Python Web Scraping for Beginners

BeautifulSoup & Requests — Complete Course Notes

SECTION 1: Introduction to Web Scraping

1.1 What is Web Scraping?

Web scraping is the process of automatically extracting data from websites using code. Instead of manually copying and pasting, you write a script that fetches the raw HTML of a webpage and pulls out exactly the data you need.

- Used for: price monitoring, research data collection, building datasets, news aggregation, and more
- This course covers the beginner-level approach using two core Python libraries: BeautifulSoup and requests
- More advanced libraries (e.g. Scrapy, Selenium) exist for dynamic or JavaScript-heavy pages and will be covered in future series

1.2 Course Overview & Structure

This series is designed as a mini-series within a broader Python tutorial. Each lesson builds on the last:

- Lesson 1 — Introduction to HTML: understanding the structure of web pages
- Lesson 2 — Inspecting a real website: using browser developer tools
- Lesson 3 — BeautifulSoup & requests: importing libraries and fetching HTML
- Lesson 4 — find() and find_all(): extracting specific data using tags, classes, and attributes
- Lesson 5 — Variable strings: extracting text content from HTML elements
- Lesson 6–8 — Full project: scraping a Wikipedia table into a pandas Data Frame and exporting to CSV

1.3 Tools & Environment

- Language: Python 3
- Environment: Jupyter Notebook (via Anaconda) — recommended for this series
- Run each cell with Shift + Enter
- Libraries required: bs4 (BeautifulSoup) and requests
  - If using Anaconda/Jupyter, both are typically pre-installed
  - If not: open your terminal and run: pip install bs4 requests

SECTION 2: HTML Fundamentals

2.1 What is HTML?

HTML (Hypertext Markup Language) is the standard language used to describe and structure all of the elements on a web page. To scrape data from a website, you must understand how HTML is structured so you can identify exactly where your target data lives.

2.2 Basic HTML Structure

HTML is built from nested tags enclosed in angled brackets. Every opening tag has a corresponding closing tag (with a forward slash). The content between tags is what gets displayed in the browser.

```
<html>
  <head>
    <title>My First Web Page</title>
  </head>
  <body>
    <p>Hello world</p>
  </body>
</html>
```

- The <html> tag wraps the entire document
- The <head> section contains metadata (title, links to stylesheets, etc.)
- The <body> section contains everything visible on the page
- Tags create a hierarchy — child elements are nested inside parent elements

2.3 Key HTML Tags for Web Scraping

Tag	Full Name / Purpose	Example Use
`<html>`	Root element — wraps entire page	Outermost tag on every page
`<head>`	Metadata section	Title, CSS links
`<body>`	Visible page content	All text, images, tables
`<p>`	Paragraph — text content	Paragraphs, descriptions
`<div>`	Division / container block	Grouping related content
`<a>`	Anchor — hyperlink	Links (href attribute holds URL)
`<table>`	Table container	Wraps an entire data table
`<tr>`	Table row	One row of data in a table
`<th>`	Table header cell	Column titles / headings
`<td>`	Table data cell	Actual data in each row
`<h1>-<h6>`	Headings (largest to smallest)	Page titles, section headers

2.4 Attributes — Class, ID, and href

HTML tags can carry additional information called attributes. These are key-value pairs inside the opening tag and are essential for targeted scraping.

- class — groups elements that share the same style or role. Multiple elements can share a class. Example:
- id — a unique identifier for one specific element on the page. Example:
- href — used inside <a> tags to store the URL of a hyperlink. Example: Click here

When scraping, class and id are the most useful attributes for filtering down to the exact element you want.

2.5 Variable Strings

A variable string is the actual text content that lives between an HTML tag's opening and closing brackets. This is typically the data you want to extract when scraping.

```
<title>My First Web Page</title>
```

In this example, 'My First Web Page' is the variable string. In BeautifulSoup, you extract it using .text or .get_text().

SECTION 3: Inspecting a Real Website

3.1 Why Use Browser Developer Tools?

Real websites (like Wikipedia or scrape.site) are far more complex than simple HTML examples. Browser developer tools let you view the live HTML structure of any webpage and identify exactly where your target data is located — without guessing.

