Week 1: Web Scraping & Data Collection

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra
IIT Gandhinagar

Today's Agenda (90 minutes)

1. **Introduction** (10 min)
    - Why web scraping matters for AI/ML
    - Legal and ethical considerations
2. **Chrome DevTools Deep Dive** (25 min)
    - Network tab, XHR inspection
    - Copying requests as cURL/fetch
    - Hands-on demo
3. **Python Requests + BeautifulSoup** (30 min)
    - Making HTTP requests
    - Parsing HTML

Why Web Scraping for AI/ML?

Real-world AI needs real-world data

- **Training datasets**: Reviews, images, text for NLP/CV models
- **Live predictions**: Stock prices, weather, news sentiment
- **Feature engineering**: Enriching data with external sources
- **Monitoring**: Tracking competitors, prices, trends

The Challenge

> "80% of data science is data collection and cleaning"

Most interesting data isn't in nice CSV files—it's on websites!

Legal & Ethical Considerations

✅ Generally OK

- Publicly accessible data
- Respecting `robots.txt`
- Reasonable request rates (not DDoS)
- Personal/academic research

⚠️ Check First

- Terms of Service violations
- Copyrighted content
- Personal data (GDPR, privacy laws)
- Authentication-required data

Part 1: Chrome DevTools

Your Web Scraping Swiss Army Knife

DevTools lets you:

- 🔍 Inspect how websites load data
- 📡 See all network requests (APIs!)
- 🎯 Find the exact data source
- 🛠️ Generate code to replicate requests

Key Insight: Most modern websites load data via JavaScript/APIs, not in raw HTML!

Opening Chrome DevTools

Three Ways

1. **Right-click** → "Inspect"
2. **Keyboard**: `Cmd+Option+I` (Mac) / `Ctrl+Shift+I` (Windows)
3. **Menu**: View → Developer → Developer Tools

Key Tabs for Web Scraping

- **Network**: See all requests (XHR/Fetch for APIs)
- **Elements**: Inspect HTML structure
- **Console**: Test JavaScript, debug

Demo: Network Tab Basics

Let's scrape a weather website

Goal: Find how the site loads weather data

```
1. Open https://weather.com (example)
2. Open DevTools → Network tab
3. Refresh page (Cmd+R)
4. Look for XHR/Fetch requests
5. Click on a request → Preview tab
6. See JSON data!
```

Key Filters:

- `XHR` - API calls made by JavaScript
- `Fetch` - Modern API calls
- `Doc` - HTML pages

Understanding Network Requests

Anatomy of an HTTP Request

Request

```
GET /api/weather?city=Ahmedabad HTTP/1.1
Host: api.weather.com
User-Agent: Mozilla/5.0...
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "city": "Ahmedabad",
  "temp": 28,
  "humidity": 65
}
```

Finding the Data You Need

Step-by-Step Process

1. **Clear Network Log**: Click 🚫 icon to clear
2. **Trigger Action**: Scroll, click, search on the website
3. **Look for XHR/Fetch**: Filter by type
4. **Preview Data**: Click request → Preview tab
5. **Check Headers**: See URL, parameters, cookies

Pro Tips

- Use search/filter (🔍) to find keywords
- Look for URLs with `/api/`, `/data/`, `.json`

Copying Requests as Code

The Magic Feature: "Copy as..."

Right-click on any request → **Copy** → Choose format:

- **Copy as cURL**: Shell command
- **Copy as fetch**: JavaScript code
- **Copy as PowerShell**: Windows
- **Copy as Node.js fetch**: Node.js code

Why This Matters

Instead of manually constructing requests, DevTools gives you working code with:

Demo: Copy as cURL

Example: GitHub API

```
# Copied from DevTools
curl 'https://api.github.com/users/nipunbatra' \
  -H 'Accept: application/json' \
  -H 'User-Agent: Mozilla/5.0 (Macintosh)' \
  --compressed
```

Convert to Python (next section):

```
import requests

response = requests.get(
    'https://api.github.com/users/nipunbatra',
    headers={
        'Accept': 'application/json',
        'User-Agent': 'Mozilla/5.0 (Macintosh)'
```

Live Demo: Real Website

Let's scrape together!

