Module 17 - WEB SCRAPING

Welcome to Module 17! This module will equip you with the essential skills to programmatically extract data from websites. While many services offer structured data through Web APIs, the ability to perform web scraping is invaluable for gathering information directly from web pages when no official API exists.

Chapter 1: Fundamentals of Web Scraping

This chapter introduces the core concepts of web scraping, including its purpose, ethical considerations, how to retrieve raw web content, and the critical role of character encodings.

1.1 Introduction to Web Scraping

- **What is Web Scraping?** Web scraping is the automated process of collecting data from websites. It involves:
    1. Sending HTTP requests to web servers to fetch the webpage content (typically HTML or XML).
    2. Parsing the received content to identify and extract specific information.
    3. Storing the extracted data in a structured format (e.g., CSV, JSON, database).
- **Why Web Scrap?** Web scraping is employed for various purposes:
    - **Data Collection:** Gathering large datasets for research, analysis, or machine learning (e.g., product prices, news articles, real estate listings).
    - **Monitoring:** Tracking changes on websites, such as price drops, stock availability, or new job postings.
    - **Integration:** Obtaining data from websites that do not offer a public API, allowing integration into other applications.
    - **Automation:** Automating repetitive data extraction tasks that would be tedious to perform manually.
- **Ethical and Legal Considerations:** Before engaging in web scraping, it is crucial to consider the ethical and legal implications:
    - **`robots.txt`:** Always check a website's `robots.txt` file (e.g., `https://example.com/robots.txt`). This file provides guidelines to web crawlers, indicating which parts of the site they are allowed or disallowed from accessing. **Respect these rules.**
    - **Terms of Service (ToS):** Many websites' Terms of Service explicitly forbid or restrict automated scraping. Violating ToS can lead to legal action, IP blocking, or account termination.
    - **Frequency/Politeness:** Avoid overwhelming the server with too many requests in a short period. Implement delays (`time.sleep()`) between requests to mimic human behavior and prevent being flagged as a malicious bot.
    - **Copyright:** Be aware of copyright laws regarding the data you collect. The data might be copyrighted even if you can scrape it.
    - **Privacy:** Never scrape personal data without explicit consent and ensure compliance with data protection regulations (e.g., GDPR, CCPA).

- **Key Python Libraries:** Throughout this module, we will primarily use:
    - **`requests`**: A powerful and easy-to-use library for making HTTP requests.
    - **`BeautifulSoup4`**: A library for parsing HTML and XML documents, simplifying the process of navigating and searching the parsed tree.

1.2 Unicode and ASCII

- **Character Encoding Explained:** Text on the internet is represented by various character encodings. When a web server sends content, it's a stream of bytes. Your Python program needs to decode these bytes into human-readable strings using the correct encoding.
    - **ASCII (American Standard Code for Information Interchange):** An older, limited encoding primarily for English characters (0-127).
    - **Unicode:** A universal character encoding standard that encompasses nearly all characters from all written languages globally. **UTF-8** is the most common Unicode encoding on the web.
- **Importance in Web Scraping:**
    - If the incorrect encoding is used during decoding, you will encounter "Mojibake" (garbled, unreadable characters like `æ˜¯å®ƒ`).
    - The `requests` library attempts to guess the correct encoding based on HTTP headers and the content itself.
    - **`response.text`**: This attribute provides the content as a string, decoded using `requests`'s best guess of the encoding.
    - **`response.content`**: This attribute provides the raw byte content of the response, useful for binary files (like images) or when you need explicit control over decoding.
    - **`response.encoding`**: The encoding that `requests` *guessed* from the response headers.
    - **`response.apparent_encoding`**: The encoding that `requests` *detects* from the content itself. This is often more reliable if the HTTP headers are missing or incorrect.
    - **Manual Correction:** If `response.text` appears garbled, you can manually set `response.encoding = 'utf-8'` (or the correct encoding) *before* accessing `response.text` to force decoding with a specific charset.

1.3 Retrieving Content from Webpages

The `requests` library is your primary tool for fetching the raw HTML content of a webpage.

