Web Scraping

Lukas Hager

2024-04-23

Learning Objectives

- Understand the appropriate use of web scraping as a tool
- Know how to use the `requests` module to access a page's HTML
- Use `BeautifulSoup` to parse HTML
- Learn to inspect pages and understand how to best extract data

Ethics

Is Web Scraping Legal?

Ignorantia juris non excusat.

Is Web Scraping Legal?

Ignorantia juris non excusat.

- Probably

Is Web Scraping Ethical?

- That depends
  - Is the data that you're scraping publicly available?
  - Are you scraping to sell the data?
  - Are you scraping to make money off of using the data otherwise?
  - Are you disrupting use of the site by others?
- Check out `robots.txt` file if it exists
  - Guide for Google's and Yahoo's crawlers
  - Newly, also for ChatGPT and other LLMs

Basics

What is Web Scraping?

- Acquiring data from a website programmatically
- Imagine copy-pasting tabular data from webpages, but using code
- Very useful for creating datasets that are directly of interest

Basic Goal

- We want to ingest HTML from a page and return a dataframe of the data that we want from that page

requests

- Python has a built-in module for accessing webpages
- Here, we're going to access Yahoo's webpage of Apple's stock price history
    - This is hosted on my website so we don't get in trouble

```
import requests

req_obj = requests.get(
    'https://lukashager.netlify.app/econ-481/data/yahoo_apple
)
```

`status_code`

A `status_code` of 200 means that your request was successful

```
req_obj.status_code
```

```
200
```

ok

You can more directly make sure the request was successful with the ok attribute

```
req_obj.ok
```

```
True
```

Content

What did requests actually give us? The HTML of the site we requested:

```
req_obj.text[:2000]
```

```
'<!DOCTYPE html>\n<!-- saved from url=(0045)https://finance.yaho
```

HTML

- If we request a webpage, we're going to get back the page's HTML code
- If the site is very simple, the HTML generates the full output (imagine really old MS Paint websites)
- Now, normally there are scripts that also run within the page and render objects or request additional data – can make things tricky

HTML Structure

A very basic piece of HTML might look like this:

```
<div>
    <h4> This is a heading </h4>
    <p> This is a sentence </p>
</div>
```

HTML Tables

We often care about tables when web scraping – they look like this:

```
<table>
    <thead>
        <tr>
            <td> a column </td>
            <td> another column </td>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td> 1 </td>
            <td> 2 </td>
        </tr>
    </tbody>
</table>
```

a column	another column
1	2

BeautifulSoup

- A library that we can use to process HTML more easily
- Facilitates searching for specific elements in a webpage

prettify()

We can use BeautifulSoup to make the page's HTML more nicely formatted and readable – we won't actually run the command here since the output is quite large.

```python
from bs4 import BeautifulSoup

soup = BeautifulSoup(req_obj.text)
# print(soup.prettify()) this would print formatted HTML
```

Searching in the HTML

- We only want some of the HTML – how should we figure out what we want and where it is?
- Easiest way: using Chrome, press Ctrl + Shft + C on PC or Cmd + Shft + C on Mac
- Alternatively (still in Chrome) right click on the element you care about and click “Inspect”

Finding `table` in `BeautifulSoup`

`BeautifulSoup` allows us to search for specific HTML tag by name – if there's only one table in the page, using `find` with the tag's name should work well

```
table_obj = soup.find('table')
```

Making this Usable

- We have a `thead` tag
  - Represents the headers of the table
- We then have `tbody` tag
  - Represents the table content

Getting the Headers

```
table_obj.find('thead').find_all('th')
```

```
[<th class="svelte-ta1t6m">Date  </th>,
 <th class="svelte-ta1t6m">Open  </th>,
 <th class="svelte-ta1t6m">High  </th>,
 <th class="svelte-ta1t6m">Low  </th>,
 <th class="svelte-ta1t6m">Close <span class="container svelte-u
 <th class="svelte-ta1t6m">Adj Close <span class="container svel
 <th class="svelte-ta1t6m">Volume  </th>]
```

Converting BeautifulSoup to List

```
headers = [x.text for x in table_obj.find('thead').find_all('t
headers
```

```
['Date  ',
 'Open  ',
 'High  ',
 'Low  ',
 'Close    Close price adjusted for splits. ',
 'Adj Close    Adjusted close price adjusted for splits and divi
 'Volume  ']
```

Converting BeautifulSoup to List

```
headers = [x.text for x in table_obj.find('thead').find_all('
headers
```

```
['Date  ',
 'Open  ',
 'High  ',
 'Low  ',
 'Close    Close price adjusted for splits. ',
 'Adj Close    Adjusted close price adjusted for splits and divi
 'Volume  ']
```

Ugly, but usable!