Website: IITGN Course Catalog (or similar)

Steps:

1. Open DevTools Network tab
2. Navigate to courses page
3. Identify API request for course data
4. Examine request/response
5. Copy as cURL
6. We'll convert to Python next!

Your turn: Try with any website you're interested in!

Part 2: Python Requests

The HTTP Library for Python

```python
import requests

# GET request
response = requests.get('https://api.github.com/users/nipunbatra')

# Check status
print(response.status_code)  # 200 = success

# Parse JSON
data = response.json()
print(data['name'])  # "Nipun Batra"

# Get raw text
html = response.text
```

Install: `pip install requests`

HTTP Methods in Requests

Common Operations

```
import requests

# GET - Retrieve data
response = requests.get('https://api.example.com/items')

# POST - Submit data
response = requests.post(
    'https://api.example.com/items',
    json={'name': 'New Item', 'price': 100}
)

# PUT - Update data
response = requests.put('https://api.example.com/items/1',
    json={'price': 150})

# DELETE - Remove data
```

Working with Headers

Why Headers Matter

Headers tell the server:

- What format you accept (`Accept: application/json`)
- Who you are (`User-Agent`)
- Authentication (`Authorization`)
- Cookies for sessions

```
import requests

headers = {
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)',
    'Accept': 'application/json',
    'Accept-Language': 'en-US,en;q=0.9',
}
```

Query Parameters

Two Ways to Add Parameters

Method 1: In URL

```
url = 'https://api.example.com/search?q=python&limit=10'
response = requests.get(url)
```

Method 2: params dict

```
url = 'https://api.example.com/search'
params = {
    'q': 'python',
    'limit': 10,
    'sort': 'stars'
}
response = requests.get(url, params=params)
# Actual URL: /search?q=python&limit=10&sort=stars
```

Recommended: Use `params` dict—cleaner and handles encoding!

Handling Authentication

Common Auth Methods

```python
import requests

# 1. API Key in header
headers = {'Authorization': 'Bearer YOUR_API_KEY'}
response = requests.get(url, headers=headers)

# 2. API Key in params
params = {'api_key': 'YOUR_API_KEY'}
response = requests.get(url, params=params)

# 3. Basic Auth
from requests.auth import HTTPBasicAuth
response = requests.get(url,
    auth=HTTPBasicAuth('username', 'password'))

# 4. Session cookies (we'll use this later)
```

Error Handling

Always Check for Errors!

```
import requests

try:
    response = requests.get('https://api.example.com/data', timeout=10)

    # Raise exception for 4xx/5xx status codes
    response.raise_for_status()

    data = response.json()
    print(f"Success! Got {len(data)} items")

except requests.exceptions.Timeout:
    print("Request timed out!")

except requests.exceptions.HTTPError as e:
    print(f"HTTP error: {e}")

except requests.exceptions.RequestException as e:
    print(f"Error: {e}")
```

Rate Limiting & Being Polite

Don't Overwhelm Servers!

```
import requests
import time

urls = ['https://api.example.com/item/' + str(i) for i in range(100)]

for url in urls:
    response = requests.get(url)
    data = response.json()

    # Process data...

    # IMPORTANT: Wait between requests
    time.sleep(1)  # 1 second delay
```

Best Practices

Part 3: BeautifulSoup

Parsing HTML Like a Pro

When to use: When data is in HTML, not JSON APIs

```
from bs4 import BeautifulSoup
import requests

# Fetch page
response = requests.get('https://example.com/articles')
html = response.text

# Parse HTML
soup = BeautifulSoup(html, 'html.parser')

# Find elements
title = soup.find('h1').text
articles = soup.find_all('article', class_='post')
```

BeautifulSoup Selectors

Finding Elements

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(html, 'html.parser')

# By tag name
soup.find('h1')              # First <h1>
soup.find_all('p')           # All <p> tags

# By class
soup.find('div', class_='content')
soup.find_all('a', class_='link')

# By id
soup.find('div', id='main')

# By CSS selector
soup.select('div.content p')     # All <p> inside <div class="content">
soup.select_one('h1.title')      # First match
```

Extracting Data from Elements

Common Operations

```
# Get text content
title = soup.find('h1').text
title_clean = soup.find('h1').get_text(strip=True)

