Universidade do Minho

Ano Letivo: 2025/26

Mineração de Dados

Web Scraping

Miguel Rocha | mrocha@di.uminho.pt
Diana Ferreira | diana@di.uminho.pt

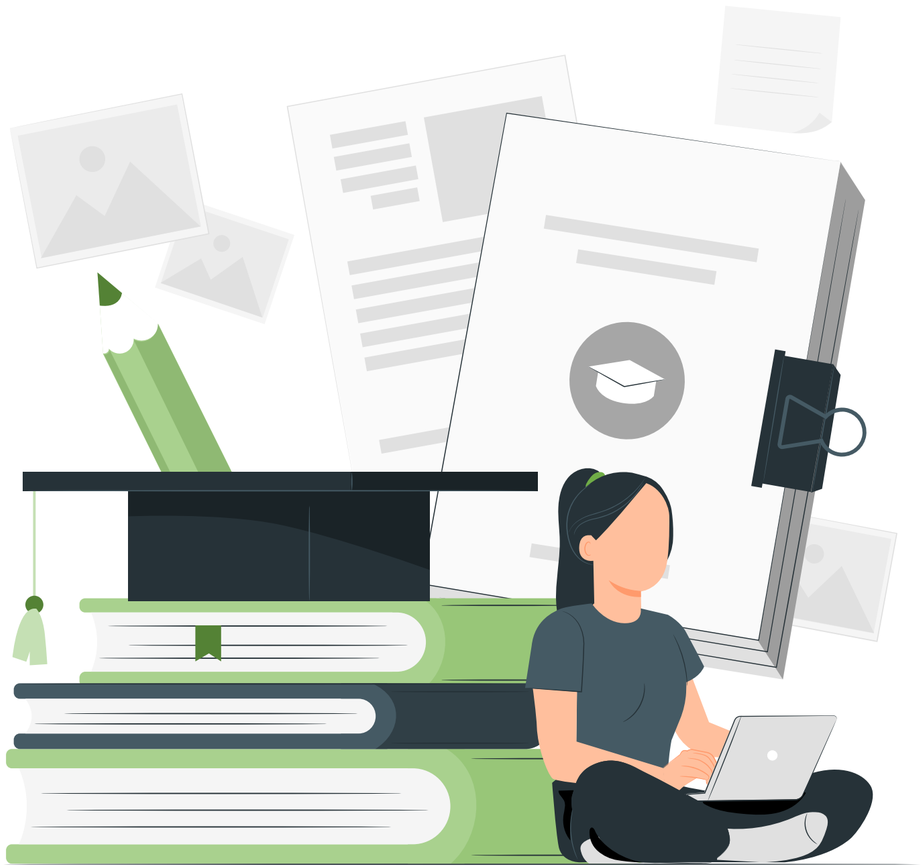
Contents

Learning Objectives

By the end of this lecture, students should be able to:

- Understand the theoretical foundations of web scraping and how it works.
- Master the main tools of the Python ecosystem.
- Develop scrapers for static and dynamic pages.
- Learn best practices, ethics, and legal aspects.

Bibliography



Books:

1. McKinney, W. (2022). Python for Data Analysis. O'Reilly
2. Mitchell, R. (2018). Web Scraping with Python. O'Reilly

Official Documentation:

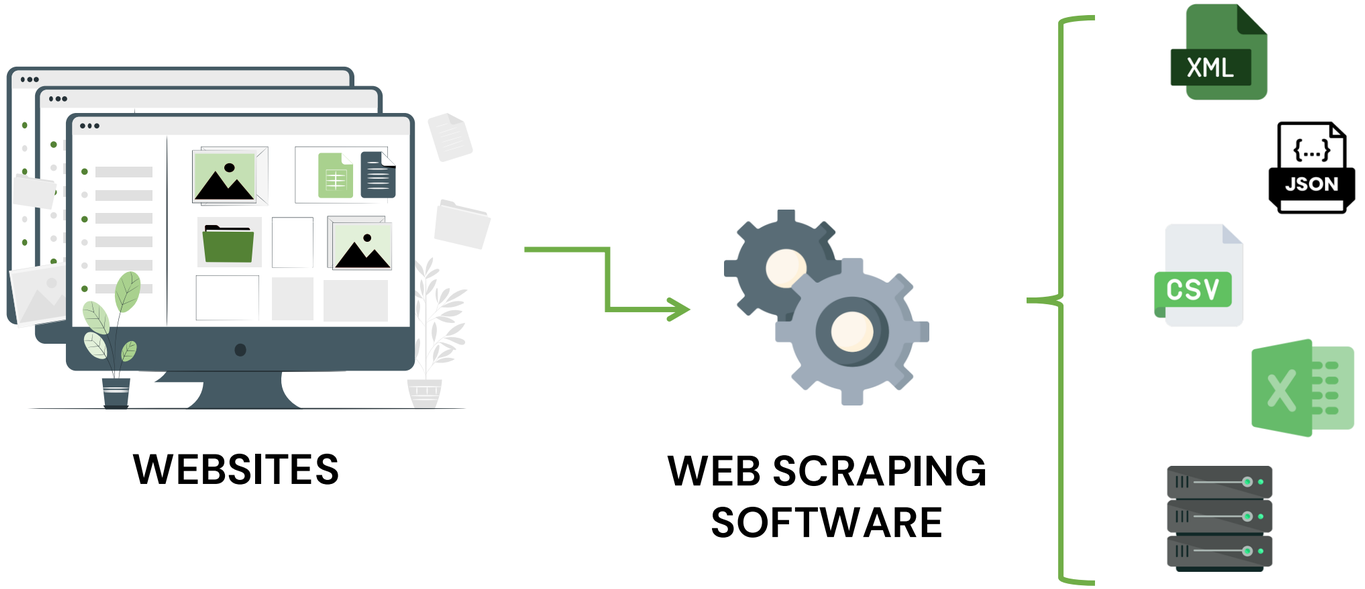
1. Requests
2. Beautiful Soup
3. Selenium
4. Scrapy

Community and Resources:

1. **Reddit:** r/webscraping
2. **GitHub:** https://github.com/lukas-bear/awesome-web-scraping

What is web scraping?