- **Installation:**

  Bash

  ```
  pip install requests
  ```

- **Example: Making a GET Request**

  Python

  ```
  import requests
  ```

```
# The URL of the page you want to scrape
# We'll use a dummy website designed for scraping practice:
books.toscrape.com
url = "http://books.toscrape.com/"

try:
    # Send an HTTP GET request to the specified URL
    response = requests.get(url)

    # Raise an HTTPError for bad responses (4xx or 5xx status codes).
    # This is good practice for robust error handling.
    response.raise_for_status()

    # Print the HTTP status code (e.g., 200 for success, 404 for not
found)
    print(f"Status Code: {response.status_code}")

    # Print the first 500 characters of the HTML content
    # response.text holds the decoded content as a string
    print("\n--- HTML Content (first 500 chars) ---")
    print(response.text[:500])

except requests.exceptions.RequestException as e:
    # Catch any request-related exceptions (e.g., connection error,
timeout, HTTP error)
    print(f"An error occurred: {e}")
```

1.4 Getting the Data (Parsing HTML)

Once you have the raw HTML content, you need to parse it to extract specific elements. `BeautifulSoup` excels at this by creating a parse tree that you can navigate.

- **Installation:**

  Bash

  ```
  pip install beautifulsoup4
  ```

- **Example: Parsing HTML and Basic Extraction**

  Python

  ```
  import requests
  from bs4 import BeautifulSoup

  url = "http://books.toscrape.com/"
  response = requests.get(url)
  response.raise_for_status()

  # Create a BeautifulSoup object by parsing the HTML content
  # 'html.parser' is a common and robust built-in parser
  soup = BeautifulSoup(response.text, 'html.parser')

  # --- Basic Data Extraction Examples ---

  # 1. Get the title of the page using dot notation
  page_title = soup.title.string
  ```

```
print(f"\nPage Title: {page_title}")

# 2. Find the first occurrence of a specific HTML tag (e.g., <h1>)
first_h1 = soup.find('h1')
if first_h1:
    # .get_text(strip=True) extracts all text within the tag,
removing leading/trailing whitespace
    print(f"First H1 Tag: {first_h1.get_text(strip=True)}")

# 3. Find all occurrences of a specific HTML tag (e.g., <a> for
links)
all_links = soup.find_all('a')
print(f"\nTotal links found: {len(all_links)}")

# Print the text and 'href' attribute of the first 5 links
print("\nFirst 5 Links:")
for link in all_links[:5]:
    link_text = link.get_text(strip=True)
    link_href = link.get('href') # Use .get() to safely retrieve
attribute values
    print(f"  Text: '{link_text}', Href: '{link_href}'")

# 4. Find elements by tag and attribute (e.g., a specific product)
# We want to find the first book on the page, typically within an
<article> tag
first_product_article = soup.find('article', class_='product_pod') #
Use class_ because 'class' is a Python keyword

if first_product_article:
    # Navigate down the tree from the article to get the title and
price
    product_title = first_product_article.h3.a.get('title')
    product_price = first_product_article.find('p',
class_='price_color').get_text(strip=True)
    print(f"\nFirst Product: {product_title}, Price:
{product_price}")
```

1.5 Storing Data Using Dictionaries

Storing your extracted data in a structured format, such as a list of Python dictionaries, is essential for easy manipulation, analysis, and saving to files.

- **Example: Extracting Multiple Books into a List of Dictionaries**

  Python

```
import requests
from bs4 import BeautifulSoup

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

books_data = [] # Initialize an empty list to store dictionaries of
book information

# Find all HTML <article> tags that have the class 'product_pod'
```

```
# Each 'product_pod' article typically represents a single book
listing
product_articles = soup.find_all('article', class_='product_pod')

for article in product_articles:
    # Extract the title: within <h3>, then <a>, then get the 'title'
attribute
    title = article.h3.a.get('title')

    # Extract the price: within a <p> tag with class 'price_color'
    price = article.find('p',
class_='price_color').get_text(strip=True)

    # Extract the rating: within a <p> tag with class 'star-rating'
    # The actual rating (e.g., 'Two', 'Three') is the last class in
the tag's 'class' attribute list
    rating_tag = article.find('p', class_='star-rating')
    if rating_tag:
        rating_class = rating_tag['class'][-1] # Get the last class
which indicates the rating (e.g., 'Two')
    else:
        rating_class = "N/A" # Handle cases where rating might be
missing

    # Create a dictionary for the current book's information
    book_info = {
        'title': title,
        'price': price,
        'rating': rating_class
    }

    # Add the book's dictionary to our list
    books_data.append(book_info)

print("\n--- Extracted Books Data (First 3) ---")
for book in books_data[:3]: # Print only the first 3 for brevity
    print(book)

print(f"\nTotal books extracted from this page: {len(books_data)}")
```

