Web Scraping Workshop

Tushar Khan (CS ‘22)

OVERVIEW

FOR THE UNINITIATED

What is Web Scraping?

Web crawling is the process of systematically browsing the World Wide Web, typically for the purpose of Web indexing.

Web scraping is the process of automatically mining data or collecting information from the World Wide Web. It might require some crawling to access information hidden behind links on a page.

Web scraping a web page involves **downloading** the page (which a browser does when you view it) and then **extracting** the content from it.

Why Web Scrape?

CDS:

- Web sites generally have the most up-to-date data
- Allows more flexibility in creating custom data sets
- More powerful than using APIs

Industry:

- Market Price Monitoring
- Competitor Price Monitoring
- Product Trend Monitoring
- Market Trend Analysis
- Consumer Sentiment Monitoring
- News and Content Monitoring
- Real-time Analytics

WEB SCRAPING

WITH PYTHON

Getting Started in Python

The following libraries are the de-facto standard when it comes to web scraping in Python. You will need to install them:

- `requests` - making HTTP requests in Python.
  - `$ pip install requests`

- `beautifulsoup4` - pulling data out of HTML and XML files.
  - `$ pip install beautifulsoup4`
  - `$ pip install lxml`

HTTP in a Nutshell

HTTP (HyperText Transfer Protocol) is the standard protocol used to structure the exchange of resources over the web.

Resources on the web are typically hosted on a **server**. A client (e.g. web browser) can perform a specific action on a resource by sending an HTTP **request**. There are a number of actions that a client can perform on a resource. For example, to read a resource it sends a GET request, to send a resource it sends a POST request, etc.

When a server receives a request from a client, it sends back information in the form of an HTTP **response**. The response contains not only the requested information, but additional metadata like a status code as well.

Requests Library - Sending Requests

The `requests` library allows you to **send HTTP requests** extremely easily with quality-of-life features like keep-alive & connection pooling, sessions with cookie persistence, and more.

- GET requests
  - `r = requests.get('https://www.google.com')`
  - `r = requests.get('https://www.amazon.com/s', params={'k': 'shoes'})`
- POST requests
  - `r = requests.post('https://example.com/', data = {'key':'value'})`

- Also supports PUT, DELETE, HEAD, and OPTIONS requests.

Requests Library - Adding Headers

HTTP **headers** are typically sent along with an HTTP request or a response to pass additional information between the client and the server.

Our plain request doesn’t contain any headers, so sites can easily figure out that it is being sent by a bot and not by a potential consumer. Thus, the server might be configured to send a response that doesn’t actually contain any data.

We can get around this by sending packets that specify a **user agent** in the header to fool the server into thinking that the request was sent by a web browser:

```
r = requests.get('https://www.amazon.com', headers={'User-Agent': 'Mozilla/5.0'})
```

Requests Library - Response Objects

The request methods all return a Response object, which contains all the information about the response sent by the server.

- Response Code

  -
    ```
    print(r.status_code)
    >>> 200
    ```
- URL

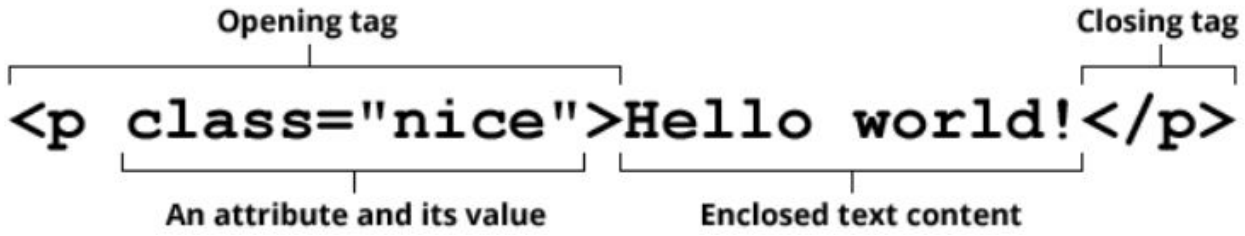
  -
    ```
    print(r.url)
    >>> https://www.amazon.com/s?k=shoes
    ```
- HTML content

  -
    ```
    print(r.text)
    >>> <!DOCTYPE html> <!--[if lt IE 7]> <html lang="en-us" class="a-no- ...
    ```

HTML In a Nutshell

HTML (HyperText Markup Language) is a descriptive language that specifies web page structure.

An HTML document is structured by nested **elements**, which are surrounded by matching opening and closing **tags**. Tags can be extended with **attributes**, which provide additional information affecting how the browser interprets the element.