Making Headers Prettier

```
import re
headers_pretty = [
    re.findall('[A-Za-z]+\s?[A-Za-z]+(?=\s+)', x)[0] for x in
]
headers_pretty
```

```
['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']
```

Exercise: Getting Table Contents

Use a similar approach to get the contents of the table using `BeautifulSoup`. Hint: the HTML tag for the body is `tbody`, the tag for rows is `tr`, and the tag for elements is `td`.

Solutions: Getting Table Contents

```python
rows = table_obj.find('tbody').find_all('tr')
data_list = [[item.text for item in row.find_all('td')] for r
data_list[:2]
```

```
[['Mar 25, 2024',
  '170.53',
  '171.94',
  '169.45',
  '170.85',
  '170.85',
  '53,895,981'],
 ['Mar 22, 2024',
  '171.76',
  '173.05',
  '170.06',
  '172.28',
  '172.28',
  '71,106,600']]
```

Putting It Together

```python
import pandas as pd

df = pd.DataFrame(
    data = data_list,
    columns = headers_pretty
).set_index('Date')
print(df.head(10))
```

```
                Open    High     Low   Close Adj Close       Vol
Date
Mar 25, 2024  170.53  171.94  169.45  170.85    170.85   53,895,
Mar 22, 2024  171.76  173.05  170.06  172.28    172.28   71,106,
Mar 21, 2024  177.05  177.49  170.84  171.37    171.37  106,181,
Mar 20, 2024  175.72  178.67  175.09  178.67    178.67   53,423,
Mar 19, 2024  174.34  176.61  173.03  176.08    176.08   55,215,
Mar 18, 2024  175.57  177.71  173.52  173.72    173.72   75,604,
Mar 15, 2024  171.17  172.62  170.29  172.62    172.62  121,664,
Mar 14, 2024  172.91  174.31  172.05  173.00    173.00   72,913,
Mar 13, 2024  172.77  173.19  170.76  171.13    171.13   52,488,
Mar 12, 2024  173.15  174.03  171.01  173.23    173.23   59,825,
```

Faster Solution (If Possible)

```
print(pd.read_html(req_obj.text)[0].head(10))
```

```
           Date    Open    High     Low  \
0  Mar 25, 2024  170.53  171.94  169.45
1  Mar 22, 2024  171.76  173.05  170.06
2  Mar 21, 2024  177.05  177.49  170.84
3  Mar 20, 2024  175.72  178.67  175.09
4  Mar 19, 2024  174.34  176.61  173.03
5  Mar 18, 2024  175.57  177.71  173.52
6  Mar 15, 2024  171.17  172.62  170.29
7  Mar 14, 2024  172.91  174.31  172.05
8  Mar 13, 2024  172.77  173.19  170.76
9  Mar 12, 2024  173.15  174.03  171.01

  Close Close price adjusted for splits.  \
0                               170.85
1                               172.28
2                               171.37
3                               178.67
4                               176.08
```

When Does `pd.read_html` Work?

- If you have a simple `<table>` tag, `pd.read_html` will probably work
- Note that the column names are still ugly – `pd.read_html` will follow our approach above
- Note that in this *specific* application, there are plenty of sites that allow direct CSV download of stock data, so scraping is superfluous

Harder Scraping Problem

Baseball Reference

```
gunnar = requests.get('https://www.baseball-reference.com/pla
gunnar.status_code
```

```
200
```

Baseball Reference

```
gunnar = requests.get('https://www.baseball-reference.com/pla
gunnar.status_code
```

```
200
```

Let's get the "Advanced Batting" table

Find “Advanced Batting”

There are a lot of tables on the page

```
gunnar_bs = BeautifulSoup(gunnar.text)
len(gunnar_bs.find_all('table'))
```

```
2
```

Find "Advanced Batting"

There are a lot of tables on the page

```
gunnar_bs = BeautifulSoup(gunnar.text)
len(gunnar_bs.find_all('table'))
```

```
2
```

Huh?

