WEB SCRAPING

IN PYTHON

Colloquium series

Yohai Meiron

2023 November 22

WHAT'S WEB SCRAPING?

Web scraping is the *art* of extracting data from websites

The basic steps are:

- Programmatically retrieve URLs
- Download each web page
  - Render dynamic content if needed
- Parse the HTML
- Store information in database
- Repeat...

Web scraping *at scale* is a high-performance computing task, but normally the computing needs are modest

WHAT IS IT GOOD FOR?

Data harvesting can be used for *research*, *commercial*, and *personal* purposes

- Statistical analysis
- Machine learning
- Creating alerts
- Visualization
- ...

OUTLINE

In this seminar we will

- discuss legal and ethical considerations
- learn the basics of an HTML document
- see how to retrieve and parse HTML in Python
- try to bypass the website and get directly to the data source
- render dynamic page content with Selenium
- talk about bot detection avoidance

There is *plenty* of further learning material online!

LEGAL AND ETHICAL CONSIDERATIONS

⚠️ Disclaimer ⚠️ I am not a lawyer or an ethicist

IS IT LEGAL?

- Scraping *publicly available* information is not against the law in Canada
- The act may constitute a breach of the terms of service of a website
- Publicly available material may still be under copyright
- If the material violates PIPEDA or other laws, storing it may be illegal

IS IT ETHICAL?

- It incurs cost to the website being scraped
- Badly done scraping constitutes a denial of service attack
- Bulk data may be offered for sale
- Broader questions about training AI on publicly available material

For academics: consult your institution’s ethics board

introduction to the World Wide Web

INTRODUCTION TO THE WORLD WIDE WEB

By the year 1991

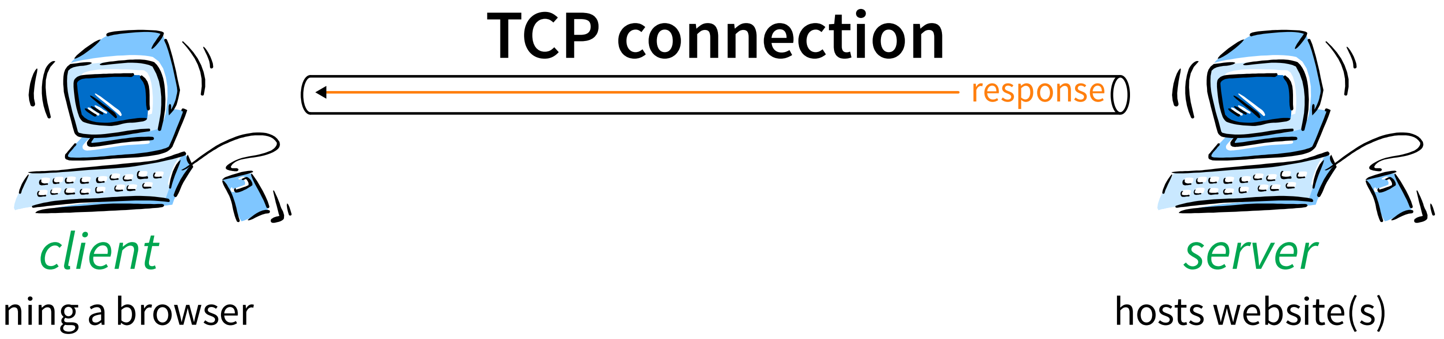
- Computer networking has become quite mature
- The Internet had many application such as
  - File transfer
  - E-mail
  - News and discussions
- It was still missing an application for content sharing *on demand*

Then came the **World Wide Web** out of CERN

- Hypertext Transfer Protocol (HTTP)
- Hypertext Markup Language (HTML)

Hypertext refers to text documents interconnected by links

HTTP CRASH COURSE



```
GET /something.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:120.0) Gecko/20100101 Firefox/120.0
```

```
HTTP/1.1 200 OK
Server: Apache/2.4.58
Date: Wed, 22 Nov 2023 17:30:00 GMT
Content-type: text/html
Content-Length: 18439
Last-Modified: Wed, 22 Nov 2023 16:00:00 GMT

<!DOCTYPE html>
<html>
  <head>
...
```

Web browsers *don't do magic*, a Python script can send requests and receive responses

(Reality in 2023 is more complicated, but the web still works on the principle of requests & responses)

