Week 1: Data Collection for Machine Learning

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

The Netflix Movie Recommendation Problem

Scenario: You work at Netflix as a data scientist.

Your manager asks:

> "We need to decide which movies to add to our catalog next month. Build a system to predict which movies will be successful with our subscribers."

This is a machine learning problem.

But before we can build any model, we need data.

The ML Pipeline

Every machine learning project follows this pipeline:

```
DATA COLLECTION → DATA CLEANING → FEATURE ENGINEERING →
MODEL TRAINING → EVALUATION → DEPLOYMENT → MONITORING
```

Week 1 (Today): Data Collection - the foundation of everything

The principle: "Garbage in, garbage out"

- Bad data leads to bad models
- No amount of sophisticated algorithms can fix poor quality data

What Data Do We Need?

To predict movie success, we need features:

Basic Information:

- Title, year, genre, runtime
- Director, main actors
- Plot summary

Performance Metrics:

- IMDb rating, number of votes
- Rotten Tomatoes score
- Box office revenue

What Data Do We Need? (continued)

Production Details:

- Budget, production companies
- Country of origin
- Language

Audience Signals:

- User reviews and sentiment
- Social media mentions
- Awards and nominations

Question: Where can we get all this data?

Data Sources on the Web

Option 1: Public APIs

- IMDb API, OMDb API, TMDb API
- Structured, reliable, well-documented
- Rate limits, authentication required

Option 2: Web Scraping

- Extract data directly from HTML pages
- Flexible, can get data not available via API
- More fragile, can break if site changes

Today: We'll learn both approaches

Part 1: Understanding the Web

How does the web actually work?

Client-Server Architecture

Source: diagrams/generate_client_server.py

Client: Your web browser (Chrome, Firefox, Safari)

Server: Computer hosting the website (e.g., netflix.com)

Communication happens via HTTP protocol

HTTP Request-Response Cycle

Step 1: You type URL in browser or click a link

Step 2: Browser sends HTTP request to server

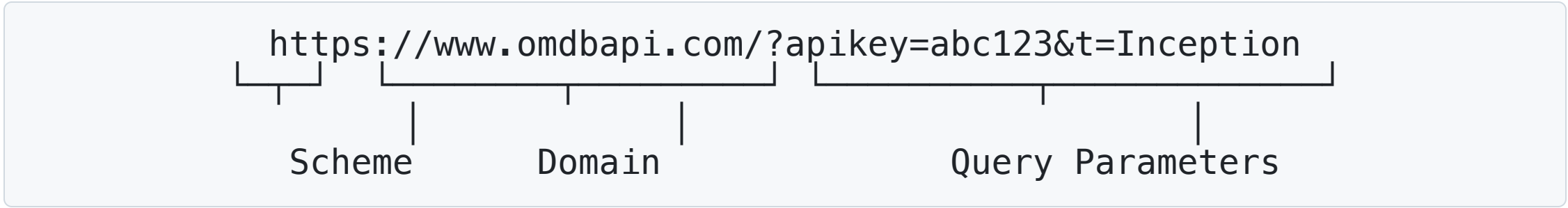
Step 3: Server processes request

Step 4: Server sends HTTP response back

Step 5: Browser renders the page

Key insight: Every web page you see is the result of this cycle

Anatomy of a URL



Scheme: Protocol (http vs https)

Domain: Server address

Query Parameters: Data sent to server (key=value pairs)

HTTP Methods

GET: Retrieve data from server

- Example: Get movie information
- Safe operation, doesn't change server state

POST: Send data to server

- Example: Submit a form, upload a file
- Can modify server state

Others: PUT, DELETE, PATCH

- For updating and deleting resources

HTTP Status Codes

2xx - Success:

- `200 OK`: Request succeeded
- `201 Created`: Resource created successfully

4xx - Client Error:

- `400 Bad Request`: Invalid request
- `401 Unauthorized`: Authentication required
- `404 Not Found`: Resource doesn't exist

HTTP Status Codes (continued)

5xx - Server Error:

- `500 Internal Server Error` : Server crashed
- `503 Service Unavailable` : Server overloaded

Why this matters: Your data collection code needs to handle these!

