Turning the Web into a Dataset

A procedural guide to scraping AmbitionBox company data with Python.

The Source

The Destination

Company Name	Rating	Reviews	HQ Location	Age	Employees
Technovate Inc.	4.5	2.3K	San Francisco, CA	8 Years	500-1000
Global Solutions Ltd.	4.2	1.8K	New York, NY	15 Years	1000-5000
Innovex Data	4.0	950	London, UK	5 Years	200-500

The Target

Extract company intelligence (Name, Rating, Reviews, HQ, Age, Employees) from AmbitionBox.

The Challenge

The site provides no API. The data is locked behind complex HTML structures and server protections.

The Approach

We will build a custom Python bot to fetch, parse, and structure this data programmatically.

The Python Extraction Toolkit

`requests`

The Messenger

Sends HTTP requests to the server to fetch the webpage's raw HTML. It acts as our browser, knocking on the server's door.

`BeautifulSoup (bs4)`

The Translator

Parses the raw HTML code, allowing us to navigate the Document Object Model (DOM) to locate specific tags and data points.

`pandas`

The Warehouse

Used to store the extracted lists and export them into a readable, analytical DataFrame format.

The First Hurdle: Access Denied

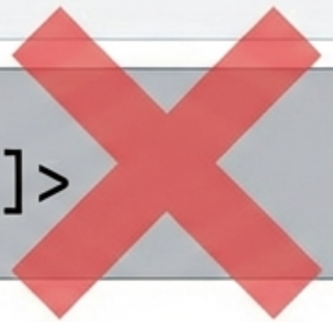
```python
import requests

url = 'https://www.ambitionbox.com/list-of-companies
response = requests.get(url)
print(response)
```

JetBrains Mono

```
<Response [403]>
```

JetBrains Mono



Status 403: Forbidden

The server has rejected our request.

Websites use 'robots.txt' and server rules to filter traffic.

Our default Python request identifies itself as a script, which the server blocks to prevent scraping.

Disguising the Bot as a Browser

The Failed Request

```
requests.get('url')
```

Identifies as “python-requests/x.x”. Blocked.

The Successful Request

```
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 6.3)...'}
requests.get('url', headers=headers)
```

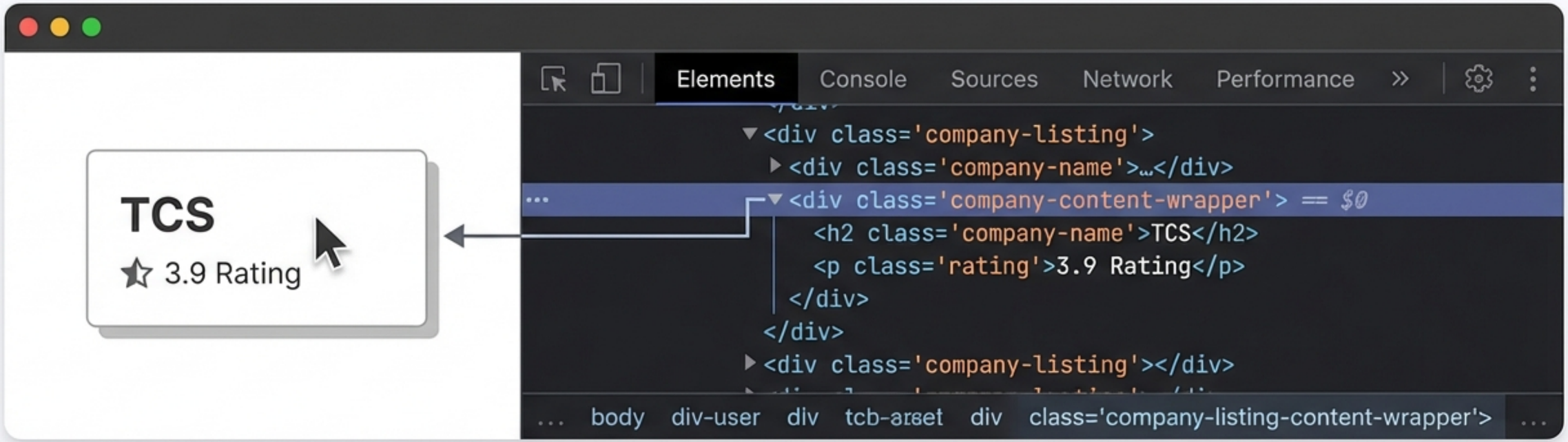
Identifies as “Mozilla/Windows”. Accepted.

The Fix: User-Agent Spoofing

By passing a “User-Agent” string in the headers, we tell the server:
“I am not a bot; I am a human using a browser.”

The response code flips from 403 to 200 (OK).

Mapping the Territory



1. Parse

Pass the Response text

Pass the response text to BeautifulSoup using the 'lxml' parser.

