Salil Seeruttun's Web Scraping Solution Report

Summary

The solution is aimed to automate the extraction of data from specific tables on web pages and process it into structured formats for further analysis. By leveraging web scraping techniques and Python programming, the goal was to streamline the data extraction process and facilitate the generation of structured datasets.

The project utilised a modular approach, separating the web scraping logic into reusable components known as "site extractors," which are responsible for extracting data from different web pages. The primary focus was on extracting data from tables on web pages using *BeautifulSoup* and processing it into *Pandas* DataFrames.

Key steps included:

- Developing utility functions for headless web scraping using *Selenium WebDriver*, saving data to CSV files, and logging errors.
- Implementing a base extractor class as an *abstract base class* (ABC) to define common methods and behaviours for site extractors.
- Creating specific site extractors, such as the NDSLScraper, to extract data from tables on the 'fpi.ndsl.co.in' website
- Utilising YAML configuration files to specify input data and logging configurations.
- Handling errors and logging them to a CSV files.

The project successfully achieved its objectives by automating the extraction and processing of data from web pages, providing structured datasets for analysis, and enabling error logging for troubleshooting purposes.

Approach and Methodology

The project adopted a systematic approach to achieve its objectives of automating data extraction from web pages. Key components of the approach included:

1. **Modular Design with Reusable Components**: The project embraced a modular design, separating the web scraping logic into reusable components known as "site extractors." These extractors were designed to encapsulate the logic for extracting data from specific web pages or sources, allowing for easy extension and maintenance.

2. **Use of Input and Parameters as CSV**: One design decision was to store input data and parameters in CSV format. This approach provided flexibility in managing input configurations and allowed for easy adjustments or additions to the extraction process without modifying the codebase. Additionally, it facilitated collaboration by enabling stakeholders to contribute to the input data without requiring programming knowledge.

3. **Error Handling and Logging**: Error handling was integrated into the project to ensure robustness and reliability. Error messages were logged to a CSV file, including details such as the timestamp, web_id, status (pass or fail), and error message. This logging mechanism enabled the tracking of errors and provided insights into potential issues during the data extraction process. The decision to include the web_id (the primary key of the input file) in the

log file allowed for easy linkage between error messages and specific entries in the input data, aiding in troubleshooting and debugging efforts.

4. **Configuration via YAML Files**: YAML configuration files were utilised to specify input data, logging configurations, and other parameters. This approach facilitated easy configuration management and improved readability compared to hardcoding parameters within the codebase.

Implementation Details

The project is organised into several key components within the src directory:

- **main.py**: The main entry point of the application, responsible for orchestrating the data extraction process and handling configuration settings.
- **utility_functions.py**: Contains utility functions for performing tasks such as headless web scraping, CSV data saving, and error logging.
- **site_extractors**: A directory containing sub-modules for different site extractors, including the base extractor and specific sub-class site extractors such as '**ndsl_site_extractor.py'.**
- **config.yaml**: Configuration file containing settings for input file paths, log file paths, and other parameters.

Notable Challenges

Web Page Fetching Challenges

- **Connection Aborted Errors**: Initially, attempts to fetch data from '**www.fpi.nsdl.co.in/web/Reports/**' using the *requests* library resulted in frequent "Connection aborted" errors, specifically *ConnectionResetError(54, 'Connection reset by peer')*. These errors were encountered due to the server's abrupt termination of the connection. To mitigate this issue, the solution transitioned to using *Selenium WebDriver* with *ChromeDriver*, allowing for more stable and reliable fetching of HTML content from the target web pages.

Requirement for Google Chrome

- Browser Compatibility: The implementation utilises Google Chrome as the headless browser for web scraping tasks. This choice is driven by compatibility considerations, as the *Selenium WebDriver* library is optimised for use with Chrome.

Assumptions

- **Uniform Data Structure**: The project assumes consistent HTML table structures across target web pages for reliable data extraction. Any variations in structure may impact the extraction process.

- **Stable Web Connectivity**: The target web pages (**'www.fpi.nsdl.co.in/web/Reports/'**) are assumed to remain accessible and maintain stable connectivity during data extraction. Disruptions or structural changes could affect reliability.

- **CSV Input Format**: The input CSV file (**'input.csv'**) is expected to follow a specific format, including required columns. Deviations may lead to unexpected behaviour during configuration parsing.

- **CSV Log File Handling**: The project assumes that the log file (**‘log.csv’**) follows a consistent structure for logging errors and status updates. Changes to the log file format may require corresponding modifications to the logging logic within the code.

- **Data Duplication**: Running the code multiple times without clearing existing output will result in duplicate data in the output files.