HTTP Headers

Headers provide metadata about the request/response

Common Request Headers:

```
User-Agent: Mozilla/5.0 (identifies your browser)
Accept: application/json (what format you want)
Authorization: Bearer token123 (authentication)
```

Common Response Headers:

```
Content-Type: application/json (what format you're getting)
Content-Length: 1234 (size in bytes)
```

Data Formats on the Web

HTML: Markup language for web pages

```
<h1>Inception</h1>
<p>Rating: 8.8</p>
```

JSON: JavaScript Object Notation (most common for APIs)

```
{"title": "Inception", "rating": 8.8}
```

XML: Extensible Markup Language (older APIs)

```
<movie><title>Inception</title></movie>
```

Part 2: curl - Command Line HTTP

Testing APIs from the terminal

What is curl?

curl: Command-line tool for making HTTP requests

Why use it?

- Quick API testing without writing code
- See exact request/response data
- Debug API issues
- Works on any platform

Installation: Already on Mac/Linux, Windows: download from curl.se

Basic curl Syntax

```
curl "https://api.example.com/movies"
```

This sends a GET request and prints the response to terminal

Common options:

```
  -X POST          # Specify HTTP method
   -H "Header"       # Add custom header
-d "data"        # Send data in request body
  -o file.json     # Save response to file
```

curl Example: OMDb API

Get information about "Inception":

```
curl "http://www.omdbapi.com/?apikey=YOUR_KEY&t=Inception"
```

Response (JSON):


```
{
  "Title": "Inception",
  "Year": "2010",
  "Rated": "PG-13",
  "imdbRating": "8.8",
  "Genre": "Action, Sci-Fi, Thriller"
}
```

Making curl Output Readable

Raw JSON is hard to read. Use `jq` to format it:

```
curl "http://www.omdbapi.com/?apikey=KEY&t=Inception" | jq
```

jq: JSON processor that pretty-prints and filters JSON

Output is now nicely formatted with colors and indentation

Extracting Specific Fields with jq

Get only the title and rating:

```
curl "http://www.omdbapi.com/?apikey=KEY&t=Inception" | \
          jq '{title: .Title, rating: .imdbRating}'
```

Output:

```
{
  "title": "Inception",
  "rating": "8.8"
}
```

jq syntax: `.FieldName` accesses a field

Part 3: Chrome DevTools

Inspecting web traffic in your browser

What is Chrome DevTools?

DevTools: Built-in browser tool for web development and debugging

How to open:

- Mac: `Cmd + Option + I`
- Windows/Linux: `F12` or `Ctrl + Shift + I`
- Right-click on page then "Inspect"

Key tabs: Elements, Network, Console

The Network Tab

Purpose: See all HTTP requests your browser makes

How to use:

1. Open DevTools
2. Go to Network tab
3. Reload the page
4. See list of all requests

What you can see: URL, method, status, size, time

Inspecting a Request

Click on any request in Network tab to see:

Headers tab:

- Request URL, method, status code
- Request headers (what you sent)
- Response headers (what you got back)

Preview/Response tab:

- Actual data returned by server
- Preview shows formatted version

Finding API Calls

Technique: Look for XHR/Fetch requests

1. Filter by "Fetch/XHR" in Network tab
2. These are AJAX requests (data fetched after page load)
3. Click to see the API endpoint
4. Copy the URL to use in your code

Example: Go to IMDb, search for a movie, watch Network tab

Copying as curl

Useful trick: Right-click request then "Copy as curl"

This gives you exact curl command to reproduce the request:

```
curl 'https://api.example.com/data' \
  -H 'authorization: Bearer token' \
   -H 'user-agent: Mozilla/5.0...'
```

Now you can test API outside the browser!

The Elements Tab

Purpose: Inspect HTML structure of a page

How to use:

1. Right-click element then "Inspect"
2. DevTools opens with that element highlighted
3. See HTML structure and CSS styles

Why this matters: Essential for web scraping!

Part 4: Python requests Library

Making HTTP requests programmatically

Why Python for Data Collection?