1.6 Content (Binary Data Handling)

- The `response.content` attribute from the `requests` library provides the raw bytes of the response body.
- This is crucial when dealing with non-text data formats such as images, audio files, video streams, or compressed archives.
- Unlike `response.text`, `response.content` does not attempt any character decoding; it gives you the exact byte stream received from the server.
- **Example: Accessing Raw Bytes**

  Python

```
import requests

# URL for a sample image
image_url =
"https://www.google.com/images/branding/googlelogo/1x/googlelogo_colo
r_272x92dp.png"
```

```
try:
    image_response = requests.get(image_url)
    image_response.raise_for_status() # Check for HTTP errors

    # Access the raw binary content
    image_bytes = image_response.content

    print(f"\nSize of image content (bytes): {len(image_bytes)}")
    # print(image_bytes[:50]) # Prints the first 50 bytes (not human-
readable text)

    # To save this, you would typically write it to a file in binary
mode ('wb')
    # with open('google_logo.png', 'wb') as f:
    #     f.write(image_bytes)
    # print("Image bytes saved to google_logo.png")

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

1.7 Text (Extracted Human-Readable Content)

- `element.get_text()` and `element.text` are `BeautifulSoup` methods used to extract all human-readable text content contained within an HTML tag and all its child tags. They effectively strip out all the HTML tags themselves.
- The `strip=True` argument in `get_text(strip=True)` is highly recommended as it removes leading/trailing whitespace and normalizes internal whitespace, providing cleaner text.
- **Example: Using `get_text()` with `strip=True`**

  Python

```
from bs4 import BeautifulSoup

html_snippet = "<div>  This is <b>some</b> <i>example</i> text.
</div>"
soup_snippet = BeautifulSoup(html_snippet, 'html.parser')
div_tag = soup_snippet.find('div')

print(f"\nOriginal HTML snippet: '{html_snippet}'")
print(f"Text using .get_text(): '{div_tag.get_text()}'") # '  This is
some example text.  '
print(f"Text using .get_text(strip=True):
'{div_tag.get_text(strip=True)}'") # 'This is some example text.'
print(f"Text using .text: '{div_tag.text}'")            # '  This is
some example text.  ' (similar to .get_text() without strip)
```

Chapter 2: Advanced Data Interaction and Storage

This chapter explores how to interact with web forms (using POST requests), manage URL parameters for dynamic content, and store your scraped data efficiently in CSV files.

2.1 Posting the Data

Sometimes, a website requires you to submit data, often through a web form (e.g., login, search, registration). This typically involves sending an HTTP POST request.

- **Steps:**
    1. **Inspect the Form:** Use your browser's developer tools to find the `<form>` tag. Note its `action` attribute (the URL to which data is sent) and `method` attribute (which should be `POST`).
    2. **Identify Input Names:** Look for `<input>`, `<textarea>`, `<select>` tags within the form and note their `name` attributes. These names will be the keys in your `data` payload.
    3. **Use `requests.post()`:** Send the request with the identified data.
- **Example: Simulating a Form Submission (Illustrative)**

  Python

```
import requests

# A public service for testing HTTP POST requests
post_test_url = "https://httpbin.org/post"

# Data to be sent in the POST request. Keys correspond to form input
'name' attributes.
payload = {
    'username': 'my_test_user',
    'password': 'secure_password123',
    'remember_me': 'on' # Checkbox value
}

try:
    # Send a POST request with the 'data' payload
    response = requests.post(post_test_url, data=payload)
    response.raise_for_status() # Check for HTTP errors

    print("\n--- POST Request Response (from httpbin.org) ---")
    # httpbin.org conveniently returns the request details in JSON
format
    print(response.json())
    print(f"Status Code: {response.status_code}")

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

    - **Important Note:** Simulating real-world form submissions (especially logins) can be complex. Real sites often use CSRF tokens, session cookies, and JavaScript, which might require more advanced techniques (e.g., session handling, parsing hidden inputs). This example primarily illustrates the `requests.post()` method with the `data` parameter.
    - For JSON-formatted POST bodies (common in APIs), use `json=payload` instead of `data=payload`.