2. Prettify

Messy raw HTML

Use soup.prettify() to organize the messy raw HTML into a readable tree structure.

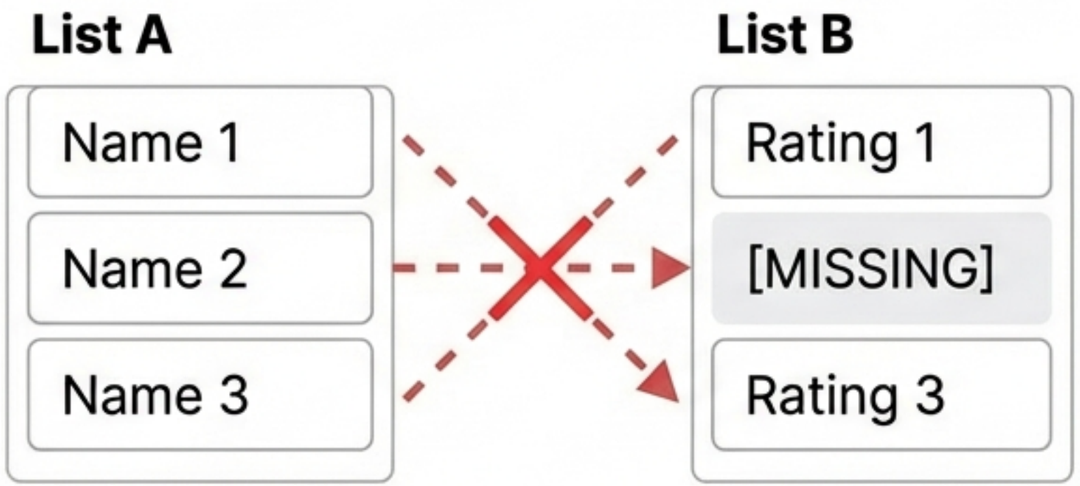
3. Inspect

Wrap our common target data

Use the browser's developer tools to identify the specific tags (div, h2, p) that wrap our target data.

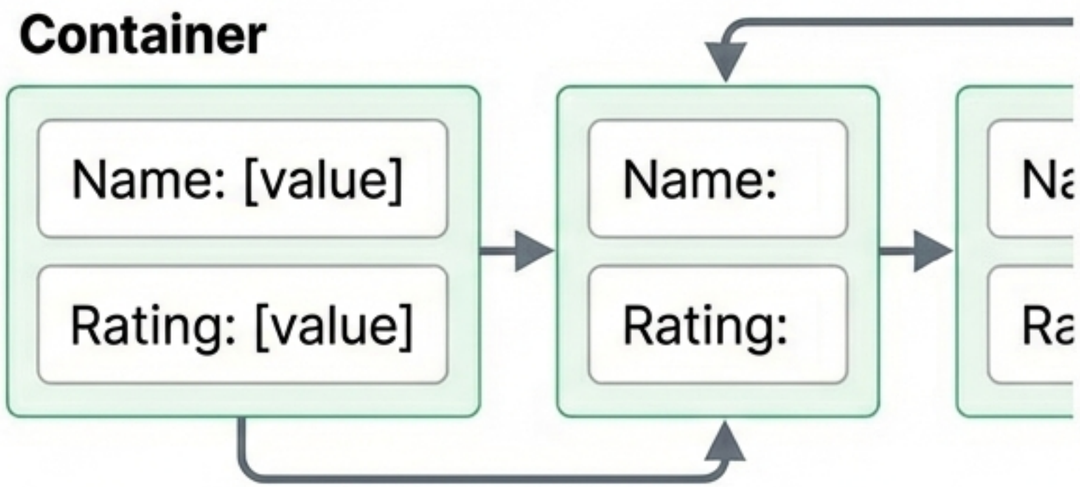
The Strategy: Container vs. Column

Fragile Method



Extracting columns independently risks misalignment if data is missing.