Web scraping is the process of collecting data from the web, converting semi-structured content (HTML) to structured formats (csv, json, excel, etc.). Although this can be a manual process (i.e. copy and pasting from websites yourself), “web scraping” generally refers to automating that process.



Imagine manually copying and pasting information from 1,000 pages. Web scraping teaches the computer to do this!

Scale: minutes vs. days of manual work

Motivation

- Massive amount of data published on the web
- Not all data is available in files, databases, or APIs.

- Web scraping enables:
  - Data-driven research
  - Market intelligence
  - Academic and industrial analytics

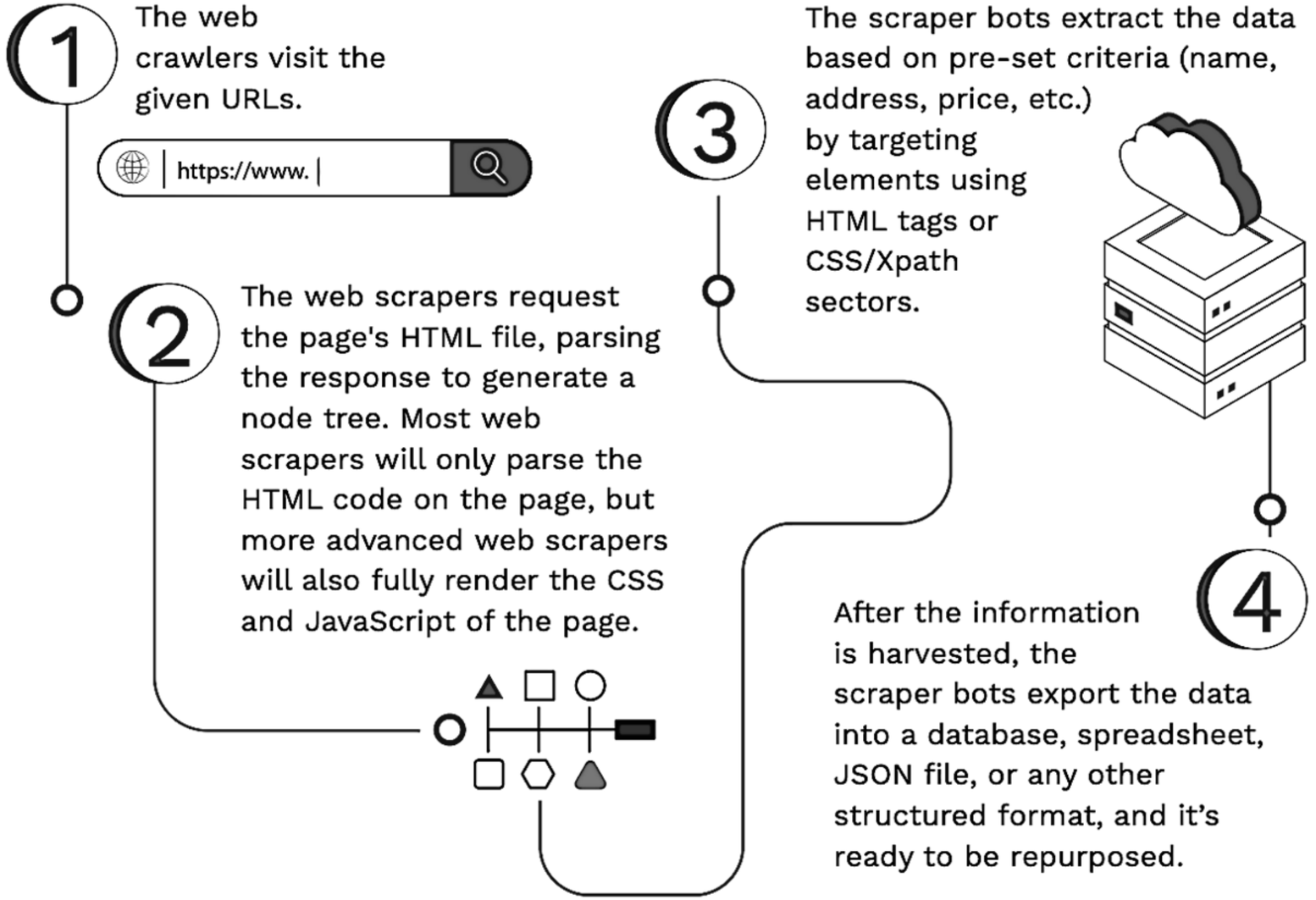
Web scraping vs. Web Crawling

Carachteristic	Web Scraping	Web Crawling *
Objective	Extract specific data	Finding/indexing URLs
Scope	Targeted pages	Link navigation
Example	Prices of a product	Googlebot
Output	Structured data	List of URLs
Manual alternatives	Copying and pasting required data into a database	Clicking through each link and storing the URL in a list

In practice: Scrapers often include crawling for navigation.

*Web crawling is the process of browsing and indexing data from the internet using a program or an automated script. These automated scripts or programs are called web crawlers, web spiders, robots, or spider bots. This is how search engines like Google collect all the data they need to return results based on your queries.

