Data Collection Techniques

In the world of data science, data collection is the cornerstone of all analytical efforts. As the saying goes, "Garbage In, Garbage Out"—the quality of insights directly depends on the quality of data collected. Whether you're building machine learning models, performing statistical analysis, or developing AI systems, your success begins with gathering accurate, reliable data.

Understanding Data Sources

Data can come from a variety of sources:

- **APIs** provide structured access to real-time information, such as weather or stock data.
- **Databases** (both SQL and NoSQL) house large volumes of structured data.
- **Webpages** offer valuable unstructured data from blogs, e-commerce platforms, and more.
- **Files** like CSV, JSON, and Excel sheets serve as local or cloud-based data repositories.
- **Sensors and logs** provide real-time inputs from IoT devices and application events.

Ethical Considerations

Ethical data collection is crucial. Always review a website’s Terms of Service and respect `robots.txt` when scraping. Data must be collected with informed consent and in compliance with regulations such as GDPR and CCPA. Attribution, licensing, and data usage rights should never be overlooked.

Tools and Techniques

A variety of tools support data collection:

- **APIs** (REST or GraphQL) for structured, remote data access.
- **Web Scraping** using tools like BeautifulSoup, Scrapy, or Selenium for HTML parsing.
- **Webhooks** for event-based data (e.g., payment notifications).
- **Manual methods** like forms or surveys for crowd-sourced insights.
- **File handling** with Python libraries such as `pandas`, `csv`, and `json`.

For working with databases, Python libraries like `sqlite3`, `SQLAlchemy`, and `pymongo` are widely used to interact with both relational and NoSQL systems.

Final Takeaway

Data collection is more than just gathering information—it’s about doing so responsibly, efficiently, and with the right tools. Python stands out as a versatile language, offering powerful libraries for virtually every type of data source. Mastering these techniques lays a strong foundation for any data science project.

What is Web Scraping?

- **Web Scraping** is the automated process of extracting information from websites.
- Instead of manually copying data, a scraper **reads** and **parses** the webpage's code to collect the needed data.
- Used widely in data science for collecting:

- Product prices
- News articles
- Job listings
- Research datasets

Why Use Web Scraping?

- **Automate** repetitive data collection tasks
- Access data that is **not available via APIs**
- Build datasets for **Machine Learning models** and **Analytics**
- Monitor websites for **price changes**, **content updates**, or **news alerts**

When Not to Use Web Scraping

- If the site offers an **official API**, prefer that (it's cleaner and more stable).
- If scraping **violates** the site's **Terms of Service**.
- If the scraping **harms** the website (excessive requests can cause server overload).

HTML Basics for Web Scraping

What is HTML?

- **HTML (HyperText Markup Language)** structures content on the web.
- It's made up of **elements** (tags) like `<div>`, `<p>`, `<h1>`, `<a>`, etc.

Example:

```html
<html>
  <body>
    <h1>Product Title</h1>
    <p class="price">$29.99</p>
    <a href="/buy-now">Buy Now</a>
  </body>
</html>
```

Important HTML Elements for Scraping

Tag	Meaning	Common Use
`<div>`	Division/Container	Group content
`<p>`	Paragraph	Text blocks
`<h1>`, `<h2>`, etc.	Headings	Titles and sections
`<a>`	Anchor (links)	URLs, navigation

Tag	Meaning	Common Use
`<img>`	Image	Pictures and icons
`<table>`	Table	Structured tabular data

Attributes Matter!

- HTML tags often have **attributes** like `id`, `class`, `href`, `src`.
- We use these attributes to **target** the correct elements.

Example:

```
<p class="price">$29.99</p>
```

Here, the `class="price"` attribute helps identify the price on the page.

Quick CSS Basics for Scraping

What is CSS?

- **CSS (Cascading Style Sheets)** controls the style and layout of HTML elements.
- For scraping, we mainly care about **CSS selectors** to **find and extract** data.