3.2 How to Open the Inspector

- Right-click anywhere on the webpage > select Inspect (or Inspect Element)
- Alternatively: press F12 (Windows) or Cmd + Option + I (Mac)
- The HTML panel appears — this is the raw structure of the page

3.3 Using the Element Selector Tool

The most efficient way to find a specific piece of data in the HTML is to use the element selector:

- Click the cursor/arrow icon at the top-left of the DevTools panel (or press Ctrl+Shift+C)
- Hover over any element on the visible webpage — the corresponding HTML is highlighted in the panel
- Click any element to jump directly to its location in the HTML tree

Example: clicking on 'Boston Bruins' in a hockey table jumps directly to the <td> tag containing that text. This reveals the exact tag name, class, and hierarchy you need to write your scraping code.

3.4 Reading the HTML Hierarchy

When you find your target element in the inspector, work upward through the hierarchy to understand the full path:

- Is the data inside a <td> tag? A <tr>? A <table> with a specific class?
- What attributes does the containing element have? (class, id, href?)
- Are there multiple elements with the same tag/class? You may need to use indexing or more specific attributes.

Example hierarchy for a table cell:

```
<body>
  <table class="table">
    <tr>
      <td class="team">Boston Bruins</td>
    </tr>
  </table>
</body>
```

This tells you: search for a <td> tag with class 'team' inside a <table> with class 'table'.

SECTION 4: BeautifulSoup & requests — Fetching HTML

4.1 Importing the Libraries

Before anything else, import both libraries at the top of your Jupyter notebook:

```
from bs4 import BeautifulSoup
import requests
```

- bs4 is the module name; BeautifulSoup is the specific class within it
- requests is a standalone library for making HTTP requests
- Run with Shift + Enter — if you see no errors, the import succeeded

4.2 Assigning the URL

Always store your target URL in a variable so you can reuse it throughout your script:

```
url = 'https://scrape.site.com'
```

- Copy the URL from your browser's address bar
- Assign it to a variable (url is convention but any name works)

4.3 Sending a GET Request

Use requests.get() to send an HTTP GET request to the URL and retrieve the page:

```
page = requests.get(url)
```

- This sends a request to the web server and receives a response object
- The response object is stored in the variable page

Checking the response status code:

```
print(page)    # e.g. <Response [200]>
```

Status Code	Meaning
200	Success — page returned correctly. This is what you want.
204	No Content — request was received but there is no content to return
400	Bad Request — invalid request; the server could not process it
401	Unauthorized — authentication required or failed
404	Not Found — the server could not find the requested page

4.4 Parsing HTML with BeautifulSoup

Once you have the page response, extract the raw HTML text and pass it to BeautifulSoup for parsing:

```
soup = BeautifulSoup(page.text, 'html.parser')
```

- page.text — retrieves the raw HTML of the page as a Python string (a snapshot of the page at that moment)
- 'html.parser' — tells BeautifulSoup to parse the content as HTML. This is Python's built-in HTML parser and requires no extra installation.
- The result (soup) is a BeautifulSoup object — a searchable, structured representation of the HTML
- The variable is conventionally called soup — named after the library (the creator described it as making 'messy HTML beautiful')

4.5 Viewing the Parsed HTML

Two ways to inspect what BeautifulSoup has parsed:

- print(soup) — prints the raw HTML without indentation (harder to read but functionally identical)
- print(soup.prettify()) — prints the HTML with hierarchical indentation, making it much easier to visualise the structure visually

Note: prettify() is useful for visual inspection only — it does not affect how you query or search the HTML.

SECTION 5: Searching HTML — find() and find_all()

5.1 Overview

Once your HTML is stored in the soup object, you use find() and find_all() to locate and extract specific elements. These are the two core methods for querying HTML in BeautifulSoup.

