CDS Web Scraping Workshop

Tanvi Bhave, Skai Nzeuton

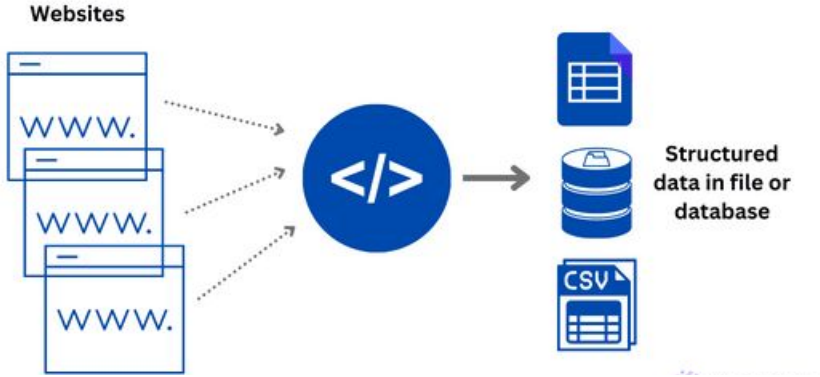
FA 24

OVERVIEW

What is Web Scraping?

The process of automatically collecting information from the internet

1. **Downloading**: visiting a webpage and retrieving the HTML
    - May involve web crawling
2. **Extracting**: parsing the site to get the important information



Why Web Scrape?

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

Example uses:

- Competitor Price Comparison
- Marketing Lead Generation
- Sentiment Analysis
- Stock Market Monitoring

WEB SCRAPING

with Python 🐍

Getting Started in Python

The following libraries are the standard when it comes to web scraping in Python. Let's install them right now:

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

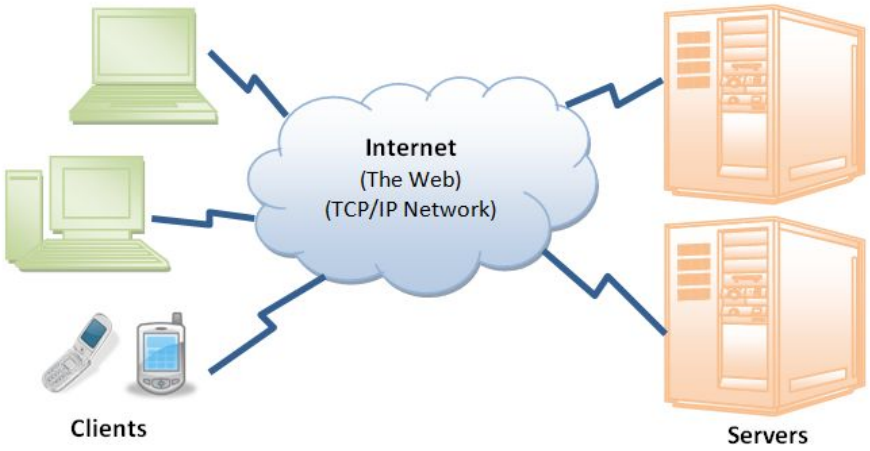
- `beautifulsoup4` - pulling data out of HTML and XML files.
  - `$ pip install beautifulsoup4`
  - `$ pip install lxml`

(Note: you may need to use the command pip2/pip3 instead of pip depending on the version of Python you have)

The Big, Beautiful Web

aka the Internet

- A massive distributed client server information system with many running applications
- How does the client and server communicate with each other?



https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/http_basics.html

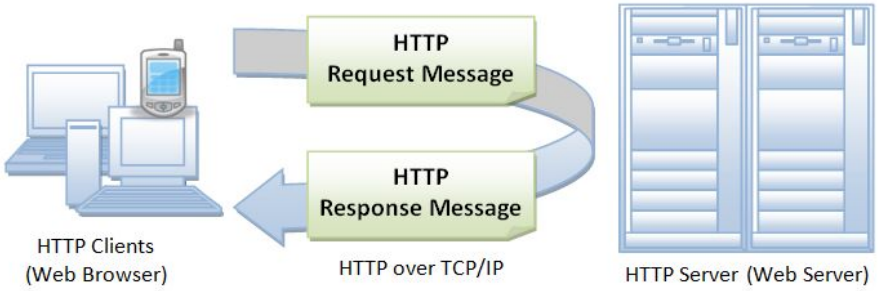
HTTP

The foundation of data communication for the Web

HTTP in a Nutshell

The standard protocol used to structure the exchange of resources over the web.

1. Request
    - Client can perform an action on a resource by sending an HTTP **request**
2. Response
    - Server sends back information in the form of an HTTP **response**



Requests Library - Sending Requests

The `requests` library allows you to **send HTTP requests** extremely easily

- GET requests: retrieve a resource
  - `r = requests.get('https://www.google.com')`
  - `r = requests.get('https://www.amazon.com/s', params={'k': 'shoes'})`
- POST requests: update a resource
  - `r = requests.post('https://example.com/', data = {'key':'value'})`
- Also supports PUT, DELETE, HEAD, and OPTIONS requests.

