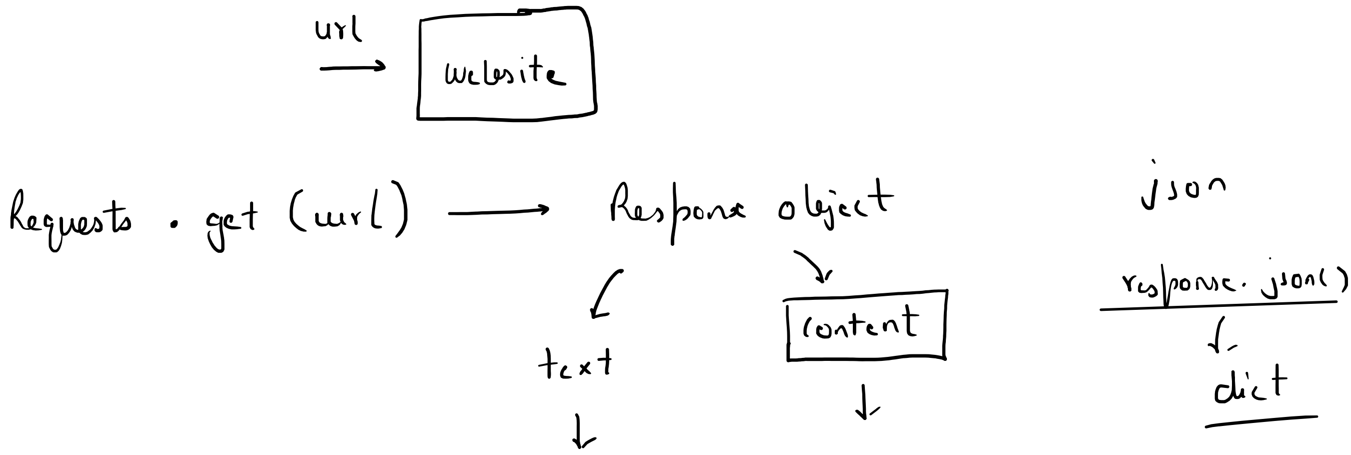
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Page</title>
</head>
<body>
  <header>
    <h1>Welcome to My Website</h1>
  </header>
<main>
    <p>This is a paragraph of text on the page.</p>
    <a href="https://example.com">Click Here</a>
  </main>
<footer>
    <p>&copy; 2025 My Website</p>
  </footer>
</body>
</html>
```

Flow

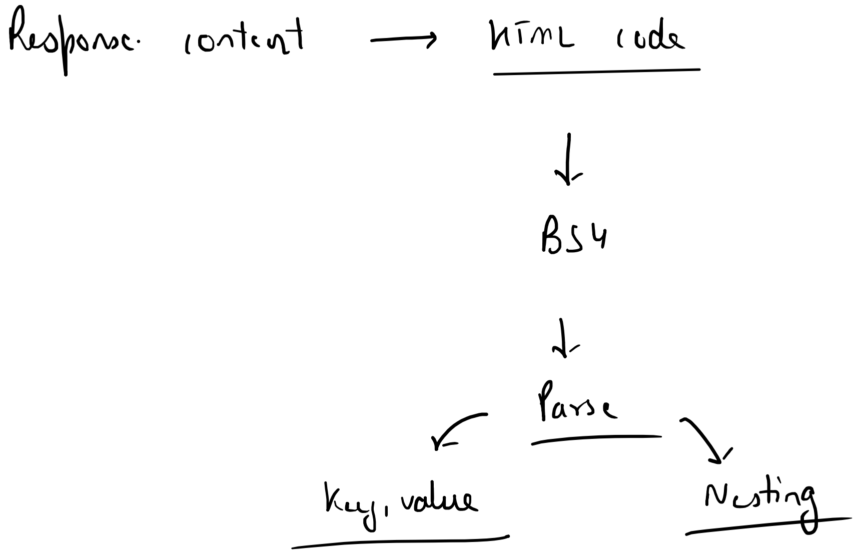


🔄 Flow of Web Scraping in Python

1. **Define the Goal**:
   - ○ Determine the data you want to scrape (e.g., product prices, articles, stock prices).
2. **Inspect the Website**:
   - ○ Use browser developer tools (Inspect Element) to examine the HTML structure of the target webpage.
   - ○ Identify the tags, classes, or IDs where the data resides.|
3. **Send a Request to the Website**:
   - ○ Use a library like requests to send an HTTP request to the webpage.