5.2 soup.find() — First Match Only

find() searches through the HTML and returns only the FIRST matching element it encounters:

```
result = soup.find('div')   # returns first <div> tag found
```

- Returns a single BeautifulSoup Tag object (not a list)
- If no match is found, it returns None
- Useful when you know there is only one instance of what you're looking for, or when you want to extract text from a specific element

5.3 soup.find_all() — All Matches

find_all() searches through the HTML and returns ALL matching elements as a Python list:

```
results = soup.find_all('div')   # returns list of all <div> tags
```

- Returns a ResultSet (behaves like a list)
- Each item in the list is a BeautifulSoup Tag object
- If no matches are found, returns an empty list []
- Most commonly used method in practice — gives you everything, then you filter further

5.4 Filtering by Tag and Class

Searching by tag alone often returns too many results. Narrow results by also specifying a CSS class:

```
# Search by tag name only
soup.find_all('p')


# Search by tag AND class (use class_ to avoid conflict with Python's class
keyword)
soup.find_all('div', class_='col-md-12')
soup.find('p', class_='lead')
```

- Always pair the tag name with the class when filtering
- class_ (with underscore) is used in Python to avoid clashing with Python's built-in class keyword
- The class value must match exactly what appears in the HTML (copy it from the inspector)
- IDs can also be used: soup.find('div', id='main-content')

5.5 Searching by Attribute (href)

You can search for elements by any attribute, including href for hyperlinks:

```
# Find all anchor tags with an href attribute
links = soup.find_all('a')

# Access the href attribute value
for link in links:
    print(link.get('href'))
```

- <a> tags (anchor tags) contain hyperlinks; the href attribute holds the URL
- Attributes like class, id, and href are all searchable via BeautifulSoup

5.6 Extracting Text with .text and .strip()

Once you have found your target element, extract the visible text content using .text:

```
# Using find() — returns a single element, .text works
element = soup.find('p', class_='lead')
print(element.text)           # raw text with possible whitespace
print(element.text.strip())   # clean text with whitespace removed
```

- .text — returns all the text content inside the tag as a string
- .strip() — removes leading and trailing whitespace and newline characters (\n)
- IMPORTANT: .text does NOT work on the result of find_all() because find_all() returns a list, not a single element. Switch to find() when you want to extract text from one specific element.

Common error to remember:

```
# This will FAIL:
result = soup.find_all('p', class_='lead')
print(result.text)   # AttributeError: ResultSet has no .text attribute

# This works:
result = soup.find('p', class_='lead')
print(result.text.strip())   # works correctly
```

SECTION 6: Scraping Table Data — Practical Techniques

6.1 Understanding Table Structure in HTML

HTML tables follow a strict nested structure. Understanding this hierarchy is essential for correctly targeting the data you want:

```
<table class="wikitable sortable">
  <tr>
    <th>Rank</th>  <th>Name</th>  <th>Industry</th>   <!-- column headers -->
  </tr>
  <tr>
    <td>1</td>  <td>Walmart</td>  <td>Retail</td>      <!-- data row -->
  </tr>
</table>
```

- <table> — wraps the entire table
- <tr> — each row of the table (including the header row)
- <th> — header cells (column titles). Only in the first row.
- <td> — data cells. Every data row contains <td> tags, one per column.

Key insight: <th> tags appear once (headers); <td> tags repeat for every data row. This distinction is critical for separating headers from data.

6.2 Handling Multiple Tables on One Page

Many real web pages (like Wikipedia) contain more than one table. Searching with soup.find('table') only returns the first table found — which may not be the one you want.

Strategy 1 — Use find_all() with indexing:

```
tables = soup.find_all('table')
target_table = tables[1]   # index 1 = second table on the page
```

Strategy 2 — Use find() with a class attribute:

```
target_table = soup.find('table', class_='wikitable sortable')
```

- find_all() returns a list — use Python list indexing [0], [1], etc. to select the right table
- Index 0 = first table, index 1 = second table, and so on
- Always inspect the page to verify which table index contains your data
- IMPORTANT: Once you have isolated your target table, run all further find() and find_all() calls on the table object — NOT on soup. Running on soup searches the entire page and may pull data from other tables.

```
# WRONG — searches entire page, may include other tables:
headers = soup.find_all('th')


# CORRECT — searches only within your target table:
headers = target_table.find_all('th')
```

