

Data Acquisition, Processing and Mining for AI

Week 04 — Scraping, ethics, and storage

Silviu Maniu

Extracting Web Data with no API

acquire → understand → clean → mine

- Last week: a contract (documented URLs, JSON, a rate limit)
- Today: what to do when it does not, and what you are allowed to do
- Parquet details

A title ending in (*) marks optional material: context or an extension, dropped first if we run short.

Scraping vs APIs

	output	problems
file / dump	everything, once	staleness
API	data according to contract	versioning, quotas, auth
scraping	Web page content	page changes, redesigns

- An API change is **announced / documented**, page changes **are not**
- Parsing a **presentation format**
- **Always check for an API first.**
- Page built by JavaScript: call its **hidden API** (week 3), or render it with **Playwright**
- Scraping is taught for illustration

One book, in HTML

```
<article class="product_pod">
  <h3><a href="a-light-in-the-attic_1000/index.html"
    title="A Light in the Attic">A Light in the ...</a></h3>
  <p class="star-rating Three"><i class="icon-star"></i>...</p>
  <p class="price_color">£51.77</p>
  <p class="instock availability">In stock</p>
</article>
```

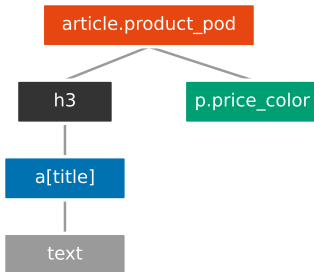
- **books.toscrape.com** = site built to be scraped
- One `article.product_pod` per book, **20 per page**, 50 pages
- **Read the markup.** What the browser displays and what the tag actually holds are **different things**

HTML and the DOM

the markup

```
<article class="product_pod">
  <h3>
    <a title="A Light in the Attic">
      A Light in the ...
    <p class="price_color">£51.77
```

the tree the parser builds



a selector names a path through this tree

- **HTML** consists of nested tags
- the **DOM** (Document Object Model) is the **tree** those tags describe
- Not search in the text, but **walk the tree**
- `class` and `id` are labels that define page style

Parsers and BeautifulSoup

```
soup = BeautifulSoup(html, "html.parser")  
#      \_ the library           \_ the parser
```

- A **parser** turns the markup **text** into the **tree**. Python has `html.parser`; `lxml` is a faster C one. **Both build the same tree**
- **BeautifulSoup** (the package is `bs4`) → the **API you navigate that tree with**
- Why not just search the text? Tags **nest**, attributes come in **any order**, and real HTML is **malformed**

CSS selectors

<code>article.product_pod</code>	tag and class
<code>h3 a</code>	an <code><a></code> anywhere inside an <code><h3></code>
<code>ul.breadcrumb > li</code>	direct children only
<code>p.price_color</code>	class <code>price_color</code> on a <code><p></code>
<code>a[href]</code>	has the attribute <code>href</code>

- **CSS selector** same as the ones you use in a **stylesheet** to style a page
- Selectors are a **W3C standard**: the same syntax works in a stylesheet, in a browser's console, and in `soup.select()`
- Prefer **content classes** (`price_color`) over layout ones (`col-lg-3`)

The real page, in a browser

Books to Scrape We love being scraped!

[Home](#) / [All products](#)

Books

- Travel
- Mystery
- Historical Fiction
- Sequential Art
- Classics
- Philosophy
- Romance
- Womens Fiction
- Fiction
- Childrens
- Religion
- Nonfiction
- Music
- Default
- Science Fiction
- Sports and Games
- Add a comment
- Fantasy
- New Adult
- Young Adult
- Science
- Poetry
- Paranormal
- Art
- Psychology
- Autobiography
- Parenting
- Adult Fiction

All products

1000 results - showing 1 to 20.

Warning! This is a demo website for web scraping purposes. Prices and ratings here were randomly assigned and have no real meaning.



★★★★★

A Light in the ...

£51.77

✓ In stock

Add to basket



★★★★★

Tipping the Velvet

£53.74

✓ In stock

Add to basket



★★★★★

Soumission

£50.10

✓ In stock

Add to basket



★★★★★

Sharp Objects

£47.82

✓ In stock

Add to basket

CSS selectors on a real page

```
<ol class="row">
  <li class="col-xs-6 col-sm-4 col-md-3 col-lg-3">
    <article class="product_pod">
      <div class="image_container">...</div>
      <p class="star-rating Three">
        <i class="icon-star"></i> ...
      </p>
      <h3><a href="a-light-in-the-attic_1000/index.html"
        title="A Light in the Attic">
        A Light in the ...</a></h3>
      <div class="product_price">
        <p class="price_color">£51.77</p>
        <p class="instock availability">...</p>
        <form>...</form>
      </div>
    </article>
  </li>
  ... 19 more <li>, one per book
</ol>

<ul class="pager">
  <li class="current">Page 1 of 50</li>
  <li class="next"><a href="page-2.html">next</a></li>
</ul>
```