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- ...`

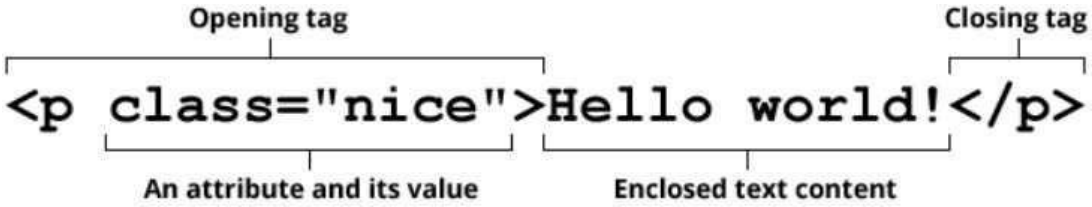
HTML

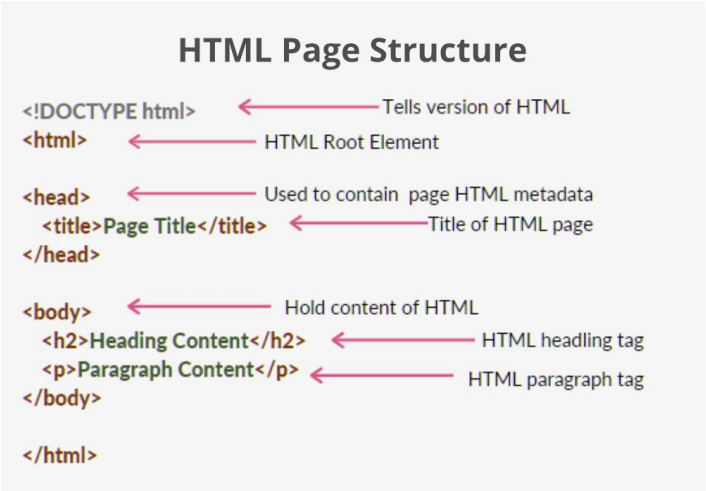
and soup that is beautiful

HTML In a Nutshell

HTML is a language that specifies web page structure.

Contains **tags** and **attributes** that can be used to identify relevant information while web scraping.

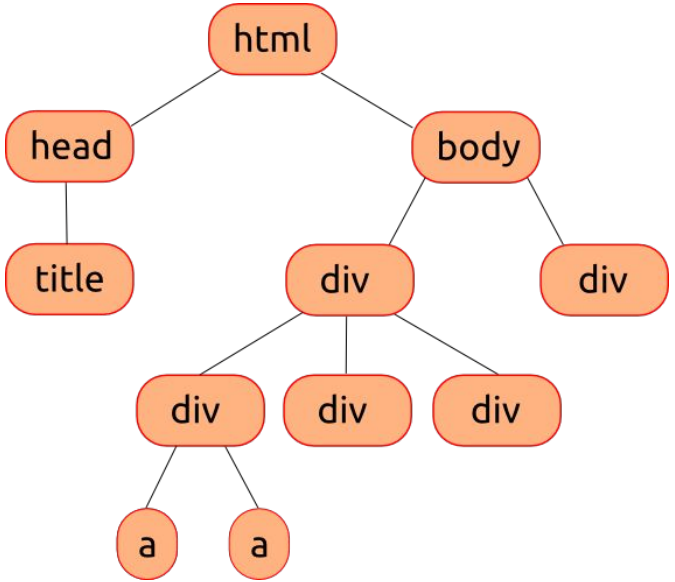




BeautifulSoup Library - HTML Parsing

The `BeautifulSoup` library allows you to **parse data** from the HTML document by constructing a parse tree.

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



BeautifulSoup Library - Searching for Tags

Extract specific tags from the parse tree:

- `find(): finds the first tag`
- `find_all(): generates a list of all tags`

- 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

The `find()` and `find_all()` methods return a Tag object or a list of Tag objects

- Tag objects correspond to a tag in the original HTML document.