6.3 Extracting Table Headers Using List Comprehension

After isolating your table, extract the column headers from the <th> tags using list comprehension for clean, efficient code:

```
# Step 1: get all <th> elements from the table
raw_titles = target_table.find_all('th')


# Step 2: extract and clean the text using list comprehension
world_table_titles = [title.text.strip() for title in raw_titles]
print(world_table_titles)
# Output: ['Rank', 'Name', 'Industry', 'Revenue', 'Employees', 'Headquarters']
```

- List comprehension: [expression for item in iterable] — a compact loop that builds a list
- .text.strip() — extracts text and removes surrounding whitespace in one step
- The result is a clean Python list of column names, ready to use as DataFrame column headers

6.4 Extracting Row Data Using Nested Loops

Each data row is a <tr> tag containing multiple <td> tags. You need two loops: one to iterate rows, one to iterate cells within each row.

```
# Get all rows from the table
column_data = target_table.find_all('tr')


# Loop through each row (skip row 0 — it's the header row)
for row in column_data[1:]:
    row_data = row.find_all('td')   # get all <td> cells in this row
    individual_row_data = [data.text.strip() for data in row_data]
    print(individual_row_data)
```

- column_data = target_table.find_all('tr') — gets all rows including the header row
- column_data[1:] — Python list slicing; skips index 0 (the header row with <th> tags)
- row.find_all('td') — gets all data cells within the current row
- List comprehension again extracts and cleans text from each <td>
- The result is individual_row_data — a list with one item per column for that row

Why slicing matters: if you include the header row (index 0), it produces an empty list for that row's <td> tags (since headers use <th>), which causes a 'mismatched columns' error when loading into a DataFrame.

SECTION 7: Loading into pandas & Exporting to CSV

7.1 What is a DataFrame?

A pandas DataFrame is a two-dimensional table structure — like a spreadsheet — that you can manipulate, filter, sort, and analyse entirely within Python. Once your scraped data is in a DataFrame, you have access to the entire pandas toolkit.

- import pandas as pd — the standard alias for pandas
- Each column corresponds to one field from your scraped table
- Each row corresponds to one record from your scraped data

7.2 Creating an Empty DataFrame with Column Headers

Before filling the DataFrame with data, create it with the correct column names using the headers you extracted:

```
import pandas as pd


df = pd.DataFrame(columns=world_table_titles)
print(df)    # empty DataFrame with the right column headers
```

- pd.DataFrame(columns=...) creates an empty DataFrame with named columns
- world_table_titles is the list of header strings you extracted from the <th> tags
- Creating the empty shell first ensures your columns are set up correctly before data is added

7.3 Appending Rows to the DataFrame Inside a Loop

The challenge: each iteration of the loop produces one row of data. A simple assignment (df = individual_row_data) would overwrite the previous row. Instead, you append each new row to the next available position using df.loc[]:

```
for row in column_data[1:]:
    row_data = row.find_all('td')
    individual_row_data = [data.text.strip() for data in row_data]

    length = len(df)              # current number of rows
    df.loc[length] = individual_row_data  # append at next position
```

- len(df) — returns the current number of rows in the DataFrame
- df.loc[length] — targets the row at index position equal to the current length
- Since the DataFrame starts empty (length = 0), first row goes to index 0, second to index 1, and so on
- Each loop iteration appends one row, so the DataFrame grows row by row

How df.loc[] works as an appender: when you assign to an index that doesn't yet exist in the DataFrame, pandas creates a new row at that position. Since len(df) always equals the next available index, this effectively appends each row in sequence.

7.4 Verifying the DataFrame

After the loop completes, print the DataFrame to verify all data was captured correctly:

```
print(df)
```

- Check that the number of rows matches the number of data rows on the original webpage
- Columns with '...' in the output simply mean the content is too wide to display — the data is still there
- Use df.shape to see the dimensions: (rows, columns)
- Use df.head() to see the first 5 rows

7.5 Exporting to CSV

Once your DataFrame is built, export it to a CSV file for saving, sharing, or further analysis:

```
df.to_csv(r'C:\path\to\output\companies.csv', index=False)
```

- df.to_csv('filepath') — writes the DataFrame to a comma-separated values file
- The r before the string creates a raw string — backslashes are treated as literal characters (important on Windows)
- The filepath specifies where to save the file and what to name it (include .csv extension)
- index=False — CRITICAL: this prevents pandas from writing the DataFrame's internal row numbers (0, 1, 2...) as an extra column in the CSV. Without this, your output has an unwanted index column.