HTML CRASH COURSE

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Simple HTML page</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <h1>Header</h1>
    <p class="fancy centred">Text with a <a href="other.html">link</a></p>
    <img id="logo" src="logo.jpg" alt="SciNet Logo" width="200" height="60">
  </body>
</html>
```

- An HTML document comprises of multiple *elements* nested within the “root” `<html>` element
- An element has a *tag*, and possibly *attributes*
  - Normal elements have start and end tags, and can have child elements
  - Some are *void elements*, they only have a start tag no children
- `<head>` is the metadata element, while `<body>` is what is being rendered
- `id` and `class` are especially important attributes

After loading the HTML page, the browser will make additional HTTP requests to the server for needed resources (`styles.css`, `logo.jpg`, ...)

THE BASIC WEB SCRAPING TOOLS

Python is a great language for this task

The bottleneck is usually the network, so a “fast” language won’t do any better

- Making HTTP requests using the *requests* package
  - httpx as an alternative
- Parsing HTML responses using the *BeautifulSoup* package
  - selectolax, lxml as alternatives
- Storing data anyway you like
  - SQLAlchemy is a good choice
  - For simplicity, we’ll just use `print`

Scrapy is a Python framework for web scraping

There are tonnes of *commercial options* including “coding free” ones

EXAMPLE 0

Scrape weather information from the following web site:

https://climate.weather.gc.ca/climate_data/daily_data_e.html?StationID=51459

Twist: data are easily available in CSV format

EXAMPLE 1: STATIC WEB PAGE

Scrape book information from the following web site:

https://books.toscrape.com/index.html

```python
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin

url = 'https://books.toscrape.com/index.html'

while True:
    response = requests.get(url)
    response.encoding = 'UTF-8'
    soup = BeautifulSoup(response.text, 'html.parser')

    article_list = soup.select('article')
    for article in article_list:
        title = article.select_one('h3 a')['title']
        price = float(article.select_one('p.price_color').text[1:])
        stars_number = article.select_one('p.star-rating')['class'][1]
        numbers = {'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
        stars = numbers[stars_number]
        print(f'"{title}",{price},{stars}')

    if next_link := soup.select_one('li.next a'):
        url = urljoin(url, next_link['href'])
    else: break
```

COMMENTS

- Error handling *is a must*
- Checkpoints when scraping massive amounts
- books.toscrape.com is a scraping-*friendly* website

DYNAMIC CONTENT & JAVASCRIPT

JavaScript is a scripting language used for creating content dynamically by *manipulating the DOM*

```html
<!DOCTYPE html>
<html>
  <body>
    <p id="content">Hello</p>
    <script>
      document.getElementById("content").innerHTML += ", world!";
    </script>
  </body>
</html>
```

- When the page loads, it will initially show a paragraph with the text “Hello”
- The browser will then execute the JavaScript instructions in the `<script>` element
- That will modify the text to “Hello, world!”
- Dynamic web content cannot be scraped like in the book store example
  - The requests Python package only retrieves the HTTP response (the HTML source code)
  - It cannot execute the JavaScript and render the page like a browser
- The page may *take some time* to fully render if the script is complex

EXAMPLE 2: API REQUESTS

Scrape movie information from: http://www.scrapethissite.com/pages/ajax-javascript/

Here we essentially bypass the web page and go directly to the data source

```python
import requests, json

url = 'https://www.scrapethissite.com/pages/ajax-javascript/?ajax=true&year={year}'

for year in range(2010, 2016):
    response = requests.get(url.format(year=year))
    data = json.loads(response.text)
    for movie in data:
        print('"{title}",{year},{awards},{nominations}'.format(**movie))
```

COMMENTS

- This is hardly *real* web scraping
  - The “hard” part was figuring out the API access point
- The data came to us in JSON format, which is much easier than HTML
- In real situations, API requests may be refused unless a *cookie* (or another header) is provided
  - The cookie can be transplanted from a browser, but it may expire quickly

EXAMPLE 3: DYNAMIC HTML CONTENT

Scrape book information from the following web site:

https://quotes.toscrape.com/js/

When we can’t get to the data source (or it’s not useful):

- Selenium WebDriver can be used to control an *actual web browser* from Python
  - Meant for website *testing* primarily
- That is much slower than retrieving using requests
- Selenium alternatives: Puppeteer, Playwright

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from bs4 import BeautifulSoup

url = 'https://quotes.toscrape.com/js/'

driver = webdriver.Firefox()
driver.get(url)
while True:
    rendered_html = driver.page_source
    soup = BeautifulSoup(rendered_html, 'html.parser')

    tag_list = soup.select('a.tag')
    for tag in tag_list:
        print(tag.text)

    try:
        next_link = driver.find_element(By.CSS_SELECTOR, 'li.next a')
        next_link.click()
    except: break

driver.quit()
```

COMMENTS

- We could get the “next” link like in the book store example
- Rendering the page with JavaScript could take some time
  - Selenium has mechanisms to *wait* for an element to appear on the page
- The browser can usually run in *headless* mode

BOT DETECTION & AVOIDANCE

- Check the www.example.com/robots.txt file for site-specific rules
- Try to appear more like a normal web browser by including a realistic *user-agent* header
  - Also rotate user-agents occasionally
- Add a little bit of *random sleep* between requests
- Rotate IPs using a *proxy* service
- If Selenium is detected as a bot, you could
  - Tweak the web driver (hard)
  - Use *Undetected ChromeDriver* (easy, doesn't always work)
- Captchas are difficult but not impossible to tackle
  - Solve the captcha yourself if you have time
  - Use a captcha solving service