Typical Web Scraping Pipeline



https://www.scraperapi.com/wp-content/uploads/scraperapi-web-scrapping-the-basics-explained.pdf

Web Scraping Use Cases

Academic research
Collecting datasets for ML

Market monitoring
Competitor pricing

Real estate analysis
Market trends

Recruitment
Professional profiles

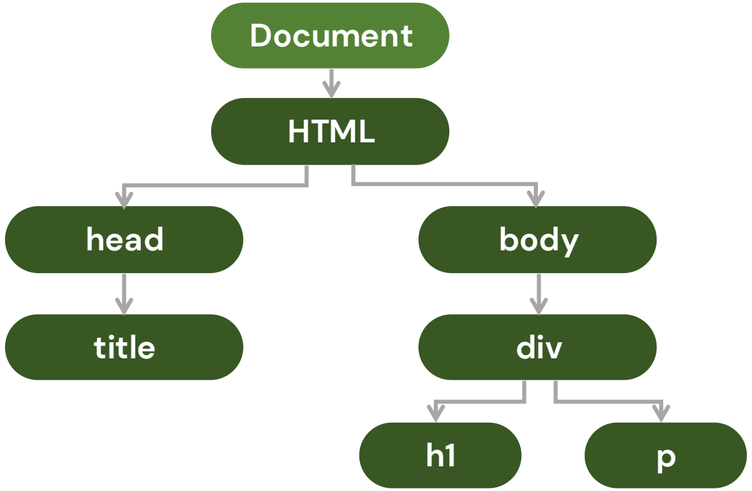
Data journalism
Automated fact-checking

Big Data projects often begin with scraping for data acquisition

Structure of a Web Page

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title> Webpage title </title>
    </head>
    <body>
        <div>
            <h1>This is a heading tag</h1>
            <p>This is some text content</p>
        </div>
    </body>
</html>
```

HTML (HyperText Markup Language)
Semantic content and structure



DOM (Document Object Model)
Tree representation of the HTML document, where each node is an object that can be manipulated.

```javascript
const checkDecimals = value => {
  const strValue = value.toString();
  let nDecimals;

  if (strValue.indexOf(".") > -1) {
    const index = strValue.indexOf(".");
    nDecimals = strValue?.size - index - 1;
  } else {
    nDecimals = 0;
  }
```

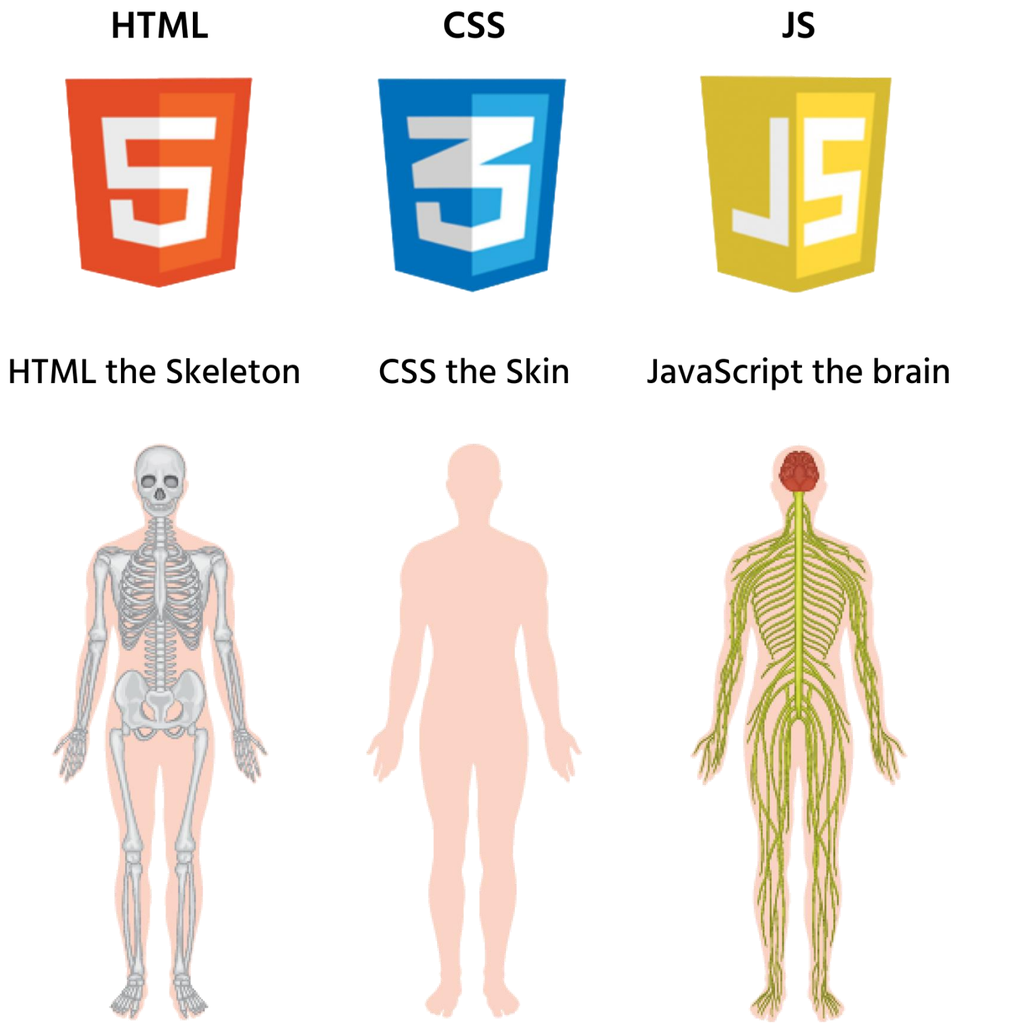
JavaScript
Dynamic behavior and interactivity