Use pandas

```
pd.read_html(gunnar.text)
```

```
[         Date   Tm Unnamed: 2  Opp       Result Pos   AB   R   H  2
 0  2024-05-12  BAL        NaN  ARI        L 2-9  SS    4   0   0
 1  2024-05-11  BAL        NaN  ARI  W 5-4 (11)  SS    4   1   2
 2  2024-05-10  BAL        NaN  ARI        W 4-2  SS    3   1   1
 3  2024-05-08  BAL          @  WSN  W 7-6 (12)  SS    5   2   2
 4  2024-05-07  BAL          @  WSN        L 0-3  SS    4   0   0

    ROE  BOP    WPA   aLI    cWPA  acLI    RE24  PO  A
 0     0    1 -0.095  0.93  -0.07%  1.08  -0.998   2  1
 1     0    1  0.145  1.83   0.10%  2.10   1.332   1  4
 2     0    1  0.076  0.77   0.06%  0.92   0.817   2  2
 3     0    1  0.040  1.61   0.03%  1.88   0.598   4  5
 4     0    1 -0.096  0.92  -0.07%  1.10  -0.876   0  2

 [5 rows x 30 columns],
             Year            Age           Tm            Lg
 0           2019             18         BAL-min             Rk
 1           2021             20         BAL-min        A+ A AA
```

id tag

```
gunnar_bs.find_all('div', {'id': 'div_batting_advanced'})
```

```
[]
```

id tag

```
gunnar_bs.find_all('div', {'id': 'div_batting_advanced'})
```

```
[]
```

Huh?

Search the Raw HTML

```
re.findall('Advanced Batting', gunnar.text)
```

```
['Advanced Batting',
 'Advanced Batting',
 'Advanced Batting',
 'Advanced Batting',
 'Advanced Batting']
```

Search the Raw HTML

```python
re.findall('Advanced Batting', gunnar.text)
```

```
['Advanced Batting',
 'Advanced Batting',
 'Advanced Batting',
 'Advanced Batting',
 'Advanced Batting']
```

- So in the HTML there is an Advanced Batting table somewhere – it's just being stored in a bizarre format.
- At this point, best to just inspect the raw HTML

Getting Tables in Comments

We know that tables are sneakily hidden in comments – how can we get around this?

Getting Tables in Comments

We know that tables are sneakily hidden in comments – how can we get around this?

```
all_tables = re.findall(
    '\<table.*?\</table\>',
    gunnar.text,
    flags=re.DOTALL
)
len(all_tables)
```

```
47
```

Get The Table We Care About

```
adv_batting = [x for x in all_tables if 'batting_advanced' in
len(adv_batting)
```

```
1
```

Use pd.read_html

```
from io import StringIO # used to wrap raw text passed to panc

pd.read_html(StringIO(adv_batting[0]))
```

```
[  Unnamed: 0_level_0 Unnamed: 1_level_0 Unnamed: 2_level_0 Unna
                 Year                Age                 Tm
 0               2022                 21                BAL
 1               2023                 22                BAL
 2               2024                 23                BAL
 3              3 Yrs              3 Yrs              3 Yrs
 4       MLB Averages       MLB Averages       MLB Averages

  Batting                           Batting Ratios          ... Batted
     rOBA Rbat+  BAbip    ISO                 HR%    SO%  ...
 0   0.348   128  0.333  0.181               3.0%  25.8%  ...
 1   0.349   125  0.306  0.234               4.5%  25.6%  ...
 2   0.395   159  0.300  0.297               6.8%  25.6%  ...
 3   0.358   131  0.309  0.238               4.7%  25.6%  ...
 4   0.320   100  0.293  0.158               3.0%  22.6%  ...
```

Baseball Reference is Tricky

This behavior is driven by the site identifying that we're a bot, not a human – the HTML they serve to a human works great:

```
human_tables = pd.read_html('https://lukashager.netlify.app/e
len(human_tables)
```

```
31
```