2.2 Params (URL and Form Parameters)

The `params` argument in `requests` is used for sending data as part of the URL query string in GET requests. The `data` argument (as seen above) or `json` argument is used for sending data in the request body for POST/PUT requests.

- **1. URL Query Parameters (GET Requests):**
  - These parameters appear after a `?` in the URL (e.g., `?query=value&key=value`).
  - `requests` automatically encodes the dictionary you provide to the `params` argument into the correct URL query string.

  Python

```
import requests

# Example: Performing a search on Google (real Google search URL)
search_url = "https://www.google.com/search"

# Parameters to be included in the URL query string
search_params = {
    'q': 'web scraping tutorial python', # 'q' is Google's search
query parameter
    'sourceid': 'chrome',
    'ie': 'UTF-8'
}

try:
    # Send a GET request with the 'params' payload
    response = requests.get(search_url, params=search_params)
    response.raise_for_status() # Check for HTTP errors

    print(f"\n--- GET Request with Params ---")
    print(f"Constructed URL: {response.url}") # Displays the full URL
built by requests
    print(f"Status Code: {response.status_code}")
    # print(response.text[:500]) # Uncomment to see part of the
search results HTML

except requests.exceptions.RequestException as e:
    print(f"Error during GET request with params: {e}")
```

- **2. Form Data Parameters (POST Requests):**
  - As demonstrated in the "Posting the Data" section, the `data` argument in `requests.post()` is used to send form-encoded data (parameters) in the request body.

2.3 - 2.6 Filtering the Data

Efficiently filtering and locating the desired data within the parsed HTML is a core skill in web scraping. `BeautifulSoup` provides several powerful methods.

- **1. Filtering by Tag Name:**
  - `soup.find('tag_name')`: Returns the *first* matching tag.
  - `soup.find_all('tag_name')`: Returns a list of *all* matching tags.

Python

```
# (Revisiting examples from 1.4)
# all_links = soup.find_all('a')         # Finds all anchor tags
# first_h1 = soup.find('h1')             # Finds the first <h1> tag
```

- **2. Filtering by Attributes (Class, ID, etc.):**
  - You can pass attribute values directly as keyword arguments. For `class`, use `class_` because `class` is a Python reserved keyword. For `id`, use `id`.
  - Alternatively, use the `attrs` dictionary for more complex attribute matching.

Python

```
# Find all <p> tags with the class 'price_color'
price_tags = soup.find_all('p', class_='price_color')
print(f"\nPrices found (by class): {[p.get_text(strip=True) for p in
price_tags[:3]]}...")

# Example: Finding an element by its ID (if 'main-content' ID
existed)
# main_content_div = soup.find('div', id='main-content')
# if main_content_div:
#     print(f"Content from main-content:
{main_content_div.get_text(strip=True)[:100]}...")

# Find elements with any attribute, using the 'attrs' dictionary
# elements_with_data_attr = soup.find_all(attrs={'data-category':
'fantasy'})
# print(f"Elements with data-category='fantasy':
{len(elements_with_data_attr)}")
```

- **3. Traversing the Parse Tree:**
  - Once you have an element, you can navigate its relationships within the HTML structure.
    - `.parent`: Accesses the immediate parent tag.
    - `.children`: Provides a generator for direct child tags.
    - `.find_next_sibling()`, `.find_previous_sibling()`: Finds adjacent siblings on the same level.
    - `.find_parent()`, `.find_parents()`: Finds all parent tags up the tree.