```css
.italic {
  font-style: italic;
}

.uppercase {
  text-transform: uppercase;
}
```

CSS (Cascading Style Sheets)
Styling and layout

Structure of a Web Page



https://content-media-cdn.codefinity.com/courses/c468052d-6d47-4677-b1d1-fb11ace719b1/web-development/comparison+of+HTML+and+CSS+and+JS.png

Understanding HTML

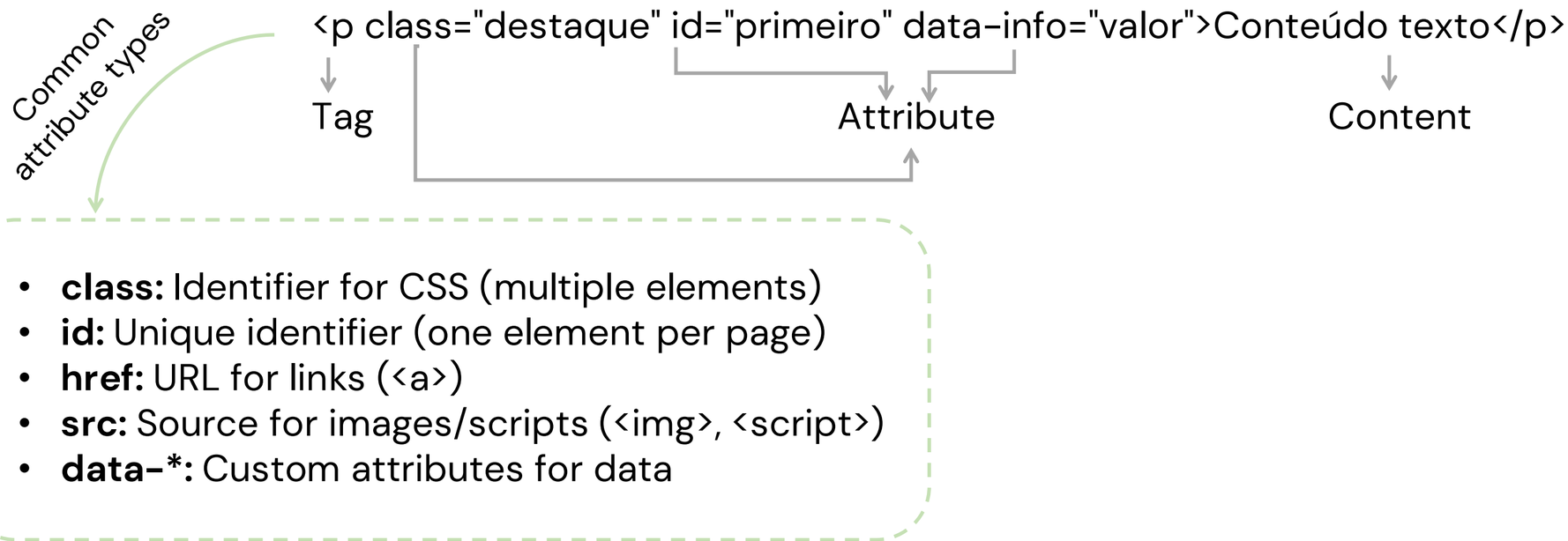
Key elements:

- Tags (div, span, a, table)
- Attributes (id, class, href)
- Hierarchical structure (tree)

Importance for:

- Locating data accurately
- Writing reliable selectors

Components of an HTML element:



Why is DOM crucial for scraping?

- It is the structure that parsing libraries (BeautifulSoup, Cheerio) build.
- It allows programmatic navigation through elements.
- It reflects the actual hierarchy of the document.

Important concepts:

Parent: Node that contains other nodes

Child: Node directly contained in another node

Siblings: Nodes with the same parent

Descendants: All nodes within an element

Ancestors: Path to the root

Why is DOM crucial for scraping?

➡ Types of Nodes in the DOM

Node Type	Example	Importance
Element	, <p>, <a>	Contains data and structure
Text	“visible content”	The data itself
Attribute	href="url," class="name"	Metadata and references

In the code (BeautifulSoup):

```
element = soup.find('div')          # Element node
text = element.text                 # Text node
url = element.get('href')           # Attribute node
```

Why is DOM crucial for scraping?

➡ Navigating the DOM with Selectors

- CSS Selectors (recommended for 90% of cases):

Selector	Example	Selects
Tag	div	All
Class	.product	Elements with class="product"
ID	#header	Element with id="header"
Descendant (space)	div p	<p> inside
Direct child (>)	div > p	<p> direct child of
Multiple classes	.highlight.blue	Elements with both classes

- **XPath** (more powerful, but verbose):
  - //div - All elements
  - //div[@class="product"] - Equivalent to .product
  - //a[contains(@href, "product")] - Links with “product” in the URL

Why is DOM crucial for scraping?

➡ Practical Example of DOM Navigation

Example HTML (e-commerce website):

```
<div class="produto">
  <h2 class="titulo">Notebook Pro</h2>
  <p class="preco">2.000€</p>
  <div class="avaliacoes">
    <span class="estrelas">★★★★☆</span>
    <a href="/avaliacoes/123">Ver 45 avaliações</a>
  </div>
</div>
```

Extracting data with BeautifulSoup:

```
# Navegação descendente
produto = soup.find('div', class_='produto')
titulo = produto.find('h2', class_='titulo').text
preco = produto.find('p', class_='preco').text

# Navegação por irmãos (mais complexa)
avaliacoes = produto.find('div', class_='avaliacoes')
link = avaliacoes.find('a')['href']
```

Browser Tools for Scraping

⚠️ Chrome DevTools (F12) is your best friend!

Essential features:

Elements – Inspect DOM and copy selectors

Network – Analyze HTTP requests

Console – Test selectors in real time

Testing selectors in the Console:

```
// Test if your CSS selector works
document.querySelectorAll('.produto .titulo')

