selenium_navigation.py

1.0 Introduction

selenium_navigation.py is the “Web Crawler” component of the program. It’s job is to crawl through every job entry and retrieve each job's ID. it then creates the URL and has scraper.py extract the job data, and data_formatter.py format the data.

2.0 Architectural and component-level design

1. Crawler.__init__(is_headless):
    a. Initializes the sqlalchemy engine for detecting duplicates.
    b. Creates the browser using the is_headless parameter to determine if the webdriver will be visible to the user.
        i. This is done by calling Crawler.init_browser.
    c. Creates two output files, one for formatted output if needed, one for passing job posts to the formatter.
2. Crawler.is_duplicate_id(id):
    a. Query the database to find if the ID exists in the database.
    b. Return true if an ID exists. Return false if the ID does not exist.
3. Crawler.init_browser(is_headless):
    a. Creates a Selenium options object, and sets it to headless mode as specified
    b. Checks the OS using platform.system() to determine which webdriver needs to be used.
    c. Create the crawler object and use Selenium functionality to log into linkedin.
    d. Return the driver object.
4. Crawler.scrape_jobs(job_name, num_pages, num_jobs, print_results, do_upload)
    a. The primary function of the Crawler class, and arguably the most important function in the backend. This uses the initialized web driver to perform all of

the needed functionality, including calling on data_formatting, which in turn calls the database upload function.

i. The parameters are as follows:

1. job_name: The search term that the crawler scrapes jobs from
2. num_pages: The number of pages that the scraper will scrape. Up to 40 pages can be scraped.
3. num_jobs: The number of jobs scraped per page. Up to the number of jobs on the page can be scraped.
4. print_results: If set to true, the function will print a list of formatted dictionaries of data to formatted.txt.
5. Do_upload: If set to true, data processed will be uploaded to the database. Otherwise, data will not be uploaded.

b. First, the webdriver is navigated to the search bar, and the given job title is entered
c. Then begins a while loop that will loop through the results pages, up to either the maximum, or the given number of pages
d. Using BeautifulSoup4, every job post result's html class is stored into an array.
e. Then, each job post element is processed to extract the 10 digit job ID, and it is added to a list
f. In order to advance the crawler to the next page, we use BS4 to get a list of page number buttons.
g. For each page button, we use two methods to find the specific button for the next page. One of them is used in a specific case where only a button represented by three ellipses is available, (this variable is called exception_button).
h. We extract the html id values from the page number, and then click the button to the next page and increment loop variables.
i. Next, we get values needed for constructing the job page URLs. We determine how many of these numbers we

will actually scrape, and then begin the loop to scrape individual jobs.

j. For every ID in the list, we first check if the ID is a duplicate. Then, we construct the URL and use the Scraper.get_page() and scraper.scrape_linkedin() functions to output the job description to a text file
k. The text file is formatted with the word BREAK followed by the job id, so that the formatter can differentiate between job posts.
    i. We use a file here because it makes testing the formatter easier, as well as to process data in batches.
l. We call the formatter on the output file, and print the results to an output file if specified
    i. The formatter will upload data to the database if specified as well.
m. The output file is cleared and the id list is emptied for the next iteration.

5. Crawler.end_crawling():
    a. All this does is deinstanciate the web drivers for both the scraper and the crawler.

3.0 Testing

1. Primary Testing Method: Due to the nature of the web scraper, conventional unit testing was not viable or useful. The web scraper gathers data from websites through a browser, and as such is expected to handle a non-finite number of possible inputs, in this case job descriptions. Scraping the same job description, or the same page of job descriptions, as a unit test would not prove or disprove functionality. As such, I decided to use full system tests with parameters that could be controlled from web_scraper.py.

4.0 Known Bugs

- Crawler.scrape_jobs():
    - The very first page the web scraper scrapes will have 0 jobs scraped. Every subsequent scrape will perform as normal. This error has been investigated thoroughly, but no root cause has been found. This bug has a limited, if not negligible, effect on the functionality of the product.
        - If the script has ended in the middle of execution either due to an error or user interruption, while data is scraped from job posts, but before it has been sent to the database, on subsequent runs of the program, during the first 0 page scrape, said data will be processed and sent to the database. This is unintended behavior, but does not affect functionality. In fact, this proves to be a useful feature to make sure all scraped data is formatted and sent to the database.