# Get attribute value
link = soup.find('a')['href']
link_alt = soup.find('a').get('href')  # Safer—returns None if missing

# Get all text from element and children
content = soup.find('article').get_text(separator='\n', strip=True)

# Check if element exists
if soup.find('div', class_='error'):
    print("Error found!")

# Navigate the tree
```

Real Example: Scraping Quotes

Let's scrape http://quotes.toscrape.com

```
import requests
from bs4 import BeautifulSoup

url = 'http://quotes.toscrape.com/'
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

# Find all quote containers
quotes = soup.find_all('div', class_='quote')

for quote in quotes:
    # Extract text, author, tags
    text = quote.find('span', class_='text').text
    author = quote.find('small', class_='author').text
    tags = [tag.text for tag in quote.find_all('a', class_='tag')]
```

Handling Pagination

Scraping Multiple Pages

```
import requests
from bs4 import BeautifulSoup
import time

base_url = 'http://quotes.toscrape.com'
page = 1
all_quotes = []

while True:
    url = f'{base_url}/page/{page}/'
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')

    quotes = soup.find_all('div', class_='quote')
    if not quotes:  # No more quotes
        break

    all_quotes.extend(quotes)

    page += 1
```

Common BeautifulSoup Patterns

Tips & Tricks

```
# Handle missing elements safely
title = soup.find('h1')
if title:
    print(title.text)
else:
    print("No title found")

# Or use .get_text() with default
title = (soup.find('h1') or {}).get_text(default='No title')

# Extract URLs (handle relative paths)
from urllib.parse import urljoin

link = soup.find('a')['href']
absolute_url = urljoin(base_url, link)  # Converts relative to absolute

# Remove unwanted elements before extracting text
```

Combining Requests + BeautifulSoup

Complete Scraping Workflow

```
import requests
from bs4 import BeautifulSoup
import time
import json

def scrape_article(url):
    """Scrape a single article"""
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')

    return {
        'title': soup.find('h1').text.strip(),
        'author': soup.find('span', class_='author').text,
        'content': soup.find('article').get_text(strip=True),
        'url': url
    }

# Scrape multiple articles
article_urls = ['https://blog.com/post1', 'https://blog.com/post2']
articles = []

for url in article_urls:
    article = scrape_article(url)
    articles.append(article)
    time.sleep(2)
```

Part 4: Browser Automation with Playwright

Why Playwright?

Playwright vs Selenium

Feature	Playwright	Selenium
Speed	⚡ Faster	Slower
Modern	✅ 2020+	2004
Auto-wait	✅ Built-in	❌ Manual
API	🎯 Cleaner	More verbose
Browsers	Chrome, Firefox, Safari	All browsers
Maintenance	Active (Microsoft)	Community

Bottom line: Playwright is the modern choice for web automation!

When Do You Need Browser Automation?

Use Cases

1. **JavaScript-rendered content**
    - Single Page Applications (React, Vue, Angular)
    - Infinite scroll
    - Content loaded after page load
2. **User interactions**
    - Click buttons, fill forms
    - Login flows
    - Navigate multiple pages
3. **Dynamic content**

Installing Playwright

Quick Setup

```
# Install Playwright
pip install playwright

# Install browsers (Chrome, Firefox, WebKit)
playwright install

# Or install specific browser
playwright install chromium
```

Size warning: Downloads ~300MB of browsers, but you only do this once!

Your First Playwright Script

Basic Example

```python
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    # Launch browser
    browser = p.chromium.launch(headless=False)  # headless=True for no GUI

    # Create new page
    page = browser.new_page()

    # Navigate to URL
    page.goto('https://example.com')

    # Get page title
    title = page.title()
    print(f"Page title: {title}")

    # Take screenshot
    page.screenshot(path='screenshot.png')
```

Finding & Interacting with Elements

Playwright Selectors

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    page = browser.new_page()
    page.goto('https://google.com')

    # Find element and type
    page.fill('input[name="q"]', 'web scraping')

    # Click button
    page.click('input[value="Google Search"]')

    # Wait for navigation
    page.wait_for_load_state('networkidle')

    # Extract data
    results = page.query_selector_all('h3')
    for result in results[:5]:
```

Common Playwright Patterns

Waiting for Elements

```
# Wait for element to appear (auto-waits up to 30s)
page.click('button')  # Automatically waits for button to be clickable

# Explicit wait
page.wait_for_selector('div.results', timeout=10000)  # 10 seconds

# Wait for network to be idle
page.wait_for_load_state('networkidle')

# Wait for specific URL
page.wait_for_url('**/results')

# Wait for function to return true
page.wait_for_function('() => document.querySelectorAll(".item").length > 10')
```

Key advantage: Playwright auto-waits for most actions—less flaky tests!