Baseball Reference is Tricky

This behavior is driven by the site identifying that we're a bot, not a human – the HTML they serve to a human works great:

```
human_tables = pd.read_html('https://lukashager.netlify.app/ec
len(human_tables)
```

```
31
```

To get the table we want, we should specify the `match` keyword

```
pd.read_html(
    'https://lukashager.netlify.app/econ-481/data/gunnar_hend
    match = 'Oppo%'
)
```

```
[  Unnamed: 0_level_0 Unnamed: 1_level_0 Unnamed: 2_level_0 Unna
                 Year                Age                 Tm
 0               2022                 21                BAL
 1               2023                 22                BAL
 2              2 Yrs              2 Yrs              2 Yrs
```

Using APIs

Data Population on Page

- It's possible that a page is sending a request itself to get the data that it uses to populate a page
- If possible, it's more efficient to try to request that API directly instead of scraping the HTML
- Can sometimes be hard – depending on the API, you may need to authenticate or pass cookies

Western States

- The oldest and most prestigious ultramarathon in the United States
- 100 miles from Olympic Valley, CA to Auburn, CA
- Used to be a horse race
  - Gordy Ainsleigh ran it in 24:42 in 1974

Figure 1: Gordy Ainsleigh, courtesy of Western States 100

Western States Results

Stored in tabular format – can we use `pd.read_html`?

```
pd.read_html('https://ultrasignup.com/results_event.aspx?did=9
```

```
[          0       1       2       3
 0       NaN     NaN     NaN     NaN
 1    2023.0  2022.0  2021.0  2020.0
 2    2019.0  2018.0  2017.0  2016.0
 3    2015.0  2014.0  2013.0  2012.0
 4    2011.0  2010.0  2009.0  2007.0
 5    2006.0  2005.0  2004.0  2003.0
 6    2002.0  2001.0  2000.0  1999.0
 7    1998.0  1997.0  1996.0  1995.0
 8    1994.0  1993.0  1992.0  1991.0
 9    1990.0  1989.0  1988.0  1987.0
 10   1986.0  1985.0  1984.0  1983.0
 11   1982.0  1981.0  1980.0  1979.0
 12   1978.0  1977.0  1976.0  1974.0
 13      NaN     NaN     NaN     NaN]
```

Western States Results

Stored in tabular format – can we use `pd.read_html`?

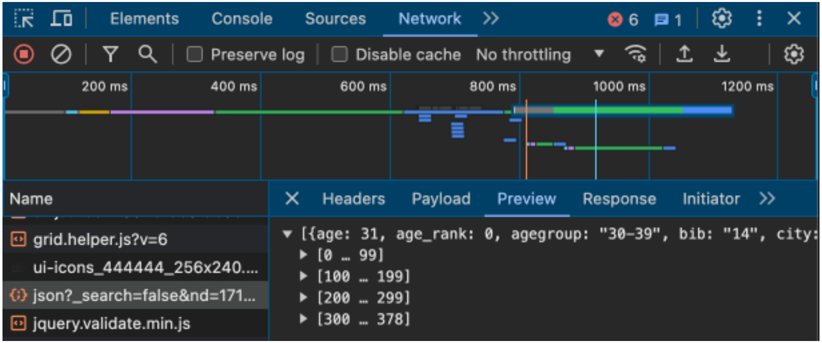
```
pd.read_html('https://ultrasignup.com/results_event.aspx?did=
```

```
[        0       1       2       3
 0     NaN     NaN     NaN     NaN
 1  2023.0  2022.0  2021.0  2020.0
 2  2019.0  2018.0  2017.0  2016.0
 3  2015.0  2014.0  2013.0  2012.0
 4  2011.0  2010.0  2009.0  2007.0
 5  2006.0  2005.0  2004.0  2003.0
 6  2002.0  2001.0  2000.0  1999.0
 7  1998.0  1997.0  1996.0  1995.0
 8  1994.0  1993.0  1992.0  1991.0
 9  1990.0  1989.0  1988.0  1987.0
 10 1986.0  1985.0  1984.0  1983.0
 11 1982.0  1981.0  1980.0  1979.0
 12 1978.0  1977.0  1976.0  1974.0
 13    NaN     NaN     NaN     NaN]
```

Network Tab in Chrome

- This shows what a webpage is doing as it loads
  - For example, we can see the images it loads, the scripts it deploys
  - Importantly: can see the APIs it requests
- In this case, we see that it's requesting a `json` that looks like what we want