Python

```
# Example: Finding a book's rating, then traversing up to its parent
article
# to find the book's title (illustrates how to combine navigation)
first_book_rating_tag = soup.find('p', class_='star-rating')
if first_book_rating_tag:
    # Traverse up to the parent <article> tag which contains all book
details
    parent_article = first_book_rating_tag.find_parent('article')
    if parent_article:
        # Now find the title within that specific parent article
        title_from_parent = parent_article.h3.a.get('title')
        print(f"\nTitle found via parent traversal:
{title_from_parent}")
```

- **4. Using CSS Selectors (`select()` and `select_one()`):**
    - This is often the most concise and powerful way to select elements, especially if you're familiar with CSS syntax.
    - `soup.select('css_selector')`: Returns a *list* of all matching elements.
    - `soup.select_one('css_selector')`: Returns the *first* matching element.

    Python

    ```
    # Select all book titles. CSS: 'article' tag with class
    'product_pod',
    # then descendant 'h3', then descendant 'a'.
    titles_css = soup.select('article.product_pod h3 a')
    print(f"\nTitles found with CSS selector: {[t.get('title') for t in
    titles_css[:3]]}...")

    # Select prices. CSS: 'article' tag with class 'product_pod',
    # then descendant 'p' with class 'price_color'.
    prices_css = soup.select('article.product_pod p.price_color')
    print(f"Prices found with CSS selector: {[p.get_text(strip=True) for
    p in prices_css[:3]]}...")

    # Select the rating for the very first book using select_one
    first_book_rating_css = soup.select_one('article.product_pod p.star-
    rating')
    if first_book_rating_css:
        rating_class_css = first_book_rating_css['class'][-1]
        print(f"First book rating with CSS selector: {rating_class_css}")
    ```

    - **Common CSS Selectors Used in Scraping:**
        - `tag`: Selects all elements with that tag (e.g., `a`, `div`).
        - `.class_name`: Selects elements with that specific class (e.g., `.price_color`).
        - `#id_name`: Selects the element with that specific ID (e.g., `#main-content`).
        - `tag.class_name`: Selects elements that are both a `tag` and have a specific `class` (e.g., `p.price_color`).
        - `parent_tag > child_tag`: Selects direct children.
        - `ancestor_tag descendant_tag`: Selects any descendant of an ancestor.
        - `[attribute="value"]`: Selects elements with a specific attribute having a specific value (e.g., `a[title="A Light in the Attic"]`).

2.7 Saving Data to a CSV File

Saving your extracted data to a CSV (Comma Separated Values) file is a common and convenient way to store structured data for later analysis, import into spreadsheets, or sharing. Python's built-in `csv` module makes this straightforward.

- **Example: Writing Extracted Book Data to CSV**

    Python

    ```
    import csv
    ```

```
import requests
from bs4 import BeautifulSoup

# Re-run the scraping logic to ensure we have the data
url = "http://books.toscrape.com/"
response = requests.get(url)
response.raise_for_status()
soup = BeautifulSoup(response.text, 'html.parser')

books_data = []
product_articles = soup.find_all('article', class_='product_pod')

for article in product_articles:
    title = article.h3.a.get('title')
    price = article.find('p',
class_='price_color').get_text(strip=True)
    rating_tag = article.find('p', class_='star-rating')
    rating_class = rating_tag['class'][-1] if rating_tag else "N/A"

    books_data.append({
        'Title': title, # Using more descriptive column names for CSV
        'Price': price,
        'Rating': rating_class
    })

# Define the CSV file name
csv_file = 'scraped_books.csv'

# Determine the header row for the CSV from the keys of the first
dictionary
# This assumes all dictionaries in books_data have the same keys
if books_data:
    csv_headers = books_data[0].keys()
else:
    # Fallback if no data was scraped
    csv_headers = ['Title', 'Price', 'Rating']

try:
    # Open the CSV file in write mode ('w')
    # 'newline='': Important to prevent extra blank rows on Windows
    # 'encoding='utf-8'': Ensures proper handling of various
characters (e.g., non-English)
    with open(csv_file, 'w', newline='', encoding='utf-8') as f:
        # Create a DictWriter object, which works with dictionaries
        writer = csv.DictWriter(f, fieldnames=csv_headers)

        # Write the header row to the CSV file
        writer.writeheader()

        # Write all the book data rows from the list of dictionaries
        writer.writerows(books_data)

    print(f"\nData successfully saved to {csv_file}")
except IOError as e:
    print(f"Error writing to CSV file: {e}")
```

Chapter 3: Advanced Scraping Techniques

This chapter delves into more specialized scraping techniques, including handling binary files like images and building a practical, continuous monitoring tool such as a price tracer.