Extracting Data with Playwright

Getting Content

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    page = browser.new_page()
    page.goto('https://news.ycombinator.com')

    # Get all story titles
    stories = page.query_selector_all('span.titleline > a')

    for story in stories[:10]:
        title = story.inner_text()
        url = story.get_attribute('href')
        print(f"{title}\n  {url}\n")

    browser.close()
```

Handling JavaScript-Heavy Sites

Example: Infinite Scroll

```
from playwright.sync_api import sync_playwright
import time

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    page = browser.new_page()
    page.goto('https://twitter.com/explore')

    # Scroll down 5 times to load more content
    for _ in range(5):
        page.evaluate('window.scrollTo(0, document.body.scrollHeight)')
        time.sleep(2)  # Wait for content to load

    # Now extract all loaded tweets
    tweets = page.query_selector_all('article')
    print(f"Loaded {len(tweets)} tweets")
```

Playwright vs Requests+BeautifulSoup

When to Use What?

Requests + BeautifulSoup ✅

- Static HTML content
- API endpoints available
- Fast scraping needed
- Low resource usage
- Simple pagination

Example: News articles, blogs, product listings (server-rendered)

Playwright ✅

- JavaScript-rendered content

Example: SPAs, social media, web apps

Practical Example: IITGN Announcements

Let's Build a Real Scraper

Goal: Scrape latest announcements from IITGN website

```
import requests
from bs4 import BeautifulSoup

url = 'https://iitgn.ac.in/news'
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

announcements = soup.find_all('div', class_='announcement-item')

for announcement in announcements[:5]:
    title = announcement.find('h3').text.strip()
    date = announcement.find('span', class_='date').text.strip()
    link = announcement.find('a')['href']
```

Best Practices Summary

Do's ✅

- Check `robots.txt` before scraping
- Add delays between requests (1-2 seconds)
- Use appropriate User-Agent headers
- Handle errors gracefully
- Cache responses when possible
- Respect rate limits

Don'ts ❌

- Don't overwhelm servers (DDoS)
- Don't scrape personal data without permission

Debugging Tips

Common Issues & Solutions

```
# 1. 403 Forbidden → Add User-Agent
headers = {'User-Agent': 'Mozilla/5.0...'}
response = requests.get(url, headers=headers)

# 2. 429 Too Many Requests → Add delay
import time
time.sleep(2)

# 3. Empty results → Check if JavaScript-rendered
# Use Playwright instead of Requests

# 4. Connection timeout → Increase timeout
response = requests.get(url, timeout=30)

# 5. SSL errors → Disable verification (careful!)
response = requests.get(url, verify=False)
```

Tools & Resources

Essential Tools

- **Chrome DevTools**: Built into Chrome
- **Postman**: Test API requests
- **curl**: Command-line HTTP client
- **jq**: Command-line JSON processor
- **HTTPie**: User-friendly curl alternative

Learning Resources

- requests docs
- BeautifulSoup docs
- Playwright docs

Lab Preview (3 hours)

What You'll Build Today

Part 1: DevTools Practice (45 min)

- Find hidden APIs on real websites
- Convert cURL to Python

Part 2: Requests + BeautifulSoup (90 min)

- Scrape quotes website (practice)
- Build a news aggregator
- Handle pagination

Part 3: Playwright (45 min)

Questions?

Get Ready for Lab!

What to have installed:

```
pip install requests beautifulsoup4 playwright pandas
playwright install chromium
```

What to bring:

- Laptop with Python 3.8+
- Curiosity about websites you use daily
- Ideas for data you want to collect

See You in Lab! 🚀

Remember: With great scraping power comes great responsibility!

Next week: Data validation, cleaning, and labeling