Verifying the output:

- Open the CSV file in Excel or a text editor to confirm the data looks correct
- All columns should match your DataFrame headers
- No extra index column should appear

SECTION 8: Full Project — Wikipedia Scraping Walkthrough

8.1 Project Goal

Scrape the 'List of largest companies in the United States by revenue' table from Wikipedia and export it to a CSV file. This project brings together every skill covered in the course.

- Target site: Wikipedia — List of largest US companies by revenue
- Target data: Rank, Name, Industry, Revenue, Revenue Growth, Employees, Headquarters
- Output: a clean pandas DataFrame exported as a .csv file

8.2 Complete Code Walkthrough

Step 1 — Import libraries:

```
from bs4 import BeautifulSoup
import requests
import pandas as pd
```

Step 2 — Set URL and fetch HTML:

```
url = 'https://en.wikipedia.org/wiki/List_of_largest_companies...'
page = requests.get(url)
soup = BeautifulSoup(page.text, 'html.parser')
```

Step 3 — Find the correct table (page has 2 tables with similar class names):

```
# Option A: use find_all and select by index
tables = soup.find_all('table', class_='wikitable sortable')
table = tables[0]   # the first wikitable sortable IS the target

# Option B: use find with class
table = soup.find('table', class_='wikitable sortable')
```

Step 4 — Extract column headers:

```
world_titles = table.find_all('th')   # search table, NOT soup
world_table_titles = [title.text.strip() for title in world_titles]
```

Step 5 — Create empty DataFrame:

```
df = pd.DataFrame(columns=world_table_titles)
```

Step 6 — Loop through rows, extract data, append to DataFrame:

```
column_data = table.find_all('tr')
for row in column_data[1:]:          # skip header row
    row_data = row.find_all('td')
    individual_row_data = [data.text.strip() for data in row_data]
    length = len(df)
    df.loc[length] = individual_row_data
```

Step 7 — Verify and export:

```
print(df)
df.to_csv(r'C:\Users\...\companies.csv', index=False)
```

8.3 Common Errors & How to Fix Them

Error / Problem	Cause	Fix
AttributeError: .text on find_all()	find_all() returns a list; lists don't have .text	Switch to find() for single-element text extraction
Wrong table selected	soup.find('table') returns first match, not desired table	Use find_all() + indexing [1], or filter by class name
Extra column in CSV	DataFrame index written to file by default	Add index=False to df.to_csv()
'Mismatched columns' error	First row (header row) included in data loop — returns empty list for <td>	Use column_data[1:] to skip the header row
Wrong headers / extra headers	find_all('th') on soup pulls <th> from ALL tables	Always call find_all('th') on table, not soup
ModuleNotFoundError: bs4	bs4 not installed	Run: pip install bs4 in terminal
Response code 404	URL is incorrect or page has moved	Verify the URL in your browser first

8.4 Full Workflow Summary

- 1. Import BeautifulSoup, requests, and pandas
- 2. Assign the target URL to a variable
- 3. requests.get(url) — send GET request; verify response is 200
- 4. BeautifulSoup(page.text, 'html.parser') — parse the HTML into soup
- 5. Inspect the page in browser DevTools to find your table and its class
- 6. Isolate the correct table using find_all() + indexing or find() + class
- 7. Extract <th> tags from the TABLE (not soup) → list comprehension → clean list of column headers
- 8. Create empty DataFrame: pd.DataFrame(columns=headers)
- 9. find_all('tr') on the table; loop through rows using [1:] to skip the header
- 10. Within each row: find_all('td') → list comprehension with .text.strip()
- 11. df.loc[len(df)] = individual_row_data — append each row to the DataFrame
- 12. print(df) to verify; df.to_csv('path/file.csv', index=False) to export

— End of Course Notes —