```
<p class="nice">Hello world!</p>
```

BeautifulSoup Library - HTML Parsing

The `BeautifulSoup` library allows you to **extract data from HTML** (or XML) content. It does this by constructing a parse tree from the nested elements of the document and providing a plethora of ways to query objects in the tree.

The parse tree is constructed when we create a BeautifulSoup object. The first argument contains the HTML and the second argument specifies the parser:

- `soup = BeautifulSoup(r.text, 'html.parser')`
- `soup = BeautifulSoup(r.text, 'lxml')`

Note: I suggest using the lxml parser because it is much faster.

BeautifulSoup Library - Searching for Tags

`BeautifulSoup` makes it very easy to extract specific tags from the parse tree using the `find()` and `find_all()` methods. These methods respectively find the first tag or a list of all tags in the HTML parse tree that match the query you provide.

- Search by tag type
  - `tags = soup.find('body') # finds the first <body> tag`
- Search by id or CSS class attributes
  - `tags = soup.find_all(id='bar', class_='icon')`
- Search using a filter function
  - `tags = soup.find(lambda t: t.has_attr('href') and not t.has_attr('img'))`
- Search the nested structure of the document
  - `nested_tags = soup.find('div', id='preview').find_all('a')`

BeautifulSoup Library - Tag Objects

A Tag object corresponds to a tag in the original HTML document. The `find()` and `find_all()` methods return a Tag object or a list of Tag objects respectively.

The attributes of the tag can be retrieved using dictionary syntax. The text enclosed by the tag can be retrieved using the `.text` or `.string` attributes.

```
soup = BeautifulSoup('<div><p align="left">Ho ho ho</p><p>Yo</p></div>', 'lxml')
tag = soup.find('div').find('p') # returns the first <p> tag in the first <div> tag
print(tag['align']) # prints the value associated with the 'align' key
>>> left
print(tag.text) # prints text enclosed by tag
>>> Ho ho ho
```

Inspecting Web Pages

We’ve just learned how to extract data from HTML content, but how do we know which elements we actually want to extract?

The answer is by **inspection** of the actual page.

- Windows: Ctrl + Shift + C
- Mac: ⌘ + Shift + C

It is helpful to first write the returned page to a file to inspect the HTML:

```
with open('page.html', mode='wb') as f:
    f.write(r.content)
```

ETHICS

NO ONE MAN SHOULD HAVE ALL THAT POWER

Denial of Service Attacks

As a human browsing the web, you can only send so many requests to a server at a time. But with a web crawler, you can easily execute thousands of requests in just a few seconds. If the server receives more requests than it can handle, it could become overloaded and **stop fulfilling legitimate requests** altogether. This is called a Denial of Service (DoS) attack.

You should be smart about how often you send a request. For example, you should only **send one request per page** and save it locally if you need to use it again. If you are sending many requests, **send them slowly** instead of all at once.

Some websites will even throttle or block your IP address altogether if they detect an unusually high amount of requests.

Robots Exclusion Standard

The robots exclusion standard is a web standard used by websites to communicate to web crawlers which pages it can and cannot request. It is primarily used to manage crawler traffic by whitelisting and blacklisting certain parts of the site.

It can be found by simply appending “robots.txt” to the base URL of any website (e.g. https://www.reddit.com/robots.txt, https://www.amazon.com/robots.txt, etc.)

This is just a standard, meaning it is not explicitly enforced. But as a programmer utilizing a site’s resources at their expense, you should respect this standard.

If you really need to access a blacklisted part of a website, your web crawler should emulate a human by **throttling the rate of requests** to prevent getting blocked.