3.1 - 3.4 Downloading Images

Downloading images involves accurately identifying image URLs, handling different URL formats, making binary requests, and saving the content to files, often with proper directory management and error handling.

- **1. Identify Image URLs:** Images are usually found within `<img>` tags, with their URL specified in the `src` attribute.
- **2. Handle Relative URLs:** Image `src` attributes can be relative (e.g., `/static/images/logo.png`). You need to combine these with the base URL of the webpage to form a complete, absolute URL using `urllib.parse.urljoin`.
- **3. Download Binary Content Efficiently:** For images, always use `requests.get(url, stream=True)` and save the `response.iter_content()` in chunks. This prevents large image files from consuming excessive memory.
- **4. Directory Management and Error Handling:** Create a dedicated directory for downloads and include robust `try-except` blocks to handle network issues or corrupted image downloads.
- **Example: Downloading Images from a Page**

  Python

```
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin # For correctly joining base URL and
relative URLs
import os
import time # To introduce delays and be polite to the server

base_url = "http://books.toscrape.com/"
download_directory = "downloaded_book_covers" # Name of the directory
to save images

# Create the download directory if it does not already exist
if not os.path.exists(download_directory):
    os.makedirs(download_directory)
    print(f"Created download directory: '{download_directory}'")

try:
    response = requests.get(base_url)
    response.raise_for_status() # Check for HTTP errors
    soup = BeautifulSoup(response.text, 'html.parser')

    # Find all <img> tags on the page
    img_tags = soup.find_all('img')

    print(f"\n--- Starting Image Download Process ---")
    for img_tag in img_tags:
        img_src = img_tag.get('src') # Get the value of the 'src'
attribute
```

```
        if img_src:
            # Construct the absolute URL from the base URL and the
(potentially relative) image source
            absolute_img_url = urljoin(base_url, img_src)

            # Extract the filename from the absolute URL
            filename = os.path.basename(absolute_img_url)
            # Construct the full path where the image will be saved
            filepath = os.path.join(download_directory, filename)

            try:
                # Download the image. Use stream=True for large files
to download in chunks.
                img_response = requests.get(absolute_img_url,
stream=True)
                img_response.raise_for_status() # Check for errors in
image download

                # Open the file in binary write mode ('wb') and save
content in chunks
                with open(filepath, 'wb') as f:
                    for chunk in
img_response.iter_content(chunk_size=8192): # Iterate in 8KB chunks
                        f.write(chunk)
                print(f"  Downloaded: '{filename}'")

            except requests.exceptions.RequestException as e:
                print(f"  Error downloading '{absolute_img_url}':
{e}")
            except IOError as e:
                print(f"  Error saving '{filename}': {e}")

        time.sleep(0.5) # Pause for 0.5 seconds to be polite to the
server

except requests.exceptions.RequestException as e:
    print(f"Error accessing the base URL: {e}")

print("\n--- Image Download Process Complete ---")
```

3.5 - 3.6 Price Tracer

A price tracer is a practical application of web scraping that monitors the price of a product on an e-commerce website and can notify you if the price drops below a specified threshold.

- **Core Logic:**
    1. **Configuration:** Define the product's URL, the target price you're looking for, and potentially HTTP headers (like `User-Agent`) to make your request appear more like a regular browser.
    2. **Scrape Regularly:** Periodically send an HTTP GET request to the product page.
    3. **Extract Price:** Parse the HTML content to accurately locate and extract the current price of the product. This step is the most website-specific and often requires careful inspection of the HTML.
    4. **Compare:** Compare the extracted current price with your predefined target price.

