Intermediate Data Science

Web Scraping

Joanna Bieri DATA201

Important Information

- Email: joanna_bieri@redlands.edu
- Office Hours take place in Duke 209 – Office Hours Schedule
- Class Website
- Syllabus

Web Scraping

Web scraping is the process of automatically extracting data from websites using code. It often involves sending HTTP requests, retrieving HTML content, and parsing it to collect specific information such as text, images, or structured data. Web scraping can be useful for research, business intelligence, price comparison, academic projects, or building datasets that are not otherwise publicly available in a structured form.

Ethical Issues

There are some things to be aware of before you start scraping data from the web.

- Some data is private or protected.
- Some websites have rules against scraping and will cut of service to users who are clearly scraping data.
- The line between web scraping and plagiarism can be very blurry.
- Ethics are different depending on if you are using the data for a personal project vs if you are using the project for your business or website

Ethical Issues

The Ethical Scraper (from https://towardsdatascience.com/ethics-in-web-scraping-b96b18136f01):

Ethical Issues

I, the web scraper will live by the following principles:

- If you have a public API that provides the data I'm looking for, I'll use it and avoid scraping all together.
- I will always provide a User Agent string that makes my intentions clear and provides a way for you to contact me with questions or concerns.
- I will request data at a reasonable rate. I will strive to never be confused for a DDoS attack.
- I will only save the data I absolutely need from your page. If all I need it OpenGraph meta-data, that's all I'll keep.

Ethical Issues

I, the web scraper will live by the following principles:

- I will respect any content I do keep. I'll never pass it off as my own.
- I will look for ways to return value to you. Maybe I can drive some (real) traffic to your site or credit you in an article or post.
- I will respond in a timely fashion to your outreach and work with you towards a resolution.
- I will scrape for the purpose of creating new value from the data, not to duplicate it.

Basics of HTML for Web Scraping

Web scraping relies on understanding the structure of **HTML (HyperText Markup Language)**, since most web pages present their content using HTML tags. Each page is built from a tree-like structure called the **DOM (Document Object Model)**, which organizes elements hierarchically.

Key HTML Elements

- `<html>`: The root element of an HTML page.
- `<head>`: Contains metadata (title, links to CSS/JS).
- `<body>`: Holds the visible content of the page.

Common Tags

- **Headings (`<h1>`, `<h2>`, ... `<h6>`)**: Define section titles.
- **Paragraph (`<p>`)**: Holds blocks of text.
- **Links (`<a href="...">`)**: Anchor tags contain hyperlinks.
- **Images (`<img src="...">`)**: Embed images via a source URL.
- **Lists (`<ul>`, `<ol>`, `<li>`)**: Ordered and unordered lists.
- **Tables (`<table>`, `<tr>`, `<td>`)**: Represent structured tabular data.
- **Divisions (`<div>`)**: Generic container often used for layout.
- **Spans (`<span>`)**: Inline container for styling or grouping.

Attributes

HTML tags often include **attributes** that provide additional information: - `id`: Unique identifier for an element.
- `class`: Groups elements with the same style or purpose.
- `href` (in <a>): Specifies the destination URL.
- `src` (in <img>): Provides the image source.

Importance for Web Scraping

- Scraping tools (like **BeautifulSoup**, **lxml**, or **Selenium**) locate elements by tags, attributes, or text.
- Common methods include selecting by `id`, `class`, or tag type.
- Understanding the **nested structure** of HTML helps identify where the target data resides in the DOM.

Basics of HTML for Web Scraping

Example Snippet:

```
<div class="product">
  <h2 class="title">Book Title</h2>
  <span class="price">$19.99</span>
</div>
```

Challenges in Creating Web Scraping Code

While web scraping is a powerful technique for gathering data, developers often face several significant challenges when implementing it:

Challenges in Creating Web Scraping Code

1. Changing Website Structures

- Websites frequently update their HTML layout or CSS classes.
- Even small changes (e.g., renaming a `class` or moving content into a new `<div>`) can break scraping scripts.
- This requires ongoing maintenance and monitoring.

Challenges in Creating Web Scraping Code

2. JavaScript-Rendered Content

- Many modern sites load data dynamically using JavaScript (AJAX requests, React, Angular, Vue).
- Static HTML parsers (like BeautifulSoup) cannot capture this directly.
- Developers may need to use tools like **Selenium**, **Playwright**, or network traffic analysis to retrieve dynamic content.