Common CSS Selectors

Selector	Meaning	Example
`.class`	Selects elements by class	`.price`, `.title`
`#id`	Selects an element by id	`#main`, `#product-title`
`tag`	Selects all elements of a tag	`h1`, `div`, `p`
`tag.class`	Selects a tag with class	`p.price`, `div.container`

Example: Selecting Elements

HTML:

```
<p class="price">$29.99</p>
```

CSS selector to target this:

```
p.price
```

In Python using BeautifulSoup:

```
soup.select_one("p.price")
```

Tools We'll Use for Web Scraping

- `requests` → To download the HTML content of a webpage.
- `BeautifulSoup` → To parse and extract data from the HTML.
- **(Optional)** `Selenium` → For websites that load data dynamically with JavaScript.

Summary

- Web scraping automates data extraction from websites.
- Understanding basic **HTML structure** and **CSS selectors** is crucial.
- Python provides powerful libraries to make web scraping easy.

What is HTML

HTML (HyperText Markup Language) is the standard language used to structure content on the web. When you load a web page, your browser interprets HTML to display the layout, text, images, and other content.

Basic Structure of an HTML Document

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <h1>Welcome!</h1>
    <p>This is a sample page.</p>
  </body>
</html>
```

Key Elements

- `<!DOCTYPE html>`: Declares the document type.
- `<html>`: Root element of the page.
- `<head>`: Contains metadata, styles, and scripts.
- `<body>`: Contains visible content.

Common HTML Tags Used in Scraping

Most Frequently Encountered Tags

Tag	Purpose
`<div>`	Section or container for content
`<span>`	Inline container
`<a>`	Anchor tag for hyperlinks
`<img>`	Displays images (uses `src`)
`<ul>`, `<ol>`, `<li>`	Lists and list items
`<table>`, `<tr>`, `<td>`	Tables and cells
`<h1>` to `<h6>`	Headers (various sizes)
`<p>`	Paragraph text
`<form>`, `<input>`, `<button>`	Form elements

Attributes in HTML

HTML tags often include attributes that provide metadata or instructions:

```
<a href="https://example.com" class="nav-link">Visit Site</a>
```

Common Attributes

- `href`: Hyperlink reference
- `src`: Image or media source
- `class`: CSS class (commonly used for scraping)
- `id`: Unique identifier for an element
- `name`: Often used in form elements
- `type`: Used in `<input>` tags

Navigating HTML Structure in Scraping

Scraping tools like BeautifulSoup and Selenium use tag names and attributes to locate elements.

Example Targets

- By tag: `soup.find('div')`
- By class: `soup.find('div', class_='product')`
- By ID: `soup.find(id='header')`
- By attribute: `soup.find('a', {'href': True})`

Understanding Nested Elements

HTML is hierarchical. Tags can contain other tags.

```
<div class="article">
  <h2>Title</h2>
  <p>This is a summary.</p>
</div>
```

To extract both the title and summary, first locate the parent `<div class="article">`, then its children.

Practical Tips

- Use browser DevTools (Right-click > Inspect) to examine HTML structure.
- Target elements with unique `id` or descriptive `class` attributes.
- Use tag nesting logic to extract specific parts of a page.

Useful Python Libraries for HTML Parsing

- `requests`: For sending HTTP requests
- `BeautifulSoup`: For parsing and traversing HTML
- `lxml`: Fast parser for large documents
- `Selenium`: For interacting with JavaScript-rendered pages

Using Requests for Web Scraping

Today we will see how to scrape websites and use requests module to download the raw html of a webpage. In this section we can safely use https://quotes.toscrape.com/ and https://books.toscrape.com/ for scraping demos

1. What is `requests`?

- `requests` is a Python library used to send HTTP requests easily.
- It allows you to fetch the content of a webpage programmatically.
- It is commonly used as the first step before parsing HTML with BeautifulSoup.

---

2. Installing `requests`

To install `requests`, run:

```
pip install requests
```

---

3. Sending a Basic GET Request

Example

```python
import requests

url = "https://example.com"
response = requests.get(url)

# Print the HTML content
print(response.text)
```

Key points:

- `url`: The website you want to fetch.
- `response.text`: The HTML content of the page as a string.

---

4. Checking the Response Status

Always check if the request was successful:

```python
print(response.status_code)
```

Common Status Codes

- `200`: OK (Success)
- `404`: Not Found
- `403`: Forbidden
- `500`: Internal Server Error

Good practice:

```python
if response.status_code == 200:
    print("Page fetched successfully!")
else:
    print("Failed to fetch the page.")
```

---

5. Important Response Properties

Property	Description
`response.text`	HTML content as Unicode text
`response.content`	Raw bytes of the response
`response.status_code`	HTTP status code

Property	Description
`response.headers`	Metadata like content-type, server info

6. Adding Headers to Mimic a Browser

Sometimes websites block automated requests. Adding a `User-Agent` header helps the request look like it is coming from a real browser.

```
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)"
}

response = requests.get(url, headers=headers)
```

7. Handling Connection Errors

Wrap your request in a try-except block to handle errors gracefully:

```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()  # Raises an HTTPError for bad responses
    print(response.text)
except requests.exceptions.RequestException as e:
    print(f"An error occurred: {e}")
```

8. Best Practices for Fetching Pages

- Always check the HTTP status code.
- Use proper headers to mimic a browser.
- Set a timeout to avoid hanging indefinitely.
- Respect the website by not making too many rapid requests.

9. Summary

- `requests` makes it simple to fetch web pages using Python.
- It is the starting point for most web scraping workflows.
- Combining `requests` with BeautifulSoup allows for powerful data extraction.

Using BeautifulSoup for Web Scraping

1. What is BeautifulSoup?

- **BeautifulSoup** is a Python library used to parse HTML and XML documents.
- It creates a **parse tree** from page content, making it easy to extract data.
- It is often used with `requests` to scrape websites.

---

2. Installing BeautifulSoup

Install both **`beautifulsoup4`** and a parser like **`lxml`**:

```
pip install beautifulsoup4 lxml
```

---

3. Creating a BeautifulSoup Object

Example

```python
from bs4 import BeautifulSoup
import requests

url = "https://example.com"
response = requests.get(url)

soup = BeautifulSoup(response.text, "lxml")
```

- `response.text`: HTML content.
- `"lxml"`: A fast and powerful parser (you can also use `"html.parser"`).

---

4. Understanding the HTML Structure

BeautifulSoup treats the page like a tree.
You can search and navigate through **tags**, **classes**, **ids**, and **attributes**.

Example HTML:

```html
<html>
  <body>
    <h1>Title</h1>
    <p class="description">This is a paragraph.</p>
    <a href="/page">Read more</a>
  </body>
</html>
```

---

5. Common Methods in BeautifulSoup

5.1 Accessing Elements

- Access the **first occurrence** of a tag:

```
soup.h1
```

- Get the **text** inside a tag:

```
soup.h1.text
```

5.2 `find()` Method

- Finds the **first matching** element:

```
soup.find("p")
```

- Find a tag with specific attributes:

```
soup.find("p", class_="description")
```

5.3 `find_all()` Method

- Finds **all matching** elements:

```
soup.find_all("a")
```

5.4 Using `select()` and `select_one()`

- Select elements using **CSS selectors**.

```
soup.select_one("p.description")
```

```
soup.select("a")
```

---

6. Extracting Attributes

Get the value of an attribute, such as `href` from an `<a>` tag:

```
link = soup.find("a")
print(link["href"])
```

Or using `.get()`:

```
print(link.get("href"))
```

---

7. Traversing the Tree

- Access parent elements:

```
soup.p.parent
```

- Access children elements:

```
list(soup.body.children)
```

- Find the next sibling:

```
soup.h1.find_next_sibling()
```

---

8. Handling Missing Elements Safely

Always check if an element exists before accessing it:

```
title_tag = soup.find("h1")
if title_tag:
    print(title_tag.text)
else:
    print("Title not found")
```

---

9. Summary

- BeautifulSoup helps parse and navigate HTML easily.
- Use `.find()`, `.find_all()`, `.select()`, and `.select_one()` to locate data.
- Always inspect the website's structure before writing scraping logic.

- Combine BeautifulSoup with `requests` for full scraping workflows.