5. **Notify:** If the current price is below or at your target, trigger a notification (e.g., print to console, send an email, send an SMS).
6. **Loop/Schedule:** Implement a loop with a delay (`time.sleep()`) to repeatedly check the price at set intervals. For production, more robust scheduling tools (e.g., cron jobs, `APScheduler`) would be used.

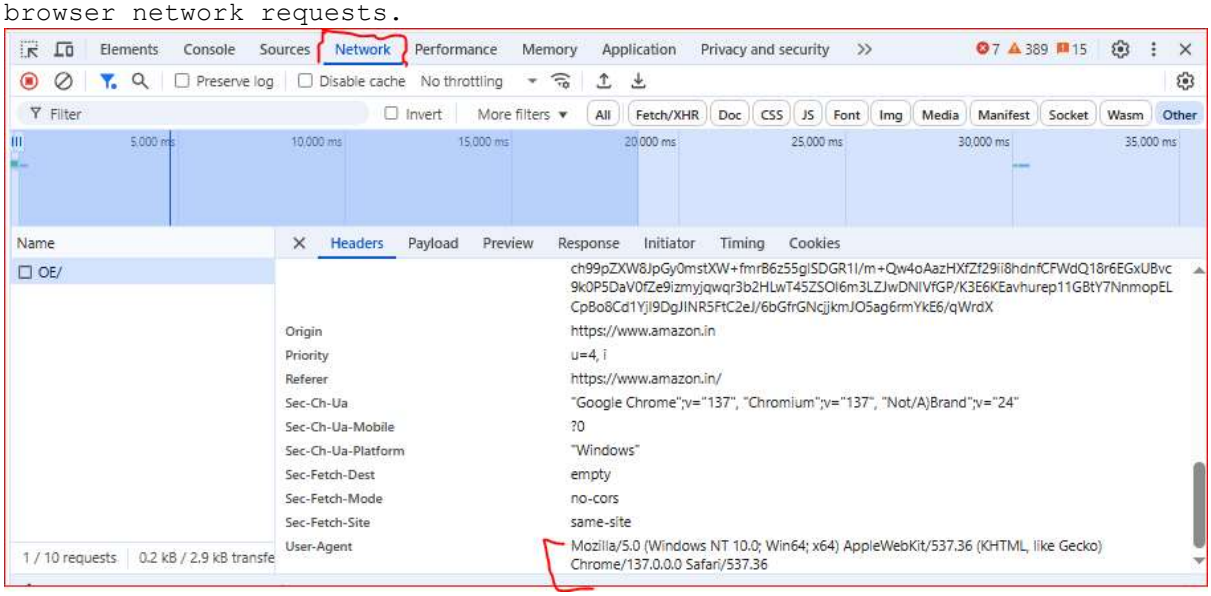
- **Simplified Example: Console-Based Price Tracer**