Network Tab json Request



grid.helper.js?v=6
ui-icons_444444_256x240....
json?_search=false&nd=171...
jquery.validate.min.js

```
▼ [{age: 31, age_rank: 0, agegroup: "30-39", bib: "14", city:
  ▶ [0 … 99]
  ▶ [100 … 199]
  ▶ [200 … 299]
  ▶ [300 … 378]
```

Figure 2: The JSON

Requesting Directly

If the page makes this request, it stands to reason that we can as well

```
url_2023 = 'https://ultrasignup.com/service/events.svc/results
ws_req = requests.get(url_2023)
ws_req.ok
```

```
True
```

Passing json to DataFrame

```
print(pd.DataFrame(ws_req.json()))
```

```
     age  age_rank agegroup  bib          city distance_time  dr
0     26         0    20-29   34      Missoula             0
1     31         0    30-39         Cedar City             0
2     29         0    20-29   19     Massillon             0
3     38         0    30-39           Portland             0
4     30         0    30-39            Boulder             0
..   ...       ...      ...  ...           ...           ...
378   40         0    40-49  419        London             0
379   39         0    30-39          Sonneberg             0
380   34         0    30-39           Masevaux             0
381   70         0      70+  235          Bend             0
382   45         0    40-49  165  Peterborough             0

     firstname formattime gender  ...  lastname participant_id p
0         Adam   15:13:48      M  ...  Peterman        1854799
1       Hayden   15:47:27      M  ...     Hawks        2310798
2        Arlen   15:56:17      M  ...     Glick        2392278
3        Tyler   15:57:10      M  ...     Green        2392344
```

Scraping Multiple Pages

Data Stored on Multiple Pages

- It's uncommon that we get everything we need from one page
  - At that point, unclear what the value of scraping is
- Often have to iterate over multiple pages and combine results

Scraping Multiple Years of WS Results

- Say we wanted results from 2022 and 2023
- Inspecting our URL leads us to believe that all we need to change is the event ID
- We can grab the 2022 and 2023 IDs:

```
ev_ids = ['87878', '97204']
```

Plugging Into API

Warning

ALWAYS put breaks in your code. If you do not, you may crash the site and get into ***serious*** trouble.

```python
import time

df_list = []
for ev_id in ev_ids:
    r = requests.get(
        f'https://ultrasignup.com/service/events.svc/results/
    )
    df = pd.DataFrame(r.json())
    df['event_id'] = ev_id
    df_list.append(df)
    time.sleep(5)

print(pd.concat(df_list).head(10))
```

Getting Event IDs

Kalvin made a great point – how would we get the identifiers programmatically?

```
r = requests.get('https://ultrasignup.com/results_event.aspx?
bs = BeautifulSoup(r.text)
r.ok
```

```
True
```

Getting the Elements

```
table = bs.find('table', {'id':'ContentPlaceHolder1_dlYears'}
rows = table.find_all('tr')
elements = [row.find_all('td') for row in rows]
years = [y.text.strip() for x in elements[1:-1] for y in x]
links = [y.find('a')['href'] for x in elements[1:-1] for y in
ids = [re.findall('\d+', x)[0] for x in links]
```

Putting Together

```
print(pd.DataFrame({'year': years, 'link': links, 'ev_id': id
```

```
    year                           link  ev_id
0   2023  /results_event.aspx?did=97204  97204
1   2022  /results_event.aspx?did=87878  87878
2   2021  /results_event.aspx?did=79446  79446
3   2020  /results_event.aspx?did=71208  71208
4   2019  /results_event.aspx?did=61359  61359
5   2018  /results_event.aspx?did=51243  51243
6   2017  /results_event.aspx?did=41765  41765
7   2016  /results_event.aspx?did=34773  34773
8   2015  /results_event.aspx?did=30033  30033
9   2014  /results_event.aspx?did=24962  24962
10  2013  /results_event.aspx?did=17746  17746
11  2012  /results_event.aspx?did=14050  14050
12  2011  /results_event.aspx?did=10804  10804
13  2010   /results_event.aspx?did=5752   5752
14  2009   /results_event.aspx?did=4742   4742
15  2007    /results_event.aspx?did=771    771
16  2006   /results_event.aspx?did=5705   5705
```

