Epigraphy in the Digital Age:
A Hands-on Session for Web Scraping

Md. Izhar Ashraf
ashraf@imsc.res.in

March 21, 2024

Contents

1 Introduction to Web Scraping

1.1 Definition and Purpose of Web Scraping

- Web scraping is the automated collection of data from websites, using software to extract information for analysis or use in various applications.
- Imagine you're an archaeologist, sifting through sand to uncover hidden treasures. Web scraping is similar! Websites hold a vast amount of information, but it's often coded and not readily accessible.
- Think of websites like libraries with countless books. Scraping allows you to efficiently target and collect specific information from those books, without reading everything cover to cover.

1.2 Importance of Web Scraping in Epigraphy

- Epigraphy, the study of inscriptions or epigraphs as writing, often involves the collection of data from various online databases, digital libraries, and archives.
- By employing web scraping techniques, researchers can automate the extraction of valuable epigraphic data, including texts, translations, metadata, and images.
- This not only accelerates the research process but also enables the creation of comprehensive datasets that can be used for further study.

1.3 Legal and Ethical Considerations

- While web scraping is a powerful tool for researchers, it's crucial to navigate the legal and ethical considerations involved.
- There are special files called "robots.txt" that tell web scrapers which parts of a website are okay to access. It's like a map showing where you can "dig" for information.
- We also need to consider legal aspects like copyright. The information itself might be freely available, but how it's presented might be protected.
- Finally, websites can get overloaded if too many requests come in at once. It's important to be respectful and avoid overwhelming them.
- By following these guidelines, web scraping becomes a valuable tool for research and knowledge sharing in epigraphy and beyond!

2 Understanding Web Basics

2.1 Building Blocks of a Webpage

- Websites are built with three key ingredients: HTML, CSS, and JavaScript [1].
- Think of them like the bricks, mortar, and electrical wiring of a house.
- HTML (HyperText Markup Language): The foundation, like bricks. It defines the structure and content of a webpage, like headings, paragraphs, and images.

```html
<!DOCTYPE html>
<html>
<head>
        <title>Page Title</title>
</head>
<body>
        <h1>This is a Heading</h1>
        <p>This is a paragraph.</p>
</body>
</html>
```

- CSS (Cascading Style Sheets): The decorator, like mortar. It defines the visual style of the webpage, like fonts, colors, and layout.

```html
<!DOCTYPE html>
<html>
<head>
  <title>Page Title</title>
</head>
<body>
  <h1 style="color:blue; text-align: center;">This is a Heading</h1>
  <p style="color:red; text-align: center;">This is a paragraph.</p>
</body>
</html>
```

- JavaScript (JS): The electrician, like wiring. It adds interactivity to webpages, like animations or forms that respond to your clicks.

```html
<!DOCTYPE html>
<html>
<head>
  <title>Change Text Color</title>
  <script>
    function changeColor() {
      // Get a reference to the heading element
      var heading = document.getElementById("myHeading");

      // Change the heading's text color
      heading.style.color = "green";
    }
  </script>
</head>
<body>
  <h1 id="myHeading" style="text-align: center;">This is a
    ↪  Heading</h1>
  <p style="color:red; text-align: center;">This is a
    ↪ paragraph.</p>
  <button onclick="changeColor()">Magic Button</button>
</body>
</html>
```