4. **Check for robots.txt**:
   - ○ Ensure that scraping is allowed by reviewing the website's robots.txt file to avoid legal issues. ✓
5. **Parse the HTML Content**: ✓
   - ○ Use libraries like BeautifulSoup or lxml to parse the HTML response from the webpage.
6. **Extract the Data**:
   - ○ Use parsing techniques (e.g., find by tag, class, or ID) to extract the relevant data.
   - ○ For dynamic content, you may need to use tools like Selenium to interact with JavaScript-rendered elements.
7. **Handle Data**:
   - ○ Clean, structure, and save the scraped data in a desired format (e.g., CSV, JSON, database). ✓
8. **Store the Data**:
   - ○ Save the extracted data to a file or a database for later use. ✓
9. **Respect Website Policies**:
   - ○ Add proper delays between requests (use time.sleep() or random delays) to avoid overwhelming the website.
   - ○ Use headers and rotate user agents to simulate human behavior. ✓
10. **Error Handling**:
    - ○ Implement error handling for timeouts, missing elements, or invalid responses to make the scraper more robust. ✓



str binary



Response content → HTML code

BS4

Parse

Key, value

Nesting

Selenium → mimic human actions

Frameworks in Python

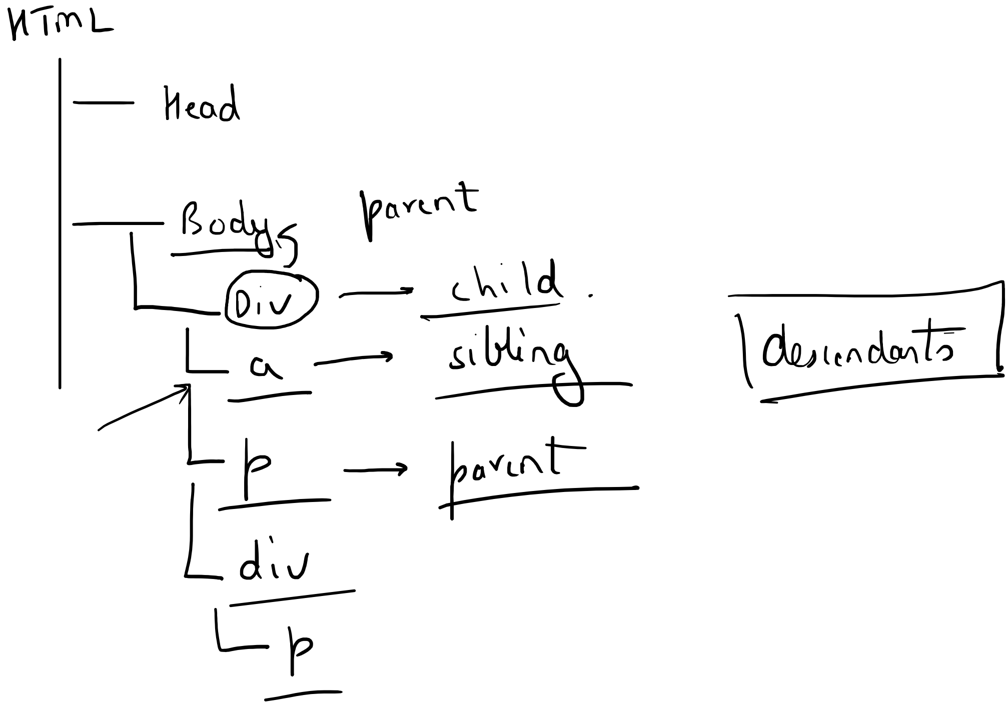
Web Scraping Frameworks in Python

1. **BeautifulSoup**:
   - A popular and easy-to-use library for parsing HTML and XML documents.
   - It provides simple methods to navigate and search the parse tree, making it ideal for web scraping tasks involving static pages.

2. **Scrapy**:
   - A powerful and fast web scraping framework designed for large-scale web scraping.
   - It allows you to create spiders that can crawl websites, extract data, and store it in various formats (JSON, CSV, etc.). Scrapy is especially suited for handling complex scraping tasks with multiple pages and links.

3. **Selenium**:
   - Primarily used for automating web browsers, Selenium can also be employed for web scraping, particularly for dynamic content rendered by JavaScript.
   - It simulates real user interactions, which allows it to extract data from pages that require interaction (e.g., clicking, scrolling).

4. **lxml**:
   - A high-performance library for parsing and extracting data from XML and HTML documents.
   - Known for its speed and support for XPath queries, making it ideal for parsing large-scale HTML pages.


HTML
Head
Bodys
parent
Div
child
a
sibling
p
parent
div
p
descendants