Challenges in Creating Web Scraping Code

3. Anti-Scraping Measures

- Websites often deploy techniques to detect and block scrapers:
  - CAPTCHAs
  - Rate limiting
  - IP blocking or blacklisting
  - Bot detection systems
- Workarounds may involve rotating proxies, adding delays, or mimicking real browser behavior.

Challenges in Creating Web Scraping Code

4. Data Quality and Cleaning

- Extracted data is often messy (inconsistent formats, missing values, duplicates).
- Developers must spend significant time cleaning, normalizing, and validating scraped data before analysis.

Start with the URL

The first thing you should inspect when scraping code is the website URL:

```
https://realpython.github.io/fake-jobs/jobs/
                    senior-python-developer-0.html
```

Start with the URL

The URL has two major parts:

- The base: https://realpython.github.io
- The path: /fake-jobs/jobs/senior-python-developer-0.html

Start with the URL

The URL might also have query or pagination information included.

1. Pagination in URLs

Pagination often appears when content is split across multiple pages.
Examples: - `https://example.com/products?page=1` - `https://example.com/products?page=2` - `https://example.com/articles?page=5&sort=latest` - `https://shop.example.com/category/shoes?p=3`
Here, the parameter `page` (or sometimes `p`) changes to load different parts of the dataset.

Start with the URL

2. Query Parameters in URLs

Query parameters are key–value pairs that appear after a `?` in the URL. Multiple parameters are separated by `&`.

Examples: - `https://example.com/search?q=laptop` - `https://example.com/search?q=laptop&sort=price_asc` - `https://example.com/api/items?category=books&limit=20&page=2` - `https://news.example.com/archive?year=2023&month=10`

Developer Tools

The next thing you want to do is inspect the website with your browsers developer tools. Developer tools let you see the HTML code that the website uses.

Developer Tools

Opening Developer Tools

- **Mac:**
  - Press `Cmd + Option + I`
  - Or right-click on the page and choose **Inspect**
- **Windows:**
  - Press `Ctrl + Shift + I` or `F12`
  - Or right-click on the page and choose **Inspect**
- **Linux:**
  - Press `Ctrl + Shift + I` or `F12`
  - Or right-click on the page and choose **Inspect**

Developer Tools

Try this! Navigate to https://realpython.github.io/fake-jobs/ and look at the code.

Click on the drop down arrows to investigate the parts of the code. The browser should highlight the parts of the website that the code controls when you hover over the code. What do you notice about the organization of this site?

Scrape the HTML Content

Install the requests package.

!conda install -c conda-forge -y requests

Scrape the HTML Content

When we call `requests.get()` it grabs whatever information the website sends back. Sometimes this is really straightforward and sometimes you have to deal with issues like JavaScript code and CAPTCHA's.

Static vs. Dynamic Websites in Web Scraping

When scraping websites, it's important to know whether the site is **static** or **dynamic**, since this affects how data is loaded and how it can be extracted.

Static vs. Dynamic Websites in Web Scraping

Static Websites

- **Content delivery:** HTML content is fully loaded when the page is requested from the server.
- **Scraping approach:** The desired data is usually visible in the page's source code and can be extracted with tools like **requests + BeautifulSoup**.
- **Example:** A blog where all posts are present in the HTML file.
- **Advantage:** Simple and predictable structure, easier to scrape.

Static vs. Dynamic Websites in Web Scraping

Dynamic Websites

- **Content delivery:** The initial HTML may be minimal, and content is loaded later using JavaScript (e.g., AJAX, React, Angular).
- **Scraping approach:** The target data may not appear in the raw HTML source; instead it must be captured by:
  - Executing JavaScript (using **Selenium**, **Playwright**, or **Puppeteer**)
  - Inspecting the **Network panel** for API requests returning JSON data
- **Example:** An e-commerce site that loads more products as you scroll.
- **Challenge:** Requires more advanced tools and often more computing resources.

Static vs. Dynamic Websites in Web Scraping

In scraping practice: Always check whether the content is present in the HTML source or if it only appears after interaction/JavaScript execution.

Parse the HTML code

Static Example

Now that you have the code as one big string in page.text you need a way to parse it. Beautiful Soup is a Python library for parsing structured data.