.col-lg-3
layout class: avoid

article.product_pod
one per book, 20 per page

p.star-rating
the rating is its 2nd class

h3 a
["title"]: the whole title

p.price_color
text, with the £

li.next a
["href"]: the next page

books.toscrape.com/catalogue/page-1.html, whitespace tidied, ... = cut

Parsing and selecting

```
import requests
from bs4 import BeautifulSoup

r = requests.get(FIRST_PAGE, headers=HEADERS, timeout=10)
r.raise_for_status()
soup = BeautifulSoup(r.text, "html.parser")

for pod in soup.select("article.product_pod"):
    title = pod.select_one("h3 a")["title"]
    price = pod.select_one("p.price_color").get_text(strip=True)
```

- `r.text`: same HTTP as in APIs
- `select` returns a list (possibly empty), `select_one` returns one or None
- `["title"]` reads an attribute; `get_text()` reads the text between the tags

What the tag actually holds

```
<a title="A Light in the Attic">A Light in the ...</a>  
<p class="star-rating Three">  
<p class="price_color">£51.77</p>
```

- Title: the text is **cut**, the full one is in the **title attribute**
- Rating: **no text at all**, only the class **Three**
- Price: **a string** with a £
- **Read the markup.**

Encodings: mojibake

```
Content-Type: text/html  
<meta charset="utf-8">
```

```
<- no charset  
<- the page says utf-8
```

```
>>> r.encoding           'ISO-8859-1'  
>>> r.text[...]          'Â£51.77'  
>>> r.apparent_encoding  'utf-8'
```

- Bytes are **not text**: they have to be decoded based on their **encoding**
- No charset in the header → requests falls back to **ISO-8859-1**, HTTP's historical default
- £ is the two bytes C2 A3 in UTF-8. Decoded as latin-1 that is **two characters**, Â£
- This is **mojibake**, the most common data-corruption bug
- Fix it **at the source**: `r.encoding = "utf-8"` before `r.text`

From tags to rows

```
RATINGS = {"One": 1, "Two": 2, "Three": 3, "Four": 4, "Five": 5}
```

```
row = {  
    "title":      pod.select_one("h3 a")["title"],  
    "price_gbp":  float(pod.select_one("p.price_color")  
                        .get_text(strip=True).lstrip("£")),  
    "rating":     RATINGS[pod.select_one("p.star-rating")["class"][1]],  
}
```

- One line per trap: title from the **attribute**, price as a **float**, rating as an **int**
- `["class"]` is a **list**, `["star-rating", "Three"]`, `[1]` is the rating word position
- Unknown word → **KeyError**: a page change **fails loudly**

Finding the next page

```
<li class="current">Page 1 of 50</li>  
<li class="next"><a href="page-2.html">next</a></li>  
nxt = soup.select_one("li.next a")  
url = urljoin(url, nxt["href"]) if nxt else None
```

- Same as the Link header, but in markup
- href is relative (page-2.html) → urljoin with the current page's URL
- The loop ends when li.next is absent
- Page 1 of 50 = how many pages to expect

What scraping is allowed to do

robots.txt: the format

```
User-agent: *  
Disallow: /api  
Disallow: /search  
Disallow: /advancedsearch
```

```
User-agent: Baiduspider  
Crawl-delay: 10
```

- `openlibrary.org/robots.txt`: always at the **root of the host**, always plain text
- **User-agent** groups the rules: **every group naming the crawler applies**; ***** only if **none does**
- Disallow takes a path pattern: ***** any characters, **\$** end of path
- Several rules match → **the longest wins**, ties go to Allow
- Crawl-delay is **not in RFC 9309**, but widely honoured

- A convention since 1994, an IETF Proposed Standard since 2022
- 404, unavailable: “the crawler MAY access any resources on the server”
- 5xx, unreachable: “the crawler MUST assume complete disallow”
- Cache it and re-fetch at least daily (longer while the server is unreachable)

robots.txt and the terms of service

- robots.txt = a **request to robots**, published by the site. Ignoring it is **not a crime**, but it can weigh against the scraper when there's an issue
- **Terms of service** = a claimed **contract**; they can forbid what robots.txt does not mention
- Whether they **bind** depends on notice, assent and jurisdiction
- Open Library: robots.txt says **Disallow: /search**, but the API guide **says use /search.json**, (with User-Agent and rate limit)
- robots.txt → **crawlers discovering pages**; the API guide → **API clients**. **Read both**, use the **documented channel**

Personal data: the GDPR

- **GDPR** (Regulation (EU) 2016/679) governs **personal data**: anything relating to an **identifiable** person
- A **username** is personal data, and so is a **comment** with its author
- **Public $\neq$ free to process**: publication and processing are **different acts**
- **Data minimization**: collect **only what the question needs**
- **Several project sources return personal data**: GitHub usernames, Hacker News comments
- State **which personal data** the pipeline touches and **how it was minimized**

