Web Scraping Skeleton Code Tutorial (Python)

This tutorial walks you through the fundamental structure of a basic, ethical web scraper using Python's two most popular libraries: **`requests`** for fetching the web page and **`BeautifulSoup`** for parsing the HTML.

The scraper targets a static, public-domain site (`http://quotes.toscrape.com/`) and extracts quotes, authors, and tags, saving the result to a CSV file.

---

1. The Skeleton Code (`scraper_skeleton.py`)

The code is organized into four main functions, following the logical flow of any scraping project: **Fetch, Parse, Save, and Main**.

```python import requests from bs4 import BeautifulSoup import csv import time

# --- Configuration ---

# Target URL for a static, public-domain website

TARGET_URL = "http://quotes.toscrape.com/" OUTPUT_FILE = "scraped_quotes.csv" DELAY_SECONDS = 1 # Ethical scraping: delay between requests

def fetch_page(url ): """Fetches the HTML content of a given URL.""" print(f"Fetching: {url}")
```

Plain Text

```
# Ethical delay
time.sleep(DELAY_SECONDS)

# Set a User-Agent header to identify the scraper (ethical practice)
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
}

try:
    response = requests.get(url, headers=headers)
    response.raise_for_status() # Raise an exception for bad status codes
```

```
(4xx or 5xx)
    return response.text
except requests.exceptions.RequestException as e:
    print(f"Error fetching {url}: {e}")
    return None
```

def parse_quotes(html_content): """Parses the HTML content to extract quotes and authors.""" if not html_content: return []

Plain Text

```
# Initialize BeautifulSoup parser
soup = BeautifulSoup(html_content, 'html.parser')

# Find all quote containers using a CSS selector
# The target site uses the class 'quote' for each quote block
quote_elements = soup.find_all('div', class_='quote')

scraped_data = []

for quote_element in quote_elements:
    # Extract the quote text
    # The quote text is inside a <span> with class 'text'
    text = quote_element.find('span', class_='text').text

    # Extract the author name
    # The author name is inside a <small> with class 'author'
    author = quote_element.find('small', class_='author').text

    # Extract the tags (optional, for demonstration)
    tags = [tag.text for tag in quote_element.find('div',
class_='tags').find_all('a', class_='tag')]

    scraped_data.append({
        'quote': text,
        'author': author,
        'tags': ', '.join(tags)
    })

return scraped_data
```

def save_to_csv(data, filename): """Saves the extracted data to a CSV file.""" if not data: print("No data to save.") return

Plain Text

```

```

```
# Define the fieldnames for the CSV header
fieldnames = ['quote', 'author', 'tags']

try:
    with open(filename, 'w', newline='', encoding='utf-8') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(data)
    print(f"Successfully saved {len(data)} records to {filename}")
except IOError as e:
    print(f"Error saving to CSV: {e}")
```

def main(): """Main function to run the scraper."""

Plain Text

```
# 1. Fetch the HTML content
html = fetch_page(TARGET_URL)

# 2. Parse the content and extract data
data = parse_quotes(html)

# 3. Save the extracted data
save_to_csv(data, OUTPUT_FILE)
```

if **name** == "**main**": main() ```

2. Step-by-Step Code Explanation

A. Configuration and Imports

The initial section sets up the environment and defines constants, which is a best practice for easy modification.

Code Section	Purpose	Key Concept
`import requests`	Handles the HTTP requests (fetching the page).	**Fetching**
`from bs4 import BeautifulSoup`	Handles the HTML parsing and data extraction.	**Parsing**
`import csv, time`	Used for saving the data and implementing ethical delays.	**Data Storage & Ethics**

`TARGET_URL`	The website address to scrape.	**Target Identification**
`DELAY_SECONDS`	A delay to prevent overwhelming the server. **Crucial for ethical scraping.**	**Rate Limiting**

B. The `fetch_page(url)` Function (Fetching)

This function is responsible for making the network request and retrieving the raw HTML.

- **`time.sleep(DELAY_SECONDS)`** : This is the technical implementation of rate limiting. It pauses the script for the configured time to be polite to the server.
- **`headers = {'User-Agent': ...}`** : Setting the `User-Agent` header is an ethical practice. It tells the website server who is making the request, which is often required to avoid being blocked, as many sites block requests that do not appear to come from a standard browser.
- **`response = requests.get(url, headers=headers)`** : This is the core request. The `requests` library handles the network communication.
- **`response.raise_for_status()`** : This is a robust error-handling technique. If the server returns an error code (like 404 Not Found or 500 Server Error), this line immediately raises an exception, preventing the scraper from trying to parse bad data.
- **`return response.text`** : Returns the raw HTML content of the page as a string.

C. The `parse_quotes(html_content)` Function (Parsing and Extraction)

This is the heart of the scraper, where the raw HTML is transformed into structured data.

- **`soup = BeautifulSoup(html_content, 'html.parser')`** : Initializes the BeautifulSoup object. The `html.parser` is the standard Python parser that turns the HTML string into a navigable tree structure (the DOM).
- **`quote_elements = soup.find_all('div', class_='quote')`** : This is the core extraction logic. It uses a **CSS selector** (`div` with class `quote`) to find all the main containers holding the data we want. This is the part that requires inspecting the target website's source code.
- **`for quote_element in quote_elements:`** : The scraper iterates through each container found.
- **`text = quote_element.find('span', class_='text').text`** : Inside each container, we use another CSS selector (`span` with class `text`) to pinpoint the quote text. The `.text` property extracts only the visible text content, stripping away the HTML tags.

D. The `save_to_csv(data, filename)` Function (Saving)

This function handles the output of the structured data.

- **`csv.DictWriter`** : This class is used to write dictionaries (our scraped data) to a CSV file.
- **`fieldnames = ['quote', 'author', 'tags']`** : Defines the column headers for the CSV file.
- **`writer.writeheader()`** : Writes the column headers to the first row of the file.
- **`writer.writerows(data)`** : Writes all the extracted data rows to the file.

E. The `main()` Function

This function orchestrates the entire process, calling the other three functions in sequence: Fetch -> Parse -> Save.

---

3. Next Steps: Expanding the Skeleton

This skeleton is a starting point for **static scraping**. To advance your skills, you would expand this code to handle:

1. **Pagination:** Modify the `main()` function to loop through multiple pages by finding the "Next" button link and calling `fetch_page()` repeatedly until no next page is found.
2. **Error Handling:** Add more specific error handling for different HTTP status codes (e.g., handling a 403 Forbidden error by rotating proxies or changing the User-Agent).
3. **Database Storage:** Replace the `save_to_csv()` function with code that connects to a database (e.g., MongoDB or MySQL) and inserts the data, as discussed in the previous guide.