Coding & Development / Python Prompts

Generate a BeautifulSoup or Selenium script to extract structured data from a described website.
Difficulty: Advanced
Model: GPT-4 / Claude / Gemini
Use Case: Web Scraping, Data Extraction, Automation
Updated: May 2026
Why This Prompt Exists
Most web scraping is done manually — inspecting HTML, writing selectors, handling errors.

You get:

- brittle selectors that break when the site changes
- no error handling for missing elements
- no rate limiting (getting blocked)
- no pagination handling
- scrapers that work once and never again

But a scraper is not a one-off script.

It is a maintainable data extraction system.

- Requests: handle headers, cookies, rate limiting
- Parsing: BeautifulSoup or Selenium for dynamic content
- Selectors: CSS or XPath (robust, not brittle)
- Error handling: missing elements, timeouts, retries
- Data storage: CSV, JSON, or database

Without structure, scrapers break and get blocked.

This framework forces AI to build robust scrapers.

The Prompt

```
Assume the role of a web scraping engineer who builds robust,
maintainable scrapers.

Your task is to generate a web scraper script.

Generate:

1. IMPORTS
   - requests, BeautifulSoup (or selenium)
   - time, csv/json

2. SCRAPER CONFIGURATION
   - Headers (User-Agent)
   - Rate limiting (time.sleep)

3. FETCH FUNCTION
   - Get page HTML
   - Handle HTTP errors
   - Retry logic

4. PARSE FUNCTION
   - Extract data using CSS selectors or XPath
   - Handle missing elements gracefully

5. PAGINATION HANDLING (if applicable)
```

   - Loop through pages
   - Stop when no more pages

6. DATA STORAGE
   - Save to CSV or JSON

7. MAIN FUNCTION
   - Orchestrate the scraping process

INPUTS:

Target URL:
[INSERT]

Data to Extract (list fields with descriptions):
[LIST]

Site Structure (how data is organized):
[E.G., "Product listings on grid, each product has title, price, rating"]

Pagination Type:
[NONE / NEXT BUTTON / URL PARAMETER / INFINITE SCROLL]

Dynamic Content (JavaScript rendered):
[YES / NO]

Rate Limit (requests per second):
[INSERT OR "1"]

```
RULES:
- Always set a User-Agent header (identify your scraper)
- Add delays between requests (be respectful)
- Use CSS selectors or XPath (not regex for HTML)
- Handle missing elements (set to None, don't crash)
- Save data incrementally (don't lose progress on error)
- Check robots.txt before scraping
- Respect website terms of service
```

How To Use It

- Check robots.txt before scraping (e.g., example.com/robots.txt).
- Start with a small test (limit pages to 5) before scaling.
- Add delays between requests (1-2 seconds minimum).
- If the site uses JavaScript, use Selenium (not BeautifulSoup).
- Save data to CSV incrementally to avoid losing progress on errors.
- Be respectful — don’t hammer the server.

Example Input

**Target URL:** https://books.toscrape.com

**Data to Extract:** Title (h3 > a title attribute), Price (price_color class), Rating (star-rating class), Availability (instock availability class)

**Site Structure:** Each book is in an article with class “product_pod”. Next page button exists.

**Pagination Type:** NEXT BUTTON

**Dynamic Content:** NO (static HTML)

**Rate Limit:** 1 request per second

Why It Works

Most scrapers are brittle and get blocked.

This framework improves outcomes by forcing:

- proper headers (avoid blocks)
- rate limiting (be respectful)
- error handling (resilience)
- pagination (completeness)
- data storage (usability)

Great web scrapers don’t just extract data — they handle errors, respect rate limits, and save progress.

## Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, Python frameworks, and practical strategies for developers and engineers.

Carefully engineered prompts for people doing real work.

### Share this:

- Share on Facebook (Opens in new window) Facebook
- Share on X (Opens in new window) X

See also The Python Function Generator