  Python

```
import requests
from bs4 import BeautifulSoup
import time # For pausing execution
from datetime import datetime # For timestamping messages
import re # For regular expressions, useful for cleaning price text

# --- Configuration for the Price Tracer ---
# The Amazon India URL for Samsung S24 FE
PRODUCT_URL = "https://www.amazon.in/Samsung-Galaxy-Smartphone-128GB-
Storage/dp/B0DHL98QM2/"
TARGET_PRICE_THRESHOLD = 50000.00 # Set your desired maximum price
for the alert (e.g., ₹50,000)

# Mimic a common desktop browser's User-Agent. This is crucial for
Amazon.
# User-Agents can be found by searching "my user agent" or inspecting
browser network requests.
```




```
HEADERS = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0
Safari/537.36",
    "Accept-Language": "en-US,en;q=0.9",
    "Accept-Encoding": "gzip, deflate, br",
    "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,ima
ge/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "DNT": "1", # Do Not Track
    "Connection": "keep-alive",
    "Upgrade-Insecure-Requests": "1"
}
```

```
def get_product_details(url, headers):
    """
    Fetches the product page and attempts to extract its current
price and name.
    Returns a tuple (product_name, current_price_float) or (None,
None) on failure.
    """
    product_name = None
    current_price = None

    try:
        print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}]
Checking: {url}")
        response = requests.get(url, headers=headers)
        response.raise_for_status() # Raise an exception for HTTP
errors (4xx or 5xx)

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

        # --- Product Name Extraction Logic ---
        # Common Amazon selector for product title
        title_tag = soup.find('span', id='productTitle')
        if title_tag:
            product_name = title_tag.get_text(strip=True)
        else:
            print("  Warning: Product title tag (id='productTitle')
not found.")

        # --- Price Extraction Logic ---
        # Amazon's price element can be tricky and vary.
        # A common pattern is a span with class 'a-offscreen' which
holds the full price text.
        price_span = soup.find('span', class_='a-offscreen')

        if price_span:
            price_text = price_span.get_text(strip=True)
            # Clean the price text: remove currency symbols (₹),
commas, and convert to float.
            # Example: "₹49,999.00" -> "49999.00" -> 49999.00
            cleaned_price_text = re.sub(r'[₹,]', '', price_text)
            try:
                current_price = float(cleaned_price_text)
            except ValueError:
                print(f"  Error: Could not convert cleaned price
'{cleaned_price_text}' to number.")
        else:
            # Fallback for other common price elements if 'a-
offscreen' isn't available
            # This might need refinement based on exact page
structure.
            price_tag_alt = soup.find('span', class_='a-price-whole')
            if price_tag_alt:
                whole_part = re.sub(r'[₹,]', '',
price_tag_alt.get_text(strip=True))
                fraction_tag = soup.find('span', class_='a-price-
fraction')
                fraction_part = fraction_tag.get_text(strip=True) if
fraction_tag else "00"
                try:
```

```
                    current_price =
float(f"{whole_part}.{fraction_part}")
                except ValueError:
                    print(f"  Error: Could not combine price parts
'{whole_part}.{fraction_part}' to number.")
            else:
                print("  Warning: Price tag (class='a-offscreen' or
'a-price-whole') not found.")

        return product_name, current_price

    except requests.exceptions.RequestException as e:
        print(f"  Network error or bad HTTP response: {e}")
        return None, None
    except Exception as e:
        print(f"  An unexpected error occurred during product detail
retrieval: {e}")
        return None, None

def notify_price_drop(product_name, current_price, target_price):
    """
    Simulates a notification for a price drop.
    In a real-world application, this function would send an email,
SMS, push notification, etc.
    """
    print(f"\n{'='*30}")
    print(f"         PRICE ALERT!         ")
    print(f"{'='*30}")
    print(f"Product: {product_name if product_name else 'N/A'}")
    print(f"Current Price: ₹{current_price:,.2f}") # Format with
commas and 2 decimal places
    print(f"Target Price: ₹{target_price:,.2f}")
    print(f"Action: Grab it now! {PRODUCT_URL}")
    print(f"{'='*30}\n")

def price_tracker_loop():
    """
    The main loop that continuously checks the price.
    """
    print("--- Starting Amazon Price Tracker ---")

    while True:
        product_name, current_price =
get_product_details(PRODUCT_URL, HEADERS)

        if current_price is not None:
            print(f"  Current price of '{product_name if product_name
else 'Unknown Product'}': ₹{current_price:,.2f}")
            if current_price <= TARGET_PRICE_THRESHOLD:
                notify_price_drop(product_name, current_price,
TARGET_PRICE_THRESHOLD)
                # In a real application, you might want to:
                # 1. Stop tracking after notification (e.g.,
sys.exit()), or
                # 2. Only notify once for a given price drop to avoid
spam, or
                # 3. Implement a cool-down period for notifications.
            else:
                print(f"  Price (₹{current_price:,.2f}) is still
above target (₹{TARGET_PRICE_THRESHOLD:,.2f}).")
        else:
```

```
            print("  Could not retrieve valid price. Retrying...")

        # Wait for a period before checking again (e.g., 2 hours =
7200 seconds)
        sleep_duration_seconds = 7200 # 2 hours
        print(f"  Waiting for {sleep_duration_seconds / 3600:.1f}
hours before next check...")
        time.sleep(sleep_duration_seconds) # Pause execution

# To run the price tracker, uncomment the line below:
# if __name__ == '__main__':
#     price_tracker_loop()
```