Robust Method



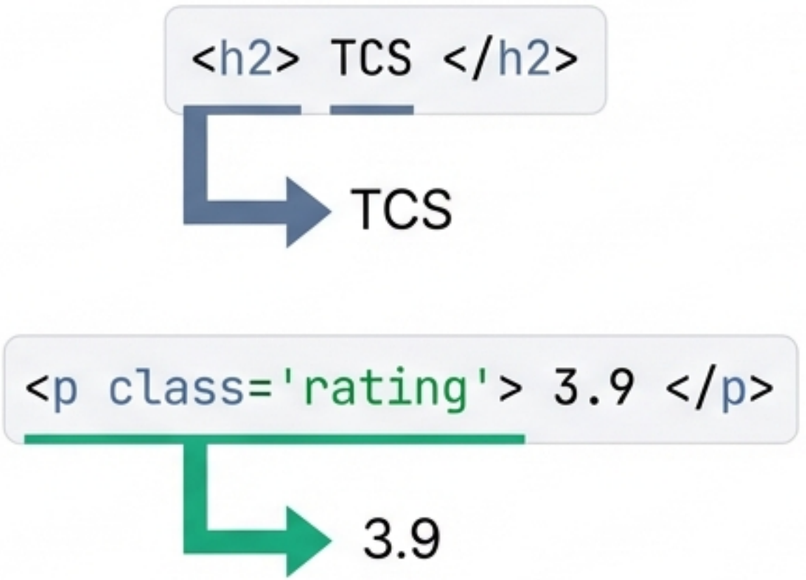
Isolate the ‘Company Card’ container first. Extract data locally within each card.

```
company_cards = soup.find_all('div', class_='company-content-wrapper')
```

Excavation Phase 1: Name and Rating

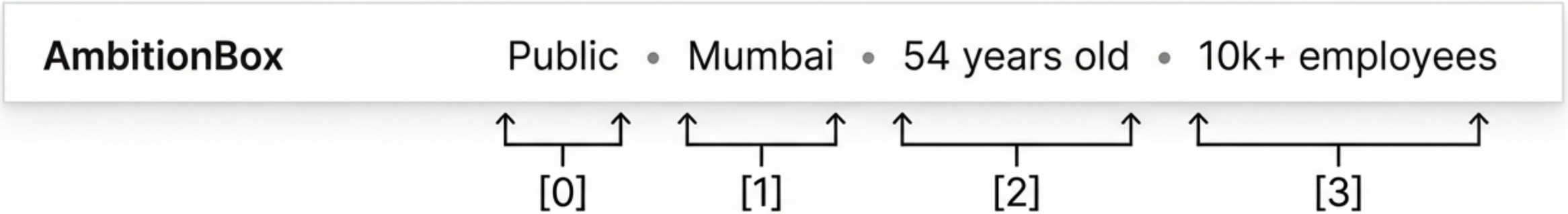
```
for item in company_cards:
    # Extract Company Name
    name = item.find('h2').text.strip()


    # Extract Rating
    rating = item.find('p',
class_='rating').text.strip()
```



- .find() locates the first matching tag inside the container.
- .text.strip() removes HTML tags and whitespace.

Excavation Phase 2: The Info Entities



These items share the exact same HTML tag and class ('p', class='infoEntity'). We cannot target them by unique names.

```
# Get list of all info entities
info = item.find_all('p', class_='infoEntity')

company_type = info[0].text.strip() # Public
hq_location = info[1].text.strip()  # Mumbai
company_age = info[2].text.strip()  # 54 years old
```

→ We use `find_all()` to return a list, then access data by position (index).

Excavation Phase 3: Review Counts

```
reviews = item.find('a', class_='review-count').text.strip()
```

The ‘Reviews’ count is a clickable link, so it **lives inside an anchor** (**<a>**) tag, not a paragraph (**<p>**) tag.

Checkpoint Status

Page Request [OK] → HTML Parse [OK] → Single Card Extraction [OK]

We now have working logic to extract all data points for a single company card.

Scaling Up: Pagination

Outer Loop: Pages 1 to 10

```
for j in range(1, 11):
    url = '...&page={}'.format(j)
```