// Check how many elements will be captured
document.querySelectorAll('.produto').length
```

💡 **Tip:** Use “Copy > Copy selector” or “Copy > Copy XPath” to get precise selectors.

The Role of JavaScript in the DOM

Example of how JavaScript modifies the DOM:

```
// Example of dynamic manipulation
document.addEventListener('DOMContentLoaded', function() {
    // Fetch API data
    fetch('/api/products')
        .then(response => response.json())
        .then(products => {
            // CREATE new elements in the DOM
            const list = document.getElementById('products');
            products.forEach(p => {
                const div = document.createElement('div');
                div.textContent = p.name;
                list.appendChild(div);
            });
        });
});
```

Implication for scraping: The initial HTML does not contain the data, it is injected later.

Static vs. Dynamic Pages

Feature	Static Page	Dynamic Page (SPA)
Content	HTML fully delivered by server	Content loaded via JavaScript
Loading	One request	Multiple AJAX requests
JavaScript	Optional (improvements)	Required (renders content)
Tool	Requests + BeautifulSoup	Selenium/Playwright
Example	quotes.toscrape.com	Page with infinite scroll

How to Identify the Page Type?

Method 1: View source code

```
import requests

response = requests.get('https://exemplo.com’)
initial_html = response.text

if 'data I expect' not in initial_html:
    print("Page is likely dynamic, requires JavaScript")
```

Method 2: Disable JavaScript in your browser

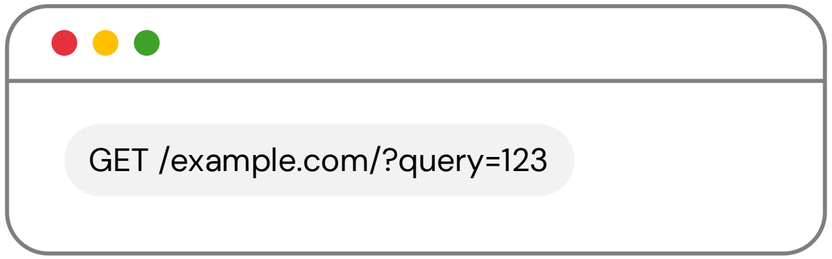
- Chrome settings: Disable JavaScript
- Reload the page
- If the content disappears, it is dynamic

HTTP Fundamentals

HTTP - HyperText Transfer Protocol

Fundamental web protocol - Client-server, stateless, text-based.

Method	Use in Scraping	Description
GET	**95% of cases**	Obtain resources/pages
POST	Forms, APIs	Send data (login, search)
HEAD	Verification	Only headers, without body



GET request

- Used to **retrieve data** from a server
- Parameters are sent in the **URL query string**
- Length of data is **limited**
- Less secure

POST request

- Used to **send data** to the server
- Data is sent in the **request body**
- Supports **large and complex data**
- More secure (but not encrypted by default)

HTTP Fundamentals

Status Codes

Code	Meaning	Scraper Action
2xx	**Success**	
200	OK	Process normally
3xx	**Redirection**	
301	Moved permanently	Update URL in favorites
302	Found (temporary)	Follow redirection
4xx	**Client Error**	
403	Forbidden	Blocked - use proxies
404	Not Found	Check URL
429	Too Many Requests	WAIT! Rate limiting
5xx	**Server Error**	
500	Internal Error	Try again after delay

HTTP Fundamentals

HTTP Headers

What Are HTTP Headers?

- Metadata sent with every HTTP request and response
- Headers are not part of the page content, but control how communication happens.

Why headers are critical?

Servers identify clients by headers. Without realistic headers, you are an “obvious bot.”

```
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
    'Accept-Language': 'pt-BR,pt;q=0.9,en;q=0.8',
    'Accept-Encoding': 'gzip, deflate, br',
    'Connection': 'keep-alive',
    'Upgrade-Insecure-Requests': '1',
    'Sec-Fetch-Dest': 'document',
    'Sec-Fetch-Mode': 'navigate',
    'Sec-Fetch-Site': 'none',
    'Sec-Fetch-User': '?1',
    'Cache-Control': 'max-age=0',
}
```

→ Identifies the client software

Why User-Agent Matters in Web Scraping

- Websites use it to:
  - Detect bots vs browsers
  - Apply rate limits
  - Block unknown clients
- Missing or default User-Agent often leads to 403 Forbidden

HTTP Fundamentals

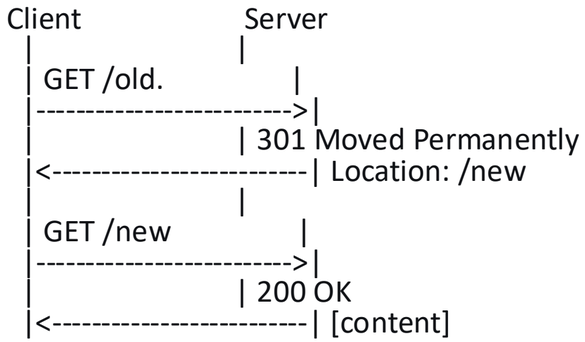
➡ The Life Cycle of an HTTP Request

What happens when you do requests.get(url):

1. DNS lookup: URL → IP (e.g., 192.0.2.1)
2. TCP connection: port 80 (HTTP) or 443 (HTTPS)
3. TLS handshake (if HTTPS)
4. GET request sent
5. Processing on the server
6. Response (HTML, JSON, etc.)
7. Termination or keep-alive

HTTP Fundamentals

HTTP Redirects



Scraping behavior:

- Requests automatically follow redirects (allow_redirects=True)
- Selenium/Playwright also follow redirects
- Problem: Multiple redirects may indicate dynamic pages

HTTP Fundamentals

Cookies and Sessions

What are cookies?

- Small pieces of data stored by the browser
- Sent with each request to the domain
- Used to maintain state (login, preferences)

Why they matter for scraping?

- Websites use cookies to track sessions
- Many require cookies for access
- Some use anti-bot cookies

Manipulating cookies with Requests:

```
# Using session automatically maintains cookies
session = requests.Session()
session.get('https://site.com/login')  # Receives cookies

# Subsequent requests automatically send cookies
response = session.get('https://site.com/dados')
```

HTTP Fundamentals

Rate Limiting and Throttling

Why servers limit requests?

- To protect against overload
- To prevent aggressive scraping
- To save resources

Common rate limiting headers:

```
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 42
X-RateLimit-Reset: 1640995200
Retry-After: 30
```

Throttling is a technique used to **control the rate of operations**, especially the number of requests sent to a system over time

Why it is used:

- Avoid triggering anti-bot protections
- Prevent server overload
- Reduce risk of IP blocking
- Behave ethically and responsibly

Concept	Applied by	Purpose
Throttling	Client	Self-imposed control
Rate limiting	Server	Enforced restriction

HTTP Fundamentals

➡ Throttling Strategies

```python
import time
import random

# Random delay between requests
def polite_wait():
    delay = random.uniform(1, 3)  # 1-3 seconds
    time.sleep(delay)

# Exponential backoff in case of error
def exponential_backoff(attempt):
    return min(300, (2 ** attempt) + random.random())
```

HTTP Fundamentals

➡ HTTPS and TLS (Transport Layer Security)

TLS ensures that data exchanged between a client and a server is:

- **Confidential** – encrypted so third parties cannot read it
- **Authentic** – the server's identity is verified
- **Integrity-protected** – data cannot be altered in transit

Without TLS → **HTTP** (data sent in plain text)

With TLS → **HTTPS** (data encrypted)

Relevance to Web Scraping

- Scrapers rely on TLS to:
  - Secure credentials and cookies
  - Safely interact with HTTPS websites
- TLS **does not prevent scraping**, but ensures secure transport

The First Source of Truth – robots.txt

What is robots.text?

A file that defines rules for crawlers: www.site.com/robots.txt

```
User-agent: *
Disallow: /admin/
Disallow: /private/
Allow: /public/
Crawl-delay: 10
```

Interpretation:

Disallow: Areas prohibited for scraping

Crawl-delay: Recommended time between requests

Golden rule: ALWAYS respect robots.txt

Tools & Languages for Web Scraping

Common choices:

- Python
- JavaScript (Node.js)
- R
- Java

Why Python dominates:

- Rich ecosystem
- Simplicity
- Community support

Popular Python Libraries

- requests – HTTP requests
- BeautifulSoup – HTML parsing
- lxml – Fast XML/HTML parsing
- Scrapy – Full scraping framework
- Selenium / Playwright – Browser automation

Tools & Languages for Web Scraping

Library	Application	When to use it?
Requests	Download pages	Always the starting point
BeautifulSoup (bs4)	Parse HTML	Simple, static HTML pages
Scrapy	Scale, Crawling	Many pages, entire websites, high performance
Selenium	Websites with JS, interactions	Need to click, scroll, fill out forms
Playwright	Modern alternative	Modern web apps, SPAs, reliable automation
Pandas	Cleaning and exporting	Organize tabular data

Tool	Static websites	Dynamic websites	Crawling Scale	Complex Interactions
Requests+BS4	🟢	🔴	🟡	🔴
Scrapy	🟢	🟡[1]	🟢	🟡[1]
Selenium	🟡	🟢	🔴	🟢
Playwright	🟡	🟢	🟡	🟢

[1] *With Playwright integration*

Environment Setup

```
# Create virtual environment
python -m venv scraping_env
source scraping_env/bin/activate  # Linux/Mac
# scraping_env\Scripts\activate   # Windows

# Install dependencies
pip install requests beautifulsoup4 pandas selenium jupyter

# For Selenium (Chrome)
# Download ChromeDriver: https://chromedriver.chromium.org/
```

Quick verification

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
print("Ambiente pronto!")
```

BeautifulSoup

Method	Return	Typical use
find()	First matching element	Single search (e.g., title)
find_all()	List of all elements	Multiple occurrences (e.g., products)
select()	List of elements by CSS	Complex selection
select_one()	First element by CSS	CSS equivalent to find()

EXAMPLES:

- soup.find('div', class_='produto') -> Find the FIRST div with class "product"
- soup.find_all('div', class_='produto') -> Find ALL divs with class "product"
- soup.find_all('a', class_='link') -> Search by class (use class_ because 'class' is a reserved word)
- soup.find(id='prod1') -> Search by ID
- soup.find(attrs={'data-categoria': 'eletrónicos'}) -> Search by custom attribute (data-*)
- soup.find('a', {'class': 'link', 'id': 'prod2'}) -> Multiple attributes

BeautifulSoup

Regular Expressions in Selectors – For more complex patterns, combine with re.compile().

```
import re
from bs4 import BeautifulSoup

html = """
<a href="/produto/123">Produto 123</a>
<a href="/categoria/456">Categoria 456</a>
<a href="/produto/789">Produto 789</a>
<a href="/promocao/abc">Promoção</a>
"""

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

# Todos os links que começam com "/produto/"
produtos = soup.find_all('a', href=re.compile('^/produto/'))
print(f"Produtos encontrados: {len(produtos)}")

# Links que contêm números
com_numeros = soup.find_all('a', href=re.compile(r'\d+'))

# Classes que começam com "prod"
elementos = soup.find_all(class_=re.compile('^prod'))
```

BeautifulSoup

BeautifulSoup supports CSS selectors through the select() and select_one() methods.