Parse the HTML code

```
!conda install -c conda-forge -y beautifulsoup4
from bs4 import BeautifulSoup
```

Parse the HTML code

Example HTML

```
html = """
<div class="product" id="p1">
  <h2 class="title">Book One</h2>
  <span class="price">$19.99</span>
  <a href="https://example.com/book1" class="buy-link">Buy</a>
</div>
<div class="product" id="p2">
  <h2 class="title">Book Two</h2>
  <span class="price">$24.99</span>
  <a href="https://example.com/book2" class="buy-link">Buy</a>
</div>
<img src="cover1.jpg" alt="Cover 1">
<img src="cover2.jpg" alt="Cover 2">
"""
```

Parse the HTML code

Parse HTML

```
example_soup = BeautifulSoup(html, "html.parser")
```

Parse the HTML code

Find by tag name*

```
info = example_soup.find_all("h2")
for i in info:
    print(i)
    print(i.name)
    print(i.attrs)
    print(i.text)
    print('--------------------')
```

Parse the HTML code

```
<h2 class="title">Book One</h2>
h2
{'class': ['title']}
Book One
--------------------
<h2 class="title">Book Two</h2>
h2
{'class': ['title']}
Book Two
--------------------
```

Parse the HTML code

Find by class

```
first_product = example_soup.find("div", class_="product")
print(first_product)
print('--------------------')
all_products = example_soup.find_all("div", class_="product")
print(all_products)
```

Parse the HTML code

```
<div class="product" id="p1">
<h2 class="title">Book One</h2>
<span class="price">$19.99</span>
<a class="buy-link" href="https://example.com/book1">Buy</a>
</div>
--------------------
[<div class="product" id="p1">
<h2 class="title">Book One</h2>
<span class="price">$19.99</span>
<a class="buy-link" href="https://example.com/book1">Buy</a>
</div>, <div class="product" id="p2">
<h2 class="title">Book Two</h2>
<span class="price">$24.99</span>
<a class="buy-link" href="https://example.com/book2">Buy</a>
</div>]
```

Parse the HTML code

Find by id

```
product_p1 = example_soup.find("div", id="p1")
print(product_p1)
```

Parse the HTML code

```
<div class="product" id="p1">
<h2 class="title">Book One</h2>
<span class="price">$19.99</span>
<a class="buy-link" href="https://example.com/book1">Buy</a>
</div>
```

Parse the HTML code

Find by other attributes

```
first_link = example_soup.find("a",
                href="https://example.com/book1")
print(first_link)
print('--------------------')
all_images = example_soup.find_all("img", src=True)
print(all_images)
```

Parse the HTML code

```
<a class="buy-link" href="https://example.com/book1">Buy</a>
--------------------
[<img alt="Cover 1" src="cover1.jpg"/>, <img alt="Cover 2" src
```

Parse the HTML code

Nested find

```
product_price = first_product.find("span",
                                   class_="price").text
print(product_price)
```

Parse the HTML code

```
$19.99
```

Parse the HTML code

Using `.find()` and `.find_all()` in BeautifulSoup

BeautifulSoup provides two main methods for locating elements in an HTML document:

- `.find()` $\rightarrow$ returns the **first matching element**.
- `.find_all()` $\rightarrow$ returns a **list of all matching elements**.

Scrape https://realpython.github.io/fake-jobs/

```
URL = "https://realpython.github.io/fake-jobs/"
page = requests.get(URL)
soup = BeautifulSoup(page.text)
results = soup.find(id="ResultsContainer")
```

Scrape https://realpython.github.io/fake-jobs/

```
job_cards = results.find_all("div", class_="card-content")
# Lets look at the first job card
print(job_cards[0].prettify())
```

Scrape https://realpython.github.io/fake-jobs/

```
<div class="card-content">
 <div class="media">
  <div class="media-left">
   <figure class="image is-48x48">
    <img alt="Real Python Logo" src="https://files.realpython.
   </figure>
  </div>
  <div class="media-content">
   <h2 class="title is-5">
    Senior Python Developer
   </h2>
   <h3 class="subtitle is-6 company">
    Payne, Roberts and Davis
   </h3>
  </div>
 </div>
```

Scrape https://realpython.github.io/fake-jobs/

```
jobs = dict()

for i,job_card in enumerate(job_cards):
    # Get the information from each job card
    title_element = job_card.find("h2", class_="title")
    company_element = job_card.find("h3", class_="company")
    location_element = job_card.find("p", class_="location")
    # Add the information to the dictionary
    jobs[i] = {'job':title_element.text.strip(),
               'company':company_element.text.strip(),
                'location':location_element.text.strip()}

df = pd.DataFrame(jobs).T
df
```