2.2 Understanding the Structure of a Webpage

- Imagine a webpage as a well-organized document. HTML tags create a hierarchy, like headings and subheadings, to structure the content.
- Each element has tags (like labels) that define its purpose (e.g., heading, paragraph, image).
- By understanding this structure, we can pinpoint the specific data we want to scrape.

```html
<!DOCTYPE html>
<html>
<head>
  <title>Change Text Color</title>
  <style>
    .center-content {
      text-align: center;
    }
    table {
      margin: auto; /* Centers the table */
    }
  </style>
  <script>
    function changeColor() {
      // Get a reference to the heading element
      var heading = document.getElementById("myHeading");

      // Change the heading's text color
      heading.style.color = "green";
    }
  </script>
</head>
<body>
  <h1 id="myHeading" style="text-align: center;">This is a
    ↪ Heading</h1>
```

```html
<p style="color:red; text-align: center;">This is a
  ↪ paragraph.</p>
<br>

<div class="center-content">
  <button onclick="changeColor()">Magic Button</button>
  <br>
  <br>
  <!-- Adding a simple table -->
  <table id="table1" border="1">
    <caption>Demo Table 1</caption>
    <tr>
      <th>Column 1</th>
      <th>Column 2</th>
      <th>Column 3</th>
    </tr>
    <tr>
      <td>Row 1, Cell 1</td>
      <td>Row 1, Cell 2</td>
      <td>Row 1, Cell 3</td>
    </tr>
    <tr>
      <td>Row 2, Cell 1</td>
      <td>Row 2, Cell 2</td>
      <td>Row 2, Cell 3</td>
    </tr>
  </table>

  <br>

  <!-- Adding another simple table -->
  <table id="table2" border="1">
    <caption>Demo Table 2</caption>
    <tr>
      <th>ID</th>
      <th>Site Name</th>
    </tr>
    <tr>
      <td>101</td>
      <td>Harappa</td>
    </tr>
    <tr>
      <td>102</td>
      <td>Mohenjo-daro</td>
    </tr>
    <tr>
      <td>103</td>
      <td>Rakhigarhi</td>
      </tr>
    </table>
  </div>
</body>
</html>
```

2.3 Introduction to the Document Object Model (DOM)

- The Document Object Model (DOM) is a way to represent a webpage's structure as a tree.
- Each element on the page becomes a node in this tree, with parent-child relationships reflecting the HTML hierarchy.
- Understanding the DOM is crucial for web scraping tools like BeautifulSoup to navigate and extract specific data from a webpage.

```
document
|- html
   |- head
   |   |- title: "Change Text Color"
   |   |- style
   |   |   `- (.center-content and table styles)
   |   `- script: function changeColor()
   `- body
      |- h1#myHeading (style="text-align: center")
      |- p (style="color:red; text-align: center")
      |- br
      `- div.center-content
         |- button (onclick="changeColor()")
         |- br
         |- br
         |- table#table1 (border="1")
         |   |- caption: "Demo Table 1"
         |   |- tr
         |   |   |- th: "Column 1"
         |   |   |- th: "Column 2"
         |   |   `- th: "Column 3"
         |   |- tr
         |   |   |- td: "Row 1, Cell 1"
         |   |   |- td: "Row 1, Cell 2"
         |   |   `- td: "Row 1, Cell 3"
         |   `- tr
         |       |- td: "Row 2, Cell 1"
         |       |- td: "Row 2, Cell 2"
         |       `- td: "Row 2, Cell 3"
         |- br
         `- table#table2 (border="1")
```

```
            |- caption: "Demo Table 2"
            |- tr
            |   |- th: "ID"
            |   `- th: "Site Name"
            |- tr
            |   |- td: "101"
            |   `- td: "Harappa"
            |- tr
            |   |- td: "102"
            |   `- td: "Mohenjo-daro"
            `- tr
                |- td: "103"
                `- td: "Rakhigarhi"
```

3 Tools and Libraries for Web Scraping

Python is ideal for web scraping with its simple syntax and powerful libraries like BeautifulSoup, Requests, and Scrapy make data extraction efficient and accessible for all skill levels.

3.1 BeautifulSoup

Beautiful Soup [2], started by Leonard Richardson, is a Python package used for parsing HTML and XML documents, including those with malformed markup. It creates a parse tree that facilitates data extraction from HTML, making it highly useful for web scraping. However, it lacks the capability to download web pages on its own, a function complemented by the Requests library.

3.1.1 Requests

Requests is a straightforward Python library for HTTP requests, essential for retrieving web content. Paired with BeautifulSoup, it efficiently manages web scraping tasks. Beautiful Soup is a Python library for parsing HTML and XML documents. It provides idiomatic ways of navigating, searching, and modifying the parse tree. Here's a concise cheat sheet of its basic syntax and functions:

Creating a Soup Object

To parse a document, you first need to create a BeautifulSoup object, which represents the document as a nested data structure:

```python
from bs4 import BeautifulSoup
# Parse HTML from a string
soup = BeautifulSoup(html_doc, 'html.parser')
# Parse HTML from a file
with open("index.html") as file:
    soup = BeautifulSoup(file, 'html.parser')
# Parses XML content
soup = BeautifulSoup(xml_content, 'lxml')
```