Automation: Collect data for hundreds of movies

Error Handling: Retry on failure, handle rate limits

Data Processing: Clean and structure data immediately

Integration: Save to database, CSV, or use in ML pipeline

requests library: Most popular HTTP library for Python

Installing requests

```
pip install requests
```

Import in your code:

```
import requests
```

Documentation: https://requests.readthedocs.io/

Basic GET Request

```
import requests

url = "http://www.omdbapi.com/"
params = {
    "apikey": "YOUR_KEY",
    "t": "Inception"
}

response = requests.get(url, params=params)
print(response.status_code)  # 200
print(response.json())       # Python dict
```

Note: `params` dict is converted to query string automatically

Handling the Response

```
response = requests.get(url, params=params)

# Status code
if response.status_code == 200:
    print("Success!")

# Headers
print(response.headers["Content-Type"])

# Body as text
print(response.text)

# Body as JSON (parsed into Python dict)
data = response.json()
print(data["Title"])  # "Inception"
```

Error Handling: Network Errors

Network requests can fail! Always handle exceptions:

```
try:
    response = requests.get(url, timeout=10)
    response.raise_for_status()
    data = response.json()
except requests.exceptions.Timeout:
    print("Request timed out")
except requests.exceptions.ConnectionError:
    print("Network error")
except requests.exceptions.HTTPError as e:
    print(f"HTTP error: {e}")
```

Error Handling: Invalid JSON

API might return invalid JSON:

```
                try:
                    data = response.json()
except requests.exceptions.JSONDecodeError:
        print("Response is not valid JSON")
    print(response.text)  # Print raw response
```

Common cause: API returned error page in HTML instead of JSON

Error Handling: Missing Fields

API might not return expected fields:

```
data = response.json()

# Bad: Will crash if Title doesn't exist
title = data["Title"]

# Good: Provide default value
title = data.get("Title", "Unknown")

# Better: Check if key exists
if "Title" in data:
    title = data["Title"]
else:
    print("No title in response")
```

Complete Example: Fetch Movie Data

```
import requests

def get_movie_data(title, api_key):
    url = "http://www.omdbapi.com/"
params = {"apikey": api_key, "t": title}
                try:
        response = requests.get(url,
                                params=params,
                                  timeout=10)
            response.raise_for_status()
              data = response.json()
        if data.get("Response") == "False":
                    return None
                return data
except requests.exceptions.RequestException:
                return None
```

Using the Function

```
api_key = "YOUR_KEY"
movie = get_movie_data("Inception", api_key)

if movie:
    print(f"Title: {movie['Title']}")
    print(f"Year: {movie['Year']}")
    print(f"Rating: {movie['imdbRating']}")
    print(f"Genre: {movie['Genre']}")
```

Output:

```
Title: Inception
Year: 2010
Rating: 8.8
Genre: Action, Sci-Fi, Thriller
```

Collecting Multiple Movies

```
titles = ["Inception", "The Matrix", "Interstellar"]
movies = []

for title in titles:
    movie = get_movie_data(title, api_key)
    if movie:
        movies.append(movie)

print(f"Collected {len(movies)} movies")
```

Problem: What if we have 1000 movies? This is slow!

Rate Limiting

Problem: APIs limit how many requests you can make

OMDb Free Tier: 1000 requests per day

Solution: Add delay between requests

```
import time

for title in titles:
    movie = get_movie_data(title, api_key)
    if movie:
        movies.append(movie)
    time.sleep(1)  # Wait 1 second between requests
```

Exponential Backoff

Better solution: Retry with increasing delays

```
import time

def get_movie_with_retry(title, api_key, max_retries=3):
    for attempt in range(max_retries):
        movie = get_movie_data(title, api_key)
        if movie:
            return movie
        wait_time = 2 ** attempt
        print(f"Retry {attempt + 1} after {wait_time}s")
        time.sleep(wait_time)
    return None
```

Part 5: BeautifulSoup - Web Scraping

Extracting data from HTML pages

When to Use Web Scraping

Use scraping when:

- No API available
- API doesn't provide all needed data
- API is too expensive or restrictive