Scrape https://realpython.github.io/fake-jobs/

	job	company	loca
0	Senior Python Developer	Payne, Roberts and Davis	Stev
1	Energy engineer	Vasquez-Davidson	Chri
2	Legal executive	Jackson, Chambers and Levy	Port
3	Fitness centre manager	Savage-Bradley	East
4	Product manager	Ramirez Inc	Nort
...	...	...	...
95	Museum/gallery exhibitions officer	Nguyen, Yoder and Petty	Lake
96	Radiographer, diagnostic	Holder LLC	Jacq
97	Database administrator	Yates-Ferguson	Port
98	Furniture designer	Ortega-Lawrence	Nort
99	Ship broker	Fuentes, Walls and Castro	Mich

Extracting Attributes

We have already seen how to extract text from one of our elements. The text is just one part of the html code. Sometimes we want other parts of the content. For example, we may want to know about some of the attributes that come before the text. Attributes are not typed out to the website, but rather act on the text or help organize the elements on the page.

Common HTML Attributes

HTML attributes provide additional information about elements and are often used in web scraping to locate or filter data. Some of the most common attributes include:

- **id**
  - Uniquely identifies an element on the page.
  - Example: `<div id="main-content">`
- **class**
  - Groups elements for styling or scripting. Multiple elements can share the same class.
  - Example: `<span class="price">`

Common HTML Attributes

- **href**
  - Used in <a> tags to specify the link destination.
  - Example: `<a href="https://example.com">Visit</a>`
- **src**
  - Specifies the source for media elements like images, videos, or scripts.
  - Example: `<img src="cover.jpg" alt="Book Cover">`

Common HTML Attributes

- **alt**
  - Provides alternative text for images, shown if the image cannot load.
  - Example: `<img src="cover.jpg" alt="Book Cover">`
- **title**
  - Gives additional information about an element, usually shown as a tooltip.
  - Example: `<a href="#" title="Click here for more info">Link</a>`

Common HTML Attributes

- **style**
  - Inline CSS styling for an element.
  - Example: `<p style="color:red;">Important text</p>`
- **name**
  - Often used in form elements to identify input fields.
  - Example: `<input type="text" name="username">`

Common HTML Attributes

- **type**
  - Specifies the type of input or button in forms.
  - Example: `<input type="password">`
- **target**
  - Defines where to open linked documents (new tab, same tab, etc.).
  - Example: `<a href="https://example.com" target="_blank">Visit</a>`

Common HTML Attributes

- **data-* (custom data attributes)**
  - Custom attributes used to store extra information on elements.
  - Example: `<div data-id="123" data-category="books">`

Common HTML Attributes

```
# From the example code above: get the first link
link_example = example_soup.find('a')
link_example

link_example.text
link_example['class']
link_example['href']
```

Common HTML Attributes

```
<a class="buy-link" href="https://example.com/book1">Buy</a>
```

```
'Buy'
```

```
['buy-link']
```

```
'https://example.com/book1'
```

Functions and Beautiful Soup

You can sometimes leverage functions inside the beautiful soup method to extract things in a more pythonic way

```
python_jobs = results.find_all("h2",
            string=lambda x: "python" in x.lower())
```

Functions and Beautiful Soup

Print them in a loop:

```
Senior Python Developer
Software Engineer (Python)
Python Programmer (Entry-Level)
Python Programmer (Entry-Level)
Software Developer (Python)
Python Developer
Back-End Web Developer (Python, Django)
Back-End Web Developer (Python, Django)
Python Programmer (Entry-Level)
Software Developer (Python)
```

More advanced Parsing

Why does this give an error?

```
title_element = job.find("h2", class_="title")
title_element.text
```

```
AttributeError: 'NoneType' object has no attribute 'text'
```

Searching Hierarchy

You can also search starting from an interior element. Here job is just one “h2” element, but we can look at its parent containers.

```
print(job)
print('--------------------')
print(job.parent)
print('--------------------')
print(job.parent.parent)
print('--------------------')
print(job.parent.parent.parent)
```

Searching Hierarchy

```
<h2 class="title is-5">Software Developer (Python)</h2>
--------------------
<div class="media-content">
<h2 class="title is-5">Software Developer (Python)</h2>
<h3 class="subtitle is-6 company">Moreno-Rodriguez</h3>
</div>
```

Searching Hierarchy

```
--------------------
<div class="media">
<div class="media-left">
<figure class="image is-48x48">
<img alt="Real Python Logo" src="https://files.realpython.com/
</figure>
</div>
<div class="media-content">
<h2 class="title is-5">Software Developer (Python)</h2>
<h3 class="subtitle is-6 company">Moreno-Rodriguez</h3>
</div>
</div>
```