Tag attributes can be retrieved using dictionary syntax.

```
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 **inspecting** the actual page. (inspect element)

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

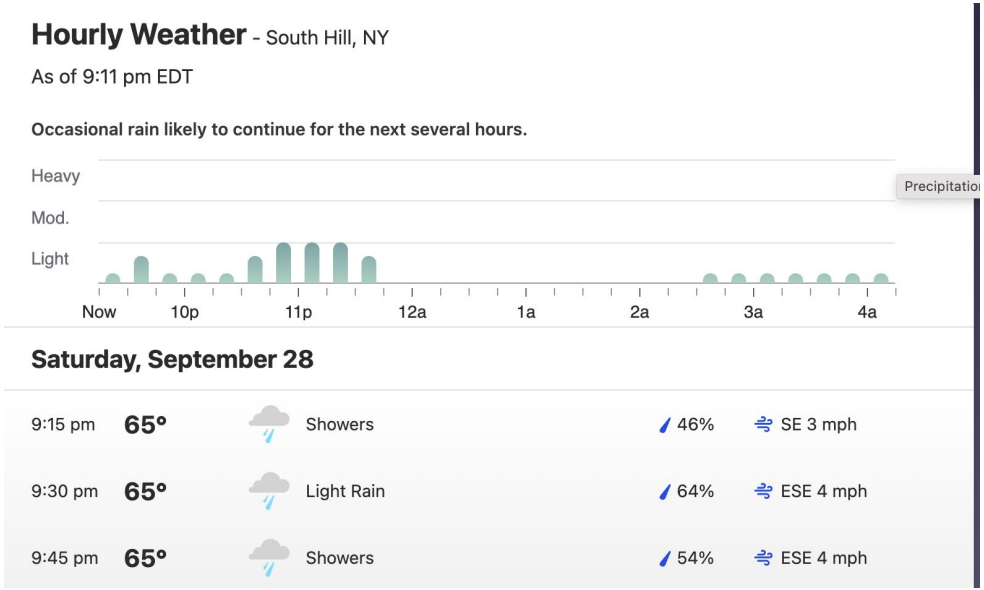
It is helpful to first write the returned page to a file to inspect the HTML:

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

INTERACTIVE DEMO

It’s time to scrape

Demo Project - Scraping Weather Forecasts



Task: Build a dataset that shows

- Date
- Time
- Temperature
- Forecast

0. Setting Up Google Colab

Import necessary libraries:

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

1. Downloading the Forecasts Page

We want to scrape this week's weather data from: https://weather.com/weather/hourbyhour/l/a7728cedd4dfc3c22a182fa00be4cd3826e62adb314498d4dc714c94e5fa09fb

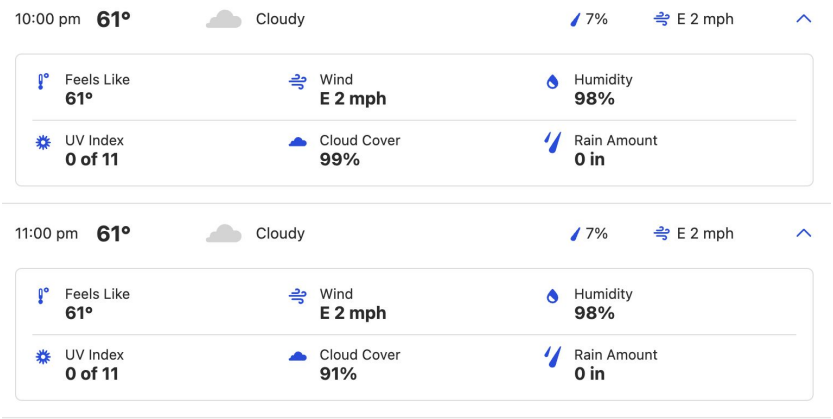
We can request the page like so:

```
base_url = 'https://weather.com/weather/hourbyhour…'

response = rq.get(base_url)
```

2. Inspecting the Forecasts Page

We need to find the specific HTML tag, id, class, etc. each feature uniquely corresponds to.




```
<h2 class="HourlyForecast--longDate--J_Pdh" id="currentDateId0">Sunday, September 29</h2>,
<h2 class="DetailsSummary--daypartName--kbngc" data-testid="daypartName">9 pm</h2>,
```

3. Extracting Data from HTML

Now that we've found a search parameter for the HTML elements we are trying to extract, we can easily collect those elements using `BeautifulSoup`.

```
h2Tags = soup.findAll('h2')
for header in h2Tags:
    if header.has_attr('id') and "currentDateId" in header['id']:
      currentDay = header.text
    if header.has_attr('data-testid'):
      date.append(currentDay)
      time.append(header.text)
```

The returned list contains the dates and times in order.

4. Saving the Scraped Data

Once we've scraped all the relevant data, we need to be sure to save it.

As we saw just now, one way to save our data is to store it in a `pandas` Dataframe object. Then, we can simply write that Dataframe object to a CSV, JSON, etc. after collecting the data.

```
# Define dataframe
df = pd.DataFrame({
    'Date': date,
    'Time' : time,
    'Forecast': forecast,
    'Temperature': temperature
})

# Save it as a csv file
data.to_csv('weather.csv', index=False)
```

Resulting Dataset

WOOOOOOO!!!!!!!

	Date	Time	Forecast	Temperature
0	Sunday, September 29	9 pm	Cloudy	61°
1	Sunday, September 29	10 pm	Cloudy	61°
2	Sunday, September 29	11 pm	Cloudy	61°
3	Monday, September 30	12 am	Mostly Cloudy Night	61°
4	Monday, September 30	1 am	Mostly Cloudy Night	60°
5	Monday, September 30	2 am	Mostly Cloudy Night	59°

UN-ETHICS

"With great power comes great responsibility”

Denial of Service Attacks

A human browser can only send so many requests to a server at a time, but a web crawler can easily execute thousands of requests in a few seconds.

The server could become overloaded and **stop fulfilling legitimate requests.**

- This is called a Denial of Service (DoS) attack.

You should be smart about how often you send a request:

- **send one request per page** and save it locally
- **send requests slowly**

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.

THANK YOU

Questions?

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'})
```