Navigating the Tree

BeautifulSoup offers multiple ways to navigate and search the parse tree:

```python
# Accessing tag names
soup.title
# Accessing tag attributes
soup.title.name
soup.title.string
soup.title['attribute']
# Navigating using tag names
soup.body.a
# Navigating using methods
soup.find_all('a')
soup.find(id='link3')
```

Searching the Tree

BeautifulSoup's search methods allow you to find elements based on their attributes, text content, or filter functions:

```python
# Find all tags with a specific name
soup.find_all('a')
# Find the first tag with a specific name
soup.find('title')
# Find tags using keyword arguments
soup.find_all(id='link2')
soup.find_all(href=True)
# Searching by CSS class
soup.find_all("a", class_="sister")
# Using string arguments
soup.find_all(string="Elsie")
# Using regular expressions
import re
soup.find_all(string=re.compile("Dormouse"))
# Using a list
soup.find_all(["a", "b"])
```

Modifying the Tree

You can also modify the HTML or XML tree in various ways:

```python
# Changing tag names and attributes
tag.name = 'blockquote'
tag['attribute'] = 'new value'
# Adding and removing tags
new_tag = soup.new_tag('a', href="http://www.example.com")
soup.body.append(new_tag)
tag.extract()  # removes a tag from the tree
```

This cheat sheet covers the basics to get started with BeautifulSoup for web scraping tasks. For more detailed information, refer to the official BeautifulSoup documentation.

Example code

```python
import requests
from bs4 import BeautifulSoup

# The URL of the website you want to scrape
url = 'http://example.com/news'
# Use Requests to get the webpage content
response = requests.get(url)
# Create a BeautifulSoup object and specify the parser
soup = BeautifulSoup(response.text, 'html.parser')
# Find all 'h2' elements with a class 'article-title'
article_titles = soup.find_all('h2', class_='article-title')
# Loop through the list of titles and print them
for title in article_titles:
    print(title.text.strip())
```

3.1.2 XPath (XML Path Language) Syntax

XPath (XML Path Language) is a query language designed for navigating and selecting nodes from an XML document. It allows for precise location of elements, attributes, text, and more within XML files using a path-like syntax.
XPath expressions can be used to navigate through elements and attributes in an XML document, making it a powerful tool for XML querying and transformation tasks.

Example code

```python
from lxml import etree
import requests
# Fetch the HTML content
url = 'https://www.example.com/'
response = requests.get(url)
html_content = response.text
# Parse the HTML
parser = etree.HTMLParser()
tree = etree.fromstring(html_content, parser)
out_list= tree.xpath("//div/text()")
```

Table 1: XPath syntax

Syntax	**Description**
`nodename`	Selects nodes with the name `nodename`
`/`	Selects from the root node
`//`	Selects nodes from the current node that match the selection no matter where they are
`.`	Selects the current node
`..`	Selects the parent of the current node
`@`	Selects attributes
`*`	Selects all elements/nodes regardless of their name
`@*`	Selects all attributes of the current node
`node()`	Selects all nodes (element, attribute, text, namespace, processing-instruction, comment, and document nodes)
`[n]`	Selects the n-th node (1-based index)
`|`	Combines two expressions, selecting nodes that match either expression
`//book[1]`	Selects the first `book` element
`//book[last()]`	Selects the last `book` element
`//book[position()<3]`	Selects the first two `book` elements
`//book[@attr]`	Selects all `book` elements that have an attribute named `attr`
`//book[@attr='value']`	Selects all `book` elements where the `attr` attribute has the value 'value'
`//book[price>35.00]`	Selects all `book` elements with `price` elements having a value greater than 35.00
`//title[@lang='en']`	Selects all `title` elements with a `lang` attribute value of 'en'

3.2 Scrapy

For more complex web scraping projects, Scrapy, a comprehensive open-source web crawling and scraping framework, comes into play. Unlike BeautifulSoup, which is more suited for simple, direct web page extraction, Scrapy is built to scrape and crawl at scale. It allows for the extraction of data from websites and the automation of web interactions, making it a formidable tool for gathering data from multiple pages or even entire websites.