Example: Rotten Tomatoes audience reviews

- No public API
- Must scrape from website

How Web Scraping Works

1. Send HTTP request to get HTML page
2. Parse HTML into a tree structure
3. Find specific elements (e.g., all movie titles)
4. Extract text or attributes
5. Clean and structure the data

Tool: BeautifulSoup library

Installing BeautifulSoup

```
pip install beautifulsoup4
pip install lxml  # HTML parser
```

Import:

```
from bs4 import BeautifulSoup
import requests
```

Basic BeautifulSoup Example

```
html = """
  <html>
<h1>Inception</h1>
<p class="rating">Rating: 8.8</p>
  <p class="genre">Sci-Fi</p>
  </html>
  """

soup = BeautifulSoup(html, 'lxml')

  # Find first h1 tag
  title = soup.find('h1')
print(title.text)  # "Inception"

  # Find element by class
rating = soup.find('p', class_='rating')
print(rating.text)  # "Rating: 8.8"
```

Scraping a Real Website

```
url = "https://www.imdb.com/title/tt1375666/"
response = requests.get(url)
soup = BeautifulSoup(response.text, 'lxml')

# Find movie title
title_elem = soup.find('h1')
if title_elem:
title = title_elem.text.strip()
print(f"Title: {title}")
```

Warning: Websites change! This might break.

Finding Elements

By tag name:

```
soup.find('h1')           # First h1
soup.find_all('p')        # All p tags
```

By class:

```
soup.find('div', class_='movie-info')
soup.find_all('span', class_='rating')
```

By id:

```
soup.find('div', id='main-content')
```

Finding Elements (continued)

By CSS selector:

```
soup.select('div.movie > h1')
soup.select('#main-content')
   soup.select('.rating')
```

CSS selectors are very powerful for complex queries

Extracting Data

```
# Get text content
elem = soup.find('h1')
text = elem.text.strip()

# Get attribute value
link = soup.find('a')
href = link['href']
# or: href = link.get('href')

# Get all text in children
div = soup.find('div', class_='info')
all_text = div.get_text(strip=True)
```

Finding Multiple Elements

```
# Find all movie cards on a page
movies = soup.find_all('div', class_='movie-card')

for movie in movies:
    title = movie.find('h2').text.strip()
    rating = movie.find('span', class_='rating').text
    print(f"{title}: {rating}")
```

Pattern: Find container elements, then extract data from each

Handling Missing Elements

```
movie = soup.find('div', class_='movie-card')

# Bad: Crashes if h2 doesn't exist
title = movie.find('h2').text

# Good: Check if element exists
title_elem = movie.find('h2')
if title_elem:
    title = title_elem.text.strip()
else:
    title = "Unknown"

# Alternative: Use try-except
try:
    title = movie.find('h2').text.strip()
except AttributeError:
    title = "Unknown"
```

Complete Scraping Example

```
def scrape_imdb_top_movies():
    url = "https://www.imdb.com/chart/top/"
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'lxml')
    movies = []

    for row in soup.select('tbody.lister-list tr')[:10]:
        title_col = row.find('td', class_='titleColumn')
        rating_col = row.find('td', class_='ratingColumn')
        if title_col and rating_col:
            title = title_col.a.text
            rating = rating_col.strong.text
            movies.append({'title': title,
                           'rating': rating})
    return movies
```

Web Scraping Ethics

Important rules:

1. **Check robots.txt**: `example.com/robots.txt`
   - Specifies what bots can/can't scrape
2. **Respect rate limits**: Don't overload servers
   - Add delays between requests
3. **Check Terms of Service**: Some sites prohibit scraping
4. **Use APIs when available**: They're designed for this!

Part 6: Playwright - JavaScript-Heavy Sites

When requests/BeautifulSoup isn't enough

The JavaScript Problem

Problem: Many modern sites load data dynamically with JavaScript

Example: Netflix loads movie thumbnails after page loads

When you use `requests.get()`:

- You get initial HTML
- JavaScript hasn't run yet
- Data you want isn't in the HTML!