Advantages:

- Familiar syntax for those who know CSS
- More concise for complex selections
- Better performance in some scenarios

```
# select_one() - retorna o PRIMEIRO elemento (equivalente ao find())
primeiro = soup.select_one('.produto')

# select() - retorna LISTA de elementos (equivalente ao find_all())
todos = soup.select('.produto’)

# Exemplos práticos
tags_p = soup.select('p')                    # Todos os <p>
produtos = soup.select('.produto')            # Classe "produto"
cabecalho = soup.select_one('#cabecalho')     # ID "cabecalho"
```

BeautifulSoup

CSS selectors for filtering by attributes with patterns

Selector	Example	Meaning
[attr]	[href]	Ements with attribute href
[attr="valor"]	[type="submit"]	Exact value
[attr^="valor"]	[href^="https"]	Begins with "valor"
[attr$="valor"]	[src$=".jpg"]	Ends with "valor"
[attr*="valor"]	[class*="prod"]	Contains "valor"
[attr~="valor"]	[class~="destaque"]	Contains exact word

```
# Links externos (https)
links_https = soup.select('a[href^="https"]')

# Imagens JPEG
imagens_jpg = soup.select('img[src$=".jpg"]')

# Elementos com "prod" na classe
elementos_prod = soup.select('[class*="prod"]')
```

BeautifulSoup

Text and Attribute Extraction

```
html = """
<div class="produto" data-id="123">
    <h3 class="titulo">Notebook Pro</h3>
    <a href="/produto/123" class="link">Ver detalhes</a>
    <img src="/images/notebook.jpg" alt="Notebook">
</div>
"""

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

# Extrair texto
titulo = produto.select_one('.titulo').text          # "Notebook Pro"
# Alternativa
titulo = produto.select_one('.titulo').get_text()    # mesmo resultado

# Extrair atributos
link = produto.select_one('.link')['href']           # "/produto/123"
img_src = produto.select_one('img')['src']           # "/images/notebook.jpg"
data_id = produto['data-id']                          # "123"

# Extrair com fallback (evita erros se atributo não existir)
alt_text = produto.select_one('img').get('alt', 'Sem descrição')
```

Scraping Static Websites

➡ Hands on – first scraper

Training website: http://quotes.toscrape.com/

Objective: To extract famous sentences and their authors

1st step – Make the request

```
import requests
from bs4 import BeautifulSoup

url = "http://quotes.toscrape.com/"
headers = {'User-Agent': 'Mozilla/5.0'}
response = requests.get(url, headers=headers)

if response.status_code == 200:
    print("Página carregada com sucesso!")
else:
    print(f"Erro: {response.status_code}")
```

2nd step – Analyze the HTML

```
soup = BeautifulSoup(response.text, 'html.parser')

# Encontrar todas as citações
quotes = soup.find_all('div', class_='quote')

print(f"Encontradas {len(quotes)} citações na página")
```

Scraping Static Websites

➡ Hands on – first scraper

3rd step – Extract specific data

```
for quote in quotes:
    text = quote.find('span', class_='text').text
    author = quote.find('small', class_='author').text

    # Tags (list)
    tags = [tag.text for tag in quote.find_all('a', class_='tag')]

    print(f"Frase: {text}")
    print(f"Autor: {author}")
    print(f"Tags: {', '.join(tags)}")
    print("-" * 50)
```

Scraping Static Websites

➡ Hands on – first scraper

4th step - Organize in data structures

```
dados = []

for quote in quotes:
    item = {
        'frase': quote.find('span', class_='text').text,
        'autor': quote.find('small', class_='author').text,
        'tags': [tag.text for tag in quote.find_all('a', class_='tag')]
    }
    dados.append(item)

# Visualizar
for i, item in enumerate(dados[:3]):
    print(f"{i+1}. {item['autor']}: {item['frase'][:50]}...")
```

Scraping Static Websites

➡ Hands on – first scraper

5th step - Save data

```python
import pandas as pd
import json
import csv

# Para CSV (tabular) - tags viram string
df = pd.DataFrame(dados)
df['tags'] = df['tags'].apply(lambda x: ', '.join(x))
df.to_csv('quotes.csv', index=False, encoding='utf-8')

# Para JSON (preserva estrutura)
with open('quotes.json', 'w', encoding='utf-8') as f:
    json.dump(dados, f, ensure_ascii=False, indent=2)

print("Dados exportados com sucesso!")
```

Approach:
- Send HTTP request
- Parse HTML response
- Extract data via CSS/XPath selectors

Advantages:
- Faster
- Lower resource usage

Scraping Static Websites

➡ Hands on – first scraper

❌ Problem: Website has multiple pages of results

Solution 1: Loop based on “next”

```
base_url = "http://quotes.toscrape.com"
page_url = "/page/1/"
all_quotes = []

while page_url:
    print(f"Raspando: {base_url + page_url}")
    response = requests.get(base_url + page_url, headers=headers)
    soup = BeautifulSoup(response.text, 'html.parser')

    # ... extraction code...

    # find link "Next"
    next_li = soup.find('li', class_='next')
    page_url = next_li.find('a')['href'] if next_li else None

    # be polite
    time.sleep(1)

print(f"Total de {len(all_quotes)} citações raspadas")
```

Scraping Dynamic Websites

➡ Hands on – second scraper

⚠ Many modern websites load data dynamically using JavaScript. In such cases, scraping using ***requests*** and ***BeautifulSoup*** may not work, as these libraries only retrieve the raw HTML and not the dynamically generated content.

Site with JS rendering: http://quotes.toscrape.com/js/

Test with Requests:

```
response = requests.get("http://quotes.toscrape.com/js/")
soup = BeautifulSoup(response.text, 'html.parser')
quotes = soup.find_all('div', class_='quote')
print(f"Encontradas: {len(quotes)}")  # Result: 0!
```

↳ The initial HTML does not contain the data, it is loaded via JavaScript after the page loads.