CNIL 2025: lawful scraping, with conditions

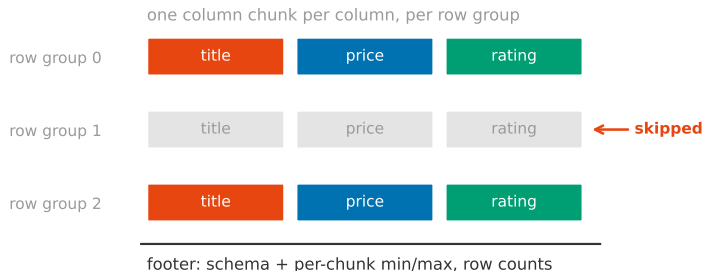
- June 2025, CNIL (the French data-protection regulator) on *moissonnage* (scraping) to build AI training databases
- Such collection may rest on **legitimate interest**: one of the GDPR's six lawful bases, no consent needed, but weighed against the person's rights
- The safeguards it names:
 - collection criteria defined in advance = the project's registered scope
 - exclusion filters, and respect for robots.txt and CAPTCHAs
 - immediate deletion of anything irrelevant or sensitive
- Good practice beyond AI training too; with personal data, read the full text

Polite scraping

- Identify yourself in the User-Agent, with a contact address
- One request at a time, with a delay
- Cache every response: re-running the parser re-fetches nothing
- Take only the pages needed: “grab everything, filter later” is what the CNIL text explicitly forbids
- Dump or API published → use it

Parquet as storage

Parquet: the real file layout



- **Row group** = a horizontal slice, size **picked by the writer** (often $\sim 1\text{M}$ rows)
- Inside it, one **column chunk** per column
- **Footer**: the **schema**, and **per-chunk min/max statistics** (optional, written by default here)
- Example: `price > 50` $\rightarrow$ compare with **min/max**, **skip whole row groups**

CSV and Parquet

```
books.to_parquet("books.parquet", compression="zstd")
```

```
back = pd.read_parquet("books.parquet")
```

```
price = pd.read_parquet("books.parquet", columns=["price_gbp"])
```

- **CSV has no types**: every read **re-guesses** them. One "unknown" in a numeric column → a **column of strings**
- Parquet stores the **schema**: types **round-trip**
- `columns=` reads **only those column chunks** → fast on wide tables
- Compressed **per column** (e.g., `zstd` algorithm): repeated values cost almost nothing → the taxi table is **21.7 MB as CSV, 5.4 MB as Parquet**
- Footer = **fixed overhead**: the **60 books** are **8.5 kB as CSV, 7.6 kB as Parquet, 3.1 kB of it footer**

Limits of Parquet

- Stores **what it is given**: a column of "£51.77" strings works but is **still unusable**
- Lists come back as **numpy.ndarray**
- **No indexes, no constraints, no updates**: appending = **writing another file**
- A **storage** decision. **Modelling** decisions remain: one row per what, which key, missing versus zero

Datasets on the Hub

```
from datasets import load_dataset
```

```
ds = load_dataset("stanfordnlp/imdb", split="train")  
df = ds.to_pandas()
```

- **HuggingFace Hub**: tens of thousands of datasets, **one call**
- Mostly **Parquet files** and HTTP range requests
- A **fourth acquisition channel**, next to dump, API and scrape, acquisition is **already done**
- Their data **decisions come with it**: the dataset card says (or omits) what was collected, when, and what was dropped
- **Not a project source**; but fine for a model training / testing

Lab and next week

Today's lab

- Three exercises are mandatory: a listing parser, a pagination loop, and a stored dataset with a log
- Solved: robots.txt with urllib.robotparser, encoding and attribute traps
- Offline: set OFFLINE=1 to replay cached pages
- Optional, at home: detail pages, the join, and a check against a redesign

Before next week

- M1 is due before the week-5 lab: acquisition log + script + Parquet shapes
- Next lecture: EDA for data quality, summary statistics, missingness, outliers

References

- RFC 9309, *Robots Exclusion Protocol*. <https://www.rfc-editor.org/rfc/rfc9309>
- GDPR — Regulation (EU) 2016/679. <https://eur-lex.europa.eu/eli/reg/2016/679/oj>
- CNIL, *Focus : l'intérêt légitime et la collecte par moissonnage* (2025-06-19).
<https://www.cnil.fr/fr/focus-interet-legitime-collecte-par-moissonnage>
- BeautifulSoup: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- Apache Parquet, file format specification:
<https://parquet.apache.org/docs/file-format/>
- Hugging Face datasets: <https://huggingface.co/docs/datasets>
- Books to Scrape (today's site): <https://books.toscrrape.com>