Solution: Use a real browser that runs JavaScript

What is Playwright?

Playwright: Library to control a real browser programmatically

Features:

- Supports Chrome, Firefox, Safari
- Runs JavaScript, waits for elements
- Can screenshot, interact with page
- Much slower than requests!

When to use: Only when necessary (JS-heavy sites)

Installing Playwright

```
pip install playwright
playwright install chromium
```

This downloads Chromium browser (100+ MB)

Basic Playwright Example

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    page.goto("https://www.example.com")
    page.wait_for_selector('h1')
    html = page.content()
    browser.close()
```

Playwright with BeautifulSoup

```
from playwright.sync_api import sync_playwright
from bs4 import BeautifulSoup

def scrape_js_site(url):
with sync_playwright() as p:
browser = p.chromium.launch(headless=True)
page = browser.new_page()
page.goto(url)
page.wait_for_selector('.movie-card')
html = page.content()
browser.close()
soup = BeautifulSoup(html, 'lxml')
return soup
```

When to Use Playwright vs requests

Use requests + BeautifulSoup:

- Simple static websites
- Speed is important
- Scraping many pages

Use Playwright:

- JavaScript-heavy sites
- Need to interact (click, scroll, type)
- Data loads after page load

Rule of thumb: Try requests first, use Playwright if needed

Part 7: Working with APIs

Best practices for production data collection

REST API Basics

REST: Representational State Transfer

Key concepts:

- Resources identified by URLs
- HTTP methods indicate action (GET, POST, PUT, DELETE)
- Stateless (each request independent)
- Responses usually in JSON

Example: OMDb API is a REST API

API Authentication

Why: Prevent abuse, track usage, monetize

Common methods:

1. **API Key** (simplest):

```
params = {"apikey": "YOUR_KEY"}
```

2. **Bearer Token**:

```
headers = {"Authorization": "Bearer YOUR_TOKEN"}
```

3. **OAuth** (complex): Multi-step authentication flow

API Keys Best Practices

Never hardcode API keys!

```
# Bad
api_key = "sk_live_abc123xyz"

# Good: Use environment variables
import os
api_key = os.environ.get("OMDB_API_KEY")

# Better: Use python-dotenv
from dotenv import load_dotenv
load_dotenv()
api_key = os.getenv("OMDB_API_KEY")
```

Using .env Files

Create `.env` file:

```
OMDB_API_KEY=your_key_here
TMDB_API_KEY=another_key
```

Add to `.gitignore`:

```
.env
```

Load in code:

```
from dotenv import load_dotenv
import os

load_dotenv()
api_key = os.getenv("OMDB_API_KEY")
```

Rate Limiting

APIs limit requests to prevent abuse

OMDb: 1000/day (free tier)
TMDb: 40 requests per 10 seconds

How to handle:

1. Read API documentation
2. Add delays between requests
3. Implement exponential backoff
4. Cache responses locally

Implementing Rate Limiting

```
import time
from datetime import datetime, timedelta

class RateLimiter:
    def __init__(self, max_calls, period):
        self.max_calls = max_calls
        self.period = period
        self.calls = []

    def wait_if_needed(self):
        now = datetime.now()
        self.calls = [c for c in self.calls
                      if now - c <
                      timedelta(seconds=self.period)]
        if len(self.calls) >= self.max_calls:
            sleep_time = self.period - \
                (now - self.calls[0]).seconds
            time.sleep(sleep_time)
            self.calls.append(now)
```

Pagination

Problem: APIs don't return all data at once

Example: TMDb returns 20 movies per request

Solution: Make multiple requests with page parameter

```
all_movies = []
for page in range(1, 6):  # Get 5 pages
  params = {"apikey": key, "page": page}
response = requests.get(url, params=params)
data = response.json()
all_movies.extend(data["results"])
time.sleep(1)  # Rate limiting
```

Caching API Responses

Why: Avoid redundant requests, save API quota