Scraping Dynamic Websites

➡ Hands on – second scraper

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
import time

# Config driver (headless - without graphical interface)
options = Options()
options.add_argument("--headless")
driver = webdriver.Chrome(options=options)

driver.get("http://quotes.toscrape.com/js/") # load page (execute JS)

time.sleep(2) # Wait for JavaScript to render the content - not ideal

quotes = driver.find_elements(By.CLASS_NAME, "quote")
print(f"Encontradas: {len(quotes)}")

for quote in quotes:
    text = quote.find_element(By.CLASS_NAME, "text").text
    author = quote.find_element(By.CLASS_NAME, "author").text
    print(f"{author}: {text[:50]}...")

driver.quit() # Close the browser
```

Scraping Dynamic Websites

➡ Hands on – second scraper

- Better than time.sleep():

```python
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver.get("http://quotes.toscrape.com/js/")

# Wait until the elements are present
try:
    WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.CLASS_NAME, "quote"))
    )

    quotes = driver.find_elements(By.CLASS_NAME, "quote")
    print(f"Carregadas {len(quotes)} citações")

except Exception as e:
    print(f"Timeout: {e}")
```

Advantage: Waits exactly as long as necessary, not fixed

Scraping Dynamic Websites

➡ Hands on – second scraper

```python
from bs4 import BeautifulSoup

# Use Selenium for rendering
driver.get("http://quotes.toscrape.com/js/")
WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.CLASS_NAME, "quote"))
)

# Pass HTML to BeautifulSoup (easier parsing)
html = driver.page_source
soup = BeautifulSoup(html, 'html.parser')

# Use BeautifulSoup for extraction
quotes = soup.find_all('div', class_='quote')
for quote in quotes:
    text = quote.find('span', class_='text').text
    author = quote.find('small', class_='author').text
    print(f"{author}: {text}")

driver.quit()
```

Strategy: Selenium for rendering, BeautifulSoup for parsing

But Is Web Scraping Legal?

Web scraping itself is legal, but it can be illegal depending on what type of data you scrape and how you scrape it. In general, you can legally scrape the internet as long as:

- The data is publicly available
- You do not scrape private information
- You do not scrape copyrighted data
- You do not need to create an account and log in to access the website, OR you have read and fully understand the Terms and Conditions (T&Cs)

But Is Web Scraping Legal?

On the other hand, scraping becomes problematic when:

- Any of the previous conditions are not respected.
- Collecting personal data without a legal basis: Scraping names, emails, addresses, or any other personal data from European citizens without explicit consent or a valid and documented legitimate interest justification violates the GDPR.
- Circumventing technical barriers: Using means to bypass login systems, CAPTCHAs, or IP blocks.
- Scraping data from a competitor to replicate their service and steal their user base can lead to legal action.

POLEMIC CASES: LinkedIn vs. hiQ + Facebook vs. Power Ventures

Best Practices

- **Always consult robots.txt and the Terms of Service**
- **Prioritize the use of official APIs:** If the website offers an API, use it. It is the safest, most stable, and legal way to access data.
- **Be polite (“The Polite Crawler”):** Implement delays (time.sleep()) between requests so as not to overload the target website's server.
- **Minimize data collection:** Collect only the information strictly necessary for your purpose.
- **Avoid personal and sensitive data:** Implement filters to automatically exclude names, emails, and other personal information.
- **Document everything:** If you are basing data collection on “legitimate interest” (especially under the GDPR), document your impact assessment and the safeguards implemented in detail.

Challenges

Technical Challenges

- Dynamic content loaded via **JavaScript**
- Frequent **website layout changes**
- Pagination, infinite scrolling, and lazy loading

Data Quality Challenges

- Missing or inconsistent data
- Encoding and formatting issues

Operational Challenges

- Scalability and performance
- Maintenance and monitoring
- Reproducibility of results

Anti-Scraping Defenses

- Rate limiting and IP blocking
- CAPTCHAs and bot detection
- Browser fingerprinting

Legal & Ethical Challenges

- ToS, copyright and privacy regulations (e.g., GDPR)
- Ethical responsibility to avoid server overload

Future Trends

Increasing Technical Complexity

- More sophisticated anti-bot and fingerprinting systems
- More JavaScript-heavy websites
- Widespread use of client-side rendering

Stronger Legal & Ethical Oversight

- Stricter enforcement of Terms of Service
- Expansion of data protection regulations
- Increased focus on ethical data collection
- Greater legal scrutiny

AI-Assisted Scraping

- Use of machine learning for:
  - Automatic element detection
  - Layout change adaptation
  - Data extraction from semi-structured pages
- Natural language–driven scraping tools

Kahoot

Goal

- To **reward and compensate students** who actively attend and participate in classes
- To reinforce key concepts in an engaging and competitive way

Rules

- Only students **physically present in class** may participate
- No external help (notes, phones, AI tools, or collaboration)
- Any form of cheating results in **disqualification**

Scoring & Rewards

🥇 **1st place**: +0.5 bonus points

🥈 **2nd place**: +0.3 bonus points

🥉 **3rd place**: +0.2 bonus points

(Bonus points are added to the continuous assessment component.)

⚠️ Bonus points are **non-transferable** and **cannot exceed course limits!**

Application to the KE Project

Web scraping is naturally integrated into the data ingestion phase, serving as a primary source of unstructured and semi-structured data and directly feeding the RAG pipeline and the vector database.

Best practices:

- Batch scraping, not real-time
- Data stored in:
  - MongoDB (documents)
  - Vector DB (embeddings)
- Reproducible pipeline (versioned scripts)
- Ingestion logging

What to avoid:

- Scraping during user interaction
- Fragile code without error handling
- Scraping without justifying the source
- Ignoring legal/ethical aspects

Practical Application

➡ **Worksheet PL3**