Authentication

Problem: Sites Recognize Bots

- We've seen a few times that websites will recognize that we're not people
- This is due to a few possible reasons:
  - Our requests don't have payloads that indicate that we're humans
  - There's some sort of human verification on the page
- The former we can resolve – the latter, not so much

Examples

- Baseball Reference hides their data in comments
- UltraSignup hides their data entirely

Potential Solution: Authenticate

- Some sites will return different HTML to scrapers depending on whether or not the scraper is an authenticated account
- In this case, we can do the following:

1. Send a POST request to the site to authenticate
2. Use a GET request to get the data from the site/API

Issue

- I don't want to hardcode any of my passwords into this presentation
- I don't have a good example where I can create a throwaway account

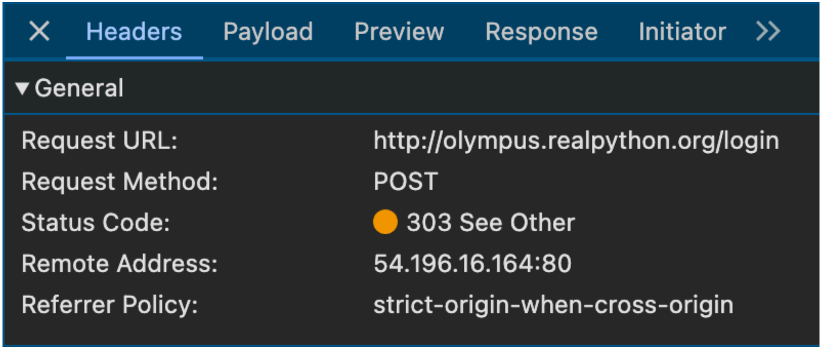
Very Simple – Olympus

- Thanks to Real Python there's a practice login page located here
- We can inspect the form and see what the POST request needs to look like to authenticate

Exercise: Olympus

Find the POST request that occurs on this page when you log in using username “zeus” and password “ThunderDude”. Remember to use Chrome and use CTRL+SHIFT+C.

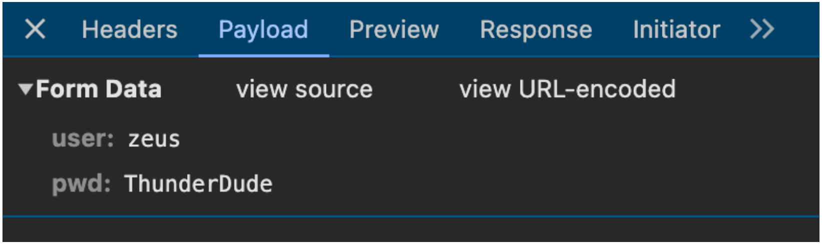
Olympus Headers



Request URL:	http://olympus.realpython.org/login
Request Method:	POST
Status Code:	● 303 See Other
Remote Address:	54.196.16.164:80
Referrer Policy:	strict-origin-when-cross-origin

Figure 3: Headers

Olympus Payload



```
user: zeus
pwd: ThunderDude
```

Figure 4: Payload

Constructing the Request

This is all we need to get the HTML behind the login:

```
headers = {'user': 'zeus', 'pwd': 'ThunderDude'}
r = requests.post(
    'http://olympus.realpython.org/login',
    data = headers
)
```

Exercise: Extracting Links

Use `BeautifulSoup` to extract all the links from the logged-in Olympus page.

Solutions: Extracting Links

We can use `BeautifulSoup`'s `find_all` with the a tag.

```
olympus_bs = BeautifulSoup(r.text)
[x['href'] for x in olympus_bs.find_all('a')]
```

```
['/profiles/aphrodite', '/profiles/poseidon', '/profiles/dionysu
```

UW Economics Database

Goal (Scraping Non-Tabular Data)

We want to list every graduate student in Economics at UW as well as whatever data the Department of Economics makes publicly available about them.

Exercise: Strategy

Come up with the strategy we should use to accomplish this task. Don't write any code, but figure out what pages would be helpful, and think about what sort of code we'll need to write.