```
import json
import os

def get_cached_or_fetch(movie_id, api_key):
    cache_file = f"cache/{movie_id}.json"
    if os.path.exists(cache_file):
        with open(cache_file) as f:
            return json.load(f)
    data = get_movie_data(movie_id, api_key)
    os.makedirs("cache", exist_ok=True)
    with open(cache_file, 'w') as f:
        json.dump(data, f)
    return data
```

Combining Multiple APIs

Strategy: Use IMDb ID as common key

```
def enrich_movie_data(imdb_id):
    omdb_data = get_omdb_data(imdb_id)
    tmdb_data = get_tmdb_data(imdb_id)

    movie = {
        "title": omdb_data.get("Title"),
        "year": omdb_data.get("Year"),
        "rating_imdb": omdb_data.get("imdbRating"),
        "rating_tmdb": tmdb_data.get("vote_average"),
        "budget": tmdb_data.get("budget"),
        "revenue": tmdb_data.get("revenue")
    }
    return movie
```

Handling API Changes

APIs change! Be defensive:

1. **Version your API calls**:

```
url = "https://api.example.com/v3/movies"
```

2. **Validate responses**:

```
required_fields = ["title", "year", "rating"]
if not all(field in data for field in required_fields):
    print("API response missing required fields")
```

3. **Monitor for errors**: Log when structure changes

Data Collection Best Practices

1. **Start small**: Test with 5-10 items before scaling
2. **Save raw responses**: Keep original JSON for debugging
3. **Handle errors gracefully**: Don't crash on one failure
4. **Log progress**: Know where you stopped if interrupted
5. **Validate data**: Check for missing/invalid fields
6. **Document your process**: Future you will thank you!

Building a Production Collector

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def collect_movies(movie_ids, api_key):
    results = []
    for i, movie_id in enumerate(movie_ids):
        logger.info(f"Collecting {i+1}/{len(movie_ids)}")
        try:
            data = get_movie_data(movie_id, api_key)
            if data:
                results.append(data)
            if i % 10 == 0:
                save_results(results,
                    "partial_results.json")
        except Exception as e:
            logger.error(f"Failed: {e}")
    return results
```

Saving Progress

```
import json

def save_results(results, filename):
    with open(filename, 'w') as f:
        json.dump(results, f, indent=2)

def load_results(filename):
    try:
        with open(filename) as f:
            return json.load(f)
    except FileNotFoundError:
        return []

# Resume from where you left off
existing = load_results("partial_results.json")
existing_ids = {r["imdbID"] for r in existing}
remaining = [id for id in all_ids
             if id not in existing_ids]
```

Summary: Data Collection Workflow

For our Netflix movie dataset:

1. **Identify data sources**: OMDb API, TMDb API, IMDb scraping
2. **Test with small sample**: 5-10 movies
3. **Implement collectors**: Functions for each source
4. **Handle errors**: Try-except, retries, validation
5. **Respect rate limits**: Delays, exponential backoff
6. **Combine data**: Merge using IMDb ID
7. **Save incrementally**: Don't lose progress
8. **Validate output**: Check for missing fields

Next Steps

Week 2: Data Validation

- Clean and validate the collected data
- Handle missing values and inconsistencies
- Use Pydantic for validation
- Analyze with csvkit and pandas

Week 3: Data Labeling

- Learn about annotation tasks
- Use Label Studio for labeling
- Measure inter-annotator agreement
- Build high-quality training datasets

Key Takeaways

1. **Data collection is foundational**: Bad data = bad models
2. **Multiple approaches**: APIs (preferred) and web scraping
3. **Error handling is critical**: Network issues, missing fields, rate limits
4. **Be a good citizen**: Respect robots.txt, rate limits, ToS
5. **Save raw data**: You can always reprocess
6. **Start small, scale gradually**: Test thoroughly before running at scale

Resources

APIs:

- OMDb API: http://www.omdbapi.com/
- TMDb API: https://www.themoviedb.org/documentation/api

Libraries:

- requests: https://requests.readthedocs.io/
- BeautifulSoup: https://www.crummy.com/software/BeautifulSoup/
- Playwright: https://playwright.dev/python/

Tools:

- curl: https://curl.se/
- jq: https://stedolan.github.io/jq/

Questions?

Next class: Lab session - hands-on data collection practice