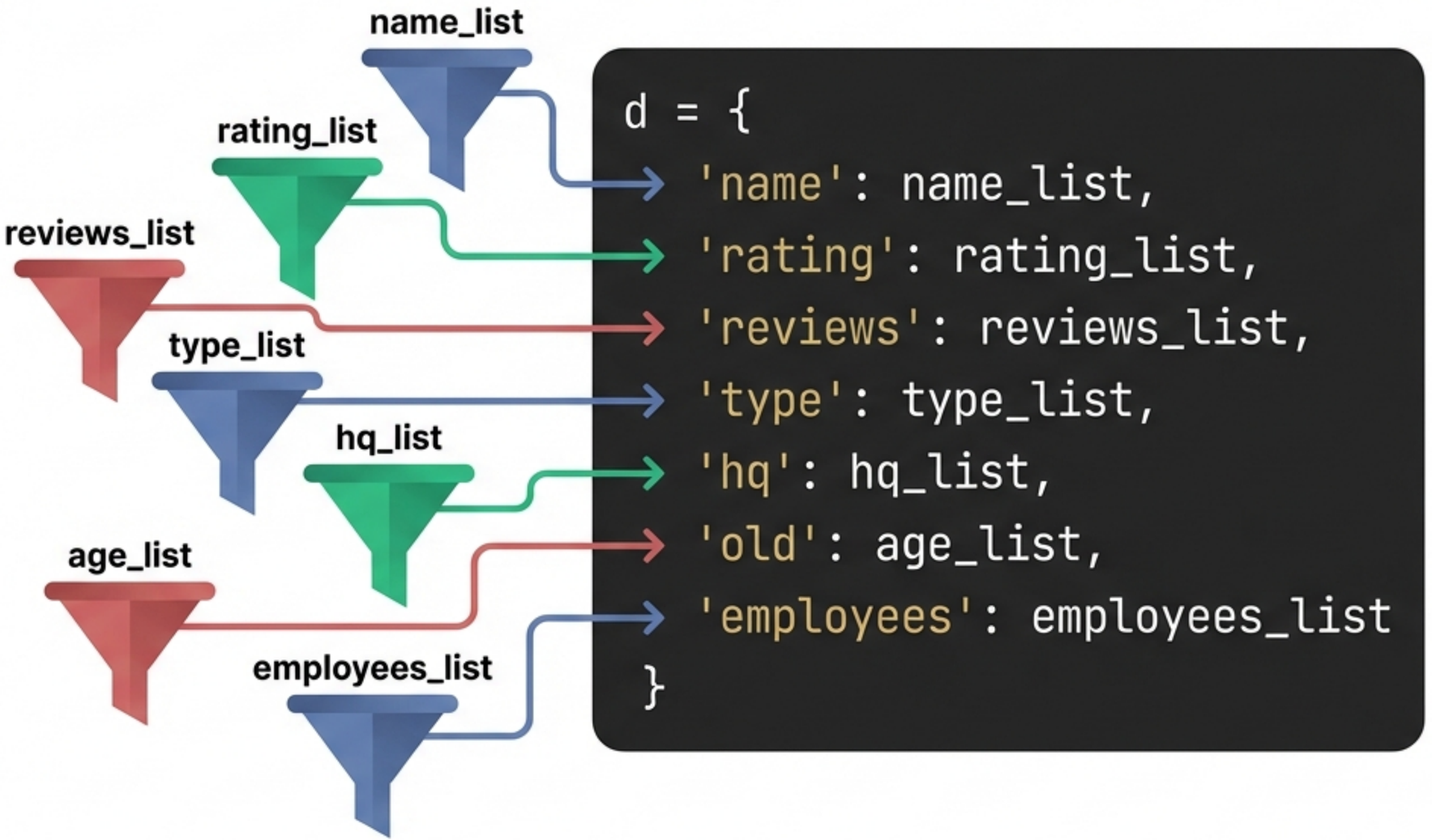
Inner Loop: Company Cards

```
requests.get(url)
soup.find_all('div'...)
Extract & Append Data
```

To scrape the whole site, we wrap our logic in a master loop.

Dynamic URL: The page number in the URL changes with every iteration (`j`), allowing us to traverse pages 1 through 333.

Assembling the Final Structure



During the loop, we appended extracted data to temporary lists.

Now, we assemble those lists into a single Dictionary ‘d’. This dictionary is the raw material required by Pandas.

From Chaos to Order: The DataFrame

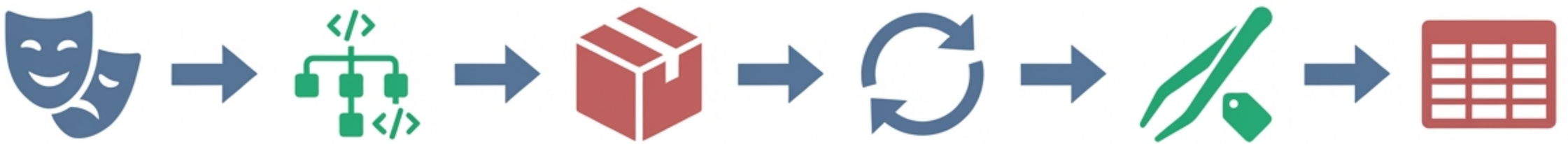
```
df = pd.DataFrame(d)
```

	name	rating	reviews	type	hq	old	employees
1	TCS	3.9	50k	Public	Mumbai	54 yrs	10k+
2	Infosys	4.0	48k	Public	Bangalore	41 yrs	10k+
3	Wipro	3.8	45k	Public	Bangalore	77 yrs	10k+
4	HCLTech	4.1	32k	Public	Noida	46 yrs	10k+
5	Tech Mahindra	3.7	29k	Public	Pune	37 yrs	10k+

Shape (3300 rows × 7 columns)

The final result: Unstructured HTML has been converted into a structured, analytical dataset ready for Data Science projects.

The Workflow Recap



1. Request (User-Agent)
2. Parse (BeautifulSoup)
3. Identify Container (div)
4. Loop Pages & Cards
5. Extract Tags (h2, p, a)
6. Store (DataFrame)

Web scraping empowers us to create our **own datasets** when APIs don’t exist, unlocking infinite possibilities for analysis.