Approach

1. List all of the links on this page
2. For each link, extract all of the relevant data from wherever it's stored.

Step 1

```
r1 = requests.get('https://econ.washington.edu/people/graduat
assert r1.ok
r1_bs = BeautifulSoup(r1.text)
```

To get all the links, we could try what we did before:

```
links = r1_bs.find_all('a')
links
```

```
[<a class="visually-hidden focusable skip-link" href="#main-cont
     Skip to main content
   </a>,
 <a class="uw-link" href="https://www.washington.edu">
 <div class="w-logo"><svg aria-labelledby="W_Title" data-name="W
 <div class="university-wordmark show-for-medium-up"><svg aria-l
 </a>,
 <a class="artsci-link show-for-medium-up" href="https://artsci.
 <a data-drupal-link-system-path="node/636" href="/support-us" t
 <a href="https://uw.edu/directory" title="">Directories</a>,
```

Cutting Down to Useful Links

```
url_pattern=re.compile('https://econ.washington.edu/people/[a
links = [x['href'] for x in r1_bs.find_all('a', {'href': url_
links[:5]
```

```
['https://econ.washington.edu/people/amre-abken',
 'https://econ.washington.edu/people/afsana-adiba',
 'https://econ.washington.edu/people/shabab-ahmed',
 'https://econ.washington.edu/people/alireza-aminkhaki',
 'https://econ.washington.edu/people/erik-andersen']
```

Step 2

Note that the useful data is stored in `<div>` elements with consistently formatted class names:

```
r2 = requests.get('https://econ.washington.edu/people/amre-abl
assert r2.ok
r2_bs = BeautifulSoup(r2.text)

data_dict = {}
field_names = ['email', 'office', 'office-hours', 'biography']
for field_name in field_names:
    search_crit = {'class': re.compile(f'field-name-field-{fie
    search_obj = r2_bs.find_all('div', search_crit)
    if len(search_obj) > 0:
        data_dict[field_name] = search_obj[0].text.strip()
data_dict
```

```
{'email': 'abken@uw.edu',
 'office': 'Savery Hall 319F',
 'office-hours': 'Office Hours\n\nMonday 330-430pm; Wednesday 33
 'biography': 'Nazarbayev University'}
```

DataFrame

```python
dfs = []
for url in links[:5]:
    r2 = requests.get(url)
    assert r2.ok
    r2_bs = BeautifulSoup(r2.text)

    data_dict = {}
    field_names = ['email', 'office', 'office-hours', 'biogra
    for field_name in field_names:
        search_crit = {'class': re.compile(f'field-name-field
        search_obj = r2_bs.find_all('div', search_crit)
        if len(search_obj) > 0:
            data_dict[field_name] = search_obj[0].text.strip(
    name = re.findall('(?<=https://econ.washington.edu/people
    dfs.append(pd.DataFrame(data = data_dict, index=[name]))
    time.sleep(3)
print(pd.concat(dfs))
```

```
                          email             office  \
amro-abken         abken@uw.edu  Savery Hall 319F
```

Aside: Email Addresses

This is exactly why you should think hard about putting your email address anywhere on a page – they're extremely easy to extract and spam.

Cookies

What is a Cookie?

- Identifiers for a site about your computer
- For example, keeping you logged in
- Broadly, allowing the site to remember things about your visit
- Can be useful/necessary to convince a site that your scraper should be served real stuff

Cookies with `requests`

We can save cookies from a request with `requests.session()`:

```
s = requests.session()
google_r = s.get('https://www.google.com')
google_r.ok
```

```
True
```

Look at Cookies

```
[x for x in s.cookies]
```

```
[Cookie(version=0, name='AEC', value='AQTF6Hwzzklpypcw3bzQBd_Nar
 Cookie(version=0, name='NID', value='514=cibMpLwvWqw33GdmEqqUWC
```

Passing These Cookies

You could now use these cookies in a new request with

```
requests.get(<url>, cookies=s.cookies)
```

A common use case is authenticating, and then passing the cookies resulting from the authentication to future requests.

Headless Browsers

Use Cases

- Sometimes if a page is sufficiently complicated (think Amazon), you can use a headless browser to scrape a page
- This is essentially giving a browser instructions on what to do
  - First, click this button, then type this text in this field, etc.
- Not obvious that it's super valuable in my experience
- If interested, look at Selenium