Scrapy Project Setup

Start a new Scrapy project:

```
scrapy startproject myproject
```

This creates a new Scrapy project with the name `myproject`.
Generate a Spider:

```
scrapy genspider myspider example.com
```

This command generates a spider named `myspider` for the domain `example.com`.

Scrapy Components

Spider: Classes that define how a site will be scraped, including how to perform the crawl (i.e., follow links) and how to extract structured data from their pages.
Basic spider example:

```python
import scrapy

class MySpider(scrapy.Spider):
    name = 'example_spider'
    start_urls = ['http://example.com']

    def parse(self, response):
        # Parsing code here
```

Item: Standard Python classes used to define the data structure for items you scrape.
Example:

```python
import scrapy

class MyItem(scrapy.Item):
    name = scrapy.Field()
    description = scrapy.Field()
```

Item Pipeline: Process and filter the items returned by spiders. Defined in ITEM_PIPELINES setting.
Pipeline example:

```python
class MyPipeline:
    def process_item(self, item, spider):
        # Process item here
        return item
```

Basic Commands

Running a Spider:

```
scrapy crawl myspider
```

This command runs the spider named `myspider`.
Shell:

```
scrapy shell 'http://example.com'
```

Opens the Scrapy shell for the given URL, allowing you to test your extraction code interactively.

Selectors

Scrapy uses selectors to extract data from HTML documents. There are two types of selectors: CSS and XPath.
CSS:

```
response.css('title::text').get()
```

XPath:

```
response.xpath('//title/text()').get()
```

Requests and Responses

Following Links:

```
yield response.follow(next_page, self.parse)
```

Form Requests:

```
yield scrapy.FormRequest(url='http://example.com/login',
                         formdata={'user': 'john', 'pass': '
                           ↪ secret'},
                         callback=self.after_login)
```

3.3 Selenium

Essential for dynamic web pages, Selenium automates browser actions, enabling interaction with JavaScript-driven content. Originally for testing, its real-user simulation is crucial for scraping JavaScript-heavy sites, making it invaluable for accessing dynamically loaded data, complementing BeautifulSoup, Requests, and Scrapy in the web scraping toolkit.
Import Selenium:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By

chrome_options = Options()
driver = webdriver.Chrome(options=chrome_options)
```

Navigate to a Page:

```
driver.get('http://example.com')
```

Locate Elements:

- By ID:

  ```
  element = driver.find_element(By.ID, 'elementId')
  ```

- By Name:

  ```
  element = driver.find_element(By.NAME,'name')
  ```

- By XPath:

  ```
  element = driver.find_element(By.XPATH, '//tag[@attribute
      ↪ ="value"]')
  ```

- By CSS Selector:

  ```
  element = driver.find_element_by(By.CSS_SELECTOR, 'tag.
      ↪ class#id')
  ```

- For multiple elements (returns a list):

  ```
  elements = driver.find_elements_by(By.TAG, 'tag')
  ```

Interact with Elements:

- Click a button:

  ```
  button.click()
  ```

- Enter text in a text field:

  ```
  text_field.send_keys('text to enter')
  ```

Close the Browser:

```
driver.quit()
```

4 Setting & installation of required packages

To update and install Python 3 and pip on Ubuntu, use the following commands in your terminal:

```
sudo apt update
sudo apt upgrade -y
sudo apt install python3 python3-pip
```

Create a directory for your scraping project, set up a Python virtual environment, and activate it:

```
mkdir my_scraping_project
cd my_scraping_project
python3 -m venv demoVirEnv
source demoVirEnv/bin/activate
```

Install BeautifulSoup, Requests for fetching web content, Scrapy for more complex scraping, and Selenium for dynamic web pages:

```
pip install beautifulsoup4
pip install requests
pip install lxml
pip install scrapy
pip install selenium
```

References

[1] W3School HTML. http://https://www.w3schools.com/

[2] Beautiful Soup Documentation. https://beautiful-soup-4.readthedocs.io/en/latest/#beautiful-soup-documentation

[3] XPather. http://xpather.com/

[4] Scrapy Tutorial. https://docs.scrapy.org/en/latest/intro/tutorial.html