Searching Hierarchy

```
--------------------
<div class="card-content">
<div class="media">
<div class="media-left">
<figure class="image is-48x48">
<img alt="Real Python Logo" src="https://files.realpython.com/
</figure>
</div>
<div class="media-content">
<h2 class="title is-5">Software Developer (Python)</h2>
<h3 class="subtitle is-6 company">Moreno-Rodriguez</h3>
</div>
</div>
<div class="content">
<p class="location">
        Martinezburgh, AE
```

Dynamic Web Scraping Process Using Selenium

Dynamic websites load content using JavaScript, or other dynamic processes. Unlike static pages, the HTML source may initially contain little data, requiring tools like **Selenium** to interact with the page and extract information.

Dynamic Web Scraping Process Using Selenium

Prepare the driver/browser

1. We need to install selenium and webdriver-manager
2. Then we import all the packages - my example is using chrome
3. Choose options that work for your computer

Dynamic Web Scraping Process Using Selenium

Now we are ready to scrape

1. Start the driver
2. Get the html - waiting for the JavaScript to load
3. Execute the script to get the HTML
4. Use Beautiful Soup to parse the HTML
5. Close the browser

Dynamic Web Scraping Process Using Selenium

Key Benefit: Selenium allows scraping of content that is not present in the initial HTML source, making it ideal for modern, JavaScript-heavy websites.

Dynamic Web Scraping Process Using Selenium

Look at our dynamic site

`'https://en.wikipedia.org/wiki/Cat'`

Dynamic Web Scraping Process Using Selenium

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from webdriver_manager.chrome import ChromeDriverManager
import time
```

Dynamic Web Scraping Process Using Selenium

Driver Options

```
options = Options()
options.add_argument("--headless=new") # So a browser window d
options.add_argument("--no-sandbox")
options.add_argument("--disable-dev-shm-usage")
options.add_argument("--disable-gpu")
options.add_argument("--remote-debugging-port=9222")
```

Dynamic Web Scraping Process Using Selenium

Choose a service (browser)

```
service = Service(ChromeDriverManager().install())
```

Dynamic Web Scraping Process Using Selenium

Start the browser

```
driver = webdriver.Chrome(service=service,options=options)
```

Get the webpage

```
driver.get(URL_dynamic)
```

Sometimes you need to use ‘`time.sleep()` to give the page enough time to load

Dynamic Web Scraping Process Using Selenium

Extract content

```
html_source_code = driver.execute_script("return
                    document.body.innerHTML;")
# Save to beautiful soup
soup = BeautifulSoup(html_source_code, 'html.parser')
```

Close the browser

```
driver.quit()
```

Dynamic Web Scraping Process Using Selenium

Now we can use the same Beautiful Soup commands that we used above.

```
cats = soup.find_all('div', class_="tsingle")

for cat in cats:
    photo = cat.find('a')
    print('https://en.wikipedia.org/'+photo['href'])
```

Dynamic Web Scraping Process Using Selenium

```
https://en.wikipedia.org//wiki/File:Cat_August_2010-4.jpg
https://en.wikipedia.org//wiki/File:Gustav_chocolate.jpg
https://en.wikipedia.org//wiki/File:Orange_tabby_cat_sitting_o
https://en.wikipedia.org//wiki/File:Siam_lilacpoint.jpg
https://en.wikipedia.org//wiki/File:Felis_catus-cat_on_snow.jp
https://en.wikipedia.org//wiki/File:Sheba1.JPG
https://en.wikipedia.org//wiki/File:Louvre_egyptologie_21.jpg
https://en.wikipedia.org//wiki/File:Cat_birds_MAN_Napoli_Inv99
https://en.wikipedia.org//wiki/File:PSM_V37_D105_English_tabby
https://en.wikipedia.org//wiki/File:Black_Cat_(7983739954).jpg
```

Practice! Practice! Practice!

The more you practice web scraping the better you will get at dealing with the curveballs that get thrown your way. Here are some great places to try out your skills

Practice! Practice! Practice!

https://www.scrapethissite.com/pages/

https://webscraper.io/test-sites

https://realpython.github.io/fake-jobs/

https://books.toscrape.com/

https://quotes.toscrape.com/

https://www.wikipedia.org/ - NOT A FAKE SITE - but good for practice. Please follow their rules for scraping: https://wikitech.wikimedia.org/wiki/Robot_policy, basically this means don't send too many requests, if you are looking for large amounts of data use the API.